PARALLEL PROGRAMMING LANGUAGES FOR COLLECTIONS

DAN SUCIU

A DISSERTATION

in

COMPUTER AND INFORMATION SCIENCE

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy.

1995

Val Tannen

Supervisor of Dissertation

Peter Buneman

Graduate Group Chairperson

To Adina, and to my parents

Acknowledgments

I am mostly grateful to my advisor Val Tannen, who influenced my work in more ways than in a typical advisor-student relationship. I first met him at the Theoretical Computer Science seminar organized by Octavian Stănăşila at the Polytechnic Institute of Bucharest: I had just entered college, Val had just finished it, and he initiated me in areas of mathematics related to Computer Science, like category theory and universal algebras. Later, in December 1989, he was first to suggest to me that I should start a PhD in Computer Science. After I started such a program at the University of Pennsylvania, he encouraged me to pursue my fascination with parallel languages, and to work in parallel query languages. We had numerous exciting and fruitful discussions and he helped me organizing and crystallizing my thoughts. The Chapters 3, 4, and 5 in this theses are based in part on two papers we wrote together.

I got my background in Computer Science from the Polytechnic Institute of Bucharest, where I first attended as a student, and later joint the faculty. There, I was mostly influenced by Cristian Giumale, who first introduced me to functional programming languages, encouraged me to do research, and taught me how to organize a course. I benefited and learned a lot from working with Cristian, and from the stimulating discussions we had in the research group led by him. I am also in debt to Irina Athanasiu for her constant encouragement.

Peter Buneman got me interested in database query languages, after I came to the University of Pennsylvania. He played a special role for me during my years at Penn. I am grateful for his constantly challenging me with innumerable questions, problems, conjectures, ideas, and for organizing the effervescent database seminar at Penn. The "bounded recursion" (Subsection 3.4.2) is based on his idea of bounding fixpoints over nested relations.

I whish to thank the members of my committee: Serge Abiteboul and Guy Blelloch for their comments, Susan Davidson for her comments and for the discussions we had, Jonathan Smith for opening my eyes to the intellectual challenge of good experimental work, Scott Weinstein for constantly revealing me the beauty of pure mathematics, and for enlightening me on finite model theory, and, of course, Peter Buneman and Val Tannen, for the numerous discussions we had and their constant encouragement.

I found the regular discussions at our informal database seminar at Penn highly stimulating. The members of this group were Peter Buneman, Wolfram Clauss, Susan Davidson, Wenfei Fan, Anthony Kosky, Lenoid Libkin, Rona Machlin, Val Tannen, Limsoon Wong and myself.

While I was working at my thesis, I had the privilege of being invited in a couple of research institutions: the contacts and discussions I made during these visits were invaluable for my work. I whish to thank Laurence Puel for inviting me to spend five weeks at the Universitè de Paris Sud and Serge Abiteboul for inviting me to visit INRIA in the fall 1994, Jan Paredaens for inviting me to visit the University of Antwerpen in November 1994, Dirk Van Gucht for inviting me to visit Indiana University in February 1995, and Victor Vianu for inviting me to visit the University of California at San Diego, in the Spring of 1995. I spent the summer of 1994 at AT&T Bell Laboratories, working in Dave MacQueen's group with Lorenz Huelsbergen on an implementation of data-parallel operations in ML.

I got valuable comments from many people who read the papers on which part of this thesis is based, as well as from people attending my talks where I had the opportunity to present parts of the thesis. It is impossible to mention all of them here, and I apologize for unintentionally omitting some names. For their feedback, comments and encouragements, I wish to thank Catriel Beeri, Guy Blelloch, Peter Buneman, Susan Davidson, Leonidas Fegaras, Peter Freyd, Jean Gallier, Stephane Grumbach, Marc Gyssens, Kyle Hart, Rick Hull, Neil Immerman, Paris Kanellakis, Anthony Kosky, Lenoid Libkin, Steven Lindell, Rona Machlin, Dale Miller, Chris Overton, Jan Paredaens, Jon Riecke, Andre Scedrov, Eric Simon, Jonathan Smith, Val Tannen, Victor Vianu, Patrick Valduriez, Jan Van den Bussche, Dirk Van Gucht, Moshe Vardi, Scott Weinstein, and Limsoon Wong.

During this work I 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.

ABSTRACT

PARALLEL PROGRAMMING LANGUAGES FOR COLLECTIONS

Dan Suciu

Advisor: Val Tannen

The thesis discusses the design, expressive power, and implementation of *parallel programming languages for collections*, the fragment dealing with collections of an object-oriented query language.

The Relational Algebra has a simple, intrinsic parallel semantics, which enabled the successful development of parallel relational database systems. But the implementation techniques of these systems do not carry over to the more complex object-oriented databases. In order to develop efficient parallel object-oriented database systems, one needs to (1) design their query languages with parallelism in mind, and (2) find new implementation techniques, specially designed for these languages. Here we pursue these goals for parallel languages for collections.

The collections of interest for us are sets, bags, and sequences (lists). We start by describing a basic collection calculus and additional forms of recursion on collections. They have an idealized parallel "execution", assuming unbounded resources and instant communication, which gives us high-level parallel complexity measures.

An interesting fragment of the calculus expresses exactly the queries in the parallel complexity class NC. Here the salient construct is *divide and conquer recursion on sets.* Sublanguages obtained by imposing a bound k on the number of recursion nesting correspond to the subclasses AC^k, for $k \geq 1$.

We break the implementation of the calculus into three steps. First, sets and bags are implemented on sequences, using high-level parallel algorithms: we express such algorithms in a high-level language for sequences called $\mathcal{MAP}$, built around a new form of recursion. Second, we describe a complexity-preserving compilation of $\mathcal{MAP}$ on a simple vector-parallel model. Third, we implement the vector model on a parallel multiprocessor. Here we choose as target the LogP model, which can be instantiated to simulate various multiprocessors.

All but one of the vector model instructions require only restricted forms of communication patters on LogP, called *monotone communications*. These in turn admit efficient implementations on LogP.

We ran two simple benchmarks on a LogP simulator, measuring the *speedup* and the *scaleup*. We report conditions under which good speedup and scaleup can be expected.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

One of the major well-known successes of basic, theoretical research in Computer Science has been the development of the relational model for databases [33]. Query languages actually used in industry, like SQL, have a clean mathematical core, which is *First Order Logic*, or, equivalently, *Relational Algebra*. Two major benefits follow: first, the complexity of the queries expressed in these languages is well-understood, e.g. we know what we can and what we cannot express in SQL. Secondly, powerful implementation and optimization techniques have been developed for these languages. Moreover, both benefits are robust: several incremental extensions of the underlying mathematical core have been proposed and shown to admit the same benefits.

While the influence of the relational model on the design of query languages like SQL is well known, less known is the fact that the same model proved to have an unanticipated benefit in parallel computation: easy parallelization [46]. Companies like Teradata, Tandem, etc. have been offering highly successful relational parallel database systems for some time now: see [46] for a review. These systems essentially consist of well-engineered parallel implementations of the relational algebra operators, which leads to an efficient implementation of SQL. Customers running SQL applications can immediately benefit from parallelism by buying parallel data servers and recompiling their applications.

Part of the explanation of this success story lies in the highly parallel nature of the operators in the Relational Algebra and, hence, SQL. But confining a query language to these operators imposes restrictions to the set of applications which can benefit from it. Today's *object-oriented database* applications require more features than the relational model and SQL can offer. E.g. [6] enumerates, among others:

- A variety of collection types: sets, bags, lists, arrays, etc. The relational model only offers *relations*, i.e. sets of tuples.
- Nested collections: e.g. a relation in which one field can be a list. The relational model insists that the fields be atomic type.
- More complex computations than those which can be expressed in the Relational Algebra, e.g. transitive closure.
- Object id's. These would allow both object sharing and object updates.
- Object encapsulation. Adapted from the original programming language view, the database interpretation of the concept is that an object encapsulates both program and data.

Stretching the relational model and its associated Relational Algebra to accommodate some of the above features makes most of the implementation techniques developed for parallel relational database systems useless. Most existing or proposed object-oriented database systems, like O_2[44], ObjectStore [72], Gemstone [23], ODMG [27], where not designed particularly with parallelism in mind. Two notable exceptions are the Bubba database machine implementing the language FAD [9], and the object-oriented storage system SHORE [25]: both deal explicitly with parallelism on flat relations. But keeping parallelism in mind while designing such languages is important: in fact a poor design for an object-oriented query language could make an efficient parallel implementation impossible.

In this thesis we study the design of **parallel programming languages for collections** accommodating most of the above features. We focus both on the mathematical robustness of the underlying language concepts, and on the development of efficient implementation techniques. While our original motivation comes from database query language design, the applicability of the techniques we develop reaches beyond that of database query languages. In fact some of our techniques are inspired from previous work done in conjunction with general-purpose parallel programming languages [13, 49, 87, 100, 38].

Recently Breazu-Tannen, Buneman, and Wong [21, 22] have shown how query languages can be designed around the operations naturally associated to the types of the objects processed. A language like the Relational Algebra, featuring *base types*, *product types* and *set types* will have operations associated to the base types, operations associated to the product type, and operations associated to the set types. Using this clean design principle they show how a variety of well-known query languages, including the Relational Algebra, can be naturally reconstructed. We strive here to apply the same clean design principles for parallel query languages.

As mentioned before, the clean mathematical core of query languages like SQL has enabled a number of studies on its expressive power, and expressive powers of its various extensions. Immerman [61, 62] and Vardi [104] show that, over ordered databases, the Relational Algebra extended with *fixpoints* expresses exactly the queries which are in $PTIME$, while extended with *partial fixpoints* expresses exactly the queries which are in $PSPACE$. Immerman [64], and Abiteboul, Vardi, and Vianu [1] show how various extension of the Relational Algebra capture the classes $DLOGSPACE$, $NLOGSPACE$, $PTIME$, NP, $PSPACE$, and $EXPTIME$.

Parallel programming languages for collections should have a similar well understood mathematical core, and a clean connection with parallel complexity classes. Moreover, the parallel complexities of queries expressed in these languages should be clearly visible in the high-level language. Work in this direction has been done by Immerman [65], who has shown that the Relational Algebra expresses essentially those queries which are computable in constant parallel time on a PRAM with polynomially many processors. Thus queries expressed in the Relational Algebra are, theoretically, easy to compute in parallel. Moreover, the parallel time complexity of a query expressed in the Relational Algebra with iterations corresponds to the number of iterations.

An good implementation of a parallel query language should guarantee its high-level parallel complexity. This can be achieved by using high-level parallel algorithms with guaranteed parallel running time. Many high-level parallel, concurrent, and distributed programming languages have been designed and implemented recently, which could, more or less easily, express such algorithms. There exist parallel extensions of FORTRAN, like High Performance Fortran [71] and PTRAN [5, 41], parallel extensions of C, like C* [100], Split-C [38], and Concert-C [7], parallel extensions of C++, like COOL [30], and Charm++ [68], and applicative parallel programming languages, like NESL [13, 15, 16], Sisal [49, 92, 48], Crystal [31], Proteus [81, 51, 87], and Data-parallel ML [50, 57, 58]. Most of them however have been developed as high-level abstractions or particular parallel architectures, and do not admit a machine-independent definition of the parallel complexity: this makes them good candidates for efficient implementations on those architectures, but poor tools for expressing high-level parallel algorithms. A major exception is NESL, designed by Guy Blelloch [13]. NESL is a powerful, general-purpose parallel functional languages which comes with a high-level definition of the parallel complexity. Certain compilation techniques developed for a simple vector-parallel machine model [12] guarantee the preservation of the time complexity for programs satisfying certain conditions. One of these techniques, *flattening nested parallelism* [17, 12], is essential in the implementation of high-level, parallel query languages on nested collections.

Culler et al. [39] take a critical look at popular models of parallel computation, observing that they reward programming techniques which are hard to implement on existing multiprocessor architectures, by assuming 0-cost communication among processors and infinite bandwidth in these communications. They propose a new low-level model of parallel computation, called the LogP model, which captures accurately the cost of communications on existing hardware. These communication costs should be taken into account by implementations of parallel query languages.

Our grail in this work is to develop a theory of the design and implementation of parallel programming languages for collections unifying a number of the seemingly dispersed principles, results, and techniques enumerated above. We would like to have clean design principles and solid bridges to well-understood parallel complexity classes; complexity-preserving compilation techniques; and provably optimal low-level implementation on realistic models of parallel computation, like the LogP model.

1.2 Thesis

Query languages for collections are part of object-oriented database systems. Here we start from high-level query languages for collections, and end on distributed memory parallel architectures. We do this in several steps.

Collection Languages The collections of interest for us here are *sets*, *bags*, and *sequences* (lists). We identify a certain common structure for languages dealing with these collections. Based on that, we define a basic "calculus", using design principles advocated in [21, 22], namely with operations designed around the types of the objects processed. Thus there will be operations associated to base types, to the product type, to the disjoint sum type, and to collection types. In addition to the operations applicable to all collection types, we consider special operations for bags, and special operations for sequences. Finally, we discuss several forms of recursion on collections. The generic form (for sets, bags, and sequences) of the calculus is the *Nested Collection Calculus*, $\mathcal{NCC}$, Chapter 2. When specialized to sets, we call it the *Nested Relational Calculus*, $\mathcal{NRC}$, (Chapter 3). For bags one needs to add some operations (like unique and monus) (Chapter 2), while for sequences we need to add $\mathbb{N}$ as a basic type, and several other operations ($\mathsf{zip}, \mathsf{enumerate}$, etc., see Chapter 2).

All these operations have a natural, in some sense ideal, parallel "execution" assuming unbounded resources and instant communication. Based on this ideal execution, we define the high-level parallel time and work complexity measures (hinted at in Chapter 2 and developed in more detail, but only for sequences, in Chapter 4). On a given architecture with a fixed number of processors, we strive to achieve an ideal running time that comes out of these complexities using Brent's scheduling principle.

Divide and Conquer Recursion on Sets It is possible to show that an interesting fragment put together from some of theses constructs and with an additional form of recursion on sets, corresponds exactly to NC. The form of recursion is called *divide and conquer recursion on sets* (Chapter 2). Divide and conquer recursion with parameters e, f, u defines the unique function φ, in notation $\mathsf{dcr}(e, f, u)$, taking finite sets as arguments, such that:

$$\begin{aligned} \varphi(\emptyset) &\stackrel{\text{def}}{=} e \\ \varphi(\{y\}) &\stackrel{\text{def}}{=} f(y) \\ \varphi(s_1 \cup s_2) &\stackrel{\text{def}}{=} u\langle\varphi(s_1), \varphi(s_2)\rangle \text{ when } s_1 \cap s_2 = \emptyset \end{aligned}$$

For example, if we take $e \stackrel{\text{def}}{=} \mathsf{false}$, $f(y) \stackrel{\text{def}}{=} \mathsf{true}$ and $u\langle v_1, v_2\rangle \stackrel{\text{def}}{=} v_1 \ xor \ v_2$, then $\varphi = \mathsf{dcr}(e, f, u)$ computes the parity of its input set: $\varphi(s) = \mathsf{true}$ iff $|s|$ is odd. Here $|s|$ denotes the cardinality of a set s. As another example, we can compute the transitive closure of some binary relation r, by taking $e \stackrel{\text{def}}{=} \emptyset$, $f(y) \stackrel{\text{def}}{=} r$ and $u\langle r_1, r_2\rangle \stackrel{\text{def}}{=} r_1 \cup r_2 \cup r_1 \circ r_2$, where $r_1 \circ r_2$ is relation composition: then, the transitive closure of r is obtained by applying φ to the set of nodes of the relation r, namely $tc(r) = \varphi(\Pi_1(r) \cup \Pi_2(r))$, where Π_1, Π_2 are the relational projections.

In general, $\mathsf{dcr}(e, f, u)$ is well-defined when there is some set containing e and the range of f, on which u is associative, commutative and has the identity e. For parity, this is the set $\mathbb{B}$ of booleans, while for transitive closure, it is the set $\{r \cup r^2 \cup \ldots \cup r^n \mid n \geq 0\}$.

We will show in Chapter 3 that the fragment dealing with flat relations of the Nested Relational Calculus extended with dcr expresses over ordered databases exactly those queries which are in NC. Here NC is the class of functions which can be computed on a PRAM in polylogarithmic parallel time and with polynomially many processors: it is regarded as the class of tractable parallel functions, in the same way in which $PTIME$ is considered to be the class of tractable sequential functions. Moreover, $PTIME$ can be captured in a similar way, by replacing the divide and conquer recursion with *element step recursion.*

We extend this result to the whole Nested Relational Calculus and show that together with a variant of divide and conquer recursion expresses exactly the NC queries over ordered, nested databases. At a finer level, we establish connections between the nesting depth of the dcr construct and the AC^k subclasses of NC.

These results are related to previous results by Immerman and Vardi [61, 104, 62], showing that first order logic with *fixpoints* expresses exactly the $PTIME$ queries over ordered databases.

map-Recursion A desirable object-oriented query language may contain all or only some of the constructs in the Nested Collection Calculus. Here we show how to implement all of them. The implementation consists of several steps. Sets and bags, together with their associated operations, will be implemented on sequences (Chapter 4). To implement the set and bag operations efficiently, we use high-level, parallel algorithms on sequences, such as Valiant's parallel merge algorithm. Expressing these algorithms requires a more powerful form of recursion than dcr. We choose a certain recursion schema, called map-recursion (Chapter 4), as central form of recursion in our implementation, because it is sufficiently powerful to allow us to express the parallel algorithms we need, and at the same time can be implemented with while-loops with just a little increase in the work complexity.

The map-recursion restricts the programmer to define recursive functions f only according to the schema in Figure 1.1. Namely to compute $f(x)$, the input x is first *divided* into a number of subproblems, say $x_0, x_1, \ldots, x_{n-1}$, then f is recursively applied in *parallel* on each of the n subproblems, and finally the n results are *combined*. We show in Chapter 4 that many familiar recursion schema can be automatically translated into map-recursive definitions. In fact, there are very few useful algorithms which cannot be expressed using map-recursion.

Thus, the sequence query language, extended with map-recursion, gives us a language called $\mathcal{MAP}$, in which all the collection query language constructs can be implemented efficiently (Chapter 4).

$$
\begin{array}{l}
\mathsf{fun}\ f(x) = \mathsf{let}\ \mathsf{val}\ [x_0, x_1, \ldots, x_{n-1}] = d(x) \\
\qquad\qquad\qquad\quad \mathsf{val}\ [y_0, y_1, \ldots, y_{n-1}] = [f(x_0), f(x_1), \ldots, f(x_{n-1})] \\
\qquad\qquad\quad \mathsf{in}\ c\langle x, [y_0, y_1, \ldots, y_{n-1}]\rangle \\
\qquad\qquad\quad \mathsf{end}
\end{array}
$$

Figure 1.1: Basic map-recursive schema

Complexity-Preserving Compilation The next problem is to implement $\mathcal{MAP}$, and this is further broken down into a compilation on a BVRAM, a vector-parallel machine model described in Chapter 5, and an implementation of the BVRAM on a parallel multiprocessor. We prove that, for any $\varepsilon > 0$, it is possible to compile a $\mathcal{MAP}$ program with parallel time complexity T and work complexity W into a BVRAM program with parallel time complexity $O(T)$ and work complexity $O(W^{1+\varepsilon})$.

Moreover, this can be done while keeping the BVRAM very simple: it has only limited forms of permutations on sequences, and a bounded number of sequence registers. These limitations will be explored further. In particular, the BVRAM admits an provably efficient implementation on a butterfly network. As a consequence, we can design efficient algorithms for the butterfly network in the high-level language $\mathcal{MAP}$ (Chapter 5).

Monotone Communications The last step is to implement the BVRAM on an actual multiprocessor architecture. Given that the technology is moving rapidly, we choose to do it on the LogP model developed by Culler et al. [39], which can be instantiated to capture various architectures, see Chapter 6. Under this model, a parallel computation is performed by a set of P independent nodes, connected by a network enabling point-to-point communications: the L, o, and g parameters stand for the *latency*, *overhead*, and *gap* of the network, and together describe with precision the cost and limitations of the communications [39].

We describe in Chapter 6 an implementation of our low-level, parallel vector model, the BVRAM, on the LogP model. A key observation for an efficient implementation is the fact that the all but one of the BVRAM instructions require only special kinds of communications in the LogP model, called *monotone communications.* We show in Chapter 6 that a monotone communication without message replication (which we call a *one-to-one monotone communication*) admits a provably optimal implementation on the LogP model. This, together with the provably optimal algorithms for broadcast and summation for the LogP model described by Karp, Sahay, Santos, and Schauser [69], allows us to design an efficient implementation of the BVRAM on the LogP model.

Experiments Finally we run two simple benchmarks on a LogP simulator to test the feasibility of our implementation, and measured the *speedup* and the *scaleup*, two widely accepted performance metrics for parallel database systems [46]. The speedup measures the ability of an implementation to run faster on the same database, as the number of processors is increased, while the speedup measures its ability to run in the same time, when both the number of processors and the size of the database are increased at the same rate. We tried to engineer carefully our implementation of the BVRAM on the LogP model, to allow communications and computations to overlap.

The first benchmark is a flat database benchmark, while the second is an object-oriented database benchmark. The experiments show that good speedup and scaleup characteristics are achievable for both benchmarks, provided that two conditions are met: (1) the amount of local data per processor is reasonably large (say $\geq 10,000$ words), and (2) the "residual work", i.e. the useful work which each of the P processors has to perform after the communication phase, is sufficiently large, in order to justify the relatively high communication cost paid for uniformly distributing this work on the processors.

1.3 Overview of Results

Query languages and parallel complexity classes In Chapter 3 we study the relationship between various query language and parallel complexity classes. Our first result here is that the Relational Algebra extended with divide and conquer recursion on sets expresses over ordered databases exactly the queries which are in NC, Theorem 3.7.1. Second, we prove that the Nested Relational Algebra extended with a variant of the divide and conquer recursion expresses over ordered nested databases exactly the queries which are in NC. Both results can be refined to establish relationships between fragments of these query languages and certain subclasses of NC: the language obtained by restricting the nesting depths of divide and conquer recursion to $\leq k$ capture exactly the queries which are in AC^k, for $k \geq 1$.

As mathematical constructs, *finite sets* can be described in two different ways as freely generated algebraic structures. One such description gives rise to a programming construct which inspired divide and conquer recursion on sets. The other inspires a different form of recursion, called here *element step recursion* on sets. It can be shown that $PTIME$ is captured in exactly the same way by element step recursion on sets, in which NC is captured by divide and conquer recursion, Proposition 3.7.7. Thus, the two complexity classes NC and $PTIME$ can be described in terms of two alternative ways of recurring on sets.

Along the way we prove a number of results relating divide-and-conquer recursion and element-step-recursion with other forms of recursion considered in the past, prove that the side-conditions for divide-and-conquer recursion are undecidable, and relate different encoding techniques for complex objects.

Design of a high-level implementation language, $\mathcal{MAP}$ In Chapter 4 we design the implementation language $\mathcal{MAP}$, by adding map-recursion to our language for sequences. We define in an machine-independent manner its parallel execution, and its parallel complexities. As evidence for its expressiveness, we give a list of parallel algorithms expressed in $\mathcal{MAP}$, and show how our parallel constructs for sets and bags, and divide and conquer recursion, can be efficiently implemented in $\mathcal{MAP}$.

Efficient compilation of $\mathcal{MAP}$ We describe the target machine, the BVRAM, in Chapter 5. Our main result here consists in showing that $\mathcal{MAP}$ can be compiled on the BVRAM while preserving the parallel time complexity, and with an arbitrarily small overhead in the work complexity, Theorems 5.2.1 and 5.4.1. This can be done despite the simplicity of the BVRAM model: it has a fixed number of vector registers, and its instructions allow us to perform only limited kinds of permutations on a sequence.

The simplicity of the BVRAM pays off in the form of more efficient implementations on another model of parallel computation: the butterfly network. Our second result in Chapter 5 shows that each of the BVRAM instructions can be efficiently implemented on a butterfly network, Proposition 5.3.1: this result would fail, had we considered a more powerful version of the BVRAM. By combining these two results, one can express efficient algorithms for a butterfly network in $\mathcal{MAP}$, Section 5.5.

The third result gives a lower bound on the cost of performing an arbitrary permutation of a sequence in $\mathcal{MAP}$ or on a BVRAM. Thus, as models of parallel computations, both the language $\mathcal{MAP}$ and the BVRAM make the cost of communication visible.

As a last application of the compilation theorem, we validate $\mathcal{MAP}$ as a model of parallel computation, by comparing it with traditional models, like the PRAM, and the Alternating Turing Machines.

Implementation of the BVRAM on the LogP model In Chapter 6 we discuss the implementation of the BVRAM instructions on the LogP model. Our only *theoretical* result here consists in showing that *monotone, one-to-one* communications admit a provably optimal implementation on the LogP model. Moreover, the algorithm implementing the optimal communication is simple and straightforward.

Our main results in this part of the thesis are *practical*. Namely we show that our approach of implementing a declarative, parallel query language in $\mathcal{MAP}$, compiling that into BVRAM and implementing the latter on the LogP model passes, under reasonable conditions, important tests in parallel databases: *speedup* and *scaleup*. We show this by running experiments both on a flat relational benchmark, and on an object-oriented database benchmark. The conditions are that (1) enough data resides at each processor, and (2) the benchmark is computation-intensive: then the cost of communications needed to balance the work among the physical processors is compensated by the gain of dividing the useful work among the processors.

Chapter 2

Parallel Computations on Collections

This Chapter is an overview of the operations on collections considered in this thesis. We start by describing generic operations which apply to all collections considered here: sets, bags, and sequences. These operations rely on a common structure of the three collection types, and they form a generic calculus, the *Nested Collection Calculus*. Next we discuss additional operations needed for bags, and especially for sequences, as well as forms of recursion associated to collections. All operations have a natural parallel execution associated to them, and from that we derive a machine-independent notion of parallel time complexity, assuming infinite resources and 0-cost communication: a complete definition of the parallel complexity is postponed however until Chapter 4 however. Finally, we discuss existing multiprocessor architectures, and describe the ideal running time for a parallel query implemented on such an architecture.

2.1 Types

In this chapter we present the basic operations occuring in parallel query languages for sets, bags and sequences. All operations are *strongly typed*. The **types**, are constructed from a set of base types by combinations of *product type*, *sum type*, and *collection type* constructors. The base types are denoted by $\mathbb{D}_1, \mathbb{D}_2, \ldots$ and may include types like *string*, *real*, *int*, $\mathbb{N}$, etc. The types are given by the grammar:

$$t ::= \mathbb{D}_1 \mid \mathbb{D}_2 \mid \ldots \mid t \times \ldots \times t \mid t + \ldots + t \mid \lceil t \rfloor$$

We will denote with $\mathbb{D}$ some arbitrary domain of the base type, instead of $\mathbb{D}_i$ with $i \geq 1$. The type $t_1 \times \ldots \times t_n$ denotes the **product type** and contains all n-tuples $\langle x_1, \ldots, x_n \rangle$ where $x_1 \in t_1, \ldots, x_n \in t_n$. The next construct, $t_1 + \ldots + t_n$, is the **disjoint sum 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 $t_1 \times \ldots \times t_n$: *unit* has only one 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.

The salient type construct for our languages however is the **collection type** construct, $\lceil t \rfloor$. It contains all finite "collections" $\lceil x_0, \ldots, x_{n-1} \rfloor$, where $x_0, \ldots, x_{n-1} \in t$. We shall consider three particular kinds of collections: sets, bags (i.e. multi-sets) and sequences:

- *Sets* are unordered and have no duplicates. We denote sets like $\{4, 3, 6\}$. Then $\{3, 6, 4, 3\}$ is the same as $\{4, 3, 6\}$.

- *Bags* are still unordered, but here the number of elements matters. We will use the notation $\{\!|3, 6, 4, 3|\!\}$ for a bag; $\{\!|3, 6, 4, 3|\!\}$ is the same as $\{\!|6, 3, 3, 4|\!\}$, but different from $\{\!|4, 3, 6|\!\}$.

- *Sequences* are ordered collections, which we write as $[6, 3, 4, 3]$. The sequences $[6, 3, 4, 3]$, $[6, 3, 3, 4]$ and $[6, 3, 4]$ are all different.

The three different collection types find applications in different areas:

- Sets and bags are fundamental types in relational databases. The parallel functional languages we discuss in Chapter 3 are intended as a core of a parallel database query languages.

- Sequences play a dual role. First they are fundamental types, together with sets and bags, in object-oriented databases, see e.g. [27]: they can be used to represent a variety of datastructures, like graphs, trees, etc, see [12]. Secondly we use them as central types in the design of a parallel implementation language, see Chapter 4.

2.2 The Nested Collection Calculus, $\mathcal{NCC}$

The *Nested Collection Calculus* is a functional language with variables. The name "calculus" should be understood in the spirit of the "lambda-calculus", i.e. a notation for values and functions, and not in the spirit of the "relational calculus", i.e. a logic-based language.

The language is parameterized by a set of *base types*, *constants* and *external functions*. We write $c : t_c$ for a constant c of type t_c, and $p : d_p \to c_p$ for an external function with domain d_p and codomain c_p. Examples of constants are: numbers, e.g. $0, 1, 2, \ldots, -1, -2, \ldots$, strings, e.g. "abcxd", etc. The external functions may be operations on the base types (like $+, -, *, /$, string concatenation, etc.), or user-defined library functions. We shall mention whenever our results depend on certain assumptions on these constants and external functions. We denote with Σ the set of base types, constants, and external functions, by which our languages are parameterized.

We assume an infinite set of variables $\mathcal{X}$ to be given. We define a **type context** Γ to be a set of the form $\Gamma = \{x_1 : s_1, \ldots, x_n : s_n\}$, where $x_1, \ldots, x_n$ are distinct variables and $s_1, \ldots, s_n$ are types. The expressions of the calculi fall into two distinct syntactic categories: **term expressions** and **function expressions**. We shall write $\Gamma \rhd e : t$ whenever the term expression e has type t under the type context Γ. Similarly we write $\Gamma \rhd f : t_1 \to t_2$ whenever the function expression f has type $t_1 \to t_2$ under the type context Γ.

The basic idea in [21, 22], which proved quite fruitful, is to design languages for collections

$$\frac{}{x : t, \Gamma \rhd x : t}$$

$$\frac{}{\Gamma \rhd c : t_c}(c \in \Sigma) \qquad \frac{}{\Gamma \rhd p : d_p \to c_p}(p \in \Sigma)$$

$$\frac{\Gamma \rhd e_1 : \mathbb{D} \quad \Gamma \rhd e_2 : \mathbb{D}}{\Gamma \rhd e_1 = e_2 : \mathbb{B}}$$

Figure 2.1: The definition of $\mathcal{NCC}$: Variables, Constants, External Functions, Equality

$$\frac{\Gamma \rhd e_1 : t_1, \ldots, \Gamma \rhd e_n : t_n}{\Gamma \rhd \langle e_1, \ldots, e_n \rangle : t_1 \times \ldots \times t_n} \qquad \frac{\Gamma \rhd e : t_1 \times \ldots \times t_n}{\Gamma \rhd \pi_i^{t_1 \times \ldots \times t_n}(e) : t_i}(i = 1, n)$$

Figure 2.2: The definition of $\mathcal{NCC}$: operations associated to the product type

$$\frac{\Gamma \rhd e : t_i}{\Gamma \rhd in_i^{t_1 + \ldots + t_n}(e) : t_1 + \ldots + t_n}(i = 1, n)$$

$$\frac{\Gamma \rhd e : t_1 + \ldots + t_n \quad x_1 : t_1, \Gamma \rhd e_1 : t \quad \ldots \quad x_n : t_n, \Gamma \rhd e_n : t}{\Gamma \rhd (\mathsf{case}\ e\ \mathsf{of}\ in_1^{t_1 + \ldots + t_n}(x_1) \Rightarrow e_1 \mid \ldots \mid in_n^{t_1 + \ldots + t_n}(x_n) \Rightarrow e_n) : t}$$

Figure 2.3: The definition of $\mathcal{NCC}$: operations associated to the sum type

$$\frac{x : t_1, \Gamma \rhd e : t_2}{\Gamma \rhd \lambda x{:}t_1.e : t_1 \to t_2} \qquad \frac{\Gamma \rhd f : t_1 \to t_2 \quad \Gamma \rhd e : t_1}{\Gamma \rhd f(e) : t_2}$$

Figure 2.4: The definition of $\mathcal{NCC}$: Functions

$$\frac{}{\Gamma \rhd \lceil\rfloor : \lceil t \rfloor} \qquad \frac{\Gamma \rhd e : t}{\Gamma \rhd \lceil e \rfloor : \lceil t \rfloor} \qquad \frac{\Gamma \rhd e_1 : \lceil t \rfloor \quad \Gamma \rhd e_2 : \lceil t \rfloor}{\Gamma \rhd e_1 @ e_2 : \lceil t \rfloor}$$

$$\frac{\Gamma \rhd f : t_1 \to \lceil t_2 \rfloor}{\Gamma \rhd \mathsf{ext}(f) : \lceil t_1 \rfloor \to \lceil t_2 \rfloor} \qquad \frac{\Gamma \rhd e : \lceil t \rfloor}{\Gamma \rhd \mathsf{empty}(e) : \mathbb{B}}$$

Figure 2.5: The definition of $\mathcal{NCC}$: operations common to all collections

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

Figure 2.6: The definition of $\mathcal{NCC}$: Weakening

by considering tuples, sums, and collections as orthogonal. Hence, there will be primitives that work on tuples (Figure 2.2), primitives that work on sums (Figure 2.3), and primitives that work on collections (Figure 2.5). In addition we have constants, variables, external functions, and equality at all base types (Figure 2.1), constructs dealing with functions defined in $\mathcal{NCC}$(Figure 2.4), and finally a rule allowing us to use more variables than strictly needed (Figure 2.6). We emphasize the fact that there are no higher-order functions definable in $\mathcal{NCC}$, e.g. the type of a function cannot be $(t_1 \to t_2) \to t_3$.

Altogether, these rules define **the Nested Collection Calculus** $\mathcal{NCC}(\Sigma)$; there will be some additional operations on bags and sequences, which we discuss in the following Section. We abbreviate this language with $\mathcal{NCC}$ when Σ contains only base types and constants (i.e. no external functions).

The meaning of these expressions is as follows. $\langle e_1, \ldots, e_n \rangle$ denotes an n-tuple. $\pi_i^{t_1 \times \ldots \times t_n}$, for $i = 1, n$, are the projections: $\pi_i^{t_1 \times \ldots \times t_n} \langle x_1, \ldots, x_n \rangle \stackrel{\text{def}}{=} x_i$. $\lceil\rfloor$ is the empty collection, $\lceil e \rfloor$ denotes the singleton collection, and $e_1 @ e_2$ is the "union" of the collections e_1 and e_2; it is true union in the case of sets, bag addition for bags, and sequence concatenation for sequences. We shall use the symbols $\cup$ and $\oplus$ for set union and bag addition, and use @ only for sequence concatenation. When the kind of collection is unspecified however, we will use the symbol @. Finally $\mathsf{ext}(f)(\lceil x_0, \ldots, x_{n-1} \rfloor) \stackrel{\text{def}}{=} f(x_0) @ \ldots @ f(x_{n-1})$. The lambda expression $\lambda x{:}t.e$ denotes a function in which x is the input variable.

We shall drop the type superscripts and the types in the λ-abstractions whenever they can be deduced from the context, and we shall 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 λ-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 $\lambda z{:}t_1 \times t_2 . \langle \pi_2^{t_1 \times t_2}(z), \pi_1^{t_1 \times t_2}(z) \rangle$.

As usual, we distinguish between *free* and *bound* variables: x becomes bound in $\lambda x{:}t.e$. $f(e)$ denotes function application. We define a **query** to be a closed function expression $f : t_1 \to t_2$ (i.e. with no free variables).

As an alternative for the $\mathsf{ext}(-)$ construction we will sometimes consider the constructs

$$\frac{\Gamma \rhd f : t_1 \to t_2}{\Gamma \rhd \mathsf{map}(f) : \lceil t_1 \rfloor \to \lceil t_2 \rfloor} \qquad \frac{\Gamma \rhd e : \lceil \lceil e \rfloor \rfloor}{\Gamma \rhd \mathsf{flatten}(e) : \lceil t \rfloor}$$

Figure 2.7: The definition of $\mathcal{NCC}$: map and $\mathsf{flatten}$ - alternatives for ext

$\mathsf{map}(-)$ and $\mathsf{flatten}$, whose types are given in Figure 2.7.

$\mathsf{map}(-)$ and $\mathsf{flatten}$ can be defined in terms of $\mathsf{ext}(-)$:

$$\begin{aligned} \mathsf{map}(f) &\stackrel{\text{def}}{=} \mathsf{ext}(\lambda x{:}t_1.\lceil f(x) \rfloor) \\ \mathsf{flatten} &\stackrel{\text{def}}{=} \mathsf{ext}(\lambda x{:}\lceil t \rfloor.x) \end{aligned}$$

Their meanings are given by:

$$\begin{aligned} \mathsf{map}(f)(\lceil x_0, \ldots, x_{n-1} \rfloor) &= \lceil f(x_0), \ldots, f(x_{n-1}) \rfloor \\ \mathsf{flatten}(\lceil x_0, \ldots, x_{n-1} \rfloor) &= x_0 @ \ldots @ x_{n-1} \end{aligned}$$

Conversely, $\mathsf{ext}(-)$ can be expressed in terms of $\mathsf{map}(-)$ and $\mathsf{flatten}$:

$$\mathsf{ext}(f)(x) \stackrel{\text{def}}{=} \mathsf{flatten}(\mathsf{map}(f)(x))$$

We shall mention in the sequel whenever we assume the $\mathsf{ext}(-)$ or the $\mathsf{map}(-)$ and $\mathsf{flatten}$ presentation of the calculus.

Example 2.2.1 *The database projections* $\Pi_i : \lceil t_1 \times \ldots \times t_i \rfloor \to \lceil t_i \rfloor$, $i = 1, n$, *are defined by* $\Pi_i = \mathsf{map}(\pi_i)$.

Example 2.2.2 *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 by: $distribute_{left} = \lambda\langle x{:}t_1, Y{:}\lceil t_2 \rfloor\rangle.(\mathsf{ext}(\lambda y{:}t_2.\langle x, y \rangle)(Y))$. *Note the essential role of the free variable* x *in the function* $\lambda y{:}t_2.\langle x, y \rangle$. *Also,* $distribute_{right} : \lceil t_1 \rfloor \times t_2 \to \lceil t_1 \times t_2 \rfloor$ *can be defined similarly.*

$$\frac{\Gamma \rhd e_1 : \mathbb{B} \quad \Gamma \rhd e_2 : t \quad \Gamma \rhd e_3 : t}{\Gamma \rhd \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : t}$$

Figure 2.8: The types for the if construct, an alternative for case and sum types

Example 2.2.3 *For $x_1 : \lceil t_1 \rfloor, x_2 : \lceil t_2 \rfloor$, the Cartesian product is defined by:*

$$\times \stackrel{\text{def}}{=} \lambda\langle x_1{:}\lceil t_1 \rfloor, x_2 : \lceil t_2 \rfloor\rangle.\mathsf{ext}(\lambda u{:}t_1.\mathit{distribute}_{\mathit{left}}\langle u, x_2\rangle)(x_1)$$

For sets and bags, this is equivalent to:

$$\times' \stackrel{\text{def}}{=} \lambda\langle x_1, x_2\rangle.\mathsf{ext}(\lambda v.\mathit{distribute}_{\mathit{right}}\langle x_1, v\rangle)(x_2)$$

but for sequences the two expressions differ. E.g. for $x_1 = [a_0, \ldots, a_{m-1}], x_2 = [b_0, \ldots, b_{n-1}]$, we have:

$$\begin{aligned} x_1 \times x_2 &= [\langle a_0, b_0\rangle, \langle a_0, b_1\rangle, \ldots, \langle a_0, b_{n-1}\rangle, \langle a_1, b_0\rangle, \ldots, \ldots, \langle a_{m-1}, b_{n-1}\rangle] \\ x_1 \times' x_2 &= [\langle a_0, b_0\rangle, \langle a_1, b_0\rangle, \ldots, \langle a_{m-1}, b_0\rangle, \langle a_0, b_1\rangle, \ldots, \ldots, \langle a_{m-1}, b_{n-1}\rangle] \end{aligned}$$

Example 2.2.4 *For three expressions $e_1 : \mathbb{B}, e_2 : t, e_3 : t$, we abbreviate* (if e_1 then e_2 else e_3) *for* (case e_1 of $in_1^{unit+unit}(x_1) \Rightarrow e_1 \mid in_2^{unit+unit}(x_2) \Rightarrow e_2$)*. Then all boolean operations* not, and, or *at type $\mathbb{B}$ are easily definable. E.g. x* and *y* $\stackrel{\text{def}}{=}$ if x then y else false.

In fact, the sum types are redundant: they can be replaced with a primitive type $\mathbb{B}$, and an if − then − else construct, as given in Figure 2.8. Let us denote with $\mathcal{NCC}^-(\Sigma)$ the language $\mathcal{NCC}(\Sigma)$ without sum types, but extended with $\mathbb{B}$ and if − then − else. We state the following proposition without proof. Wong [108] proves a similar statement for sets and bags: it is easy to extend that proof to sequences.

Proposition 2.2.5 *$\mathcal{NCC}^-(\Sigma)$ and $\mathcal{NCC}(\Sigma)$ have the same expressive power, i.e. they can express exactly the same functions over types not involving the sum type operator $+$.*

Example 2.2.6 *Let $p : t \to \mathbb{B}$ be some predicate expressible in the language. Then, we define filter$(p) : \lceil t \rfloor \to \lceil t \rfloor$, as follows: filter$(p)$* $\stackrel{\text{def}}{=}$ ext($\lambda x.$if $p(x)$ then $\lceil x \rfloor$ else $\lceil \rfloor$)*. From here we can derive: select : $\lceil \mathbb{B} \times t \rfloor \to \lceil t \rfloor$ as select$(x) = \Pi_2(\mathit{filter}(\lambda\langle b, _\rangle.b)(x))$.*

Since bags and sequences have more structure than sets, $\mathcal{NCC}$ has additional operations for these two collection types, which we discuss next.

2.3 Particular Languages for Sets, Bags, and Sequences

2.3.1 Languages for Sets and Bags

In the case of sets, $\mathcal{NCC}$ contains all operations of interest for us. In Chapter 3 we will study the expressive power of the restriction of $\mathcal{NCC}$ to sets: we will call this restriction $\mathcal{NRC}$, for **Nested Relational Calculus**. In order to simplify some technical details, we drop the sum types from $\mathcal{NRC}$, and add the boolean type $\mathbb{B}$ and $\mathsf{if} - \mathsf{then} - \mathsf{else}$ as a new primitives: Proposition 2.2.5 tells us that there is no loss in generality in doing that. Thus the types of $\mathcal{NRC}$ are given by the grammar:

$$t ::= \mathbb{D} \mid \mathbb{B} \mid t \times \ldots \times t \mid \{t\}$$

We shall use more traditional notations in $\mathcal{NRC}$, namely $\cup$ instead of @ and $\emptyset$ instead of $\{\}$.

For the purpose of a good match with parallel complexity classes we include in $\mathcal{NRC}$ a new operation, get, whose type is given in Figure 2.9. Its meaning is $\mathsf{get}\langle s, y\rangle = x$ when s is the singleton $\{x\}$ and $\mathsf{get}\langle s, y\rangle = y$ when s is not a singleton. Strictly speaking get is not inherited from $\mathcal{NCC}$: we add it to $\mathcal{NRC}$ because it is not expressible in terms of the other operations (Subsection 3.4.3), and without it the connection with parallel complexity classes given in Chapter 3 wouldn't be so smooth. But we will show (Subsection 3.4.3) that get does not really affect the expressive power of $\mathcal{NRC}$: all functions returning sets in $\mathcal{NRC}$ are expressible without get too. We insist in keeping get somehow separate from the $\mathcal{NRC}$ operations inherited from $\mathcal{NCC}$ in order to facilitate a comparison between $\mathcal{NRC}$ and other query languages for nested relations considered in the literature [2, 91, 101, 84, 22] which do not have a get-like function.

Example 2.3.1 *In $\mathcal{NRC}$, we can express the equality predicate $=_t: t \times t \to \mathbb{B}$ at every type*

$$\overline{\mathsf{get} : \{\mathbb{D}\} \times \mathbb{D} \to \mathbb{D}}$$

Figure 2.9: The Function get in $\mathcal{NRC}$

t, by induction on the type t. Indeed, once we have $=_{t_1}$, $=_{t_2}, \ldots,$ *and* $=_{t_n}$, *we can define* $=_{t_1 \times \ldots \times t_n}$ *by:*

$$(x_1 =_{t_1 \times t_2} x_2) \stackrel{\text{def}}{=} (\pi_1(x_1) =_{t_1} \pi_1(x_2) \text{ and } \ldots \text{and } \pi_n(x_1) =_{t_n} \pi_n(x_2))$$

Now suppose we have an equality predicate on t *and we want to defined one on* $\{t\}$*. We proceed as follows:*

- *First define* $member_t : t \times \{t\} \to \mathbb{B}$*.* $member_t\langle x, Y\rangle$ *returns true iff* $x \in Y$*.*
- *Next define* $included_t : \{t\} \times \{t\} \to \mathbb{B}$*.* $included_t\langle X, Y\rangle$ *returns true iff* $X \subseteq Y$*.*
- *Finally define equality at type* $\{t\}$*.*

Namely:

$$\begin{array}{rcl}
member_t & \stackrel{\text{def}}{=} & \lambda\langle x{:}t, Y{:}\{t\}\rangle.\mathsf{not}(\mathsf{empty}(\mathsf{ext}(\lambda(y{:}t).\mathsf{if}\ x =_t y\ \mathsf{then}\ \{\langle\rangle\}\ \mathsf{else}\ \emptyset)(Y))) \\
included_t & \stackrel{\text{def}}{=} & \lambda\langle X{:}\{t\}, Y : \{t\}\rangle.\mathsf{empty}(filter(\lambda x{:}t.\mathsf{not}(member_t\langle x, Y\rangle))(X)) \\
(X =_{\{t\}} Y) & \stackrel{\text{def}}{=} & included_t\langle X, Y\rangle \mathsf{and}\ included_t\langle Y, X\rangle
\end{array}$$

When no external functions are present, $\mathcal{NRC}$ has the same expressive power as other languages for complex objects discussed in the literature:

Proposition 2.3.2 *[34, 108, 21, 22]* $\mathcal{NRC}$ *has essentially the same expressive power as Abiteboul and Beeri's* algebra without powerset *[2], as Schek and Scholl's* NF^2 relational algebra *[91], as Thomas and Fischer's* algebra *[101], and as Paredaens and Van Gucht's* nested algebra *[83, 84].*

In the **Relational Algebra** [33, 3] the only types are *flat relation types*, i.e. products of types of the form $\{\mathbb{D} \times \ldots \times \mathbb{D}\}$. The operations are: $\cup, -, \times, \Pi_i, \sigma_p$, where $-$ is set difference and σ_p is *selection* – what we call here *filter*. Paredaens and Van Gucht [84], and Wong [107] prove the following *conservativity property*: the functions expressible in $\mathcal{NRC}$ over flat relation types coincide with those expressible in the Relational Algebra.

Early work on *nested relations* [91, 101, 83] has focused on *nest* and *unnest* as salient operations associated to nested relations. Here $unnest : \{t \times \{t'\}\} \to \{t \times t'\}$ is a variation of flatten, and is defined in $\mathcal{NRC}$ as $unnest(s) \stackrel{\text{def}}{=} \mathsf{flatten}(\mathsf{map}(distribute_{left})(x))$, while $nest : \{t \times t'\} \to \{t \times \{t'\}\}$ has the following meaning: $nest(s) = \{\langle u, V\rangle \mid \exists v.\langle u, v\rangle \in s \wedge V = \{v \mid \langle u, v\rangle \in s\}\}$. It can be expressed in $\mathcal{NRC}$ too, as:

$$nest(s) = \mathsf{map}(\lambda u.\mathsf{ext}(\lambda\langle u', v\rangle.\mathsf{if}\ u = u'\ \mathsf{then}\ \{v\}\ \mathsf{else}\ \emptyset)(s))(\Pi_1(s))$$

Note that $unnest(nest(s)) = s$, but $nest(unnest(s)) \neq s$ in general.

Using the conservativity result we conclude that $\mathcal{NRC}$ cannot express *transitive closure* and *parity*. Here, transitive closure is defined as $tc : \{\mathbb{D} \times \mathbb{D}\} \to \{\mathbb{D} \times \mathbb{D}\}$, $tc(x) \stackrel{\text{def}}{=} \{\langle u, v\rangle \mid \exists u_0, u_1, \ldots, u_n, u_0 = u, u_n = v, \forall i.\langle u_{i-1}, u_i\rangle \in x\}$, and $parity : \{\mathbb{D}\} \to \mathbb{B}$ $parity(x) = \mathsf{true}$ iff $|x|$ is odd, where $|x|$ denotes the cardinality of the set x. In Chapter 3 we show that $\mathcal{NRC}$ extended with a form of recursion called *divide and conquer recursion on sets* captures exactly the parallel complexity class NC. In particular, it can express both transitive closure and parity.

From the point of view of parallel evaluation, $\mathcal{NRC}$ expresses only functions of a very low parallel complexity: namely we prove in Proposition 3.7.5 that all queries expressible in $\mathcal{NRC}$ are in AC^0, i.e. can be computed in constant parallel time on a CRCW PRAM with polynomially many processors.

In the case of bags, it turns out that a number of tractable operations on bags cannot be expressed with the operations so far in $\mathcal{NCC}$. Indeed, Libkin and Wong [77] consider the following operations on bags:

$$\mathsf{max} \quad : \quad \{\!|t|\!\} \times \{\!|t|\!\} \to \{\!|t|\!\}$$

$$\begin{aligned}
\mathsf{min} &: \{\!|t|\!\} \times \{\!|t|\!\} \to \{\!|t|\!\} \\
\mathsf{subbag} &: \{\!|t|\!\} \times \{\!|t|\!\} \to \mathbb{B} \\
\mathsf{unique} &: \{\!|t|\!\} \to \{\!|t|\!\} \\
\mathsf{monus} &: \{\!|t|\!\} \times \{\!|t|\!\} \to \{\!|t|\!\}
\end{aligned}$$

Their meaning is easiest expressed in terms of the function $count : t \times \{\!|t|\!\} \to \mathbb{N}$, defined to be $count\langle x, s\rangle \stackrel{\text{def}}{=} n$ iff the number of occurences of x in s is n. Then:

$$\begin{aligned}
count\langle x, \mathsf{max}\langle s_1, s_2\rangle\rangle &\stackrel{\text{def}}{=} \max\langle count\langle x, s_1\rangle, count\langle x, s_2\rangle\rangle \\
count\langle x, \mathsf{min}\langle s_1, s_2\rangle\rangle &\stackrel{\text{def}}{=} \min\langle count\langle x, s_1\rangle, count\langle x, s_2\rangle\rangle \\
\mathsf{subbag}\langle s_1, s_2\rangle &\stackrel{\text{def}}{=} (\forall x. count\langle x, s_1\rangle \leq count\langle x, s_2\rangle) \\
count\langle x, \mathsf{unique}(s)\rangle &= \begin{cases} 1 & \text{if } count\langle x, s\rangle > 0 \\ 0 & \text{if } count\langle x, s\rangle = 0 \end{cases} \\
count\langle x, \mathsf{monus}\langle s_1, s_2\rangle\rangle &\stackrel{\text{def}}{=} count\langle x, s_1\rangle \mathbin{\dot{-}} count\langle x, s_2\rangle
\end{aligned}$$

Here $m \mathbin{\dot{-}} n \stackrel{\text{def}}{=} m - n$ if $m \geq n$, and 0 otherwise.

We illustrate with two simple examples:

$$\begin{aligned}
\mathsf{unique}\{\!|a, a, a, b, c, c|\!\} &= \{\!|a, b, c|\!\} \\
\mathsf{monus}\langle \{\!|a, a, a, b, c, c, d|\!\}, \{\!|a, b, b, c, e|\!\}\rangle &= \{\!|a, a, c, d|\!\}
\end{aligned}$$

Libkin and Wong prove that all above operations are tractable, that they are not expressible in a language obtained by instantiating $\mathcal{NCC}$ to bags, and that all can be expresses with only two of them: monus and unique. We therefore include monus and unique in $\mathcal{NCC}$.

2.3.2 Languages for Sequences

Sequences differ from sets and bags in that their elements have an associated *position*, which is a natural number. We assume that $\mathbb{N}$ is a base type in the language, and that basic arithmetic operations $+, *, /$ are included in Σ. Also we assume *monus* to be in Σ, written $m \mathbin{\dot{-}} n$, defined as $m - n$ when $m \geq n$ and 0 otherwise.

$$\frac{}{\Gamma \rhd \Omega^t : t} \qquad \frac{\Gamma \rhd e : [t]}{\Gamma \rhd \mathsf{length}(e) : \mathbb{N}} \qquad \frac{\Gamma \rhd e : [t]}{\Gamma \rhd \mathsf{get}(e) : t}$$

$$\frac{\Gamma \rhd e_1 : [t_1] \quad \Gamma \rhd e_2 : [t_2]}{\Gamma \rhd \mathsf{zip}\langle e_1, e_2\rangle : [t_1 \times t_2]} \qquad \frac{\Gamma \rhd e : [t]}{\Gamma \rhd \mathsf{enumerate}(e) : [\mathbb{N}]} \qquad \frac{\Gamma \rhd e : [t] \quad \Gamma \rhd e' : [\mathbb{N}]}{\Gamma \rhd \mathsf{split}\langle e, e'\rangle : [[t]]}$$

Figure 2.10: Specific operations for sequences

As for the case of bags, we observe that some desirable tractable operations on sequences cannot be expressed in $\mathcal{NCC}$ with the primitives described so far. We will add to $\mathcal{NCC}$ the operations on sequences listed in Figure 2.10, and describe their semantics below.

Ω^t denotes an error value at type t: error values arise naturally in conjunction with operations on sequences. The function $\mathsf{get}(x)$ expects x to be a sequence of length one, and in that case returns the only element of that sequence. zip and $\mathsf{enumerate}$ are defined by:

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

zip returns an error (i.e. Ω) when the two sequences have different lengths. Finally, $\mathsf{split}(x, y)$ is a kind of an inverse to $\mathsf{flatten}$. 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. $\mathsf{split}([a, b, c, d, e], [2, 0, 3]) = [[a, b], [], [c, d, e]]$.

Example 2.3.3 *Equality at all types can be expressed in $\mathcal{NCC}$ in a similar way to Example 2.3.1. Namely:*

$$(X =_{[t]} Y) \stackrel{\text{def}}{=} (\mathsf{length}(X) =_{\mathbb{N}} \mathsf{length}(Y)) \text{ } \mathsf{and} \text{ } (\mathsf{empty}(\mathit{filter}(\lambda\langle x, y\rangle.\mathsf{not}(x =_t y))(\mathsf{zip}\langle X, Y\rangle)))$$

Example 2.3.4 *The* bounded monotone routing *$bmRoute : [t'] \times [\mathbb{N}] \times [t] \to [t]$ has the following meaning: $bmRoute\langle u, d, x\rangle$ replicates each element of the sequence x a number of times dictated by the corresponding element in d. Moreover, u should match in length the final result; u is called* the bound. *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. E.g. $bmRoute\langle [u_0, u_1, u_2, v_0, v_1], [3, 0, 2], [a, b, c]\rangle = [a, a, a, c, c]$. The* *bound u prohibits us from constructing a very long sequence in constant parallel time. The definition is:* $bmRoute\langle u,d,x\rangle \stackrel{\text{def}}{=} \Pi_2(\mathsf{flatten}(\mathsf{map}(distribute_{right})(\mathsf{zip}\langle\mathsf{split}\langle u,d\rangle,x\rangle)))$.

Example 2.3.5 *Familiar operations on lists are easily derived. first and tail can be defined by:*

$$\begin{aligned} first(x) &\stackrel{\text{def}}{=} \mathsf{get}(\mathsf{get}(bmRoute\langle[\langle\rangle],[1,0],\mathsf{split}(x,[1,\mathsf{length}(x)\dot{-}1])\rangle)) \\ tail(x) &\stackrel{\text{def}}{=} \mathsf{get}(bmRoute\langle[\langle\rangle],[0,1],\mathsf{split}(x,[1,\mathsf{length}(x)\dot{-}1])\rangle) \end{aligned}$$

If x is empty, split *will produce an error. Similarly we can define last and remove_last, which return the last element, and delete the last element from a sequence, respectively. Here $x \dot{-} y$ (*monus*) is defined on natural numbers to be $x-y$ when $x \geq y$, and 0 otherwise.*

Example 2.3.5 suggests that sequences behave like lists: in fact, we can even append two sequences in one step, with the primitive operation @. But we will see that we can also extract in one step an element from a given position of a sequence, see the function *index* in Section 4.6: in this respect, sequences behave more like arrays.

2.4 Recursion

We discuss here recursion schemas to be added on top of $\mathcal{NCC}$. *Structural recursion* is a declarative construct, which is derived from the fundamental mathematical properties of the various collection types. It seems to be to restrictive for sets, so we slightly relax its definition to *divide and conquer recursion* on sets. Finally, map-*recursion* is a functional programming construct, obtained by generalizing divide and conquer recursion even further. Note however that none of these recursion schemas is strictly speaking in $\mathcal{NCC}$: when added to $\mathcal{NCC}$, each of them increases its expressive power.

2.4.1 Structural Recursion

Breazu-Tannen and Subrahmanyam [19] discuss what kind of operations are *naturally* associated with a collection type $\lceil t \rfloor$. They start from the observation that in all three examples

$$\frac{e : t' \quad f : t \to t' \quad u : t' \times t' \to t'}{\mathsf{sru}(e, f, u) : \lceil t \rfloor \to t'}$$

$$\begin{array}{rcl} \varphi(\lceil\rfloor) & \stackrel{\mathrm{def}}{=} & e \\ \varphi(\lceil y \rfloor) & \stackrel{\mathrm{def}}{=} & f(y) \\ \varphi(s_1 @ s_2) & \stackrel{\mathrm{def}}{=} & u(\varphi(s_1), \varphi(s_2)) \end{array}$$

Figure 2.11: Structural recursion on the union presentation

of collections, the structure $(\lceil t \rfloor, @, \lceil\rfloor)$ is a monoid, which in some sens is freely generated by the set t. More precisely:

- $([t], @, [])$ is the monoid freely generated by t.
- $(\{\!|t|\!\}, \oplus, \{\!|\,|\!\})$ is the commutative monoid freely generated by t. Indeed, bag addition $\oplus$ is commutative, $s_1 \oplus s_2 = s_2 \oplus s_1$.
- $(\{t\}, \cup, \emptyset)$ is the commutative and idempotent monoid freely generated by t. Indeed, set union $\cup$ is both commutative and idempotent (i.e. $s \cup s = s$).

It follows that whenever (t', u, e) is a similar kind of monoid as $(\lceil t \rfloor, @, \lceil\rfloor)$ (that is, commutative, or commutative and idempotent), any function $f : t \to t'$ can be uniquely extended to a monoid homomorphism $\varphi : \lceil t \rfloor \to t'$. This lead [19] to define a programming construct called *structural recursion on the union presentation.* We discuss it here.

The **structural recursion on the union presentation** with parameters e, f, and u, defines a function $\varphi : \lceil t \rfloor \to t'$ as in Figure 2.11. Formally we write $\varphi \stackrel{\mathrm{def}}{=} \mathsf{sru}(e, f, u)$.

In order for φ to be well defined, we need to impose conditions on the parameters e and u (no conditions are needed for f). These conditions depend on the particular collection type considered. They are:

- For *sequences*:

$$\begin{aligned} u\langle e, x\rangle &= u\langle x, e\rangle = x && \text{Identity} \\ u\langle x_1, u\langle x_2, x_3\rangle\rangle &= u\langle u\langle x_1, x_2\rangle, x_3\rangle && \text{Associativity} \end{aligned}$$

- For *bags* we impose both conditions for sequences, and add a third condition:

$$u\langle x_1, x_2\rangle = u\langle x_2, x_1\rangle \qquad \text{Commutativity}$$

- For *sets* we impose all three conditions above and add:

$$u\langle x, x\rangle = x \qquad \text{Idempotence}$$

Example 2.4.1 *We can express the function reverse* : $[t] \to [t]$ *on sequences using structural recursion as*

$$reverse \stackrel{\text{def}}{=} \mathsf{sru}([], \lambda y.[y], \lambda\langle s_1, s_2\rangle.s_2@s_1)$$

Obviously, the function $\lambda\langle s_1, s_2\rangle.s_2@s_1$ *has identity* $[]$ *and is associative, hence reverse is well defined.*

Example 2.4.2 *We can express parity* : $\{\!|t|\!\} \to \mathbb{B}$, *which returns* true *on bags with an odd number of elements, as: parity* $\stackrel{\text{def}}{=} \mathsf{sru}(\mathsf{false}, \lambda y.\mathsf{true}, \lambda\langle b_1, b_2\rangle.b_1 \mathsf{xor}\ b_2)$*. Here* xor *is* exclusive or *between* b_1 *and* b_2*. Since* xor *is associative, commutative, and has identity* false*, it follows that parity is well defined.*

Structural recursion on the union presentation is a high-level, declarative construct. It has an equational theory, namely that of Figure 2.11, which opens the door to powerful optimizations. Still, it suffers a major drawback: checking the side-conditions is, in general, undecidable (see Theorem 3.4.16 for sets, and [19] for sequences and bags). This remains true for all other forms of recursions with side-conditions which we will consider further. In certain cases we know how to define decidable sublanguages with the same expressive power, but these sublanguages are not so elegant.

$$\begin{aligned}
\varphi(\{\}) &\stackrel{\text{def}}{=} e \\
\varphi(\{y\}) &\stackrel{\text{def}}{=} f(y) \\
\varphi(s_1 \cup s_2) &\stackrel{\text{def}}{=} u\langle\varphi(s_1), \varphi(s_2)\rangle \text{ when } s_1 \cap s_2 = \emptyset
\end{aligned}$$

Figure 2.12: Divide and conquer recursion on sets

We found however that for sets, its expressive power is limited. For example we cannot define *parity* for sets in the way we defined it for bags in Example 2.4.2, because xor is not idempotent. This lead us to consider a generalization of sru for sets, which we call *divide and conquer recursion.*

2.4.2 Divide and Conquer Recursion

Operationally, a function $\varphi : \lceil t \rfloor \to t'$ defined by structural recursion on the union presentation, $\varphi = \mathsf{sru}(e, f, u)$, is computed as follows. To compute $\varphi(s)$: (1) decompose s as $s = s_1 @ s_2$, (2) compute recursively $x_1 = \varphi(s_1), x_2 = \varphi(s_2)$, and (3) compute the result as $u\langle x_1, x_2\rangle$. The conditions imposed on u and e make this definition mathematically elegant, by allowing the decomposition at step (1) to be arbitrary. The structural recursion on the union presentation is a *declarative* construct, allowing the compiler (interpreter) to choose the most convenient way of computing the function: this paves the way for database optimizations.

In practice, many useful algorithms are expressed by controlling the decomposition at step (1) more closely. We shall impose here a slight control on the way this decomposition is done for sets, by requiring s_1 and s_2 to be disjoint. We call the resulting form of recursion *divide and conquer recursion on sets.* Formally a function φ is defined by **divide and conquer recursion on sets** with parameters e, f, u, iff it is defined as in Figure 2.12.

For definedness, u has to be associative, commutative, and have identity e: what the additional condition $s_1 \cap s_2 = \emptyset$ buys us is that u is not required to be idempotent. E.g. we can express parity on sets, *parity* : $\{t\} \to \mathbb{B}$, using divide and conquer recursion on sets, in the same way as we expressed it on bags in Example 2.4.2. In fact, another way of understanding dcr is to observe that it is sru on bags, composed with the function $setToBag : \{t\} \rightarrow \{\!|t|\!\}$, which produces a bag in which each element occurs exactly once.

Any function defined by structural recursion on the union presentation on sets, $\varphi = \mathsf{sru}(e, f, u)$, is automatically defined by divide and conquer recursion, as $\varphi = \mathsf{dcr}(e, f, u)$, because we only have to "forget" that u is idempotent. We do not know whether the converse is true: more, we do not know whether *parity* can be expressed with sru with a polynomial complexity.

Finally, we note that dcr is a well-known construct. It appears under the name *pump*, in a language specifically designed for a parallel database machine, FAD [9]. Following FAD, but under the name *hom*, this construct was included in Machiavelli [82] where it fit nicely into the language's type system. Called (a form of) *transducer*, it is part of SVP [85], precisely in order to support divide and conquer parallelism. Some limitations of its theoretical expressive power were examined (under the name *hom*) by Immerman, Patnaik, and Stemple ([66] Theorem 7.8). They also note that dcr is in NC. Part of our interest in dcr lies in the fact that it fits into a natural hierarchy of query languages obtained by extending $\mathcal{NRC}$ with certain forms of structural recursion on collection types [19, 18, 21, 22] (see Section 3.4). Theoretical studies of expressiveness, such as [107, 75, 95, 76, 99] and the results presented here help us with the choice and mix of primitives.

2.4.3 map-Recursion

The map-recursion discussed here is obtained by imposing a strict control on the way a collection is divided, and by generalizing from a 2-way division to a division into an arbitrary number of subcollections.

Formally, we say that a function $\varphi : t \rightarrow t'$ is defined by **map-recursion** with parameters d and c, in notation $\varphi = \mathsf{maprec}(d, c)$, iff φ is defined as in Figure 2.13.

No conditions are imposed on d and c..

$$\frac{d : t \to \lceil t \rfloor \quad c : t \times \lceil t' \rfloor \to t'}{\mathsf{maprec}(d, c) : t \to t'}$$

$$\varphi(s) \stackrel{\text{def}}{=} c(s, \mathsf{map}(\varphi)(d(s)))$$

Figure 2.13: Map recursion

$$
\begin{array}{rcll}
\mathsf{sru}(e, f, u) & = & \mathsf{maprec}(d, c) & \text{where:} \\
d(s) & \stackrel{\text{def}}{=} & \begin{cases} [d_1(s), d_2(s)] & \text{when } \mathsf{length}(s) \geq 2 \\ [] & \text{otherwise} \end{cases} & \\
c(s, x) & \stackrel{\text{def}}{=} & \begin{cases} e & \text{when both } s \text{ and } x \text{ are empty} \\ f(y) & \text{when } s = [y] \\ u\langle x_1, x_2 \rangle & \text{when } \mathsf{length}(s) \geq 2 \text{ and } x = [x_1, x_2] \\ \text{undefined} & \text{otherwise} \end{cases} &
\end{array}
$$

Figure 2.14: Expressing sru with map-recursion

map-Recursion is the salient ingredient of a language for sequences, called $\mathcal{MAP}$, which we use as implementation language for $\mathcal{NCC}$. $\mathcal{MAP}$ is obtained by restricting $\mathcal{NCC}$ to sequences and adding to it map-recursion. We will show in Section 4.6 how all operations on sets and bags in $\mathcal{NCC}$ can be expressed in $\mathcal{MAP}$. Moreover, various forms of recursions on sets and bags can be expressed in $\mathcal{MAP}$ too. For illustration, we show here how the structural recursion on the union presentation on sequences can be translated into map-recursion. Let $\varphi : [t] \to t'$ be defined as $\varphi = \mathsf{sru}(e, f, u)$, and assume we decide to implement it such that, at each division step, the sequence is divided into two equal halves. Let $d_1, d_2 : [t] \to [t]$ be the functions extracting the first half, and the second half of a sequence respectively: $d_1([x_0, \ldots, x_{n-1}]) \stackrel{\text{def}}{=} [x_0, \ldots, x_{\lfloor \frac{n}{2} \rfloor}]$ and $d_2([x_0, \ldots, x_{n-1}]) \stackrel{\text{def}}{=} [x_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, x_{n-1}]$. Then φ can be defined with map-recursion as $\varphi = \mathsf{maprec}(d, c)$, with d and c defined as in Figure 2.14.

But map-recursion is by far more powerful than structural recursion on the union presentation. It may choose to "divide" the input collection into an arbitrary number of collections, not necessarily 2; these sequences may not necessarily be a partition of the original one, in fact they may not necessarily be smaller. We can define non-terminating functions using map-recursion: by contrast all definitions with structural recursion on the union presentation terminate on all inputs.

2.5 Parallel Evaluation

Nothing in the definition of the language $\mathcal{NCC}$ or the various recursion schemas points to any connection with parallelism. In order to make that connection, we have to formally introduce the *parallel time complexity* T. Intuitively, it should be understood as an ideal parallel running time, assuming an idealized parallel computation model with arbitrarily many processors and instant communication.

Let $f : t \to t'$ be some closed function in $\mathcal{NCC}$, or $\mathcal{NCC}$ extended with one of the recursion schemas discussed above. Also let $x \in t$ be an object of type t. We associate to f and x a number $T(f, x)$ called the **parallel time complexity** of f on x. Similarly, for a closed expression $e : t$ we will associate a number $T(e)$, called the parallel time complexity of e. Intuitively T will be the time taken to compute $f(x)$ (or e) by taking advantage of all the "parallelism" in the language. We emphasized however that T is part of the definition of the language $\mathcal{NCC}$. In fact it is the definition of T that states what is "parallel" in $\mathcal{NCC}$ and what is not.

We shall give a full definition of T in Chapter 4 for $\mathcal{NCC}$ restricted to sequences and extended with map-recursion. Here we highlight the salient parts of the definition, which make the connection of $\mathcal{NCC}$ and the recursion schemas with parallelism.

ext and map Let $x = \lceil x_1, \ldots, x_n \rfloor$. Then:

$$\begin{aligned} T(\mathsf{map}(f), \lceil x_0, \ldots, x_0 \rfloor) &\stackrel{\text{def}}{=} 1 + \max_{i=0,n-1} T(f, x_i) \\ T(\mathsf{ext}(f), \lceil x_0, \ldots, x_0 \rfloor) &\stackrel{\text{def}}{=} 1 + \max_{i=0,n-1} T(f, x_i) \end{aligned}$$

Here the intuition is that the n computations $f(x_0), \ldots, f(x_{n-1})$ are done in parallel. It takes a constant amount of additional time (say 1) to combine the n results into a collection.

First-order operations on collections Operations like flatten, @, or the specialized operations on bags and sequences of Section 2.3 all take $T = 1$. E.g.:

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

Other operations Except for the operations mentioned above, all others are "sequential". E.g. to compute the n-tuple $\langle e_1, \ldots, e_n \rangle$, we compute the n expressions $e_1, \ldots, e_n$ sequentially. Thus:

$$T(\langle e_1, \ldots, e_n \rangle) \stackrel{\text{def}}{=} 1 + \sum_{i=1,n} T(e_i)$$

map-**recursion** Essentially, $T(\mathsf{maprec}(d, c), x)$ is defined as follows. Suppose

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

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 we define:

$$T(\mathsf{maprec}(d, c)) \stackrel{\text{def}}{=} T_d + \max_{i=1,n}(T_i) + T_c \tag{2.1}$$

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

1. First compute $d(x) = \lceil x_0, \ldots, x_{n-1} \rfloor$. 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 2.1.

Structural recursion and divide-and-conquer recursion We illustrate on divide and conquer recursion, $\mathsf{dcr}(e, f, u)$. Structural recursion is handled similarly. Namely:

$$
\begin{aligned}
T(\mathsf{dcr}(e, f, u), \{\}) &\stackrel{\text{def}}{=} T(e) \\
T(\mathsf{dcr}(e, f, u), \{y\}) &\stackrel{\text{def}}{=} T(f, y) \\
T(\mathsf{dcr}(e, f, u), s) &\stackrel{\text{def}}{=} \min(\{\max(T(\mathsf{dcr}(e, f, u), s_1), T(\mathsf{dcr}(e, f, u), s_2)) + T_u \mid \\
&\qquad s_1 \cup s_2 = s, s_1 \cap s_2 = \emptyset\})
\end{aligned}
$$

Here T_u stands for $T(u, \langle \mathsf{dcr}(e, f, u)(s_1), \mathsf{dcr}(e, f, u)(s_2)\rangle)$.

Intuitively, a set s can be decomposed into two disjoint sets s_1, s_2 in may ways. An evaluator should choose the best way of doing the splitting.

2.6 Implementation Challenges

Defining the parallel time complexity T is easy. The hard part is to design implementation techniques targeting existing multiprocessors, which indeed preserve the high-level parallel time complexity. We will discuss this challenge here.

We adopt the taxonomy in [94] (see also [46]), and classify parallel database architectures in **shared-memory**, **shared-disk**, and **shared-nothing** architectures. In the first architecture the processors share both the memory and the discs; in the second they have local memories, but share the discs. Both of these two architectures rely on a powerful interconnection network, and are difficult to scale up. By contrast, in the shared-nothing architecture the processors have both local memory and local disc(s), and are connected by a communication network. This architecture is easier to scale up, and can be build with low-cost commodity components.

We will target our implementation only on a shared-nothing architecture. Suppose we want to compile some $\mathcal{NCC}$ program f with parallel time complexity $T = T(f, x)$ on a shared-nothing architecture with P processors. We cannot expect it to run in time $O(T)$, simply because the number of parallel threads in the computation of $f(x)$ may easily exceed P.

The ideal running time T_P for a good implementation on an architecture with P nodes is given instead by a formula taking into account both T and the *work complexity* of $f(x)$, in notation $W(f, x)$. As in the case of $T(f, x)$, the **work complexity** is a number $W(f, x)$ defined for every closed function $f : t \to t'$ and every input $x : t$, and is the same as the sequential time needed to compute $f(x)$. In short, W has a *sum* whenever T has a *max*: e.g. the work complexity of $\mathsf{maprec}(d, c)$ is $W(\mathsf{maprec}(d, c)) \stackrel{\text{def}}{=} W_d + \sum_{i=1,n}(W_i) + W_c$, where W_d, $W_i, i = 1, n$, and W_c are as in Equation 2.1.

Then the ideal running time of some closed function f, with complexities T, W, compiled on a multiprocessor with P nodes is T_P:

$$T_P \stackrel{\text{def}}{=} O(T + \frac{W}{P}) \tag{2.2}$$

Indeed, the implementation cannot run faster than T, even if enough processors were available. But since there are only P processors to execute the total work of W, at least $\frac{W}{P}$ steps are also necessary. This explains why $O(\max(T, \frac{W}{P})) = O(T + \frac{W}{P})$ steps are needed.

But there is a huge gap between a declarative, high-level language like $\mathcal{NCC}$ and its extensions, and a bare-bone network of P processors: achieving the running time T_P given by Equation 2.2 is quite a challenge. We break this problem down into a number of steps, and strive to achieve optimality at each step.

Implementation of sets and bags. We implement sets and bags as sequences, and use high-level parallel algorithms developed in the past to implement the high-level, declarative operations on sets and bags over sequences. For that we design an implementation language, called $\mathcal{MAP}$, which is still high-level enough to allow us to express naturally a large family of parallel algorithms. Still, it is cut down just enough from a general-purpose language, to admit an efficient implementation on a shared-nothing architectures.

We choose to implement sets and bags as sequences rather than, say, B-trees, because we intend to partition them among the physical processors of a shared-nothing architecture using the techniques for sequences described in Chapter 6.

Compiling $\mathcal{MAP}$. We design new compilation techniques for $\mathcal{MAP}$ into a simple vector-parallel machine model, which we call BVRAM. These techniques preserve the parallel time complexity T, and add an arbitrarily small overhead to the work complexity W.

Implementing BVRAM. Finally, we implement the vector-parallel model, BVRAM, on a shared-nothing architecture. Because of the simplicity of the original language $\mathcal{MAP}$, most of the communication patterns needed in the implementation are of a particular form: *monotone communications.* We design provably efficient implementation techniques for monotone communications on a shared-nothing architecture.

Chapter 3

A Query Language for NC

3.1 Introduction

NC is the complexity class of functions that are computable in polylogarithmic time with polynomially many processors on a *parallel random access machine* (PRAM). It is regarded as the class of functions which can be efficiently implemented in parallel, and has approximatively the same relevance for parallel computation as the complexity class PTIME for sequential computation. We will show here that a query language centered around *divide and conquer recursion* (dcr) of Section 2.4, can express exactly those queries which are in NC.

Recall that $\varphi = \mathsf{dcr}(e, f, u)$ is the unique function such that:

$$\begin{aligned} \varphi(\emptyset) &\stackrel{\text{def}}{=} e \\ \varphi(\{y\}) &\stackrel{\text{def}}{=} f(y) \\ \varphi(s_1 \cup s_2) &\stackrel{\text{def}}{=} u\langle\varphi(s_1), \varphi(s_2)\rangle \text{ when } s_1 \cap s_2 = \emptyset \end{aligned}$$

For parity, we take $e \stackrel{\text{def}}{=} \mathsf{false}$, $f(y) \stackrel{\text{def}}{=} \mathsf{true}$ and $u\langle v_1, v_2\rangle \stackrel{\text{def}}{=} v_1 \;\mathsf{xor}\; v_2$. To compute the transitive closure of some binary relation r, take $e \stackrel{\text{def}}{=} \emptyset$, $f(y) \stackrel{\text{def}}{=} r$ and $u\langle r_1, r_2\rangle \stackrel{\text{def}}{=} r_1 \cup r_2 \cup r_1 \circ r_2$. Then, the transitive closure of r is obtained by applying φ to the set of nodes of the relation r, namely $tc(r) = \varphi(\Pi_1(r) \cup \Pi_2(r))$, where Π_1, Π_2 are the relational projections, see Example 2.2.1. In general, $\mathsf{dcr}(e, f, u)$ is well-defined when there is some set containing e and the range of f, on which u is associative, commutative and has the identity e. For parity, this is the set $\mathbb{B}$ of booleans, while for transitive closure, it is the set $\{r \cup r^2 \cup \ldots \cup r^n \mid n \geq 0\}$.

This chapter is organized as follows. Section 3.2 reviews some basic query language constructs. Section 3.4 introduces four different forms of recursions over sets (one of which is dcr) and establishes the relationships between them, then defines bounded versions of recursion on sets, which are necessary for controlling complexity when we work with nested sets. It also briefly discusses the undecidability of the well-definedness of the dcr construct. We briefly review the parallel complexity classes AC^k and NC in Section 3.5, and extend them to classes of queries in Section 3.6. The main results are stated and discussed in Section 3.7. Finally, we prove the main results in Section 3.8.

Based on results reported in [97], we show here that dcr together with the restriction of $\mathcal{NRC}$ to flat relations expresses exactly the queries over *ordered* databases that are NC-computable. Using a result in [18] it follows that the Relational Algebra extended with dcr expresses exactly the NC-computable queries on ordered databases. We also show that a bounded version of dcr together with the Nested Relational Algebra expresses exactly the queries over *ordered* databases of complex objects that are NC-computable. In fact, we prove the more refined versions that relate k nested uses of (bounded) dcr exactly to the subclass AC^k of NC where $k \geq 1$ (the definitions of these complexity classes are reviewed in Section 3.5). Some explanations are in order:

- Computable queries are in the sense of Chandra and Harel [28], with a natural extension to complex objects (Section 3.6).

- Any language that can express the same class of queries as first-order logic would do just as well as the Relational Algebra. Similarly for complex objects, where a corresponding class of tractable queries has emerged from several equivalent formalisms. Some of these formalisms are syntactically restricted higher-order logics, others are algebraic languages, often called Nested Relational Algebras, hence our statement above. In fact, we will use the family of query languages introduced in [21, 22] because it is semantically related to dcr (Section 3.2).

- dcr and (Nested) Relational Algebra have meaning over any (nested) relational database. But, as with all known characterizations of query complexity classes below NP, we know how to capture the entire NC only over ordered databases. Formally, we do this by extending the language with an order predicate.

- A bounded version of dcr is necessary over complex objects, otherwise queries of high complexity such as *powerset* will be expressible. The bounded version is obtained by intersecting the result with a bounding set at each recursion step (Section 3.4). This is similar to the bounded fixpoints studied in [95], an idea due to Peter Buneman, and, as with fixpoints, over flat relations dcr can always be expressed through bounded dcr (Section 3.4).

Following Immerman and Vardi's influential result [61, 104, 62] that first-order logic with least fixed point captures exactly the $PTIME$-computable queries on flat relations over ordered databases, several characterizations of low complexity classes in terms of logics or algebras used in databases have been discovered with the hope that logical methods may give insights into the difficult problem of complexity class separation. We mention first a few of these characterizations which have had a direct influence on the work here.

For parallel complexity classes, Immerman [65] shows that the class of finite and ordered relational structures recognizable in parallel time $t(n)$ (n is the size of the structure) on a certain CRCW (concurrent read - concurrent write) PRAM coincides with the class of structures definable by a first-order induction [79] of depth up to $t(n)$. Denninghoff and Vianu [43] characterize NC in terms of a resource-restricted message-passing model with parallel semantics which computes object-oriented queries. For complex object databases, Grumbach and Vianu [54, 53] give a syntactic restriction of the ramified higher-order logic CALC which, together with inflationary fixpoints and in the presence of order, captures exactly the $PTIME$-computable complex-object queries. In the presence of order the same class of queries is captured by the Nested Relational Algebra augmented with an inflationary bounded fixpoint operator [95].

To the best of our knowledge, no characterization of parallel complexity classes of queries over complex objects has been given before. What is more likely to set our results apart, however, is the *intrinsic* nature of the language we are proposing: the semantics of dcr puts it naturally in *NC*; there is no need to impose logarithmic bounds on the number of iterations or recursion depth. Moreover, it can be shown that a different kind of recursion on sets, namely structural recursion on the insert presentation of sets ([19]; notation sri; definitions reviewed in Section 3.4), together with the Relational Algebra expresses exactly the *PTIME*-computable queries on ordered databases[1]. This follows from results in [66]; we state the corresponding result for complex objects in Proposition 3.7.7. Hence, at least over ordered databases, the difference between *NC* and *PTIME* boils down to two different ways of recurring on sets, divide and conquer vs. element by element.

Gurevich [56] and Compton and Laflamme [35] characterize the *DLOGSPACE*- and respectively the NC^1-computable global functions on ordered finite relational structures as algebras with certain primitive recursion schemas. Compton and Laflamme capture NC^1 also with first-order logic augmented with BIT[2], and with an operator for *defining relations by primitive recursion*. The recursion forms used in these two papers are very different from dcr because they depend on some linear ordering of the underlying structures for their actual definition, and while dcr is a form of recursion on finite sets, the recursion forms in [56, 35] are on *notations* for elements of (linearly ordered) finite sets. In Section 3.4 we consider a related form of recursion on sets, *set-divide*, whose definition relies on an order relation of the base type. This is a recursion *on sets*, hence it is different from the recursion considered in [35], which is a recursion *on the elements* of the domain; *set-divide* is related in spirit to *set-reduce* [66]. Clote [32] gives related characterizations of most parallel complexity classes, of *arithmetical* functions however.

Since dcr can be defined for all all structures, not just ordered ones, our characterization of *NC* is instead closer in style to the abovementioned fixpoint characterization of *PTIME* by Immerman and Vardi, or to Immerman's characterizations of *DLOGSPACE* and *NLOGSPACE* by first-order logic extended with deterministic and nondeterministic

[1] Of course, so does least fixpoint recursion, for example, but it is not a recursion *on* sets.

[2] A relation giving the binary representation of integers.

transitive closure [64, 63]. We must warn the reader however about one sense in which our language is not as neat as these extensions of first-order logic. For dcr to be well-defined, the operations involved in it must satisfy certain algebraic identities (associativity, commutativity, identity) and this turns out to be an undecidable condition, in fact even Π^0_1 complete; see Section 3.4. Of course, only a certain family of instances of dcr is actually needed in the simulations, and for these, the algebraic conditions always hold (Proposition 3.8.3). Hence, it is of theoretical interest that there is a decidable sublanguage of dcr plus Relational Algebra which captures exactly NC in the presence of order. In practice, we have found it useful to provide special syntax for some instances of dcr in which the algebraic conditions are automatically satisfied, but we found it counterproductive to limit dcr to these instances, as other uses kept appearing.

3.2 The Nested Relational Calculus

Our language for the NC-computable queries over flat relations is the restriction of the *Nested Relational Calculus* $\mathcal{NRC}$, see Subsection 2.3.1, to flat types (denoted by $\mathcal{NRC}^1$), extended with *divide and conquer recursion* dcr: $\mathcal{NRC}^1(\mathsf{dcr})$. It follows from [18] that $\mathcal{NRC}^1$ has the same expressive power as the Relational Algebra. Our language for the NC-computable queries over complex objects is $\mathcal{NRC}$ extended with a variant of dcr, called the *bounded* dcr.

Recall that the types of $\mathcal{NRC}$ are given by the grammar:

$$t \stackrel{\text{def}}{=} \mathbb{D} \mid \mathbb{B} \mid t \times \ldots \times t \mid \{t\}$$

Without loss of generality we shall restrict the base types to $\mathbb{D}$ and $\mathbb{B}$: all results remain true for an arbitrary collection of base types $\mathbb{D}_1, \mathbb{D}_2, \ldots$. Recall that *unit* denotes the empty product, i.e. obtained by taking $n = 0$ in $t_1 \times \ldots \times t_n$. We will call the values populating these types **complex objects**.

A **scalar type** is either one of the types $\mathbb{D}, \mathbb{B}$, or a product of scalar types. A **flat type** is either a type of the form $\{t\}$, with t a scalar type, or a product of flat types. E.g. $\mathbb{D} \times \mathbb{B} \times \mathbb{D}$

is a scalar type, while $\{\mathbb{D} \times \mathbb{D}\}$ and $\{unit\} \times \{\mathbb{D} \times \mathbb{B}\}$ are flat types. We call values of a flat type of the form $\{t\}$ **flat relations**. A **nested relation** is a value of some type $\{t\}$ which is not a flat relation: e.g. values populating the types $\{\mathbb{D} \times \{\mathbb{B} \times \mathbb{D}\}\}$ and $\{\{\mathbb{D}\}\}$ are nested relations. Although strictly speaking any value is a complex object, we will use sometimes the term complex object as a synonim to nested relations.

As for $\mathcal{NCC}$, a possible set Σ of **external functions** $p : d_p \to c_p$ can be added to $\mathcal{NRC}$: in this case, we denote the language by $\mathcal{NRC}(\Sigma)$.

In $\mathcal{NRC}$, we will call **query** any closed function expression $f : t_1 \to t_2$. In our presentation of $\mathcal{NRC}$, queries do not admit an inductive definition, because subexpressions of some query f may not be queries themselves: they may be functions $f : t_1 \to t_2$ with free variables, or they may be expressions denoting complex objects $e : t$. In order to prove properties about queries by induction on all the expressions in $\mathcal{NRC}$ (such as in the proof of Proposition 3.8.10), we associate to each expression a query, as follows. First, recall our abbreviation:

$$\lambda\langle x_1{:}t_1, x_2{:}t_2\rangle.e \stackrel{\text{def}}{=} \lambda z{:}t_1 \times t_2.e[\pi_1(z)/x_1, \pi_2(z)/x_2]$$

where $e[e'/x]$ is the result of substituting e' for x in e (with appropriate changes of bound variable names so as to avoid unintended capture of free variables). Then, to any complex object expression $e : t$, and any set of variables $x_1{:}t_1, \ldots, x_k{:}t_k$ containing at least all free variables of e, we associate the query

$$\lambda\langle x_1{:}t_1, \ldots x_k{:}t_k\rangle.e : t_1 \times \ldots \times t_k \to t$$

Similarly, to any function expression $f : t \to t'$ we associate the query

$$\lambda\langle x{:}t, x_1{:}t_1, \ldots, x_k{:}t_k\rangle.f(x) : t \times t_1 \times \ldots \times t_k \to t'$$

In the rest of this Chapter we shall omit mentioning explicitly the set of variables, and talk about "the query f associated to some expression e".

As shown in Section 2.3, $\mathcal{NRC}$ is powerful enough to express the following queries: set difference, set intersection, cartesian product, relational projections, equalities at all types, selections over predicates definable in the language, nest and unnest [21, 22].

To characterize the NC-computable queries over ordered flat relations (Theorem 3.7.2), we restrict ourselves to a fragment of $\mathcal{NRC}$ that has the same expressive power as Relational Algebra. First we define the **set height** of a type t as follows:

$$\begin{aligned} height(\mathbb{D}) &\stackrel{\text{def}}{=} height(\mathbb{B}) \stackrel{\text{def}}{=} 0 \\ height(t_1 \times \ldots \times t_n) &\stackrel{\text{def}}{=} \max(height(t_1), \ldots, height(t_n)) \\ height(\{t\}) &\stackrel{\text{def}}{=} 1 + height(t) \end{aligned}$$

Thus, $height(t) = 0$ iff t is a scalar type, and $height(t) \leq 1$ iff t is a product of scalar types and flat types. The fragment that interests us is $\mathcal{NRC}^1$, defined as the restriction of $\mathcal{NRC}$ to types of set height ≤ 1, i.e. the only types allowed in $\mathcal{NRC}^1$ as inputs, outputs and intermediate types are products of base and flat types. Indeed:

Theorem 3.2.1 ([18]) *The set of $\mathcal{NRC}^1$ queries $f : t_1 \to t_2$ with t_1, t_2 flat types coincides with the set of queries expressible in the Relational Algebra and, hence, in First Order Logic.*

3.3 Genericity

Chandra and Harel [28] make the observation that not all functions $f : t \to t'$ qualify as database queries: to qualify as such, f needs to treat its inputs of type $\mathbb{D}$ in a domain-independent way, i.e. it needs to be *generic*. In this Section we define the notion of genericity, and prove that every query expressed in $\mathcal{NRC}$ is generic. The definition is valid only for generic queries without external functions, and needs to be adjusted in the case of query languages with external functions; see [96] for a discussion.

Consider some subset $\mathbb{D}' \subseteq \mathbb{D}$. Then for every type t, let t' be the type obtained by replacing each occurrence of $\mathbb{D}$ with $\mathbb{D}'$. E.g. for the type $t = \{\mathbb{D} \times \{\mathbb{B} \times \mathbb{D}\}\}$, t' will be $\{\mathbb{D}' \times \{\mathbb{B} \times \mathbb{D}'\}\}$. Obviously $t' \subseteq t$. Let $\varphi : \mathbb{D}' \to \mathbb{D}$ be an injective function. For every type t, we define the extension of φ to t to be $\varphi_t : t' \to t$ as follows:

$$
\begin{aligned}
\varphi_{\mathbb{D}}(x) &\stackrel{\text{def}}{=} \varphi(x) \\
\varphi_{\mathbb{B}}(x) &\stackrel{\text{def}}{=} x \\
\varphi_{unit}(x) &\stackrel{\text{def}}{=} x \\
\varphi_{t_1\times\ldots\times t_n}\langle x_1,\ldots,x_n\rangle &\stackrel{\text{def}}{=} \langle\varphi_{t_1}(x_1),\ldots,\varphi_{t_n}(x_n)\rangle \\
\varphi_{\{t\}}\{x_1,\ldots,x_n\} &\stackrel{\text{def}}{=} \{\varphi_t(x_1),\ldots,\varphi_t(x_n)\}
\end{aligned}
$$

Definition 3.3.1 *We call a function $f : t_1 \to t_2$* **generic** *if, for every injective function $\varphi : \mathbb{D}' \to \mathbb{D}$, the following holds: $\forall x \in t_1'$, $\varphi_{t_2}(f(x)) = f(\varphi_{t_1}(x))$.*

Later we shall consider *ordered databases*, i.e. in which a total order relation is defined on the base type $(\mathbb{D}, \leq)$. In that case we call f generic if $\varphi_{t_2} \circ f = f \circ \varphi_{t_1}$ for all *order-preserving* injective functions $\varphi : (\mathbb{D}', \leq) \to (\mathbb{D}, \leq)$.

Definition 3.3.1 naturally extends that of Chandra and Harel [28], where φ only ranges over bijections $\varphi : \mathbb{D} \to \mathbb{D}$. Their definition is not enough for our purposes here. Indeed, in Section 3.6 we deal with ordered databases for which $\mathbb{D}$ is the set of natural numbers. Then, there is only one order-preserving bijection $\varphi : (\mathbb{D}, \leq) \to (\mathbb{D}, \leq)$, namely the identity, and in this case, under the definition in [28] all queries are generic. By slightly changing the definition to allow the domain of φ to be any subset $\mathbb{D}'$ of $\mathbb{D}$, we force f to commute with order-preserving renamings of elements of $\mathbb{D}$.

Proposition 3.3.2 *All queries in $\mathcal{NRC}$ are generic.*

Proof. (Sketch) Since the queries in $\mathcal{NRC}$ do not have an inductive structure (see the comments in Section 3.2), we prove by induction the following statement about expressions in $\mathcal{NRC}$: for any expression $e : t$ or $f : t \to t'$ in $\mathcal{NRC}$, and any set of variables $x_1, \ldots, x_k$ containing all free variables of e, or f respectively, the query associated to e, or to f respectively (see Section 3.2), is generic. We consider some of the cases for the structure of e:

Variable x. Then x must be one of the variables $x_1, \ldots, x_k$, say $x = x_i$, and the associated query is $\lambda\langle x_1, \ldots x_k\rangle.x_i$, which is π_i. This is obviously generic.

Pair $\langle e_1, \ldots, e_n\rangle : s_1 \times \ldots \times s_n$. Here the associated query is

$$f \stackrel{\text{def}}{=} \lambda\langle x_1, \ldots x_k\rangle.\langle e_1, \ldots, e_n\rangle$$

For some injection $\varphi : \mathbb{D}' \to \mathbb{D}$ we have:

$$\begin{aligned} \varphi_{s_1 \times \ldots \times s_n} \circ f &= \lambda\langle x_1, \ldots, x_k\rangle.\varphi_{s_1 \times \ldots \times s_n}\langle e_1, \ldots, e_n\rangle \\ &= \lambda\langle x_1, \ldots, x_n\rangle.\langle \varphi_{s_1}(e_1), \ldots, \varphi_{s_n}(e_n)\rangle \end{aligned}$$

By induction hypothesis we have:

$$\begin{aligned} &\lambda\langle x_1, \ldots x_k\rangle.\varphi_{s_i}(e_i) = \\ &\qquad (\lambda\langle x_1 \ldots x_k\rangle.e_i) \circ \varphi_{t_1 \times \ldots \times t_k} \end{aligned}$$

for every $i = 1, n$. Hence we conclude:

$$\begin{aligned} &\lambda\langle x_1, \ldots x_k\rangle.\langle \varphi_{s_1}(e_1), \ldots, \varphi_{s_n}(e_n)\rangle = \\ &\qquad (\lambda\langle x_1, \ldots, x_k\rangle.\langle e_1, \ldots, e_n\rangle) \circ \varphi_{t_1 \times \ldots \times t_k} \end{aligned}$$

The other cases are handled similarly.

□

Finally we comment on the operation get which we included in $\mathcal{NRC}$. It is obviously a generic function, in the sense of Definition 3.3.1, and it is computable in NC, in the sense of Definition 3.6.1. Hence it must be expressible in our language for NC. It turns out however that it is not expressible in $\mathcal{NRC}$ extended with the bounded divide and conquer recursion, unless we explicitly include it in $\mathcal{NRC}$, see Proposition 3.4.9. More annoyingly, our query language for flat relations, $\mathcal{NRC}^1(\mathsf{dcr})$ *can* express get, using dcr, see Example 3.4.3. But that comes at an additional level of iteration nesting, and taking it out from $\mathcal{NRC}$ would destroy the correspondence between $\mathcal{NRC}^1(\mathsf{dcr}^{(k)})$ and the classes AC^k.

However adding get in $\mathcal{NRC}$ does not affect the expressive power of the language over flat types or, more generally, over types which are products of set types, see Proposition 3.4.10. Hence, although the result of [18] concerns $\mathcal{NRC}^1$ without get, Theorem 3.2.1 remains true in our setting, in which $\mathcal{NRC}^1$ includes get.

3.4 Recursions on Sets

In this Section we discuss the properties of the construct that is at the core of our query languages for *NC*, divide and conquer recursion on sets (dcr), in the context of related operations.

We also discuss four forms of recursion on sets (two of which are *structural* recursions) and the relationship between them. One of these forms of recursion—dcr—will be at the core of our "query languages for *NC*", while another form, *element-step-recursion*, esr, corresponds to *PTIME*. When dealing with complex objects instead of flat relations we need to add an additional twist to these structural recursions to make them still capture *NC* and *PTIME* respectively: we call the resulting forms of recursion *bounded* recursions. All forms of recursions can be defined in the absence of order, in contrast to other forms of recursions considered in the literature which can only be defined in the presence of order.

We start by formally defining these forms of recursion, establish the relationships between them, and between the bounded and unbounded versions, prove that, in the presence of order, they have the same expressive power as order-dependent forms of recursions previously considered in the literature, and finally discuss the complexity of checking the "side-conditions" associated to recursions on sets.

3.4.1 Forms of Recursion on Sets

There seem to be two basic ways of describing the structure of finite sets. In one way, they are generated by finitely many (maybe zero) binary unions of singleton sets. We call this the *union presentation.* In another way, they are generated by finitely many insertions of one element, starting with the empty set. We call this the *insert presentation.* Recognizing the relevant algebraic identities satisfied by union (associativity, commutativity, idempotence, has $\emptyset$ as an identity) and by element insertion (left-commutativity and left-idempotence) gives us two different algebraic structures on finite sets. Both these algebras are characterized by universality properties, which amount to definitions of functions by *structural recursion* [19, 18]. As discussed in Section 2.4, we have a structural recursion construct on the union presentation, sru:

$$\frac{e : t' \quad f : t \to t' \quad u : t' \times t' \to t'}{\mathsf{sru}(e, f, u) : \{t\} \to t'}$$

$$\begin{aligned}
\mathsf{sru}(e, f, u)(\emptyset) &\stackrel{\text{def}}{=} e \\
\mathsf{sru}(e, f, u)(\{y\}) &\stackrel{\text{def}}{=} f(y) \\
\mathsf{sru}(e, f, u)(s_1 \cup s_2) &\stackrel{\text{def}}{=} u\langle \mathsf{sru}(e, f, u)(s_1), \mathsf{sru}(e, f, u)(s_2)\rangle
\end{aligned}$$

$\mathsf{sru}(e, f, u)$ is well-defined when there is some subset of t containing e and the range of f, on which u is associative, commutative, idempotent, and has the identity e. We also have a structural recursion construct on the insert presentation, sri:

$$\frac{e : t' \quad i : t \times t' \to t'}{\mathsf{sri}(e, i) : \{t\} \to t'}$$

$$\begin{aligned}
\mathsf{sri}(e, i)(\emptyset) &\stackrel{\text{def}}{=} e \\
\mathsf{sri}(e, i)(ins(y, s)) &\stackrel{\text{def}}{=} i\langle y, \mathsf{sri}(e, i)(s)\rangle
\end{aligned}$$

Here $ins(y, s)$ is the *insert* operation defined to be $\{y\} \cup s$. Note that $\mathsf{sri}(e, i)$ is well-defined when there is some subset of t containing e on which i is *left-commutative*,

$$i\langle x, i\langle y, s\rangle\rangle = i\langle y, i\langle x, s\rangle\rangle$$

and *left-idempotent*

$$i\langle x, i\langle x, s\rangle\rangle = i\langle x, s\rangle$$

Our central operation, divide and conquer recursion, introduced already in Section 2.4, is a "no duplicates" variation of sru:

$$\frac{e : t' \quad f : t \to t' \quad u : t' \times t' \to t'}{\mathsf{dcr}(e, f, u) : \{t\} \to t'}$$

$$\begin{aligned}
\mathsf{dcr}(e, f, u)(\emptyset) &\stackrel{\text{def}}{=} e \\
\mathsf{dcr}(e, f, u)(\{y\}) &\stackrel{\text{def}}{=} f(y) \\
\mathsf{dcr}(e, f, u)(s_1 \cup s_2) &\stackrel{\text{def}}{=} u\langle \mathsf{dcr}(e, f, u)(s_1), \mathsf{dcr}(e, f, u)(s_2)\rangle \\
& \quad \text{when } s_1 \cap s_2 = \emptyset
\end{aligned}$$

$\mathsf{dcr}(e, f, u)$ is defined only when u is associative, commutative and has e as identity, on some subset of t' containing $Im(f)$.

If $\mathsf{sru}(e, f, u)$ is well-defined then so is $\mathsf{dcr}(e, f, u)$ and they are equal. But dcr is potentially more expressive, since u need not be idempotent. In fact, it is an open problem whether $\mathcal{NRC}^1(\mathsf{sru})$, or even $\mathcal{NRC}^1(\mathsf{sru}, \leq)$ can express parity or transitive closure. However, over ordered databases sru together with transitive closure has the same expressive power as dcr, see Proposition 3.4.15 below.

One can also define a no-duplicates variant of sri, let's call it *element-step recursion*, esr. This is like sri, with the second clause modified as:

$$\mathsf{esr}(e, i)(ins(y, s)) \stackrel{\text{def}}{=} i\langle y, \mathsf{esr}(e, i)(s)\rangle \text{ when } y \notin s$$

where i is required to be left-commutative (but not necessarily left-idempotent). Obviously, esr can express sri [3]. The non-immediate relationships between the four forms of recursion on sets are contained in:

[3] sru and sri are easier to reason about than dcr or esr because they define functions that preserve the algebraic structure, i.e. *homomorphisms*, hence the "structural" in their names. A good way to think about $\mathsf{dcr}(e, f, u)$ is as the composition of the canonical coercion from sets to bags followed by the structural recursion on the sum presentation of bags [19], with parameters e, f, u. Similarly, esr can be expressed via structural recursion on the increment presentation of bags.

Proposition 3.4.1 ([19, 20]) *In the presence of the operations of $\mathcal{NRC}$:* sri *can express* sru *and similarly* esr *can express* dcr*; moreover,* sri *can express* esr*. All this can be done without increase in set height and with at most polynomial overhead.*

Schematically:

$$\mathsf{sru} \quad \subseteq \quad \mathsf{dcr} \quad \subseteq \quad \mathsf{esr} \quad = \quad \mathsf{sri}.$$

Precisely:

Corollary 3.4.2 $\mathcal{NRC}^1(\mathsf{sru}) \subseteq \mathcal{NRC}^1(\mathsf{dcr}) \subseteq \mathcal{NRC}^1(\mathsf{esr}) = \mathcal{NRC}^1(\mathsf{sri})$.

Example 3.4.3 *We show here that* get *can be defined with* sru*. Indeed:*

$$\mathsf{get} = \lambda\langle s, y\rangle.\pi_2(\varphi(s))$$

where, for each $y \in \mathbb{D}$, we define define $\varphi : \{\mathbb{D}\} \to \{\mathbb{D}\} \times \mathbb{D}$ to be $\varphi \stackrel{\text{def}}{=} \mathsf{sru}(e, f, u)$, with:

$$\begin{array}{rcl}
e & \stackrel{\text{def}}{=} & \langle \emptyset, y\rangle \\
f(x) & \stackrel{\text{def}}{=} & \langle \{x\}, x\rangle \\
u\langle\langle s_1, x_1\rangle, \langle s_2, x_2\rangle\rangle & \stackrel{\text{def}}{=} & \mathsf{if}\ s_1 = s_2\ \mathsf{and}\ |s_1| = 1\ \mathsf{then}\ \langle s_1 \cup s_2, x_1\rangle\ \mathsf{else}\ \langle s_1 \cup s_2, y\rangle
\end{array}$$

Note that the test $|s_1| = 1$ is expressible in $\mathcal{NRC}^1$. Then u is associative, commutative and idempotent on the subset $\{\langle\{x\}, x\rangle \mid x \in \mathbb{D}\} \cup \{\langle s, y\rangle \mid |s| \neq 1\}$ of $\{\mathbb{D}\} \times \mathbb{D}$, and this set also contains e and $Im(f)$. Hence φ is well defined.

As a consequence get *can be expressed with any of the four forms of recursion in the presence of the other $\mathcal{NRC}$ operations.*

We have thus reached the language $\mathcal{NRC}^1(\mathsf{dcr})$. Adding a linear order predicate to this language gives us the subject of one of our main theorems, the characterization of the NC-computable queries on ordered databases of complex objects (Theorem 3.7.2).

In the same theorem, we obtain a finer characterization, involving the AC-hierarchy, for which we need the notion of **the depth of recursion nesting** $depth(e)$, of some expression e. We define this to be to be the maximum depth of recursions occurring in e. More precisely, $depth(\mathsf{dcr}(e, f, u)) \stackrel{\text{def}}{=} \max(depth(e), depth(f), 1 + depth(u))$ (only u is actually iterated). We denote $\mathcal{NRC}^1(\mathsf{dcr}^{(k)})$ the restrictions of the language $\mathcal{NRC}^1(\mathsf{dcr})$ to recursion depth $\leq k$. We have thus obtained the hierarchy of languages $\mathcal{NRC}^1(\mathsf{dcr}^{(k)})$. Adding a linear order predicate to these languages gives us the subject of the finer characterization given in Theorem 3.7.2.

Noticed that some redundancy in expressibility will appear when we add dcr. Indeed, it turns out that $\mathsf{ext}(f)$ can be expressed with sru (and hence with dcr) as $\mathsf{sru}(\emptyset, \lambda x.\{x\}, \cup)$. It is important however to keep $\mathsf{ext}(f)$ as a separate construct in the language because the expression derived through dcr would be computed in $\log n$ parallel steps, when in fact a direct one-step parallel computation is possible: obtain in parallel and independently $f(x_1), \ldots, f(x_n)$, and then take their union to compute $\mathsf{ext}(f)(\{x_1, \ldots, x_n\})$.

3.4.2 Bounded Recursion

All four forms of recursion on sets are too powerful on complex objects since each of them can express *powerset*. Actually one can show that $\mathcal{NRC}(\mathsf{dcr})$, and similarly $\mathcal{NRC}(\mathsf{sru})$, $\mathcal{NRC}(\mathsf{sri})$, $\mathcal{NRC}(\mathsf{esr})$, has the same expressive power as Abiteboul and Beeri's *algebra* [2]. As a consequence, all four forms of recursion can simulate each other, using $\mathsf{powerset}$ (at the cost of a high complexity), hence $\mathcal{NRC}(\mathsf{sru}) = \mathcal{NRC}(\mathsf{dcr}) = \mathcal{NRC}(\mathsf{sri}) = \mathcal{NRC}(\mathsf{esr}) = \mathcal{NRC}(\mathsf{powerset})$. To keep our languages for complex objects tractable, we will define *bounded* versions of these recursions, an analog to Peter Buneman's idea of *bounded fixpoints* [95].

A **PS-type** (product of sets type) is either a set type, or a product of PS-types. E.g. $\{\mathbb{D} \times \{\mathbb{D}\}\} \times \{unit\}$ and $\{\mathbb{D}\} \times (\{\mathbb{B} \times \mathbb{B}\} \times \{\mathbb{D}\})$ are PS-types, while $\mathbb{D} \times \{\mathbb{D}\}$ is not. Set-theoretic operations like $\cup, \cap, \subseteq$ extend to PS-types component-wise.

The **bounded** version of dcr is defined by:

$$\frac{e : t' \quad f : t \to t' \quad u : t' \times t' \to t' \quad b : t'}{\mathsf{bdcr}(e, f, u, b) : \{t\} \to t'}$$

with the restriction that t' is a PS-type, and with the semantics:

$$\mathsf{bdcr}(e, f, u, b) \stackrel{\text{def}}{=} \mathsf{dcr}(e \cap b, f \cap b, u \cap b)$$

Here $(u \cap b)\langle s_1, s_2\rangle \stackrel{\text{def}}{=} u\langle s_1, s_2\rangle \cap b$, etc. As for dcr, we define bounded versions for the other forms of recursions on sets, $\mathsf{bsru}, \mathsf{bsri}, \mathsf{besr}$. Proposition 3.4.1 easily extends to the bounded versions of recursion:

Corollary 3.4.4 $\mathcal{NRC}(\mathsf{bsru}) \subseteq \mathcal{NRC}(\mathsf{bdcr}) \subseteq \mathcal{NRC}(\mathsf{bsri}) = \mathcal{NRC}(\mathsf{besr})$.

Over flat relations the explicit bounding is unnecessary: we show next that $\mathcal{NRC}^1(\mathsf{bdcr})$ and $\mathcal{NRC}^1(\mathsf{dcr})$ have the same expressive power.

For this we need a the technique of *type lifting*. For any type t we define the **lifted type** $\bar{t}$ to be the type obtained by putting set parenthesis around all the base types it contains. Formally:

$$\begin{aligned} \overline{unit} &\stackrel{\text{def}}{=} \{unit\} \\ \overline{\mathbb{B}} &\stackrel{\text{def}}{=} \{\mathbb{B}\} \\ \overline{\mathbb{D}} &\stackrel{\text{def}}{=} \{\mathbb{D}\} \\ \overline{\{t\}} &\stackrel{\text{def}}{=} \{t\} \\ \overline{t_1 \times \ldots \times t_n} &\stackrel{\text{def}}{=} \bar{t}_1 \times \ldots \times \bar{t}_n \end{aligned}$$

Note that for every t, $\bar{t}$ is a PS-type, and when t is a PS-type, then $\bar{t} = t$. In addition we define $lift_t : t \to \bar{t}$, which puts set parenthesis around all the elements of base type that it contains: $lift_{\mathbb{D}} \stackrel{\text{def}}{=} \{x\}$, $lift_{t_1 \times \ldots \times t_n}\langle x_1, \ldots, x_n\rangle \stackrel{\text{def}}{=} \langle lift_{t_1}(x_1), \ldots, lift_{t_n}(x_n)\rangle$, and $lift_{\{t\}}(x) \stackrel{\text{def}}{=} x$. Note that when t is a PS-type, then $\bar{t} = t$ and $lift_t(x) = x$.

For a given function $f : t \to t'$ where t' is a PS-types, we define $\bar{f} : \bar{t} \to t'$ as follows:

$$\bar{f}(x) \stackrel{\text{def}}{=} \bigcup\{f(z) \mid \mathit{lift}_t(z) \subseteq x\}$$

Recall that we extend the operations $\cup$ and $\subseteq$ are extended to PS-types. E.g. consider the particular case when $t = \mathbb{D}$ and t' is a set type, $t' = \{t''\}$. Then for some $f : \mathbb{D} \to \{t''\}$, the function $\bar{f} : \{\mathbb{D}\} \to \{t''\}$ is simply $\bar{f} = \mathsf{ext}(f)$. The following proposition has a straightforward proof, which we omit:

Proposition 3.4.5 *Whenever f is expressible in $\mathcal{NRC}^1$, $\mathcal{NRC}^1(\mathsf{dcr})$, or $\mathcal{NRC}^1(\mathsf{bdcr})$, then $\bar{f}$ is expressible in the same language.*

Example 3.4.6 *Let $t = \mathbb{D} \times \{\mathbb{D} \times \mathbb{D}\} \times \mathbb{B}$, $t' = \{\mathbb{D}\} \times \{\mathbb{D}\}$, and $f : t \to t'$. Then $\bar{f}$ is defined by:*

$$\bar{f}\langle x_1, x_2, x_3\rangle \stackrel{\text{def}}{=} \bigcup\{f\langle z_1, x_2, z_3\rangle \mid z_1 \in x_1, z_3 \in x_3\}$$

and can be expressed as:

$$\bar{f}\langle x_1, x_2, x_3\rangle = \mathsf{ext}(\lambda z_1.\mathsf{ext}(\lambda z_3.f\langle z_1, x_2, z_3\rangle)(x_3))(x_1)$$

Fact 3.4.7 *Let $f : t \to t'$, where t' is a PS-type. Then $\bar{f}(\mathit{lift}_t(x)) = f(x)$.*

We can now show that over flat relations the bounded and unbounded versions of recursion have the same expressive power.

Proposition 3.4.8 $\mathcal{NRC}^1(\mathsf{bdcr}) = \mathcal{NRC}^1(\mathsf{dcr})$. *Moreover, the equivalence preserves the nesting depth of iterations, i.e. $\forall k \geq 0$, $\mathcal{NRC}^1(\mathsf{bdcr}^{(k)}) = \mathcal{NRC}^1(\mathsf{dcr}^{(k)})$. Similarly for* sri.

Note that, if we drop get from $\mathcal{NRC}$, then this proposition fails, since $\mathcal{NRC}^1(\mathsf{dcr})$ can express get (Example 3.4.3), while $\mathcal{NRC}^1(\mathsf{bdcr})$ cannot (Proposition 3.4.9). However, they still express the same set of queries $f : t \to t'$, where t' is a PS-type.

Proof. The inclusion $\mathcal{NRC}^1(\mathsf{bdcr}) \subseteq \mathcal{NRC}^1(\mathsf{dcr})$ is obvious. For the inclusion

$$\mathcal{NRC}^1(\mathsf{dcr}) \subseteq \mathcal{NRC}^1(\mathsf{bdcr})$$

consider some expression $\mathsf{dcr}(e, f, u) : \{t\} \to t'$ in $\mathcal{NRC}^1(\mathsf{dcr})$. First we lift t' to a PS-type $\bar{t}'$. Also, we define $e' : \bar{t}'$, $f' : t \to \bar{t}'$ and $u' : \bar{t}' \times \bar{t}' \to \bar{t}'$ as follows:

$$\begin{aligned} e' &\stackrel{\text{def}}{=} \mathit{lift}_{t'}(e) \\ f' &\stackrel{\text{def}}{=} \mathit{lift}_{t'} \circ f \\ u' &\stackrel{\text{def}}{=} \overline{\mathit{lift}_{t'} \circ u} \end{aligned}$$

If u is associative, commutative, and has identity e on some set $s \subseteq t'$, then u' will still be associative, commutative, and will have identity e', on the subset $\{\mathit{lift}(x) \mid x \in s\}$ of $\bar{t}'$. Moreover, we have $\mathsf{dcr}(e', f', u') = \mathit{lift}_{t'} \circ \mathsf{dcr}(e, f, u)$. To see that, consider for illustration a set with three elements, $s = \{x_1, x_2, x_3\}$. Then:

$$\begin{aligned} &\mathsf{dcr}(e', f', u')(\{x_1, x_2, x_3\}) = \\ &= u'\langle f'(x_1), u'\langle f'(x_2), f'(x_3)\rangle\rangle \\ &= u'\langle f'(x_1), \overline{\mathit{lift}_{t'} \circ u}\langle \mathit{lift}_{t'}(f(x_2)), \mathit{lift}_{t'}(f(x_3))\rangle\rangle \\ &= u'\langle f'(x_1), \overline{\mathit{lift}_{t'} \circ u}(\mathit{lift}_{t' \times t'}\langle f(x_2), f(x_3)\rangle)\rangle \quad \text{by the definition of } \mathit{lift}_{t' \times t'} \\ &= u'\langle f'(x_1), \mathit{lift}_{t'}(u\langle f(x_2), f(x_3)\rangle)\rangle \quad \text{by Fact 3.4.7} \\ &= \mathit{lift}_{t'}(u\langle f(x_1), u\langle f(x_2), f(x_3)\rangle\rangle) \quad \text{by repeating the steps above} \\ &= \mathit{lift}_{t'}(\mathsf{dcr}(e, f, u))(\{x_1, x_2, x_3\}) \end{aligned}$$

Now assume without loss of generality that e, f, u have equivalent expressions in $\mathcal{NRC}^1(\mathsf{bdcr})$, and, hence, so do e', f', u'. Let $x_1, \ldots, x_k$ be all free variables in the expressions e, f, and u, and let s be the input set to $\mathsf{dcr}(e', f', u')$. We compute a *bound* for $\mathsf{dcr}(e', f', u')$ as follows. First we define $b_{\mathbb{D}} \in \{\mathbb{D}\}$ to be the set of all values of type $\mathbb{D}$ mentioned in $x_1, \ldots, x_k$, and s; obviously $b_{\mathbb{D}}$ can be computed in $\mathcal{NRC}^1$. Next, we define $b_{\mathbb{B}} \in \{\mathbb{B}\}$ to be $b_{\mathbb{B}} \stackrel{\text{def}}{=} \{\mathsf{false}, \mathsf{true}\}$, and $b_{unit} \stackrel{\text{def}}{=} \{\langle\rangle\}$. Finally we compute the bound b of type $\bar{t}'$ by pairing proper cartesian products of $b_{\mathbb{D}}, b_{\mathbb{B}}$, and b_{unit}. E.g. when $\bar{t}' = \{\mathbb{D}\} \times (\{\mathbb{D} \times \mathbb{D}\} \times \{\mathbb{B}\})$, we take $b \stackrel{\text{def}}{=} \langle b_{\mathbb{D}}, \langle b_{\mathbb{D}} \times b_{\mathbb{D}}, b_{\mathbb{B}}\rangle\rangle$. Then it is easy to check that $\lambda s.\mathsf{bdcr}(e', f', u', b)(s) = \mathsf{dcr}(e', f', u')$ (note that s is a free variable occuring in the expression b).

So let $s \in \{t\}$ and $r = \mathsf{dcr}(e, f, u)(s)$. We have shown how $\mathit{lift}_{t'}(r)$ can be computed in $\mathcal{NRC}^1(\mathsf{bdcr})$. Finally observe that we can recover r from $\mathit{lift}_{t'}(r)$ by using the function get.

□

Note that Proposition 3.4.8 fails in the presence of certain external functions, see Proposition 3.7.3.

3.4.3 The Role of get

We show in this Section that even at flat types dcr is slightly more expressive than bdcr, because it can extract an element of type $\mathbb{D}$ from a singleton set, as we have seen in Example 3.4.3.

Throughout this Section we shall denote with $\mathcal{NRC}_-$ the language $\mathcal{NRC}$ without get.

Proposition 3.4.9 get *cannot be expressed in* $\mathcal{NRC}_-(\mathsf{bdcr})$.

Proof. (Sketch) Here we prove by induction on the structure of an expression e in $\mathcal{NRC}_-(\mathsf{bdcr})$ that the following holds. Let $f : t \to t'$ be the associated query. Then we show that each "scalar component" of $f(x)$ is one of the "scalar components" of x. More precisely, for every type t define $scalar_t : t \to \{\mathbb{D}\}$ to be:

$$\begin{aligned}
scalar_{unit}(\langle\rangle) &\stackrel{\text{def}}{=} \emptyset \\
scalar_{\mathbb{B}}(x) &\stackrel{\text{def}}{=} \emptyset \\
scalar_{\mathbb{D}}(x) &\stackrel{\text{def}}{=} \{x\} \\
scalar_{\{t\}}(x) &\stackrel{\text{def}}{=} \emptyset \\
scalar_{t_1\times\ldots\times t_n}\langle x_1,\ldots,x_n\rangle &\stackrel{\text{def}}{=} scalar_{t_1}(x_1)\cup\ldots\cup scalar_{t_n}(x_n)
\end{aligned}$$

Then, for every query $f : t \to t'$ associated to some expression in $\mathcal{NRC}_-(\mathsf{bdcr})$ we prove that the following condition holds:

$$\forall x \in t, scalar_{t'}(f(x)) \subseteq scalar_t(x) \tag{3.1}$$

Certainly get does not satisfy condition 3.1, because $scalar_{\mathbb{D}}(\mathsf{get}\langle\{d_1\}, d_2\rangle) = scalar_{\mathbb{D}}(d_1) = \{d_1\}$, while $scalar_{\{\mathbb{D}\}\times\mathbb{D}}(\langle\{d_1\}, d_2\rangle) = \{d_2\}$.

We prove condition 3.1 by induction on the structure of an expression e in $\mathcal{NRC}_{-}(\mathsf{bdcr})$. We consider some relevant cases for e:

Projection $e = \pi_1(e')$, where $e' : t' \times t'', e : t'$. Then $f = \lambda x.\pi_1(f'(x))$, where f' is the query associated to e'. By induction hypothesis we have $scalar_{t'\times t''}(f'(x)) \subseteq scalar_t(x)$. It suffices to observe that: $scalar_{t'}(\pi_1(f'(x))) \subseteq scalar_{t'\times t''}(f'(x))$.

Union $e = e_1 \cup e_2$. Since the type t' is now a set type, $scalar_{t'}(f(x)) = \emptyset$, and there is nothing to prove.

Function application $e = g(e')$. For the sake of clarity suppose that $g : t'' \to t'$ does not have any free variables, hence its associated query is g itself. Let $f' : t \to t''$ be the query associated to e', so $f = g \circ f'$. Then, using the induction hypotheses we have:

$$\begin{aligned} scalar_{t'}(f(x)) &= scalar_{t'}(g(f'(x))) \\ &\subseteq scalar_{t''}(f'(x)) \\ &\subseteq scalar_t(x) \end{aligned}$$

Bounded recursion As in the case of union, there is nothing to prove here, since the resulting type is a PS-type.

The remaining cases are easily checked. □

This result justifies the inclusion of get in $\mathcal{NRC}$. The next result shows that the additional expressive power brought by get is largely cosmetic. Indeed, we consider getting around the function get through type lifting, the technique for transforming a non PS-type into a PS-type defined in Subsection 3.4.2.

Proposition 3.4.10 *Let $f : t \to t'$ be a query in $\mathcal{NRC}(\mathsf{bdcr})$ (or in $\mathcal{NRC}^1(\mathsf{bdcr})$). Then $lift \circ f : t \to \vec{t}'$ is expressible in $\mathcal{NRC}_{-}(\mathsf{bdcr})$ (or in $\mathcal{NRC}^1_{-}(\mathsf{bdcr})$ respectively), i.e. without* get. *Moreover, $\forall k \geq 0$, if $f \in \mathcal{NRC}(\mathsf{bdcr}^{(k)})$ (or $f \in \mathcal{NRC}^1(\mathsf{bdcr}^{(k)})$), then $lift \circ f \in \mathcal{NRC}_{-}(\mathsf{bdcr}^{(k)})$ (or $lift \circ f \in \mathcal{NRC}^1_{-}(\mathsf{bdcr}^{(k)})$ respectively).*

As a consequence, adding or dropping get to/from $\mathcal{NRC}$ does not affect in any way the queries whose result is a PS-type.

Proof. We proof by induction on the structure of an expression e in $\mathcal{NRC}^1(\mathsf{bdcr}, \mathsf{get})$ that its associated query f satisfies the above property. The case when e is the query $\mathsf{get} : \{\mathbb{D}\} \times \mathbb{D} \to \mathbb{D}$ is trivial, because $\mathit{lift}_{\mathbb{D}} \circ \mathsf{get} : \{\mathbb{D}\} \times \mathbb{D} \to \{\mathbb{D}\}$ is:

$$\lambda\langle x, y\rangle.\mathsf{if}\ |x| = 1\ \mathsf{then}\ x\ \mathsf{else}\ \{y\}$$

which is expressible in $\mathcal{NRC}^1$. The only interesting case is when e is function application, i.e. $e = g(e')$. Let $f' : t \to t''$ be the query associated to e', and assume, for sake of clarity, that g has no free variables, i.e. the query associated to g is g itself. Then $f = g \circ f'$. By induction hypothesis we know that $\mathit{lift}_{t''} \circ f'$ and $\mathit{lift}_{t'} \circ g$ are expressible in $\mathcal{NRC}^1_{-}(\mathsf{bdcr})$, say by $h : t \to \bar{t}''$ and $h' : t'' \to \bar{t}'$. We prove that $\mathit{lift} \circ f = \bar{h}' \circ h$ and, since the latter is in $\mathcal{NRC}^1_{-}(\mathsf{bdcr})$, this concludes our proof. Indeed:

$$\begin{aligned} \bar{h}'(h(x)) &= \bar{h}'(\mathit{lift}_{t''}(f'(x))) \\ &= h'(f'(x)) \qquad \text{by Fact 3.4.7} \end{aligned}$$

□

3.4.4 Recursion and Order

One way of interpreting the roles of conditions like associativity, commutativity, etc., in the definition of dcr and sri, is as simple sufficient conditions for order independence. As we shall see in Subsection 3.4.5, checking these conditions is undecidable in general. On *ordered* structures on the other hand, one can define forms of recursion on sets that do not require conditions. For instance, Immerman, Patnaik, and Stemple [66] consider under the name *set-reduce* a form of recursion on sets which resembles somewhat sri. Set-reduce does not require conditions such as left-commutativity, etc. Instead, its definition relies on the existence of a linear ordering on the elements of the sets to which it is applied. We prove that, in the presence of order, this form of recursion has the same expressive power as sri. We also formulate a similar order-dependent form of recursion that corresponds to dcr in the presence of order. Finally, an interesting relationship holds between sru and dcr in the presence of order.

The presence of order means for us the addition of an external function $\leq: \mathbb{D} \times \mathbb{D} \to \mathbb{B}$, which is understood always to denote a linear order on $\mathbb{D}$. The order relation can be lifted to all types (e.g. see [75]), that is, for every type t there is an expression in $\mathcal{NRC}(\leq)$, $\leq_t: t \times t \to t$, whose meaning is a total order on t.

We begin by describing the main technique for proving that apparently more powerful forms of recursion on ordered sets are in fact expressible with our forms of recursion in the presence of order.

Main technique Given a function $\psi : \{t\} \to t'$, in order to compute $\psi(x)$, where $x = \{x_1, \ldots, x_n\}$, $x_1 < x_2 < \ldots < x_n$, we define for every number k, $0 \leq k \leq n$, a complex object $\tilde{k}$ which "encodes' all applications of ψ to "intervals" of length $\leq k$ in x, i.e. to sets $s \subseteq \{x_1, \ldots, x_n\}$ of the form $s = \{x_p, x_{p+1}, \ldots, x_q\}$, of cardinality $\leq k$ (i.e. $q - p + 1 \leq k$). Note that there are exactly $k(2n - k + 1)/2$ intervals of length $\leq k$ in x. For the sake of clarity, since t' is a PS-type, we will assume that t' is a product of two set types, that is $t' = \{t'_1\} \times \{t'_2\}$. Then for every number k, $0 \leq k \leq n$, we define the *encoding* of k to be $\tilde{k} \in \{t \times t\} \times (\{(t \times t) \times t'_1\} \times \{(t \times t) \times t'_2\})$:

$$
\begin{aligned}
\tilde{k} \stackrel{\text{def}}{=} \langle \quad & \{\langle x_p, x_q\rangle \mid 1 \leq p, q \leq n, 0 \leq q - p + 1 \leq k\}, \\
& \langle \{\langle\langle x_p, x_q\rangle, a_1\rangle \mid 1 \leq p, q \leq n, 0 \leq q - p + 1 \leq k, a_1 \in \pi_1(\psi(\{x_p, x_{p+1}, \ldots, x_q\}))\}, \\
& \{\langle\langle x_p, x_q\rangle, a_2\rangle \mid 1 \leq p, q \leq n, 0 \leq q - p + 1 \leq k, a_2 \in \pi_2(\psi(\{x_p, x_{p+1}, \ldots, x_q\}))\}\rangle\rangle
\end{aligned}
$$

We adopt the convention $\tilde{k} \stackrel{\text{def}}{=} \tilde{n}$, whenever $k > n$.

Thus $\tilde{k}$ contains the following informations: (1) the set of all pairs $\langle x_p, x_q\rangle$ for which $\{x_p, x_{p+1}, \ldots, x_q\}$ has between 0 and k elements, and (2) for each such pair $\langle x_p, x_q\rangle$, all values of $\psi(\{x_p, x_{p+1}, \ldots, x_q\})$, tagged with the pair $\langle x_p, x_q\rangle$. Given an encoding $\tilde{k}$ and $x_p, x_q \in x$, we can extract $\psi(\{x_p, \ldots, x_q\})$. Namely we define:

$$extract\langle\tilde{k}, \langle x_p, x_q\rangle\rangle \stackrel{\text{def}}{=} \langle\{a_1 \mid \langle\langle x_p, x_q\rangle, a_1\rangle \in \pi_1(\pi_2(\tilde{k}))\}, \{a_2 \mid \langle\langle x_p, x_q\rangle, a_2\rangle \in \pi_2(\pi_2(\tilde{k}))\}\rangle$$

Obviously *extract* can be expressed in $\mathcal{NRC}$, and $extract\langle\langle x_p, x_q\rangle, \tilde{k}\rangle = \psi(\{x_p, x_{p+1}, \dots, x_q\})$, for $0 \leq q - p + 1 \leq k$.

We will use this encoding to compute ψ with dcr, or with esr (and, hence, with sri). To compute ψ with dcr, assume that the following are expressible in $\mathcal{NRC}$: $\tilde{0}$, $\tilde{1}$, and the function $\lambda\langle\tilde{k}, \tilde{k}'\rangle.\widetilde{k + k'}$. Then we can express the function $\gamma : \{t\} \to \{t \times t\} \times (\{(t \times t) \times t'_1\} \times \{(t \times t) \times t'_2\})$ defined by

$$\gamma(s) = \begin{cases} \tilde{k} & \text{when } |s| = k \leq n \\ \tilde{n} & \text{when } |s| > n \end{cases}$$

as $\gamma = \mathsf{dcr}(\tilde{0}, \lambda y.\tilde{1}, \lambda\langle\tilde{k}, \tilde{k}'\rangle.\widetilde{k + k'})$. Obviously the function $\lambda\langle\tilde{k}, \tilde{k}'\rangle.\widetilde{k + k'}$ is associative, commutative, and has identity $\tilde{0}$. Finally, to compute $\psi(x)$, we take

$$\psi(x) = extract\langle\psi(x), \langle x_1, x_n\rangle\rangle$$

Note that $height(\{t \times t\} \times (\{(t \times t) \times t'_1\} \times \{(t \times t) \times t'_2\})) \leq \max(height(\{t\}), height(t'))$. This will allow us to argue that, whenever ψ mentions only types of set height ≤ 1, then its equivalent expression with dcr or esr will have only intermediate types of set height ≤ 1.

Set-reduce [66] Let $<: t \times t \to \mathbb{B}$ be a linear ordering on t, and for $e : t, j : t \times t' \to t'$, let *set-reduce(e,j)* $: \{t\} \to t'$ be defined by:

$$\begin{aligned} set\text{-}reduce(e, j)(\emptyset) &\stackrel{\text{def}}{=} e \\ set\text{-}reduce(e, j)(\{x_1, \dots, x_n\}) &\stackrel{\text{def}}{=} j\langle x_1, set\text{-}reduce(\{x_2, \dots, x_n\})\rangle, \text{ if } x_1 < x_2 < \dots < x_n \end{aligned}$$

(No conditions are imposed on j.) Then:

Proposition 3.4.11

$$\begin{aligned} \mathcal{NRC}^1(set\text{-}reduce, \leq) &= \mathcal{NRC}^1(\mathsf{sri}, \leq) \\ \mathcal{NRC}(bset\text{-}reduce, \leq) &= \mathcal{NRC}(\mathsf{bsri}, \leq) \end{aligned}$$

where bset-reduce is the bounded version of set-reduce.

Proof. Obviously:

$$\begin{aligned} \mathcal{NRC}^1(\mathsf{sri}, \leq) &\subseteq \mathcal{NRC}^1(\textit{set-reduce}, \leq) \\ \mathcal{NRC}(\mathsf{bsri}, \leq) &\subseteq \mathcal{NRC}(\textit{bset-reduce}, \leq) \end{aligned}$$

To prove $\mathcal{NRC}(\textit{bset-reduce}, \leq) \subseteq \mathcal{NRC}(\mathsf{bsri}, \leq)$, let $\psi = \textit{bset-reduce}(e, j, b) : \{t\} \to t'$. In order to express $\psi(x)$ with sri, where $x = \{x_1, \ldots, x_n\}$, with $x_1 < x_2 < \ldots < x_n$, we will use the encoding described earlier. Since t' is a PS-type, we will assume, for sake of clarity, that it is the product of two set types, $t' = \{t'_1\} \times \{t'_2\}$. Then we associate to each number k, its *encoding* $\tilde{k} \in \{t \times t\} \times (\{(t \times t) \times t'_1\} \times \{(t \times t) \times t'_2\})$. It is easy to check that, given the fact that $\psi = \textit{bset-reduce}(e, j, b)$, the following are expressible in $\mathcal{NRC}(\mathsf{bsri}, \leq)$: $\tilde{0}$, and the function $\lambda\tilde{k}.\widetilde{k+1}$. For the latter, suppose we are given some $\bar{k}$. To compute $\widetilde{k+1}$, consider all pairs $\langle x_p, x_q\rangle \in \pi_1(\tilde{k})$ with $p > 1$. Compute x_{p-1} (using the order relation on $x = \{x_1, \ldots, x_n\}$), and compute $\psi(\{x_p, \ldots, x_q\}) = \textit{extract}(\tilde{k}, \langle x_p, x_q\rangle)$. To compute $\psi(\{x_{p-1}, x_p, \ldots, x_q\})$, observe that $\psi(\{x_{p-1}, x_p, \ldots, x_q\}) = j(x_{p-1}, \psi(\{x_p, \ldots, x_q\}))$. Finally this allows us to assemble together the value $\widetilde{k+1}$.

Then we define in $\mathcal{NRC}(\mathsf{bsri}, \leq)$ the function $\gamma : \{t\} \to \{t \times t\} \times (\{(t \times t) \times t'_1\} \times \{(t \times t) \times t'_2\})$, $\gamma = \mathsf{esr}(\tilde{0}, \lambda\langle y, \tilde{k}\rangle.\widetilde{k+1})$, whose meaning is:

$$\gamma(s) = \begin{cases} \tilde{k} & \text{when } |s| = k \leq n \\ \tilde{n} & \text{when } |s| > n \end{cases}$$

Next we convert the definition of γ from esr into besr, by noting that $\forall k$, $0 \leq k \leq n$, $\tilde{k} \subseteq b'$, where $b' = (x \times x) \times (((x \times x) \times \Pi_1(b)) \times ((x \times x) \times \Pi_2(b)))$. Hence $\gamma = \mathsf{besr}(\tilde{0}, \lambda\langle y, \tilde{k}\rangle.\widetilde{k+1}, b')$. Finally we convert it into a definition with bsri, using Proposition 3.4.1, and observe that $\psi(x) = \textit{extract}\langle\langle x_1, x_n\rangle, \gamma(x)\rangle$.

To check $\mathcal{NRC}^1(\textit{set-reduce}, \leq) \subseteq \mathcal{NRC}^1(\mathsf{sri}, \leq)$, we argue as follows. First, by extending Proposition 3.4.8 to *set-reduce*, we obtain $\mathcal{NRC}^1(\textit{set-reduce}, \leq) = \mathcal{NRC}^1(\textit{bset-reduce}, \leq)$. Next, we observe that in the translation given above of $\textit{bset-reduce} : \{t\} \to t'$ into bsri, the resulting expression uses only types of set height $\leq \max(height(\{t\}), height(t'), 1))$, hence $\mathcal{NRC}^1(\textit{bset-reduce}, \leq) \subseteq \mathcal{NRC}^1(\mathsf{bsri}, \leq)$. Finally we use Proposition 3.4.8 to argue that $\mathcal{NRC}^1(\mathsf{bsri}, \leq) = \mathcal{NRC}^1(\mathsf{sri}, \leq)$. □

Set-divide Similarly, one can conceive a form of divide and conquer recursion that relies on the ordering, which allows us to define some function by $\varphi(\{x_1, \ldots, x_n\}) \stackrel{\text{def}}{=} u(\varphi(\{x_1, \ldots, x_{\lfloor \frac{n}{2} \rfloor}\}), \varphi(\{x_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, x_n\}))$ (no conditions are imposed on u). That is, we make the arbitrary choice of dividing a set into two almost equal halves at each iteration step[4]. Again, we can prove that this form of recursion has the same expressive power as dcr. First we need a lemma:

Lemma 3.4.12 ([64]) *Let $TC_t : \{t \times t\} \to \{t \times t\}$ be the function computing the transitive closure of binary relations of elements of type t, and let $\mathcal{NRC}(TC)$ stand for $\mathcal{NRC}$ extended with TC_t for every type t. Then, the function eq-cardinality : $\{t\} \times \{t\} \to \mathbb{B}$, defined by eq-cardinality$(x, y) =$ true iff $|x| = |y|$, is expressible in $\mathcal{NRC}(TC, \leq)$.*

Proof. Let $x = \{x_1, \ldots, x_m\}$, $y = \{y_1, \ldots, y_m\}$, and assume $x_1 < x_2 < \ldots < x_m$, $y_1 < y_2 < \ldots < y_n$. Let $r \in \{(t \times t) \times (t \times t)\}$ be $r = \{\langle\langle x_i, y_j\rangle, \langle x_{i+1}, y_{j+1}\rangle\rangle \mid i = 1, m-1; j = 1, n-1\}$. Obviously r can be computed in $\mathcal{NRC}(\leq)$ from x and y. Compute the transitive closure of r, $q \stackrel{\text{def}}{=} TC_{t \times t}(r)$. Then $m = n$ iff $\langle\langle x_1, y_1\rangle, \langle x_m, y_n\rangle\rangle \in q$. □

Set-divide is defined formally as follows. Let $<: t \times t \to \mathbb{B}$ be a linear ordering on t, and for $e : t, f : t \to t', v : t' \times t' \to t'$ define $\textit{set-divide}(e, f, v) : \{t\} \to t'$ by:

$$
\begin{aligned}
\textit{set-divide}(e, f, v)(\emptyset) &\stackrel{\text{def}}{=} e \\
\textit{set-divide}(e, f, v)(\{a\}) &\stackrel{\text{def}}{=} f(a) \\
\textit{set-divide}(e, f, v)(\{x_1, \ldots, x_n\}) &\stackrel{\text{def}}{=} v(\textit{set-divide}(e, f, v)(\{x_1, \ldots, x_{\lfloor \frac{n}{2} \rfloor}\}), \\
&\qquad \textit{set-divide}(e, f, v)(\{x_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, x_n\}))
\end{aligned}
$$

(No conditions are imposed on v.) Then:

[4] Another ad-hoc way of defining recursion, related to *set-divide*, but on vectors instead of sets, can be found in the equational parallel language EL^* [89]

Proposition 3.4.13

$$\begin{aligned}\mathcal{NRC}^1(\textit{set-divide}, \leq) &= \mathcal{NRC}^1(\mathsf{dcr}, \leq)\\ \mathcal{NRC}(\textit{bset-divide}, \leq) &= \mathcal{NRC}(\mathsf{bdcr}, \leq)\end{aligned}$$

where bset-divide is the bounded version of set-divide.

Proof. To prove $\mathcal{NRC}(\textit{bset-divide}, \leq) \subseteq \mathcal{NRC}(\mathsf{bdcr}, \leq)$ we use the same idea as in Proposition 3.4.11. Let $\psi = \textit{bset-divide}(e, f, v, b)$. As before, we shall assume, for sake of argument that t' is a product of two set types, that is $t' = \{t'_1\} \times \{t'_2\}$. Then, for a given $x \in \{t\}$, $x = \{x_1, \ldots, x_n\}$, with $x_1 < x_2 < \ldots < x_n$, and for every number k, $0 \leq k \leq n$, we consider the encoding of k, $\tilde{k} \in \{t \times t\} \times (\{(t \times t) \times t'_1\} \times \{(t \times t) \times t'_2\})$:

Given that $\psi = \textit{bset-divide}(e, f, v, b)$, the following are expressible in $\mathcal{NRC}(\mathsf{bdcr}, \leq)$: $\tilde{0}$, $\tilde{1}$, and, the function $\lambda\langle\tilde{k}, \tilde{k}'\rangle.\widetilde{k + k'}$. We will explain how to express the latter. Recall that $\tilde{k}$ encodes all values of $\psi(s)$, for s and interval of $x_1 < x_2 < \ldots < x_n$, of cardinality $\leq k$. To compute $\widetilde{k + k'}$ consider all intervals $s'' \subseteq \{x_1, \ldots, x_n\}$ of cardinality $k'' \leq k + k'$ (there are $n - (k + k') + 1$ such intervals). Split $s'' = \{x_p, x_{p+1}, \ldots, x_q\}$ into two halves of cardinalities $\lfloor \frac{k''}{2} \rfloor$ and $\lceil \frac{k''}{2} \rceil$ respectively, $s = s_1 \cup s_2$, with $s_1 = \{x_p, x_{p+1}, \ldots, x_r\}$, $s_2 = \{x_{r+1}, x_{r+2}, \ldots, x_q\}$. This can be done using Lemma 3.4.12, given that bdcr can express transitive closure. Now we argue that both s_1 and s_2 have cardinalities which are less than or equal to the largest of k, k'. Indeed, if not, then $|s_1| > k$, $|s_2| > k'$, hence $|s| = |s_1| + |s_2| > k + k'$, contradiction. Therefore, assuming $k \geq k'$, it suffices to "look up" the values $\psi(s_1)$ and $\psi(s_2)$ in $\tilde{k}$, and to compute $\psi(s) = v(\psi(s_1), \psi(s_2))$.

Next, as in the proof of Proposition 3.4.11, we will consider the function

$$\gamma \stackrel{\text{def}}{=} \mathsf{dcr}(\tilde{0}, \lambda y.\tilde{1}, \lambda\langle\tilde{k}, \tilde{k}'\rangle.\widetilde{k + k'})$$

with the meaning:

$$\gamma(s) = \begin{cases} \tilde{k} & \text{when } |s| = k \leq n \\ \tilde{n} & \text{when } |s| > n \end{cases}$$

The definition of γ is obviously correct, since $\lambda\langle\tilde{k}, \tilde{k}'\rangle.\widetilde{k + k'})$ is associative, commutative, and has identity $\tilde{0}$. As in the proof of Proposition 3.4.11, we observe that $\gamma(x) =$

$\mathsf{bdcr}(\tilde{0}, \tilde{1}, \lambda\langle\tilde{k}, \tilde{k}'\rangle.\widetilde{k + k'}, b')(x)$, where $b' = (x \times x) \times (((x \times x) \times \Pi_1(b)) \times ((x \times x) \times \Pi_2(b)))$. Finally $\psi(x) = \mathit{extract}\langle\psi(x), \langle x_1, x_n\rangle\rangle$.

To prove $\mathcal{NRC}^1(\mathit{set\text{-}divide}, \leq) \subseteq \mathcal{NRC}^1(\mathsf{dcr}, \leq)$, we proceed as in the proof of Proposition 3.4.11. Namely we first extend Proposition 3.4.8 to

$$\mathcal{NRC}^1(\mathit{set\text{-}divide}, \leq) = \mathcal{NRC}^1(\mathit{bset\text{-}divide}, \leq)$$

then observe that the above construction also proves that

$$\mathcal{NRC}^1(\mathit{bset\text{-}divide}, \leq) \subseteq \mathcal{NRC}^1(\mathsf{bdcr}, \leq)$$

since the auxiliary types never exceed in set height the heights of the input and output types. Finally we use Proposition 3.4.8 to argue that $\mathcal{NRC}^1(\mathsf{bdcr}, \leq) = \mathcal{NRC}^1(\mathsf{dcr}, \leq)$. $\square$

sru vs. dcr Although superficially related to dcr, sru seems to have less expressive power. In fact it remains open whether transitive closure can be expressed in $\mathcal{NRC}^1(\mathsf{sru})$, or in $\mathcal{NRC}(\mathsf{bsru})$, or even in $\mathcal{NRC}(\mathsf{bsru}, \leq)$. We do not venture a conjecture about the last language, but there is some reason to think that the following is true:

Conjecture 3.4.14 *Transitive closure cannot be expressed neither in* $\mathcal{NRC}^1(\mathsf{sru})$*, nor in* $\mathcal{NRC}(\mathsf{bsru})$*.*

Interestingly, with order and TC, bsru becomes as powerful as bdcr:

Proposition 3.4.15

$$\begin{aligned} \mathcal{NRC}^1(\mathsf{sru}, \leq, TC) &= \mathcal{NRC}^1(\mathsf{dcr}, \leq) \\ \mathcal{NRC}(\mathsf{bsru}, \leq, TC) &= \mathcal{NRC}(\mathsf{bdcr}, \leq) \end{aligned}$$

Proof. To show $\mathcal{NRC}(\mathsf{bdcr}, \leq) \subseteq \mathcal{NRC}(\mathsf{bsru}, \leq, TC)$ consider some expression $\psi = \mathsf{bdcr}(e, f, u, b)$ in $\mathcal{NRC}(\mathsf{bdcr}, \leq)$. Without loss of generality we may assume that e, f, u, b are in $\mathcal{NRC}(\mathsf{bsru}, \leq)$.

To compute $\psi(\{x_1,\ldots,x_n\})$, $x_1 < x_2 < \ldots < x_n$, using bsru, we use the encodings of numbers $\tilde{k}$, $0 \le k \le n$ mentioned earlier. Namely we define the function φ to be $\varphi(s) \stackrel{\text{def}}{=} \langle\tilde{k}, s\rangle$, where $k = |s|$. φ can be defined using bsru. Obviously $\varphi(\emptyset) \stackrel{\text{def}}{=} \langle\tilde{0}, s\rangle$ and $\varphi(\{y\}) \stackrel{\text{def}}{=} \langle\tilde{1}, \{y\}\rangle$. For $\varphi(s \cup s')$, let $k = |s|, k' = |s'|$ and $k'' = |s \cup s'|$. We have $\varphi(s \cup s') = \langle\widetilde{k''}, s \cup s'\rangle$, so we have to argue that $\widetilde{k''}$ can be computed, given $\tilde{k}$, $\tilde{k}'$, and s, s'. First note that $\tilde{k}'' \subseteq \widetilde{k+k'}$. So all we have to do is to select from $\widetilde{k+k'}$ all those values which are tagged with pairs $\langle x_p, x_q\rangle$ with $q - p + 1 \le k''$. (Recall that $\lambda\langle\tilde{k},\tilde{k}'\rangle.\widetilde{k+k'}$ can be computed, see the proof of Proposition 3.4.13.) The latter is equivalent to $|\{x_p,\ldots,x_q\}| \le |s \cup s'|$, which can be tested in $\mathcal{NRC}(\le, TC)$, using Lemma 3.4.12. Finally we have $\psi(\{x_1,\ldots,x_n\}) = \mathit{extract}(\pi_1(\varphi(\{x_1,\ldots,x_n\})), \langle x_1, x_n\rangle)$. $\Box$

3.4.5 The Cost of Order-Independence

In contrast to the order-dependent recursion schemas *set-reduce* and *set-divide*, the algebraic conditions, besides the fact that they arise from principled mathematical characterizations of finite sets, provide us with an elegant alternative for ensuring the well-definedness of various forms of recursion on sets. Unfortunately, for a language at least as expressive as first-order logic, verifying in general most of these identities is as hard as testing the validity of a first-order formula in all finite models. We prove:

Theorem 3.4.16 *Deciding whether some expression* $\mathsf{dcr}(e, f, u)$ *in* $\mathcal{NRC}^1(\mathsf{dcr})$ *is well defined is co-r.e., but not r.e. Similar results hold for* sru, sri, *as well as for their bounded versions.*

To prove this result we use Trakhtenbrot's theorem (see, for example [47]):

Theorem 3.4.17 (Trakhtenbrot) *Assume that a first order language $\mathcal{L}$ contains some relation symbol that is not unary. Then the set of first-order sentences over $\mathcal{L}$ valid in all finite structures is Π^0_1-complete.*

Proof. (of Theorem 3.4.16) For some values of the free variables in e, f, u, let A denote the set obtained by closing $\{e\} \cup Im(f)$ under applications of u. The expression $\mathsf{dcr}(e, f, u)$ is well defined iff, for any values of the free variables in e, f, u, the function u is associative, commutative, and has identity e, on the set A. Since A is enumerable, testing whether $\mathsf{dcr}(e, f, u)$ is *not* well defined is obviously r.e. For the converse, we will reduce the validity problem over over finite models to the decision whether some expression $\mathsf{dcr}(e, f, u)$ is well defined. Let $\mathcal{L}$ be a first order language having at least some non unary relation symbol. To keep our argument simple, suppose $\mathcal{L}$ has exactly one binary symbol R. Since first order logic is equivalent to the Relational Algebra, and the latter is essentially equivalent to $\mathcal{NRC}^1$, we conclude that for any sentence φ in $\mathcal{L}$ there exists a query $g : \{\mathbb{D}\} \times \{\mathbb{D} \times \mathbb{D}\} \to \mathbb{B}$ in $\mathcal{NRC}^1$ such that φ is true in the finite model (D, R), $R \subseteq D \times D$, iff $g(D, R) = \mathsf{true}$. Then consider the following expression $\mathsf{dcr}(e, f, u) : \{\mathbb{D}\} \to \{\mathbb{D}\}$ with free variables D, R: $e \stackrel{\text{def}}{=} \emptyset$, $f \stackrel{\text{def}}{=} \lambda x^{\mathbb{D}}.x^{\mathbb{D}}$, $u(s_1, s_2) \stackrel{\text{def}}{=} \mathsf{if}\ g(D \cup \Pi_1(R) \cup \Pi_2(R), R)\ \mathsf{then}\ s_1 \cup s_2\ \mathsf{else}\ s_1 - s_2$. When φ is valid in all finite models, then $g(D \cup \Pi_1(R) \cup \Pi_2(R))$ is true for all values of the free variables D and R, hence the function u coincindes with union. In this case $\mathsf{dcr}(e, f, u)$ is well defined. On the other hand, if φ is not true in some finite model (D, R), then for that particular value of the free variables D and R the function u conicides with set diference, and hence $\mathsf{dcr}(e, f, u)$ is not well defined. □

3.5 Complexity Classes

We review here the definitions of the complexity classes AC^k and NC.

Let $F : \{0,1\}^* \to \{0,1\}^*$. For some $W \in \{0,1\}^*$ we denote $\mathsf{length}(W)$ the length of the string W.

Definition 3.5.1 ([88], pp. 766) *We say that F is in AC^k, for $k \geq 0$ iff the following conditions are met:*

1. *There is some polynomial $Q(n)$ s.t. $\forall W \in \{0,1\}^*$, $\mathsf{length}(f(W)) = Q(\mathsf{length}(W))$. Thus, F is the union of its restrictions to inputs of length n, $F = \bigcup_{n \geq 0} F_n$, where*

$F_n : \{0,1\}^n \rightarrow \{0,1\}^{Q(n)}$.

2. *There is a family of circuits* $(\alpha_n)_{n\geq 0}$, *where* α_n *is a circuit computing* F_n, *with* n *input gates,* $Q(n)$ *outputs, and made up of* NOT *gates, unbounded fan-in* AND *and* OR *gates.*

3. *For every* $n \geq 0$, $size(\alpha_n) \leq P(n)$ *for some polynomial* P *(the size is the number of gates), and* $depth(\alpha_n) = O(\log^k n)$.

4. *The family* α_n *is "uniform", as described below.*

Following Cook ([36], Proposition 4.7), we impose as uniformity condition the *DLOGSPACE-DCL* uniformity. Barrington, Immerman and Straubing in [11] give a weaker uniformity condition called *FO-DCL*-uniformity which is equivalent to the *DLOGSPACE-DCL* uniformity for the classes AC^k, $k \geq 1$, and which provide a more satisfactory characterization for AC^0. In this Chapter, only Proposition 3.7.5 deals with the class AC^0 and it remains true for the more restrictive *FO-DCL*-uniformity condition in [11].

The **direct connection language** DCL for a family α_n of circuits, is the set of quadruples (n, g, g', t), where g, g' are gate numbers in α_n, such that g is a child of g', and t is the type of g', $t \in \{NOT, AND, OR, y_1, \ldots, y_{Q(n)}\}$. When $t = y_i$, then g' is the output bit i. We use the convention that the input gates corresponding to $x_1, \ldots, x_n$ are identified by assigning them the numbers $1, \ldots, n$. We say that the family of circuits α_n is *DLOGSPACE-DCL* **uniform**, iff the DCL can be accepted by some $O(\log n)$ space deterministic Turing Machine T.

Definition 3.5.2 $NC \stackrel{\text{def}}{=} \bigcup_{k\geq 0} AC^k$.

The results in Stockmeyer and Vishkin [93] imply that NC coincides with the class of functions computable by a *CRCW PRAM* (Concurrent Read Concurrent Write Parallel Random Access Machine) in polylogarithmic time using polynomially many processors.

3.6 Computing Complex Objects Queries

In order to give precise definitions for query complexity classes, we must specify an encoding of complex objects into strings that can be given as input to a computational model such as PRAM's or families of circuits.

Our encoding of complex objects with strings over some fixed alphabet is related to that in [54]. We start with an encoding of the base type $\mathbb{D}$ into natural numbers, i.e. we assume some bijection $\Phi : \mathbb{D} \to \mathbb{N}$ to be given. When dealing with ordered databases, we require this encoding to preserve the order relation $\leq$ on $\mathbb{D}$. Next, we encode complex objects using the eight symbols from the alphabet $A = \{0, 1, \{, \}, \langle, \rangle, \mathit{comma}, \mathit{blank}\}$, as follows: elements from $\mathbb{D}$ are encoded in binary, true and false are encoded by 1 and 0 respectively, $\langle\rangle$ is encoded by $\langle\rangle$, an n-tuple is encoded by $\langle X_1, \ldots, X_n \rangle$, and a set by $\{X_1, \ldots, X_n\}$. No duplicates are allowed in the encoding of a set. However, blanks may be scattered arbitrarily inside some encoding, but not inside the binary numbers. Since the encoding of some complex object x is not unique, we define an encoding relation $x \sim X$ to denote the fact that X is a valid encoding of x. We view encodings as strings in $\{0, 1\}^*$, by further encoding each of the eight symbols in A by a string of length 3 in $\{0, 1\}^*$.

Removing duplicates is essential in the presence of recursors or iterators; else the size of some representation could grow beyond any polynomial. Duplicates can be removed in AC^0, by replacing them with blanks, and blanks can be removed (more precisely: moved at the end) in AC^1. So, within AC^0 we have the alternative choice of encoding with possible duplicates and no blanks, because there are no iterations allowed within AC^0. Within AC^k, $k \geq 1$, we could ask both for blanks and duplicate elimination. Our choice of encoding without duplicates, but to allow blanks, works across all AC^k, for $k \geq 0$.

Note that this encoding is different from that considered by Immerman in [65], who only deals with flat relations. Under that encoding, a relation of type $\{\mathbb{D}^k\}$ is represented by a string of bits of length n^k, where n is the size of the *active domain*, see Subsection 3.6.1. This encoding does not extend nicely to complex objects, because it requires exponential space to encode objects of higher types, even when their size is only polynomial in the size of the active domain. For flat relations, we show in Subsection 3.6.1 below, that we can translate between the two encodings in AC^0, see Proposition 3.6.2.

Adapting the definition in [28], we define a **database query** of type $t_1 \to t_2$ to be a function $f : t_1 \to t_2$ which is *generic*, see Definition 3.3.1. We say that a query $f : t_1 \to t_2$ is **computed** by the function $F : \{0,1\}^* \to \{0,1\}^*$ iff $\forall x \in t_1$, $\forall X \in \{0,1\}^*$, $x \sim X \Longrightarrow f(x) \sim F(X)$.

The fact that we only deal with *generic* queries gives us more liberty in the encoding complex objects, without using the bijection Φ. Suppose the query f is computed by F. To compute $f(x)$ for some input $x \in t_1$, we may proceed as follows. Let $d_0, d_1, \ldots, d_{m-1}$ be the **active domain** of x, i.e. all values of type $\mathbb{D}$ mentioned in x. In the case of ordered databases, we will also assume $d_0 < d_1 < \ldots < d_{m-1}$. In general we don't know the numbers $\Phi(d_0), \ldots, \Phi(d_{m-1})$, but we will choose to encode $d_0, \ldots, d_{m-1}$ with the numbers $0, 1, \ldots, m-1$; we call this **the minimal encoding**, to distinguish it from the **standard encoding** using Φ. Let X be the resulting minimal encoding of x. Apply F on X to get $Y = F(X)$. Since the query is generic, one can prove that the only numbers occuring in Y are $0, 1, \ldots, m-1$. Hence we can decode Y under the minimal encoding, to get y. Now we prove that y is the correct result, i.e. $y = f(x)$. Indeed, let $\mathbb{D}' = \{d'_0, d'_1, \ldots, d'_{m-1}\}$, where $d'_0, \ldots, d'_{m-1}$ are such that $\Phi(d'_i) = i$, for $i = 0, m-1$. Then the function $\varphi : \mathbb{D}' \to \mathbb{D}$, $\varphi(d'_i) \stackrel{\text{def}}{=} d_i$ for $i = 0, m-1$ is injective and, in the case of ordered databases, also order-preserving. Let $\varphi_t : t'_1 \to t_1$ be its extension to t_1, where t'_1 is obtained from t_1 by replacing every occurence of $\mathbb{D}'$ with $\mathbb{D}$ (see Section 3.3). Let $x' = \varphi_{t_1}^{-1}(x)$. Obviously X is the standard encoding of x' and, since F computes f, we have that $Y = F(X)$ is the standard encoding of $y' = f(x')$. A moment of thought will convince the reader that decoding Y under the minimal encoding yields $\varphi_{t_2}(y')$, that is $y = \varphi_{t_2}(y')$. Now we use the fact that f is generic to argue $\varphi_{t_2}(y') = \varphi_{t_2}(f(x')) = f(\varphi_{t_1}(x')) = f(x)$. Hence, when we decode Y according to the minimal encoding, we obtain $f(x)$.

Definition 3.6.1 *We say that a query f is in NC iff there is some function $F : \{0,1\}^* \to \{0,1\}^*$ computing f which is in NC. We denote by CMPX-OBJ-NC the class of queries which are in NC, and by FLAT-NC the class of queries over types of set height ≤ 1*

which are in NC. Similarly, for some $k \geq 0$, we define queries in AC^k, and the classes CMPX-OBJ-AC^k and FLAT-AC^k.

3.6.1 Alternative Encoding of Flat Relations

Immerman [65] proves a relationship between queries computable in extensions of FO and parallel complexity classes. His encoding of flat relations is different from ours; it is more elegant when dealing with flat relations, but does not extend to complex objects. We prove here that over flat relations our encoding is AC^0-equivalent to Immerman's, i.e. that the functions translating between the two encodings are in AC^0.

For sake of simplicity we will consider only the base type $\mathbb{D}$, and drop $\mathbb{B}$. Then a *flat relation type* is a type of the form $\{\mathbb{D}^k\}$. For a flat relation $x \in \{\mathbb{D}^k\}$ we define the **bit-wise encoding** of r to be the following string $X \in \{0,1\}^*$. Let $d = \{d_0, d_1, \ldots, d_{n-1}\} \in \{\mathbb{D}\}$ be a set containing the *active domain* of x, i.e. the set of all values of type $\mathbb{D}$ mentioned in x, and assume $d_0 < d_1 < \ldots < d_{n-1}$. Then the encoding X associated to x and d will have length n^k and will be defined as follows. For every $i_1, i_2, \ldots, i_k$, $0 \leq i_1, i_2, \ldots i_k \leq n-1$, $X[i_1 n^{k-1} + i_2 n^{k-2} + \ldots + i_k] = 1$ iff $\langle d_{i_1}, \ldots, d_{i_k} \rangle \in x$.

Proposition 3.6.2 *For every $k \geq 0$, the minimal encoding and the bit-wise encoding of flat relations of type $\{\mathbb{D}^k\}$ are AC^0-equivalent. More precisely: (1) There exists a function $F : \{0,1\}^* \to \{0,1\}^*$ in AC^0, mapping $\{0,1\}^{n^k}$ to $\{0,1\}^{n^k(3kn+3k+6)+3}$ for $n \geq 0$, such that, for every string X representing the bit-wise encoding of some flat relation x, we have $x \sim F(X)$ and, moreover, $F(X)$ is a minimal encoding of x. (2) There exists a function $G : \{0,1\}^* \to \{0,1\}^*$ in AC^0, mapping $\{0,1\}^n$ to $\{0,1\}^{n^k}$ for $n \geq 0$, such that for every string X encoding some flat relation x, $G(X)$ is a bit-wise encoding of x.*

Proof. To compute $F(X)$ we proceed as follows. The output string will be formed of a leading left parentheses, {, followed by n^k "cells" of $3kn + 3k + 6$ bits each. Each cell is split into a "body" built from the first $3kn + 3k + 3$ bits, and a "tail", containing the last three bits. We fill the cell corresponding to $i_1, i_2, \ldots, i_k$ either with blanks, when

$X[i_1n^{k-1}+i_2n^{k-2}+\ldots+i_k] = 0$, or with $\langle i_1, \ldots, i_k\rangle$, otherwise. For the second case observe that each $i_1, \ldots, i_k$ are between 0 and $n-1$, hence each requires only $3\lceil\log(n+1)\rceil \leq 3n$ bits, for a total of $3kn$ bits, and we need an addtional $3k+3$ bits to encode the brackets $\langle\rangle$ and the commas. In the first case we fill the "tail" with a blank, while in the second case we fill it with a comma, except for the last non-blank body, where we put a right parenthesis $\}$.

For $G(X)$ we observe first that the active domain of x has less than n elements, where $n = \mathsf{length}(X)$. We start by computing $Y = B_n^{\{\mathbb{D}^k\}}(X)$, see Lemma 3.8.7. Then, for every $i_1, \ldots, i_k$, the bit number $i_1n^{k-1} + i_2n^{k-2} + \ldots + i_k$ of the output will be 1 iff $\exists j, j'$ $0 \leq j < j' \leq n-1$ s.t. $Y[j] = Y[j'] = 1$ and $\forall k.j < k < j', Y[k] = 0$, and $X[j : j'-1]$ equals $\langle i_1, \ldots, i_k\rangle$. The latter test can be made using a circuit of depth $O(k)$ (since we need k' levels of comparisons to identify the position of $y_{k'}$, for $1 \leq k' \leq k$). $\square$

3.7 Main Results

We only state the results here and give the proofs in Section 3.8.

Theorem 3.7.1 $\mathcal{NRC}(\mathsf{bdcr}, \leq) =$ *CMPX-OBJ-NC. At a finer level of detail the following holds:* $\mathcal{NRC}(\mathsf{bdcr}^{(k)}, \leq) =$ *CMPX-OBJ-*AC^k *for every* $k \geq 1$.

Theorem 3.7.2 $\mathcal{NRC}^1(\mathsf{dcr}, \leq) =$ *FLAT-NC. At a finer level of detail the following holds:* $\mathcal{NRC}^1(\mathsf{dcr}^{(k)}, \leq) =$ *FLAT-*AC^k *for every* $k \geq 1$.

These languages are purely for complex objects, respectively relations. But many external functions of practical interest such as the usual arithmetical operations $(+, *, -, /, etc)$, and the usual aggregate functions (cardinality, sum, average, etc.) are also in NC. Can they be added in? The answer is yes for bdcr but no for dcr:

Proposition 3.7.3 *Let* Σ *be an extension consisting of possible additional base types and a set of functions computable in* NC. *Then* $\mathcal{NRC}(\Sigma, \mathsf{bdcr}) \subseteq NC$. *However,* $\mathcal{NRC}^1(\mathbb{N}, +, \mathsf{dcr})$

can express exponential space queries.

We prove the inclusion $\mathcal{NRC}(\Sigma, \mathsf{bdcr}) \subseteq NC$ in Section 3.8. The following example shows that $\mathcal{NRC}^1(\mathbb{N}, +, \mathsf{dcr})$ can express exponential space queries.

Example 3.7.4 *Consider the subset of* $\{\mathbb{N}\}$ *containing all sets of the form* $\{1, 2, \ldots, k\}$*; we view such a set as the encoding of the integer* k*. Then we can define the functions* $plus, times, exp : \{\mathbb{N}\} \times \{\mathbb{N}\} \to \{\mathbb{N}\}$*, with the meaning:* $x = \{1, 2, \ldots, m\}, y = \{1, 2, \ldots, n\} \Rightarrow plus\langle x, y\rangle = \{1, 2, \ldots, m + n\}, times\langle x, y\rangle = \{1, 2, \ldots, m * n\}, exp\langle x, y\rangle = \{1, 2, \ldots, n^m\}$*, in* $\mathcal{NRC}^1(\mathbb{N}, +, \mathsf{dcr})$*. Namely we define plus directly, while for times and exp we hint on their definition with* dcr*:*

$$
\begin{aligned}
plus\langle x, y\rangle &\stackrel{\text{def}}{=} \mathsf{ext}(\lambda\langle u, v\rangle.\{u + v\})(x \times y) \\
times\langle x \cup x', y\rangle &\stackrel{\text{def}}{=} plus(times\langle x, y\rangle, times\langle x', y\rangle) \quad \textit{when } x \cap x' = \emptyset \\
exp\langle x \cup x', y\rangle &\stackrel{\text{def}}{=} times\langle exp\langle x, y\rangle, exp\langle x', y\rangle\rangle \quad \textit{when } x \cap x' = \emptyset
\end{aligned}
$$

Certainly the function *exp* is not in NC, because the size of its output is not polynomially bounded by the size of its input. In particular Example 3.7.4 shows that dcr is strictly more powerful than bdcr over flat types, in the presence of certain external functions, see Proposition 3.4.8.

Immerman in [65] and Barrington, Immerman and Straubing in [11] prove that FO is included in FO-DCL-uniform AC^0, and that FO together with order and BIT relation has the same expressive power as AC^0. Here, we prove that $\mathcal{NRC}$ is included in AC^0, thus extending half of their result to complex objects.

Proposition 3.7.5 *Under the encoding of complex objects described in Section 3.6, all queries in* $\mathcal{NRC}(\leq)$*, are in FO-DCL-uniform* AC^0 *(see [11])*

We state two more results which help us put the main theorems in perspective. Their proofs are omitted.

Conservative extension One may wonder in what sense Theorem 3.7.2 is a "particular case" of Theorem 3.7.1. Actually, even though the proof of Theorem 3.7.2 is quite similar to that of Theorem 3.7.1, and we do present them "together" in Section 3.8, Theorem 3.7.2 in fact follows from Theorem 3.7.1, Proposition 3.4.8 and the conservative extension result presented below.

Paredaens and Van Gucht in [84], and Wong in [107] prove that $\mathcal{NRC}$ is a conservative extension of $\mathcal{NRC}^1$. Suciu in [95] proves that $\mathcal{NRC}(bfix)$ is a conservative extension of $\mathcal{NRC}^1(fix)$, where fix is the usual inflationary fixpoint, and $bfix$ is a bounded version of fix. Using the techniques in [95], we can prove the following:

Proposition 3.7.6 *Let Σ be a set of external functions which have set heights ≤ 1. Then, $\mathcal{NRC}(\Sigma, \mathsf{bdcr}, \leq)$ is a conservative extension of $\mathcal{NRC}^1(\Sigma, \mathsf{bdcr}, \leq)$.*

Note that for the case when $\Sigma = \emptyset$, we can turn the tables and Proposition 3.7.6 follows directly from the main theorems. For the case when $\Sigma \neq 0$, this proposition requires a separate proof, and we are able to do it only in the presence of order. However, we conjecture that $\mathcal{NRC}(\mathsf{bdcr})$ is a conservative extension of $\mathcal{NRC}^1(\mathsf{dcr})$.

PTIME vs. NC Immerman, Patnaik and Stemple [66] show that *PTIME* is captured by a language built around *set-reduce* (see Section 3.4). Extending their result also to complex objects we have:

Proposition 3.7.7 $\mathcal{NRC}^1(\mathsf{sri}^{(1)}, \leq) =$ *Q-PTIME [66] and* $\mathcal{NRC}(\mathsf{bsri}^{(1)}, \leq) =$ *CQ-PTIME.*

Here *Q-PTIME* and *CQ-PTIME* are the set of queries over flat relations, and over complex objects respectively, computable by some function in *PTIME*.

Thus, by the main theorems and this proposition, the difference between *PTIME* and *NC* computable queries over ordered databases can be characterized by the difference between two kinds of recursion on sets. It is interesting to note that only one level of recursion nesting suffices for sri and *PTIME*, as opposed to dcr and *NC*.

3.8 Proofs

3.8.1 Iteration Over Sets

The central technical tool in proving our main result, is to convert the two forms of recursion over sets, into more simple *loops*. The **logarithmic** and the **bounded logarithmic iterator** are defined by:

$$\frac{f : t \to t}{\mathsf{log\_loop}(f) : \{t'\} \times t \to t} \quad \frac{f : t \to t \quad b : t}{\mathsf{blog\_loop}(f, b) : \{t'\} \times t \to t}$$

with the semantics:

$$\mathsf{log\_loop}(f)\langle x, y\rangle \stackrel{\text{def}}{=} f^{(\lceil \log(|x|+1)\rceil)}(y)$$

where $|x|$ is the cardinality of x. Thus, $\mathsf{log\_loop}$ iterates some function f a number of times equal to the number of bits necessary to represent the number $|x|$. The bounded logarithmic iterator is define by:

$$\mathsf{blog\_loop}(f, b)\langle x, y\rangle \stackrel{\text{def}}{=} \mathsf{log\_loop}(f \cap b)\langle x, y \cap b\rangle$$

Similarly, we define the **iterator** and the **bounded iterator** loop and bloop, which iterates some function $|x|$ times, instead of $\lceil \log(|x| + 1)\rceil$ times:

$$\frac{f : t \to t}{\mathsf{loop}(f) : \{t'\} \times t \to t} \quad \frac{f : t \to t \quad b : t}{\mathsf{bloop}(f, b) : \{t'\} \times t \to t}$$

with the semantics:

$$\begin{aligned} \mathsf{loop}(f)\langle x, y\rangle &\stackrel{\text{def}}{=} f^{|x|}(y) \\ \mathsf{bloop}(f, b)\langle x, y\rangle &\stackrel{\text{def}}{=} \mathsf{loop}(f \cap b)\langle x, y \cap b\rangle \end{aligned}$$

We extend the definition of depth of recursion nesting to *depth of iteration nesting* for these construct, by defining $depth(\mathsf{log\_loop}(f)(e)) \stackrel{\text{def}}{=} \max(1 + depth(f), depth(e))$, etc.

Both log_loop and loop are powerful enough to express *powerset*. Hence, we will only consider the unbounded versions in conjunction with flat relations, and use their bounded versions for complex objects.

Example 3.8.1 log_loop *can express transitive closure,* $tc : \{t \times t\} \to \{t \times t\}$. *Indeed, let* $r \in \{t \times t\}$ *be some relation. First compute* $v = \Pi_1(r) \cup \Pi_2(r)$ *(the set of all elements mentioned in* r*), then, repeat* $\lceil \log(n+1) \rceil$ *times* $r \leftarrow r \cup r \circ r$*, where* $n \stackrel{\text{def}}{=} |v|$*, and* $\circ$ *is relation composition. That is* $TC(r) = \mathsf{log\_loop}(\lambda r.r \cup r \circ r)(v, r)$.

Example 3.8.2 *Let* $n = card(x)$*. Then* $\mathsf{loop}(f)$ *and* $\mathsf{log\_loop}(f)$ *allow us to iterate* n *and* $\log n$ *times respectively. To iterate* n^2 *times, it suffices to* loop *over* $x \times x$*, which has* n^2 *elements:* $f^{(n^2)}(y) = \mathsf{loop}(f)\langle x \times x, y\rangle$*. To iterate* $\log^2 n$ *times, we use two levels of iterations:* $f^{(\lceil \log(n+1) \rceil^2)}(y) = \mathsf{log\_loop}(\lambda z.\mathsf{log\_loop}(f)\langle x, z\rangle)\langle x, y\rangle$.

Immerman defines $FO(t(n))$ in [65] to be first order logic, with order and with a binary relation BIT, extended with those inductive definitions which close after $t(n)$ steps. $\mathcal{NRC}^1(\mathsf{log\_loop}, \leq, BIT)$ and $\mathcal{NRC}^1(\mathsf{loop}, \leq, BIT)$ have essentially the same expressive power as $FO[\log^{O(1)} n]$ and $FO[n^{O(1)}]$ respectively. However, without order, these two are no longer equivalent: loop can express parity, while $FO[n^{O(1)}]$ (without order and BIT) is included in $FO + LFP$, and hence it cannot express parity. Similarly, we can argue that $FO[\log^{O(1)} n]$ without order is less powerful than $\mathcal{NRC}^1(\mathsf{log\_loop})$.

The key technical lemma in proving the main results states that dcr and log_loop have the same expressive power over ordered databases. Expressions with dcr can be translated into expressions with log_loop and conversely. Moreover, this translation preserves the depth of iteration nesting.

Proposition 3.8.3 *For every* $k \geq 0$, $\mathcal{NRC}(\mathsf{blog\_loop}^{(k)}, \leq) = \mathcal{NRC}(\mathsf{bdcr}^{(k)}, \leq)$.

From this, Proposition 3.4.8, and the analogous property:

$$\mathcal{NRC}^1(\mathsf{blog\_loop}^{(k)}) = \mathcal{NRC}^1(\mathsf{log\_loop}^{(k)})$$

proved in a similar manner, we get:

Corollary 3.8.4 *For every* $k \geq 0$, $\mathcal{NRC}^1(\mathsf{log\_loop}^{(k)}, \leq) = \mathcal{NRC}^1(\mathsf{dcr}^{(k)}, \leq)$

Proof. (of Proposition 3.8.3) We start by proving

$$\mathcal{NRC}(\mathsf{bdcr}^{(k)}, \leq) \subseteq \mathcal{NRC}(\mathsf{blog\_loop}^{(k)}, \leq)$$

Consider some function $\varphi = \mathsf{bdcr}(e, f, u, b)$, $\varphi : \{t\} \to t'$. Let $x = \{x_1, \ldots, x_n\} \in \{t\}$. We will use the encoding $\tilde{k}$ described in Subsection 3.4.4. Define g to be the function $\lambda\tilde{k}.\widetilde{k+k}$ (see the techniques used in the proof of Proposition 3.4.13). Then the sequence $\tilde{1}, g(\tilde{1}), g(g(\tilde{1})), g(g(g(\tilde{1}))), \ldots$ is $\tilde{1}, \tilde{2}, \tilde{4}, \tilde{8}, \ldots$, hence it suffices to iterate g $\lceil \log(n+1) \rceil$ times, and to apply it to $\tilde{1}$, to get: $\mathsf{log\_loop}(g)\langle x, \tilde{1}\rangle = \tilde{n}$. Using the techniques from the proof of Proposition 3.4.13, given a bound b for $\mathsf{bdcr}(e, f, u, b)$, we can compute a bound b' for $\tilde{k}$, for all $k \leq n$, hence $\mathsf{log\_loop}(g)\langle x, \tilde{1}\rangle$ is equivalent to $\mathsf{blog\_loop}(g, b')\langle x, \tilde{1}\rangle$. Finally we obtain $\varphi(x) = \mathit{extract}\langle \tilde{n}, \langle x_1, x_n\rangle\rangle$.

Now we prove $\mathcal{NRC}(\mathsf{blog\_loop}^{(k)}, \leq) \subseteq \mathcal{NRC}(\mathsf{bdcr}^{(k)}, \leq)$. Consider some $\mathsf{blog\_loop}(f, b)\langle x, y\rangle$ expression; we can express it by divide and conquer recursion on the set x. The base cases are trivial:

$$\begin{aligned}
\mathsf{blog_loop}(f, b)\langle \emptyset, y\rangle &= y \\
\mathsf{blog_loop}(f, b)\langle \{z\}, y\rangle &= f(y)
\end{aligned}$$

For $\mathsf{log\_loop}(f)\langle s \cup s', y\rangle$, where $s \cap s' = \emptyset$, assume $|s| \geq |s'|$ (the case $|s| < |s'|$ is treated similarly). Then the binary representation of the number $|s \cup s'| = |s| + |s'|$ has either the same number of bits as the binary representation of $|s|$, or one bit more. So:

$$\mathsf{log\_loop}(f)\langle s \cup s', y\rangle = \begin{cases} f(\mathsf{log\_loop}\langle s, y\rangle) \text{ or} \\ \mathsf{log\_loop}\langle s, y\rangle \end{cases}$$

Namely $\mathsf{log\_loop}(f)\langle s \cup s', y\rangle = f(\mathsf{log\_loop}\langle s, y\rangle)$ if the binary representation of the number $|s \cup s'|$ has one more bit than the binary representation of the number $|s|$, and $\mathsf{log\_loop}(f)\langle s \cup s', y\rangle = \mathsf{log\_loop}\langle s, y\rangle$ otherwise. Now we explain the details. We start by constructing an ordered set $z = \{z_0, z_1, \ldots, z_{2n}\}$ having (at least) $2n+1$ values $z_0 < z_1 < \ldots < z_{2n}$, which we use as numbers. Assuming $x \neq \emptyset$, it suffices to take $z = (\{\mathsf{false}, \mathsf{true}\} \times \{\mathsf{false}, \mathsf{true}\}) \times x$, which has $4n > 2n+1$ elements, and treat the case $x = \emptyset$ separately[5]. Then we define the function $\varphi(s) \stackrel{\text{def}}{=} \langle z_k, \langle z_p, f^{(\lceil \log(k+1) \rceil)}(y)\rangle\rangle$, where $k = \min(|s|, n)$ and $p = 2^{\lceil \log(k+1) \rceil}$. I.e., $\varphi(s)$ returns: (1) k, the cardinality of s, (2) p, the next power of 2, and (3) $\mathsf{log\_loop}(f)\langle s, y\rangle$. Now we argue that φ can be expressed with dcr. Indeed, consider $s \cap s' = \emptyset$, and let:

$$\begin{aligned} \varphi(s) &= \langle z_k, \langle z_p, v\rangle\rangle \\ \varphi(s') &= \langle z_{k'}, \langle z_{p'}, v'\rangle\rangle \end{aligned}$$

where $k = |s|, k' = |s'|$, p, p' are the next powers of 2, and $v = f^{(\lceil \log(k+1) \rceil)}(y), v' = f^{(\lceil \log(k'+1) \rceil)}(y)$. Assume without loss of generality that $k \geq k'$ (and, hence, $p \geq p'$). Then we have $\varphi(s \cup s') = \langle z_{k''}, \langle z_{p''}, v''\rangle\rangle$, where k'', p'', v'' are computed as follows:

$$\begin{aligned} k'' &\stackrel{\text{def}}{=} k + k' \\ p'' &\stackrel{\text{def}}{=} \mathsf{if}\ k'' < p\ \mathsf{then}\ p\ \mathsf{else}\ p + p \\ v'' &\stackrel{\text{def}}{=} \mathsf{if}\ k'' < p\ \mathsf{then}\ v\ \mathsf{else}\ f(v) \end{aligned}$$

Following the techniques of [64], we compute addition on $z_0, \ldots, z_{2n}$ with the help of transitive closure. Namely we define the binary relation $r = \{\langle\langle z_i, z_j\rangle, \langle z_{i+1}, z_{j+1}\rangle\rangle \mid 0 \leq i, j < 2n\}$ on $z \times z$, and compute its transitive closure $q = TC(r)$. Then, given z_i and z_j, z_{i+j} is the unique element for which $\langle\langle z_0, z_i\rangle, \langle z_j, z_{i+j}\rangle\rangle \in q$. To avoid increasing the nesting depth of dcr we precompute an addition table: $a = \{\langle\langle z_i, z_j\rangle, z_{i+j}\rangle \mid 0 \leq i, j \leq 2n, i + j \leq 2n\}$ in $\mathcal{NRC}(\mathsf{bdcr}^{(1)}, \leq)$, and then compute addition by looking up the table. $\Box$

The proof of Proposition 3.8.3 has an important consequence. Recall that the conditions for well-definedness of dcr are not r.e. hence the language $\mathcal{NRC}^1(\mathsf{dcr}, \leq)$ is not r.e. But,

[5] The presence of the two constants false and true is not essential here. We could have taken $z = x \times (x \times x)$ which has $n^3 > 2n + 1$ elements for $n > 1$. Then we need to treat the cases $x = \emptyset$ and $|x| = 1$ separately.

by restricting it to the instances of dcr used in the simulation of log_loop we obtain an r.e., in fact decidable, sublanguage $\mathcal{L}$ which has the same expressive power as the whole $\mathcal{NRC}^1(\mathsf{dcr}, \leq)$.

3.8.2 Circuits

In order to prove that $\mathcal{NRC}(\mathsf{blog\_loop}) \subseteq AC$, we first establish some technical lemmas. For some string $W \in \{0,1\}^n$ of length n and numbers i, j, $0 \leq i \leq j \leq n-1$, we denote with $W[i : j]$ the substring of W containing all elements from positions i to j inclusive. Recall that we encode each symbol of the alphabet $A = \{0, 1, \{, \}, \langle, \rangle, comma, blank\}$ with a string of length 3. Thus if X is an encoding of the complex object x, then for every i for which $i \ mod \ 3 = 0$, $X[i : i+2]$ will be the binary encoding of some character in A.

Lemma 3.8.5 *For every $d \geq 0$ there exists a function $F^d = \bigcup_{n \geq 0} F_n^d$ in AC^0, $F_n^d : \{0,1\}^n \to \{0,1\}^{n^2}$, finding the "matching parentheses" of nesting depth $\leq d$.*

More precisely, for every string $X \in \{0,1\}^n$ and for every i, j, $0 \leq i, j \leq n-1$, $F_n^d(X)$ will have a 1 on position $i \cdot n + j$ iff $i mod 3 = j mod 3 = 0$, and there is a left parenthesis $\{$ on positions $i, i+1, i+2$, and a matching right parenthesis $\}$ on positions $j, j+1, j+2$ (or similarly, for a pair of matching angle parentheses $\langle$ and $\rangle$ on these positions), and the nesting depth of parentheses enclosed between the positions i and j is $< d$. E.g. consider the string $X =$ "$\}_0\{_1\{_2\}_3\{_4\{_5\}_6\{_7\}_8\}_9\}_{10}\{_{11}$" which, after encoding each parenthesis with three bits, will be translated into a string $X \in \{0,1\}^{36}$. Then $F_{36}^2(X)$ reports the matching parentheses $\{_2, \}_3$, $\{_5, \}_6$, $\{_7, \}_8$, and $\{_4, \}_9$ (i.e. it will output an 1 on the positions $36i + j$, for $i, j = \langle 6, 9\rangle, \langle 15, 18\rangle, \langle 21, 24\rangle$, and $\langle 12, 27\rangle$). It will not report the matching parentheses $\{_1, \}_{10}$, because their nesting level is 3.

Proof. (of Lemma 3.8.5) We prove the statement by induction on d. For $d = 0$ there is nothing to check: $F_n^0(X)$ just returns the string $000\ldots0$. For $d = 1$, $F_n^1(X)$ returns an 1 on those positions $i \cdot n + j$ for which: both i and j are divisible by 3, $i < j$, $X[i : i+2]$ and $X[j : j+2]$ contain $\{$ and $\}$ respectively (or $\langle$ and $\rangle$), and $\forall k$, $k \ mod \ 3 = 0$, $i < k < j$, there is no parenthesis in $X[k : k+2]$.

For $d > 1$ we proceed as follows. First we compute $Y = F_n^{d-1}(X)$. Next replace in X every character corresponding to a matched parenthesis in Y with *blank*, and call X' the result, i.e. $\forall i.0 \leq i \leq n-1$, if $i \ mod \ 3 = 0$, then, if $\exists j$ s.t. $Y[i \cdot n + j] = 1$ or $Y[j \cdot n + i] = 1$, then $X'[i : i+2] \stackrel{\text{def}}{=} blank$, else $X'[i : i+2] \stackrel{\text{def}}{=} X[i : i+2]$. Compute $Y' = F_n^1(X')$, and finally output Z, where $Z[i] \stackrel{\text{def}}{=} AND(Y[i], Y'[i])$, for $0 \leq i \leq n^2 - 1$.

It is easy to check that, for $d \geq 1$, a circuit α_n^d can be built for F_n^d, for which $height(\alpha_n^d) \leq 8d - 4$, and $size(\alpha_n^d) \leq d \cdot (2n^3 + 3n^2 + 6n) - 5n$. □

Corollary 3.8.6 *For each type t, there is some function $F^t = \bigcup_{n \geq 0} F_n^t$ in AC^0, $F_n^t : \{0,1\}^n \to \{0,1\}^{n^2}$ which identifies the pairs of parenthesis for any encoding of type t. More precisely for any encoding X of an object x of type t, $F_n^t(X)$ will have an 1 on position $i \cdot n + j$ iff there is a left parenthesis $\{$ on positions $i, i+1, i+2$, and a matching right parenthesis $\}$ on positions $j, j+1, j+2$, or similarly for a pair of matching angle parentheses $\langle\rangle$.*

Proof. This follows directly from Lemma 3.8.5, since any encoding X of an object of type x will have at most d nested parentheses, where d depends only on the type t. □

Lemma 3.8.7 *For any set type $\{t\}$, there is some function $B^{\{t\}} = \bigcup_{n \geq 0} B_n^{\{t\}}$ in AC^0, $B_n^{\{t\}} : \{0,1\}^n \to \{0,1\}^n$, which, for some encoding $\{X_1, \ldots, X_m\}$ of type $\{t\}$, returns a string containing exactly m 1's, namely on those positions where some X_i begins. Similarly, for every product type $t_1 \times \ldots \times t_m$, there exists a function $B^{t_1 \times \ldots \times t_m}$ in AC^0 which, when given an encoding $\langle X_1, \ldots, X_m \rangle$ of length n of $\langle x_1, \ldots, x_m \rangle \in t_1 \times \ldots \times t_m$, returns a string of length n containing exactly m 1's, namely on the positions where $X_1, \ldots, X_m$ start.*

Proof. The circuit computing $B_n^{\{t\}}$ identifies the outermost commas (i.e. those not included in any pair of matching parenthesis, except the outermost $\{ \}$), and returns a 1 on each first non-blank position following such a comma, or following the leading left brace. □

As a consequence, we have:

Lemma 3.8.8 *For all types t, equality of objects of type t is computable in AC^0.*

Proof. We construct a family of circuits $E_n^t : \{0,1\}^{2n} \to \{0,1\}$ which, when given two encodings X, Y of $x, y \in t$, both of length n, returns 1 iff $x = y$. We proceed by induction on the type t. For the base case $t = \mathbb{D}$, we use essentially the fact that blanks are not allowed inside the binary representation of numbers. Then $E_n^{\mathbb{D}}(XY)$ will return 1 iff $\exists i, i', j, j'$, $0 \leq i < i' \leq n-1, 0 \leq j < j' \leq n-1$, $X[i : i'] = Y[j : j']$ (in particular $i' - i = j' - j$), and in $X[0 : i-1], X[i'+1 : n-1], Y[0 : j-1], Y[j'+1 : n-1]$ there are only blanks. For the induction case $t = \{t'\}$, we have $X = \{X_1, \ldots, X_p\}$ and $Y = \{Y_1, \ldots, Y_q\}$, with possible blanks scattered. We essentially test the following two conditions:

$$\forall i \leq p.\exists j \leq q.X_i = Y_j \tag{3.2}$$

$$\forall j \leq q.\exists i \leq p.X_i = Y_j \tag{3.3}$$

To do that, we start by using Lemma 3.8.7 to compute the stings U, V of length n, which identify the p positions in X and the q positions in Y where some element starts. Then condition 3.2 becomes:

$$\begin{aligned}\forall i, i' \quad . \quad & (i < i' \land U[i] = U[i'] = 1 \land \forall i'' i < i'' < i', U[i''] = 0) \Rightarrow \\ & \exists j, j'.j < j' \land V[j] = V[j'] = 1 \land (\forall j''.j < j'' < j' \Rightarrow V[j''] = 0) \land \\ & E_n^{t'}(U[i : i'-1]V[j : j'-1]) = 1\end{aligned}$$

In fact the expression $E_n^{t'}(U[i : i'-1]V[j : j'-1])$ is not correct; we have to eliminate the trailing *comma* (i.e. replace it with *blank*) in $U[i : i'-1]$ and $V[j : j'-1]$, and add add $n - i' + i - 1$ trailing blanks to $U[i : i'-1]$ and $V[j : j'-1]$ (to make them of length n) before computing $E_n^{t'}(U[i : i'-1]V[j : j'-1])$. The details are straightforward and are left out. Condition 3.3 is handled similarly. □

Lemma 3.8.9 *For every set type $\{t\}$ there exists a "duplicate elimination function" $D^t = \bigcup D_n^t$, where $D_n^t : \{0,1\}^n \to \{0,1\}^n$, with the following meaning. Given any encoding*

$X = \{X_1, \ldots, X_m\}$ of some object of type $\{t\}$, with possible duplicates, $D^t(X)$ returns a string of the same length, in which all duplicates are replaced with blanks.

Proof. $D^t_n(X)$ will work as follows. First compute $Y = B^t_n(X)$, i.e. identify where each of the m substrings $X_1, \ldots, X_m$ begins in X. Now we have to replace each character $X[i : i+2]$ belonging to some duplicate X_p, with $0 \le i \le n-1$, $i \bmod 3 = 0$, with a *blank*. More precisely we replace $X[i : i+2]$ with *blank* iff the following holds:

$$\begin{aligned}\exists j \le i.\exists k > i \quad . \quad & Y[j] = Y[k] = 1 \wedge (\forall l.j < l < k \Rightarrow Y[l] = 0) \wedge \\ & \exists j', k'.k \le j' < k' \wedge Y[i'] = Y[k] = 1 \wedge (\forall l.j' < l < k' \Rightarrow Y[l] = 0) \wedge \\ & E^t_n(X[j : k-1]X[j' : k'-1])\end{aligned}$$

The condition says that $X[i : i+2]$ lies inside some string X_p stretching between positions j and k, and that X_p equals X_q, with $p < q$, where X_q stretches between the positions j' and k'. The expression $E^t_n(X[j : k-1]X[j' : k'-1])$ is not quite correct: the possible trailing *comma*s have to be eliminated (replaced with *blank*) in both $X[j : k-1]$ and $X[j' : k'-1]$, and then they have to be padded with *blank*s, to make them strings of length n. We omit the details. $\Box$

Proposition 3.8.10

$$\mathcal{NRC}(\mathsf{blog\_loop}^{(k)}) \subseteq \mathit{CMPX\text{-}OBJ\text{-}AC}^k \quad \textit{for all } k \ge 0$$

Hence, $\mathcal{NRC}^1(\mathsf{log\_loop}^{(k)}) \subseteq \mathit{FLAT\text{-}AC}^k$.

Proof. We prove by induction on some complex object expression $e \in \mathcal{NRC}(\mathsf{blog\_loop}^{(k)})$ of type t that for any set of variables $x_1, \ldots, x_l$ which includes all free variables in e, the associated query $\lambda\langle x_1, \ldots, x_l\rangle.e$ (see Section 3.2) is in AC^k. We illustrate only some of the cases for e.

Union $e \cup e'$. Let f and f' be the queries associated to e and e' respectively. Then the query associated to $e \cup e'$ is $g = \lambda x.f(x) \cup f'(x)$. Let f, f' be computed by the functions F and F' respectively, and let α_n and α'_n be circuits associated to F_n and F'_n. To compute g concatenate the outputs of α_n and α'_n, eliminate the braces }{ by replacing them with blanks and conditionally placing a comma (the comma is placed only when both outputs encode a nonempty set). Finally, eliminate the duplicates in the resulting set using Lemma 3.8.9.

Function application $f(e)$, where $f : t \to t'$ and $e : t$. Let $g : t \times t_1 \times t_2 \times \ldots \times t_l \to t'$ be the query corresponding to f, and $h : t_1 \times t_2 \times \ldots \times t_l \to t$ be the query corresponding to e. Then the query corresponding to $f(e)$ is $k = \lambda x.g\langle h(x), x\rangle$, and this leads us directly to a circuit. Namely let α_n be the circuit computing g and β_n be the circuit computing h; let $Q(n)$ be the size of the output of β_n. Then the circuit computing k will consists in a copy of β_n, and a copy of $\alpha_{n+Q(n)+9}$. The inputs of $\alpha_{n+Q(n)+9}$ will consists in the concatenation of the following strings:

$$\text{“}\langle\text{''} \text{ the output of } \beta_n \text{ “,” the input } X \text{“}\rangle\text{''}$$

Extension $\mathsf{ext}(f)$. For sake of clarity, suppose that $f : t \to \{t'\}$ does not have free variables, and that $l = 0$. Then the query associated to f is f itself, and let α_n be a circuit for computing f; let $Q(n)$ be the size of the output of α_n. The circuit β_n for $\mathsf{ext}(f)$ will receive an input $X = \{X_1, X_2, \ldots, X_m\}$. Essentially it has to “apply” α_n to each substring X_p. It starts by computing $Y = B^t_n(X)$. Since it cannot anticipate where each substring X_p lies, it will have a copy of the circuit α_{j-i} for every pair $\langle i, j\rangle$, $0 \leq i < j \leq n-1$ for which $i \ mod \ 3 = j \ mod \ 3 = 0$, which will receive as input $X[i : j-1]$. The output of the circuit corresponding to $\langle i, j\rangle$ will be invalidated however (overwritten with *blanks*), unless $Y[i] = Y[j] = 1$ and $\forall k, i < k < j, Y[k] = 0$, i.e. unless there exists indeed some substring X_p which stretches from position i to position $j-1$. All $n(n+1)/2$ results are concatenated, all pairs the inner parentheses }{ replaced with a *comma*, and finally we feed the resulting string (of size $\leq Q(n) \cdot n \cdot (n+1)/2$) into $E^t_{Q(n)\cdot n\cdot(n+1)/2}$ to eliminate the duplicates.

Iteration $\mathsf{blog\_loop}(f, b)$. Since the output type is a PS-types, we will assume, for sake of clarity, it is a set type, i.e. $f : \{t\} \to \{t\}$. Let $g : \{t\} \times (t_1 \times \ldots) \to \{t\}$ be the query associated to f, and $h : (t_1 \times \ldots) \rightarrow \{t\}$ be the query associate to b. Let α_n and β_n be the circuits for computing g and h respectively, and let G_n, H_n be the functions computed by them. Assume their output sizes to be $Q_g(n)$ and $Q_h(n)$ respectively. The input set y in $f\langle x, y\rangle$ cannot have more than n elements, (since at most $n/3$ characters are used to encode $\langle x, y\rangle$), hence it would suffice to generate $\lceil \log n \rceil$ copies of α_n, and to feed the output of every circuit as input to the next. However this naive approach does not work. Indeed, the output of the first α_n has size $Q_g(n)$, then the output of the second $\alpha_{Q_g(n)}$ has size $Q_g(Q_g(n))$, the output of the third $\alpha_{Q_g(Q_g(n))}$ has size $Q_g(Q_g(Q_g(n)))$, etc. Hence the output size grows more than a polynomial. We need a more subtle technique which uses the bound b in an essential way.

Let $p : (\{t'\} \times \{t\}) \times (t_1 \times \ldots) \rightarrow \{t\}$ be the query associated to $\mathsf{blog\_loop}(f, b)$. The circuit computing p will receive as input a string $\langle\langle X, Y\rangle, Z\rangle$ of length n. It will start by computing H_n on the string $\langle Y, Z\rangle$ of length n (obtained from the input by overriding some characters with blanks). Let $U = H_n\langle Y, Z\rangle$. U is the encoding of the bounding set, $u = \{u_1, \ldots, u_l\}$, and $\mathsf{length}(U) = Q_h(n)$. Next, the circuit for p contains $\lceil \log(n+1) \rceil$ copies of the circuit $\alpha_{Q_h(n)}$. The idea is that all intermediate results are subsets of u, hence $Q_h(n)$ bits will suffice to represent them. Namely we feed the first circuit $\alpha_{Q_h(n)}$ with the intersection of Y and U; we obtain Y from the input $\langle\langle X, Y\rangle, Z\rangle$ by overriding the rest with blanks, and we compute the intersection by overriding in U all elements which are not in Y with blanks, so the result has the same size as U. Similarly, we feed the $i+1$ copy of the circuit $\alpha_{Q_h(n)}$ with the output of the i's copy, intersected with U, which is again a string of size $Q_h(n)$. Of course, we have to bypass all circuits $\alpha_{Q_h(n)}$ beyond level $\lceil \log(m+1) \rceil$, where m is the cardinality of the set encoded by X. To do that, we start by computing $B_n^{t'}(X)$. Next we compute a string of length $\lceil \log(n+1) \rceil$ representing the binary encoding of the number of 1's in $B_n^{t'}(X)$, i.e. the binary encoding of m: this can be done in AC^1, since it is a particular case of the problem of adding n binary numbers. Now we establish a correspondence between the $\lceil \log(n+1) \rceil$ bits representing m and the $\lceil \log(n+1) \rceil$ copies of $\alpha_{Q_h(n)}$: the least significant bit will correspond to the first copy, etc. We will bypass copy $\alpha_{Q_h(n)}$ all bits $i, i+1, i+2, \ldots$ in m are 0.

Finally, observe that, if the circuit $\alpha_{Q(n)}$ for computing g had depth $O(\log^k n)$, then the circuit for computing p has depth $O(\log^{k+1} n)$.

The other cases are treated in a similar fashion. We skip the proof of the uniformity, which is tedious but straightforward. □

Since the inequality $\leq$ can be computed in AC^0, Proposition 3.8.10 immediately implies:

Corollary 3.8.11 *$\mathcal{NRC}(\mathsf{blog\_loop}^{(k)}, \leq) \subseteq$ CMPX-OBJ-AC^k and $\mathcal{NRC}^1(\mathsf{log\_loop}^{(k)}, \leq) \subseteq$ FLAT-AC^k*

Instead of designing a circuit for computing f, we could have shown that f can be computed in $FO[\log^k n]+ \leq +BIT$, and then using the results in [65, 11] to conclude $f \in AC^k$: in fact, this is the way we prove Proposition 3.7.5. But we chose to construct the circuit for computing f in order to suggest that how f may compiled on a CRCW PRAM.

To prove the converse inclusion we adopt techniques for simulating $PTIME$ Turing Machines with query languages over finite, ordered structures [62, 104], and over complex objects [55]. A central step consists in taking the (ordered) input x and constructing some set z whose cardinality is at least as large as the size of the minimal encoding of x. The type of z may depend on the value x: but there are a fixed number of choices, and we may test (using if's) which one is the right choice. E.g., if $t = \{\mathbb{D}\}$, then $x = \{x_1, \ldots x_m\}$, where $m = |x|$. Then x can be encoded with $n \leq 6 * m^2 + 3$ bits, since each element in x can be encoded with at most $\lceil \log(m+1) \rceil \leq m$ characters 0,1, each is followed by one additional comma (and the last element is followed by the right brace), and each character uses, in fact, 3 bits. Finally we need the left brace (hence the $\ldots + 3$). For this case, it suffices to take $z \stackrel{\text{def}}{=} x \times (x \times x)$, of type $\{\mathbb{D} \times (\mathbb{D} \times \mathbb{D})\}$ when $m > 2$, because $|z| = m^3 \geq 6 * m^2 + 3$ for $m \geq 3$. But for $m \leq 2$, we choose $z \stackrel{\text{def}}{=} \{\mathsf{true}, \mathsf{false}\}^5$ of type $\{\mathbb{B}^5\}$, which has $32 > 6 * m^2 + 3$ elements.

Lemma 3.8.12 *For every type t there exists n_t conditions $C^t_1, \ldots, C^t_{n_t} : t \to \mathbb{B}$, n_t types $s^t_1, \ldots, s^t_{n_t}$, and n_t functions $b^t_1 : t \to \{s^t_1\}, \ldots, b^t_{n_t} : t \to \{s^t_{n_t}\}$ in $\mathcal{NRC}$, such that the following hold. Let $x \in t$, and let $X \in \{0,1\}^*$ be the minimal encoding of x. Then:*

- *$\exists i$ such that $C_i^t(x) = \mathsf{true}$.*
- *$C_i^t(x) = \mathsf{true} \Rightarrow \mathsf{length}(X) \leq |b_i^t(x)|$.*

Moreover, $height(\{s_i^t\}) \leq \max(1, height(t))$, for $i = 1, n_t$.

Proof. We proof the lemma by induction on the type t. We will abbreviate the set $\{\mathsf{false}, \mathsf{true}\} \times \{\mathsf{false}, \mathsf{true}\}$ with 4.

Case $t = \mathbb{D}$ Then we take $n_t = 1$, $C_1^t(x) \stackrel{\text{def}}{=} \mathsf{true}$, $s_1^t \stackrel{\text{def}}{=} \mathbb{B} \times \mathbb{B}$, and $b_1^t(x) \stackrel{\text{def}}{=} 4$. Indeed, $\forall x \in \mathbb{D}$, the minimal encoding of x is 0, which has length 3 (recall that we encode each symbol of the alphabet A with three bits).

Case $t = \{t'\}$ Let $x \in t$, $x = \{x_1, \ldots, x_n\}$, and let $a \stackrel{\text{def}}{=} atoms(x)$, $a \in \{\mathbb{D}\}$. We first apply induction hypothesis to t', to get conditions $C_1^{t'}, \ldots, C_{n_{t'}}^{t'}$, and types $s_1^{t'}, \ldots, s_{n_{t'}}^{t'}$. Next observe that the standard encoding of X is $\{X_1, \ldots, X_n\}$. Obviously $X_1, \ldots, X_n$ are *not* standard encodings of $x_1, \ldots, x_n$ respectively (because x_j does not necessarily mention all atoms in a), but their length is at most a factor of $3 \cdot \lceil \log(|a| + 1) \rceil$ larger than the length of their minimal encoding. Observe that $3 \cdot \lceil \log(|a| + 1) \rceil \leq 4 \times |a|$, when $a \neq \emptyset$. Hence, to obtain a set of cardinality $\geq \mathsf{length}(X_j)$ when $C_i^{t'}(x_j)$ is true and $a \neq \emptyset$, it suffices to take the set $4 \times a \times b_i^{t'}(x_j)$. The latter set is included in $4 \times a \times b_i^{t'}(x)$. However, when $a = \emptyset$, then the size of X is bounded by a constant which depends only on the type t, and we take as bounding set $\{\mathsf{false}, \mathsf{true}\}^k$, with k sufficiently large. In conclusion we define $n_t \stackrel{\text{def}}{=} n_{t'} + 1$, and:

$$
\begin{aligned}
C_i^t(x) &\stackrel{\text{def}}{=} \exists x_j \in x.C(x) \wedge a \neq \emptyset \\
s_i^t &\stackrel{\text{def}}{=} \mathbb{B}^2 \times \mathbb{D} \times s_i^{t'} \\
b_i^t(x) &\stackrel{\text{def}}{=} 4 \times a \times \mathsf{ext}(b_i^{t'})(x)
\end{aligned}
$$

for $i = 1, n_{t'}$, and

$$
C_{n_{t'}+1}^t(x) \stackrel{\text{def}}{=} a = \emptyset
$$

$$\begin{aligned}
s^t_{n_{t'}+1} &\stackrel{\text{def}}{=} \mathbb{B}^k \\
b^t_{n_{t'}+1}(x) &\stackrel{\text{def}}{=} \{\mathsf{false}, \mathsf{true}\}^k
\end{aligned}$$

The other cases are handled using similar techniques. □

Proposition 3.8.13 *For all $k \geq 1$ we have:*

$$\begin{aligned}
\mathit{CMPX\text{-}OBJ\text{-}AC}^k &\subseteq \mathcal{NRC}(\mathsf{blog_loop}^{(k)}, \leq) \\
\mathit{FLAT\text{-}AC}^k &\subseteq \mathcal{NRC}^1(\mathsf{log_loop}^{(k)}, \leq)
\end{aligned}$$

Proof. Let f be in $\mathit{CMPX\text{-}OBJ\text{-}AC}^k$, $f : t \to t'$. We shall construct an expression f' in $\mathcal{NRC}(\mathsf{blog\_loop}^{(k)}, \leq)$ equivalent to f. This proves $\mathit{CMPX\text{-}OBJ\text{-}AC}^k \subseteq \mathcal{NRC}(\mathsf{blog\_loop}^{(k)}, \leq)$. For $\mathit{FLAT\text{-}AC}^k \subseteq \mathcal{NRC}^1(\mathsf{log\_loop}^{(k)}, \leq)$ it will suffice to observe that, when $\mathit{height}(t) \leq 1$ and $\mathit{height}(t') \leq 1$, then $f' \in \mathcal{NRC}^1(\mathsf{blog\_loop}^{(k)}, \leq)$ and, hence, $f' \in \mathcal{NRC}^1(\mathsf{log\_loop}^{(k)}, \leq)$, by a simple extension of Proposition 3.4.8 to $\mathsf{blog\_loop}$.

Since $f \in \mathit{CMPX\text{-}OBJ\text{-}AC}^k$ it is computed by some function $F : \{0,1\}^* \to \{0,1\}^*$. F is given by: (1) A *DLOGSPACE* Turing Machine T accepting the *DCL* of a family of circuits, and (2) Polynomials $P(n)$ and $Q(n)$ (see Section 3.5). For some input $x \in t$, let n be the length of the *minimal encoding* X of x (see Section 3.6). The simulation of F in $\mathcal{NRC}^k(\mathsf{blog\_loop}, \leq)$ is described below.

1. Apply Lemma 3.8.12 to construct some set z having a cardinality $\geq n$. The type of z may depend on the value x: but there are a fixed number of choices, and we may test (using if's) which one is the right choice. Observe that when $\mathit{height}(t) \leq 1$ then the type of z will be flat and z be computed in $\mathcal{NRC}^1$.

2. Some power z^l of z will have $p = n^l$ elements, enough to perform all the arithmetic needed in the sequel. Over this ordered set, we pre-compute the functions plus, minus, multiplication, and bit, on the numbers $0, \ldots, p-1$. See e.g. the proof of Proposition 3.8.3 for the computation of addition, and see Example 3.7.4 for a hint on how to compute multiplication using bdcr and, hence, $\mathsf{blog\_loop}$.

3. Compute the minimal encoding X of x, of length n, without blanks: a string $X \in \{0,1\}^*$ of length n is represented as a set of "numbers", i.e. a subset of z^l. The computation is done in $\mathcal{NRC}^1(\mathsf{blog\_loop}, \leq)$, the $\mathsf{blog\_loop}$ being needed to compute the sum of a set of numbers. As a byproduct of this encoding, we also obtain a translation table from atomic values in x to numbers.

4. Simulate F on X, as described below, to get $Y = F(X)$.

5. Finally "decode" Y, to get $y \in t$. Decoding is done in $\mathcal{NRC}$, i.e. no loops are necessary, and uses the translation table constructed at point 3. The decoding consists essentially in parsing the string Y and constructing y. The parser depends on the type t. E.g., when $t = \{t'\}$, then we construct y as follows. We built the set of pairs of number (i,j), $i < j$ (this is just a subset of $z^l \times z^l$), and for each such pair, we use the parser for the type t' to test whether some element is encoded in Y between the positions i and j, and if so, to decode that element. Finally we construct the set of all such elements. So essentially we apply ext on the parser for the type t'.

There are two ways of simulating F on X. One is to use the result in [65] which says that, since F is in AC^k, F is also in $FO[\log^k n] + \leq +BIT$, and to observe that, for $k \geq 1$, $FO[\log^k n] + \leq +BIT \subseteq \mathcal{NRC}^1(\mathsf{log\_loop}^{(k)}, \leq)$. The second way is to use the *DLOGSPACE-DCL*-uniformity definition of AC^k. We will follow the second path here. First, we simulate the $O(\log n)$ space Turing Machine computing the DCL of α_n: this can be done, since there are only polynomially many configurations for T, and deciding whether T accepts some input (n, g, g', t) reduces, using standard techniques, to the computation of the transitive closure of the successor relation on the set of configurations of T. Secondly, we simulate the circuit α_n itself, by computing step by step the outputs of the gates at each level: this only requires $\log^k n$ iterations, so it can be done in $\mathcal{NRC}^1(\mathsf{log\_loop}^{(k)}, \leq)$ and, hence, in $\mathcal{NRC}(\mathsf{blog\_loop}^{(k)}, \leq)$. We observe that ext is used in an essential way at each iteration step, accounting for the parallelism in the evaluation of α_n.

This proved

$$\textit{CMPX-OBJ-AC}^k \subseteq \mathcal{NRC}(\mathsf{blog\_loop}^{(k)}, \leq), \quad \forall k \geq 1$$

If t and t' are both flat types, then all the computations describe above can be expressed in $\mathcal{NRC}^1(\mathsf{blog\_loop}^{(k)}, \leq)$, which is equal to $\mathcal{NRC}^1(\mathsf{log\_loop}^{(k)}, \leq)$, by a straightforward extension of Proposition 3.4.8 to $\mathsf{log\_loop}$. Hence, $\mathit{FLAT}\text{-}\mathit{AC}^k \subseteq \mathcal{NRC}^1(\mathsf{log\_loop}^{(k)}, \leq)$. $\square$

Now we can prove our main results. Theorems 3.7.1 and 3.7.2 follow from Corollary 3.8.11 and Proposition 3.8.13. Proposition 3.7.5 follows from Proposition 3.8.10, and Proposition 3.7.3 is proven by an straightforward extension of the proof of Proposition 3.8.10.

Chapter 4

The Implementation Language $\mathcal{MAP}$

4.1 Overview

We will implement the Nested Collection Calculus $\mathcal{NCC}$ and the additional forms of recursions described in Chapter 2 using parallel algorithms expressed in a high-level programming language. There are many advantages to programming in a high-level language. However, while sequential algorithms are typically designed and evaluated in reasonably high-level terms, the situation with parallel algorithms is - by necessity, so far - more complicated. The issue is intimately connected with the existing efforts to bridge the gap between the theoretical design of parallel algorithms and practical programming on massively parallel computers.

In the case of *data parallelism*, the work of Blelloch [12, 13] and Blelloch and Sabot [17] has made substantial progress on this issue. For example, if we manage to represent an algorithm in a high-level language such as NESL with a certain work and time complexity and if the representation satisfies certain restrictions then we are guaranteed an implementation of the same algorithm with the same asymptotic time and work complexity in terms of a low-level parallel vector model, which in turn admits efficient implementations on various architectures, for example the CM2.

To implement $\mathcal{NCC}$ we follow a similar path. We start by defining an implementation language called $\mathcal{MAP}$, presented under a different name in [98]. $\mathcal{MAP}$ is is a somewhat abstract high-level, general-purpose language manipulating mostly nested sequences. We give a precise high-level definition of parallel complexity (in the work and time framework [67]) for $\mathcal{MAP}$ programs. Unlike NESL, $\mathcal{MAP}$ is based on a restricted form of recursion, called map-recursion, rather than general recursion. This will surely impose some limitations, although not that many: we will show that high-level parallel algorithms can be expressed with map-recursion, and that $\mathcal{MAP}$ is suitable as an implementation language for our high-level declarative parallel query language. By restricting to map-recursion instead of general recursion we will show in Chapter 5 that we can develop provable efficient compilation techniques. By contrast the compilation techniques [12, 17, 13] for a language with a general form of recursion, like NESL, can guarantee efficiency only for programs satisfying certain semantic conditions.

4.2 The Language $\mathcal{MAP}$

We define the language $\mathcal{MAP}$ to be the restriction of the Nested Collection Calculus $\mathcal{NCC}$ (Section 2.2) to sequences, and extended with a while-iteration and map-recursion.

As $\mathcal{NCC}$, $\mathcal{MAP}$ is parameterized by a set Σ of base types and external functions. The base types will always include $\mathbb{N}$, with the standard interpretation of natural numbers, and may include additional types like *string*, *real*, or user-defined types. We will generically denote with $\mathbb{D}$ some arbitrary base type. Hence the types in $\mathcal{MAP}$ are given by the grammar:

$$t ::= \mathbb{N} \mid \mathbb{D} \mid t \times \ldots \times t \mid t + \ldots + t \mid [t]$$

Recall that *unit* is the empty product (obtained by taking $n = 0$ in $t_1 \times \ldots \times t_n$), and $\mathbb{B} \stackrel{\text{def}}{=} \mathit{unit} + \mathit{unit}$.

$$\frac{\Gamma \rhd p : t \to \mathbb{B} \quad \Gamma \rhd f : t \to t}{\Gamma \rhd \mathsf{while}(p, f) : t \to t}$$

$$\frac{d : t \to [t] \quad c : t \times [t'] \to t'}{\mathsf{maprec}(d, c) : t \to t'}$$

Figure 4.1: The language $\mathcal{MAP}$: iterations and map-recursion

Thus $\mathcal{MAP}$ has all operations in Figures 2.1- 2.6 (Section 2.2), except that the general collection operations in Figure 2.5 are only instantiated to sequences; it has all sequence-specific operations in Figure 2.10; and it has the while and maprec constructs shown here in Figure 4.1. The map-recursion construct was defined in Subsection 2.4.3 (and we will comment more on it in Section 4.4), and the while-construct has the following meaning: $\mathsf{while}(p, f)(x) = x$ when $p(x) = \mathsf{false}$, and $\mathsf{while}(p, f)(x) = \mathsf{while}(p, f)(f(x))$ when $p(x) = \mathsf{true}$. Finally we assume, for convenience, that $\mathcal{MAP}$ is given in the $\mathsf{map}, \mathsf{flatten}$ presentation of Figure 2.7, instead of the ext-presentation.

$\mathcal{MAP}$ is a strongly, statically typed language. The type derivations in $\mathcal{MAP}$ play a crucial role, because some of the definitions of its operational semantics, and of its parallel complexities, will actually depend on the type derivation. Recall that a **type context** Γ is a set of the form $\Gamma = \{x_1 : s_1, \ldots, x_n : s_n\}$, where x_i are variables and s_i are types. We write $\Gamma \rhd e : t$, or $\Gamma \rhd f : t_1 \to t_2$, when we want to say that, under the type assumptions of Γ, the term expression e has type t, or the function expression f has type $t_1 \to t_2$. The typing rules for $\mathcal{MAP}$ are those for $\mathcal{NCC}$. A **type derivation** for $\Gamma \rhd e : t$ is a tree whose nodes are labeled with the typing rules defining the language $\mathcal{MAP}$, and whose root is labeled with $\Gamma \rhd e : t$. Similarly, we have type derivations for $\Gamma \rhd f : t \to t'$. A type derivation for, say, $\Gamma \rhd e : t$, could be understood as a proof that e has type t under type context Γ. Note that there could be several type derivations for the same conclusion $\Gamma \rhd e : t$, because of the weakening rule.

Besides being an implementation language for our parallel primitives on collections, $\mathcal{MAP}$ together with its complexities T, W to be discussed shortly, is also a model of parallel computation. In order to keep it related to other such models, like the PRAM or the

Alternating Turing Machines, we made sure that any primitive in $\mathcal{MAP}$ increases the size of its input only by a polynomial function. Had we introduced as a primitive a function like $\iota(n) \stackrel{\text{def}}{=} [0, \ldots, n-1]$, which generates an arbitrarily long list out of a number, this property would fail: for an input n of size 1 (or size $\log n$), ι returns an object of size n.

From the small set of primitives in $\mathcal{MAP}$, we can *derive* a rich set of functions. Section 2.3 gives some very simple examples of expressions in $\mathcal{MAP}$ without using iterations or map-recursion. In Section 4.6 we will show how $\mathcal{MAP}$ can express more complex algorithms.

4.3 Adding Syntactic Sugar to $\mathcal{MAP}$

$\mathcal{MAP}$ is described in a concise, mathematical style but can be easily extended to a more user-friendly language, by allowing a certain amount of block structure: definitions of global/local variables and of non-recursive functions. More precisely we will allow:

- Non-recursive let-bindings for terms and functions. E.g.:

$$
\begin{array}{l}
\mathsf{val}\ a = [4, 2, 6, 9] \\
\mathsf{fun}\ f(x) = \\
\quad \mathsf{let\ val}\ y = \mathsf{enumerate}(x) \\
\quad \mathsf{in\ map}(\lambda(v, i).v * i)(\mathsf{zip}\langle x, y\rangle) \\
\quad \mathsf{end} \\
\mathsf{val}\ b = f(a)
\end{array}
$$

  Here we define two values, a and b, and a function f. a is just a sequence literal. The function f of argument x is defined as $\mathsf{map}(\lambda(v, i).v * i)(\mathsf{zip}\langle x, y\rangle)$, where y is a local definition bound inside the function f by the let-construct. Finally b is another value obtained by applying f to a: in particular b will be $[0, 2, 12, 27]$. We allow also nested let-bindings. The let-bindings can be easily compiled away by substitutions.

- map-recursive definitions. We allow a function $f = \mathsf{maprec}(d, c)$ to be defined as:

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

- Pattern matching. E.g. we may define the function *delta* as:

$$\begin{aligned} \text{fun } delta(0) &= 1 \quad | \\ delta(x) &= 0 \end{aligned}$$

- Declaration of polymorphic functions. We choose an easy way of dealing with polymorphism: namely every polymorphic function is instantiated to all ground types at which it is used. More advanced techniques described in [59] could be used to avoid code duplication.

4.4 map-Recursion

The ability to define recursive functions is crucial for the expressiveness of a parallel programming language. We have seen in Chapter 3 how the divide-and-conquer recursion on sets offered us the needed ingredient for expressing all functions from NC. Also, standard textbooks in parallel algorithms [74, 67] often use recursion to achieve conciseness in the description of most algorithms.

However, general recursion in a parallel function language is difficult to compile. Blelloch and Sabot [17] and Blelloch [12] 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, namely the recursive functions have to be map-recursive. A compiler would simply have to verify that the function definition looks like that:

$$
\begin{aligned}
\mathsf{fun}\ g(x) &= \mathsf{if}\ p(x)\ \mathsf{then}\ s(x)\ \mathsf{else}\ c\langle g(d_1(x)), g(d_2(x))\rangle \\
\mathsf{fun}\ h(x) &= \mathsf{if}\ p(x)\ \mathsf{then}\ s(x)\ \mathsf{else}\ c(h(d(x))) \\
\mathsf{fun}\ k(x) &= \mathsf{if}\ p(x)\ \mathsf{then}\ s(x)\ \mathsf{else} \\
&\quad\ \mathsf{if}\ p'(x)\ \mathsf{then}\ c\langle k(d_1(x)), k(d_2(x))\rangle\ \mathsf{else}\ c'\langle k(d'_1(x)), k(d'_2(x)), k(d'_3(x))\rangle
\end{aligned}
$$

Figure 4.2: Other recursive schemas which can be automatically translated to map-recursion

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

Figure 4.3: Rewriting of a function as map-recursive

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

Then, in Theorem 5.2.1, we show how map-recursion can be compiled away, while preserving the parallel time complexity T.

Imposing such a restriction is not too sever. In fact map-recursion is general enough to express many existent parallel algorithms: tail recursive definitions, and what is usually meant by divide-and-conquer recursion (for instance the worked example in Subsection 4.6.2) can be easily converted to map-recursive definitions. It is easy to write a preprocessor which translates recursive schemas like those in figure Figure 4.2 into map-recursive definitions, and in the process "parallelizes" them. For example, the function g in Figure 4.2 is equivalent to $\mathsf{maprec}(d', c')$, where d', c' are given in Figure 4.3.

In fact few recursion schema cannot be translated into map-recursion. One example is Ackerman's function: $A(x, y) = A(x-1, A(x, y-1))$. We argue that most practical algorithms can be translated into map-recursion[1].

[1] A notable exception however is the worst-case median-finding algorithm ([37] p. 189, attributed to

4.5 Operational Semantics and Parallel Complexities

As promised, we will give in this Section a high-level definition of parallel time complexity T and work complexity W for $\mathcal{MAP}$ programs, in an machine independent way. The idea is for the parallel complexity of a program to be inferred from its structure in the same way in which the sequential complexity is inferred from the structure of a program in a sequential language. The parallel time complexity T is based on some idealized computation model with arbitrary many processors and instant communication. The purpose of the high-level complexities T, W allow the programer to infer the real complexity of some $\mathcal{MAP}$ program using formula 2.2 in Section 2.6.

All primitive operations (including @ and $\mathsf{flatten}$) take by definition one parallel step, while in a $\mathsf{map}(f)([x_0, \ldots, x_{n-1}])$, the n executions of f are done in parallel. The iterative constructs ($\mathsf{while}, \mathsf{maprec}$) however may count for several steps hence our definition cannot be done solely by induction on programs. Instead we provide rules for a formal operational semantics and then count the depth of the derivations. The work complexity is tied to the size of the data that is being manipulated.

Assume we have some method for encoding objects of any base type $\mathbb{D}$. Then we define **S-objects** by the grammar:

$$C ::= n \mid b \mid \langle C, \ldots, C \rangle \mid in_i^{t_1 + \ldots + t_m}(C) \mid [C, \ldots, C]$$

where $n \in \mathbb{N}$, b is some encoding for an object of the base type $\mathbb{D}$, $t_1, \ldots, t_m$ are types, and $1 \leq i \leq m$. We only consider typed S-objects.

We adopt a *unit size* complexity measure, in that every natural number has size 1, and define **the size** of some S-object as follows:

$$\begin{aligned} size(n) &\stackrel{\text{def}}{=} 1 \\ size(b) &\stackrel{\text{def}}{=} \text{the size of the encoding } b \end{aligned}$$

Blum, Floyd, Pratt, Rivest, and Tarjan).

$$
\begin{aligned}
size\langle C_1,\ldots,C_n\rangle &\stackrel{\text{def}}{=} 1 + size(C_1) + \ldots + size(C_n) \\
size(in_i^{t_1+\ldots+t_m}(C)) &\stackrel{\text{def}}{=} 1 + size(C) \qquad \text{for } 1 \le i \le m \\
size([C_0,\ldots,C_{n-1}]) &\stackrel{\text{def}}{=} 1 + \sum_{i=0,n-1} size(C_i)
\end{aligned}
$$

As before, we use true and false as abbreviations for $in_1^{unit+unit}\langle\rangle$ and $in_2^{unit+unit}\langle\rangle$. Following [40], we define **an environment** to be 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 S-objects. The variables $x_1,\ldots,x_n$ must be distinct, i.e. one can view an environment as a function from the finite set of variables $\{x_1,\ldots,x_n\}$ to complex objects. We define **the type context associated with the environment** $\rho = \{x_1 = C_1,\ldots,x_n = C_n\}$ to be $\Gamma = \{x_1 : s_1,\ldots,x_n : s_n\}$, where s_i is the type of C_i, for all $i = 1,n$.

Now we are ready to define the operational semantics of $\mathcal{MAP}$. It consists of rules which simultaneously define a ternary relation $\rho \bullet e \Downarrow C$ meaning that the term e evaluates to the S-object C under the environment ρ, and a 4-ary relation $\rho \bullet f(C) \Downarrow C'$ meaning that the function f applied to the S-object C evaluates to C'. E.g. 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 the term expression $e = \mathsf{zip}\langle x, \mathsf{enumerate}(x)\rangle$ and let ρ be the environment $\rho = \{x = [30,10,50]\}$: then $\rho \bullet e \Downarrow [\langle 30,0\rangle,\langle 10,1\rangle,\langle 50,2\rangle]$. In these figures the notation $x = C, \rho$ stands for the environment $\{x = C\} \cup \rho$ when x does not occur in ρ, and for the environment $\{x = C\} \cup (\rho - \{x = C'\})$ when $(x = C') \in \rho$.

The relations $\rho \bullet e \Downarrow C$ and $\rho \bullet f(C) \Downarrow C'$ will only be defined for environments ρ of the "right type". More precisely, they will be defined only when $\Gamma \rhd e : t$, and $\Gamma \rhd f : t \to t'$ respectively, where Γ is the type context associated with ρ. The relations $\Downarrow$ are defined in Figures 4.4- 4.12.

For each rule defining the two relations $\Downarrow$, there are corresponding formulas for computing the parallel time complexity T and the work complexity W. Here $size(\rho) \stackrel{\text{def}}{=} size(C_1) + \ldots + size(C_n)$, for $\rho = \{x_1 = C_1,\ldots,x_n = C_n\}$.

The external functions are computed sequentially. We assume that we are given a function

Rule	T, W
$\dfrac{}{x = C, \rho \bullet x \Downarrow C}$	$T((x = C, \rho), x) \stackrel{\text{def}}{=} 1$ $W((x = C, \rho), x) \stackrel{\text{def}}{=} \mathit{size}(C) + \mathit{size}(\rho)$
$\dfrac{}{\rho \bullet n \Downarrow n}$	$T(\rho, n) \stackrel{\text{def}}{=} 1$ $W(\rho, n) \stackrel{\text{def}}{=} 1 + \mathit{size}(\rho)$
$\dfrac{}{\Gamma \triangleright p : d_p \to c_p} \; (p \in \Sigma)$ For $p \in \Sigma$	$T(\rho, p, C) \stackrel{\text{def}}{=} 1 + T(p, C)$ $W(\rho, p, C) \stackrel{\text{def}}{=} T(p, C) + \mathit{size}(\rho)$

Figure 4.4: Operational semantics and T, W for variables, constants, and external functions

T such that, for every external function $p : d_p \to c_p$ in Σ and every S-object C, $T(p, C)$ is the sequential time needed to compute $p(C)$. Simple arithmetic operations like $+, -, *, /$ will usually take $T = O(1)$. Expensive user-defined external functions may run longer. In fact we give in Chapter 6 examples of parallel algorithms which are efficiently compiled for shared-nothing architectures which make heavy use of expensive external functions.

To *evaluate* some closed term e, one has to construct a *proof tree*, whose nodes are labeled with rules of the operational semantics, such that its root is labeled with some rule with conclusion $e \Downarrow C$. The complexities T and W are computed based on this proof tree, by applying the rules in Figures 4.4-4.12 at each node. In fact, there are simple guidlines for these rules. Namely for every evaluation of the form:

$$\frac{\rho_1 \bullet e_1 \Downarrow C_1, \ldots, \rho_n \bullet e_n \Downarrow C_n}{\rho \bullet e \Downarrow C}$$

T and W are defined as:

$$T(\rho, e) \stackrel{\text{def}}{=} 1 + \sum_{i=1}^{n} T(\rho_i, e_i)$$

Rule	T, W
$\dfrac{}{\rho \bullet \langle\rangle \Downarrow \langle\rangle}$	$T(\rho, \langle\rangle) \stackrel{\text{def}}{=} 1$ $W(\rho, \langle\rangle) \stackrel{\text{def}}{=} 1 + size(\rho)$
$\dfrac{\rho \bullet e_1 \Downarrow C_1, \ldots, \rho \bullet e_n \Downarrow C_n}{\rho \bullet \langle e_1, \ldots, e_n\rangle \Downarrow \langle C_1, \ldots, C_n\rangle}(i = 1, n)$ $\dfrac{\rho \bullet e \Downarrow \langle C_1, \ldots, C_n\rangle}{\pi_i(e) \Downarrow C_i}$	$T(\rho, \langle e_1, \ldots, e_n\rangle) \stackrel{\text{def}}{=} 1 + \sum_{i=1,n} T(\rho, e_i)$ $W(\rho, \langle e_1, \ldots, e_n\rangle) \stackrel{\text{def}}{=} \sum_{i=1,n} W(\rho, e_i) +$ $size\langle C_1, \ldots, C_n\rangle$ $+size(\rho)$ $T(\rho, \pi_i(e)) \stackrel{\text{def}}{=} 1 + T(\rho, e)$ $W(\rho, \pi_i(e)) \stackrel{\text{def}}{=} W(\rho, e) + size(C_i)$ $+size(\rho)$
$\dfrac{\rho \bullet e \Downarrow C}{\rho \bullet in_i(e) \Downarrow in_i(C)}$ $\dfrac{\rho \bullet e \Downarrow in_i(C) \quad x_i = C, \rho \bullet e_i \Downarrow C'}{\rho \bullet \left(\begin{array}{ccc} \mathsf{case} & e & \mathsf{of} \\ in_1(x) & \Rightarrow & e_1 \mid \\ & \ldots & \\ in_n(x_n) & \Rightarrow & e_n \end{array}\right) \Downarrow C'}$	$T(\rho, in_i(e)) \stackrel{\text{def}}{=} 1 + T(\rho, e)$ $W(\rho, in_i(e)) \stackrel{\text{def}}{=} W(\rho, e) + size(in_i(C))$ $+size(\rho)$ $T(\rho, \mathsf{case}\ldots) \stackrel{\text{def}}{=} 1 + T(\rho, e) +$ $T((x_i = C, \rho), e_i)$ $W(\rho, \mathsf{case}\ldots) \stackrel{\text{def}}{=} W(\rho, e)$ $+W((x_i = C, \rho), e_i)$ $+size(C_i) + size(\rho)$

Figure 4.5: Operational semantics and T, W for product types and sum types

Rule	T, W
$\dfrac{\rho \bullet e \Downarrow C_1 \quad \rho \bullet f(C_1) \Downarrow C_2}{\rho \bullet f(e) \Downarrow C_2}$	$T(\rho, f(e)) \stackrel{\text{def}}{=} 1 + T(\rho, e) + T(\rho, f, C_1)$ $W(\rho, f(e)) \stackrel{\text{def}}{=} W(\rho, e) + W(\rho, f, C_1)$ $+ \mathit{size}(C_2) + \mathit{size}(\rho)$
$\dfrac{x = C_1, \rho \bullet e \Downarrow C_2}{\rho \bullet (\lambda x.e)(C_1) \Downarrow C_2}$	$T(\rho, \lambda x.e, C_1) \stackrel{\text{def}}{=} 1 + T((x = C_1, \rho), e)$ $W(\rho, \lambda x.e, C_1) \stackrel{\text{def}}{=} W((x = C_1, \rho), e)$ $+ \mathit{size}(C_2) + \mathit{size}(\rho)$

Figure 4.6: Operational semantics and T, W for λ-abstraction and function application

Rule	T, W
$\dfrac{\rho \bullet p(C) \Downarrow \mathsf{false}}{\rho \bullet \mathsf{while}(p, f)(C) \Downarrow C}$	$T(\rho, \mathsf{while} \ldots, C) \stackrel{\text{def}}{=} 1 + T(\rho, p, C)$ $W(\rho, \mathsf{while} \ldots, C) \stackrel{\text{def}}{=} W(\rho, p, C) +$ $\mathit{size}(C) + \mathit{size}(\rho)$
$\dfrac{\begin{array}{rcl} \rho \bullet p(C) & \Downarrow & \mathsf{true} \\ \rho \bullet f(C) & \Downarrow & C_1 \\ \rho \bullet \mathsf{while}(p, f)(C_1) & \Downarrow & C_2 \end{array}}{\rho \bullet \mathsf{while}(p, f)(C) \Downarrow C_2}$	$T(\rho, \mathsf{while} \ldots, C) \stackrel{\text{def}}{=} 1 + T(\rho, p, C) +$ $T(\rho, f, C) + T(\rho, \mathsf{while} \ldots, C_1)$ $W(\rho, \mathsf{while} \ldots, C) \stackrel{\text{def}}{=} W(\rho, p, C) +$ $W(\rho, f, C) + W(\rho, \mathsf{while} \ldots, C_1) +$ $\mathit{size}(\rho)$ Note: $\mathit{size}(C_2)$ does not occur here.

Figure 4.7: Operational semantics and T, W for while

Rule	T, W
$$\frac{\begin{array}{rcl} \rho \bullet d(C) & \Downarrow & [C_0, \\ & & \vdots \\ & & C_{n-1}] \\ \rho \bullet \mathsf{maprec}(d,c)(C_0) & \Downarrow & C'_0 \\ & \cdots & \\ \rho \bullet \mathsf{maprec}(d,c)(C_{n-1}) & \Downarrow & C'_{n-1} \\ \rho \bullet c\langle C, [C'_0, \ldots, C'_{n-1}]\rangle & \Downarrow & C' \end{array}}{\mathsf{maprec}(d,c)(C) \Downarrow C'}$$	$$\begin{array}{l} T(\rho, \mathsf{maprec}(d,c), C) \stackrel{\text{def}}{=} \\ \quad 1 + T(\rho, d, C) + \\ \quad \max_i(T(\rho, \mathsf{maprec}(d,c), C_i)) + \\ \quad T(\rho, c, [C'_0, \ldots, C'_{n-1}]) \end{array}$$ $$\begin{array}{l} W(\rho, \mathsf{maprec}(d,c), C) \stackrel{\text{def}}{=} W(\rho, d, C) + \\ \quad \sum_i W(\rho, \mathsf{maprec}(d,c), C_i) + \\ \quad W(\rho, c, [C'_0, \ldots, C'_{n-1}]) + \\ \quad \mathit{size}([C'_0, \ldots, C'_{n-1}]) + \mathit{size}(\rho) \end{array}$$

Figure 4.8: Operational semantics and the T, W for map-recursion

$$W(M) \stackrel{\text{def}}{=} \sum_{i=1}^{n} W(\rho_i, e_i) + \mathit{size}(C) + \mathit{size}(\rho)$$

There are however two exceptions:

1. For the map-rule we replace the $\sum$ in the definition of T with max. This corresponds to the fact that the function is applied *in parallel* on all objects in the sequence.

2. For the while rule we delete $\mathit{size}(C)$ from the right-hand side of the definition of W. That is while is a *true* while construct, and not a tail-recursion.

Strictly speaking the work complexity W of a term e does not only depend on e, but also on the *type derivations* for e. There could be several different type derivations for the same term e, due to the weakening rule, see Figure 2.6 of Section 2.2. Discovering earlier or later that some variable is not actually used in a subterm can make a difference in the work complexity, because the map rule replicates the environment ρ to all n threads running in parallel. A good compiler will discover at typechecking time which free variables are not actually needed, and thus will keep the work complexity low.

Rule	T, W
$\dfrac{}{\rho \bullet [] \Downarrow []}$	$T(\rho, []) \stackrel{\text{def}}{=} 1$ $W(\rho, []) \stackrel{\text{def}}{=} 1 + \mathit{size}(\rho)$
$\dfrac{\rho \bullet e \Downarrow C}{\rho \bullet [e] \Downarrow [C]}$	$T(\rho, [e]) \stackrel{\text{def}}{=} 1 + T(\rho, e)$ $W(\rho, [e]) \stackrel{\text{def}}{=}$ $W(\rho, e) + \mathit{size}(C) + \mathit{size}(\rho)$
$\dfrac{\begin{array}{ccc} \rho \bullet e & \Downarrow & [C_0, \ldots C_{m-1}] \\ \rho \bullet e' & \Downarrow & [C'_0, \ldots, C'_{n-1}] \end{array}}{\rho \bullet e @ e' \Downarrow [C_0, \ldots, C_{m-1}, C'_0, \ldots, C'_{n-1}]}$	$T(\rho, e@e') \stackrel{\text{def}}{=} 1 + T(\rho, e) + T(\rho, e')$ $W(\rho, e@e') \stackrel{\text{def}}{=} W(\rho, e) + W(\rho, e') +$ $\mathit{size}([C_0, \ldots, C'_{n-1}]) + \mathit{size}(\rho)$
$\dfrac{\rho \bullet e \Downarrow [[C_{00}, C_{01}, \ldots], [C_{10}, C_{11}, \ldots], \ldots]}{\rho \bullet \mathsf{flatten}(e) \Downarrow [C_{00}, C_{01}, \ldots, C_{10}, C_{11}, \ldots]}$	$T(\rho, \mathsf{flatten}(e)) \stackrel{\text{def}}{=} 1 + T(\rho, e)$ $W(\rho, \mathsf{flatten}(e)) \stackrel{\text{def}}{=}$ $W(\rho, e) + \mathit{size}([C_{00}, \ldots]) + \mathit{size}(\rho)$
$\dfrac{\rho \bullet e \Downarrow [C_0, \ldots, C_{n-1}]}{\rho \bullet \mathsf{length}(e) \Downarrow n}$	$T(\rho, \mathsf{length}(e)) \stackrel{\text{def}}{=} 1 + T(\rho, e)$ $W(\rho, \mathsf{length}(e)) \stackrel{\text{def}}{=} W(\rho, e) +$ $1 + \mathit{size}(\rho)$
$\dfrac{\rho \bullet e \Downarrow [C]}{\rho \bullet \mathsf{get}(e) \Downarrow C}$	$T(\rho, \mathsf{get}(e)) \stackrel{\text{def}}{=} 1 + T(\rho, e)$ $W(\rho, \mathsf{get}(e)) \stackrel{\text{def}}{=}$ $W(\rho, e) + \mathit{size}(C) + \mathit{size}(\rho)$

Figure 4.9: Operational semantics and T, W for operations on collections

Rule	T, W
$$\frac{\rho \bullet f(C_0) \Downarrow C'_0 \quad \ldots \quad \rho \bullet f(C_{n-1}) \Downarrow C'_{n-1}}{\rho \bullet \mathsf{map}(f)([C_0, \ldots, C_{n-1}]) \Downarrow [C'_0, \ldots, C'_{n-1}]}$$	$T(\rho, \mathsf{map}(f), \ldots) \stackrel{\text{def}}{=} 1 + \max_{i=0,n-1}(T(\rho, f, C_i))$ $W(\rho, \mathsf{map}(f), \ldots) \stackrel{\text{def}}{=} \sum_{i=0,n-1} W(\rho, f, C_i) +$ $\mathit{size}([C'_0, \ldots, C'_{n-1}]) +$ $\mathit{size}(\rho)$

Figure 4.10: Operational semantics and T, W for map

4.6 Expressing Parallel Algorithms in $\mathcal{MAP}$

We present in this Section a collection of useful parallel algorithms expressed in $\mathcal{MAP}$ and analyze their complexity. We start with *routing* algorithms, i.e. algorithms which rearrange the elements of a sequence, and continue with *sorting* algorithms. Finally we show how these algorithms can be used to implement efficiently the operations on *sets* and *bags* discussed in Chapter 2.

4.6.1 Routings

We call *routings* those operations which rearrange the order of the elements of a sequence $X = [X_0, \ldots, X_{n-1}]$, by specifying which element to be placed on which output position. This can be done essentially in two ways. **Reading**, in which we are given a sequence $[i_0, i_1, \ldots, i_{m-1}]$ and have to construct the sequence $Y = [x_{i_0}, x_{i_1}, \ldots, x_{i_{m-1}}]$, and **writing**, in which we are given a sequence $[j_0, \ldots, j_{n-1}]$ representing a permutation of $0, 1, \ldots, n-1$, and are required to construct a sequence $Y = [y_0, \ldots, y_{n-1}]$ where $y_{j_k} = x_k$, for $k = 0, n-1$.

We show here that an arbitrary routing problem can be solved in constant parallel time, with a work complexity which is arbitrarily close to linear: $T = O(1), W = O(n^{1+\varepsilon})$. Also, we show that if the routing is monotone (i.e. $i_0 \leq i_1 \leq \ldots \leq i_{m-1}$ in the reading problem),

Rule	T, W
$$\frac{\begin{array}{lcl}\rho \bullet e & \Downarrow & [C_0, \ldots, C_{n-1}] \\ \rho \bullet e' & \Downarrow & [C'_0, \ldots, C'_{n-1}]\end{array}}{\rho \bullet \mathsf{zip}\langle e, e' \rangle \Downarrow [\langle C_0, C'_0 \rangle, \ldots, \langle C_{n-1}, C'_{n-1} \rangle]}$$	$T(\rho, \mathsf{zip}\langle e, e' \rangle) \stackrel{\text{def}}{=} 1 + T(\rho, e) + T(\rho, e')$ $W(\rho, \mathsf{zip}\langle e, e' \rangle) \stackrel{\text{def}}{=} W(\rho, e) + W(\rho, e') + \mathit{size}([\langle C_0, C'_0 \rangle, \ldots]) + \mathit{size}(\rho)$
$$\frac{\rho \bullet e \Downarrow [C_0, \ldots, C_{n-1}]}{\rho \bullet \mathsf{enumerate}(e) \Downarrow [0, \ldots, n-1]}$$	$T(\rho, \mathsf{enumerate}(e)) \stackrel{\text{def}}{=} 1 + T(\rho, e)$ $W(\rho, \mathsf{enumerate}(e)) \stackrel{\text{def}}{=} W(\rho, e) + n + 1 + \mathit{size}(\rho)$
$$\frac{\begin{array}{lcl}\rho \bullet e_1 & \Downarrow & [C_0, \ldots C_{n_0+\ldots+n_{m-1}-1}] \\ \rho \bullet e_2 & \Downarrow & [n_0, \ldots, n_{m-1}]\end{array}}{\begin{array}{c}\rho \bullet \mathsf{split}\langle e_1, e_2 \rangle \quad \Downarrow \\ \begin{array}{llcll}[[& C_0, & \ldots, & C_{n_0-1} &], \\ {[} & C_{n_0}, & \ldots, & C_{n_0+n_1-1} &], \\ & & \ldots & & \\ {[} & C_{n_0+\ldots+n_{m-2}}, & \ldots, & C_{n_0+\ldots+n_{m-1}-1} &]]\end{array}\end{array}}$$	$T(\rho, \mathsf{split}(\ldots)) \stackrel{\text{def}}{=} 1 + T(\rho, e_1) + T(\rho, e_2)$ $W(\rho, \mathsf{split}(\ldots)) \stackrel{\text{def}}{=} W(\rho, e_1) + W(\rho, e_2) + \mathit{size}([[C_0, \ldots], \ldots]) + \mathit{size}(\rho)$

Figure 4.11: Operational semantics and T, W for operations on sequences

Rule	T, W
$$\frac{\rho \bullet e \Downarrow C}{X = C', \rho \bullet e \Downarrow C}$$	$T((x = C', \rho), e) \stackrel{\text{def}}{=} 1 + T(\rho, e)$ $W((x = C', \rho), e) \stackrel{\text{def}}{=} W(\rho, e) + \mathit{size}(C) + \mathit{size}(x = C', \rho)$
$$\frac{\rho \bullet f(C_1) \Downarrow C_2}{X = C', \rho \bullet f(C_1) \Downarrow C_2}$$	T, W same as above

Figure 4.12: Operational semantics and T, W for the weakening rules

$$
\begin{array}{l}
\mathsf{fun}\ \mathit{Index}\langle X, I\rangle = \\
\quad \mathsf{let\ val}\ X\_pos = \mathsf{zip}\langle X, \mathsf{enumerate}(X)\rangle \\
\qquad \mathsf{fun}\ \mathit{select}(i) = \Pi_1(\mathit{filter}(\lambda\langle x, p\rangle.i = p)(X\_pos)) \\
\quad \mathsf{in\ flatten}(\mathsf{map}(\mathit{select})(I)) \\
\quad \mathsf{end}
\end{array}
$$

Figure 4.13: The function *Index* expressed in $\mathcal{MAP}$

then the work complexity can be kept linear $W = O(n)$.

Indexing m elements from n, $T = O(1), W = O(mn)$ This is a "reading" problem:

$$\mathit{Index} : [t] \times [\mathbb{N}] \to [t]$$

with the meaning:

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

Here $m = \mathsf{length}(I)$, $n = \mathsf{length}(X)$.

When $\mathsf{length}(I) = \mathsf{length}(X) = n$ and I contains the numbers $0, 1, 2, \ldots, n-1$, then it generates an arbitrary permutation of the input sequence X.

Example: $\mathit{Index}([a, b, c, d, e, f, g], [3, 0, 1, 3, 1]) = [d, a, b, d, b]$. Here $n = 7, m = 5$.

The method consists in replicating X m times. From each copy k, select element number i_k. See the algorithms in Figure 4.13. The complexities are $T = O(1), W = O(mn)$. For the work complexity W, observe that in the expression $\mathsf{map}(\mathit{select})(I)$, the function *select* has X_pos as a free variable. When we evaluate the function, we carry the value of X_pos, a sequence of length m, in the environment ρ, which the rule for map replicates a number of times equal to the number of elements in I, m in our case. Hence, the $O(mn)$ component in W.

Using *Index*, we derive the subscript function $\mathit{index} : [t] \times \mathbb{N} \to t$, with the meaning $\mathit{index}(X, i) = x_i$, with $T = O(1), W = O(n)$ as $\mathit{index}\langle X, i\rangle = \mathsf{get}(\mathit{Index}\langle X, [i]\rangle)$.

```
fun small_sort⟨X, m⟩ =
    let val I = filter(λk.k < m)(enumerate(X))
        fun select(i) = filter(λ⟨x, p⟩.i = p)(X)
    in flatten(map(select)(I))
    end
```

Figure 4.14: The function *small_sort* expressed in $\mathcal{MAP}$

Sorting n small integers, $T = O(1), W = O(mn)$

$$small\_sort : [t \times \mathbb{N}] \times \mathbb{N} \to [t \times \mathbb{N}]$$

sorts a sequence with integer keys, under the assumption that all keys are less than a given number. More precisely:

$$small\_sort\langle X, m\rangle = Y$$

s.t. (1) $m \leq \mathsf{length}(X)$, and X satisfies the restriction $\forall\langle x, k\rangle \in X,\ k \in \{0, 1, \ldots, m-1\}$, (2) Y contains all elements in X sorted in increasing order of its second component. The complexities are $T = O(1), W = O(mn)$, where $n = \mathsf{length}(X)$.

We call it *sorting of small integers*, because there is an upper bound on the size of the input keys. The function is more easily understood in the particular case in which the set of keys, $I = \Pi_2(X)$, is a permutation of $0, 1, \ldots, n-1$: then *small_sort* is just a "writing" problem.

Example:

$$small\_sort\langle[\langle a, 3\rangle, \langle b, 1\rangle, \langle c, 0\rangle, \langle d, 2\rangle, \langle e, 1\rangle, \langle f, 0\rangle]\rangle = [\langle c, 0\rangle, \langle f, 0\rangle, \langle b, 1\rangle, \langle e, 1\rangle, \langle d, 2\rangle, \langle a, 3\rangle]$$

Here $m = 4$, $n = 6$.

The method consists in replicating X m times, and selecting from each copy numbered k, $k = 0, n-1$, those elements having the key k. Figure 4.14 gives the algorithm expressed in $\mathcal{MAP}$: its complexity is $T = O(1), W = O(mn)$.

Monotone routing, $T = O(1), W = O(m+n)$ This is a "reading" problem

$$m\_Index : [t] \times [\mathbb{N}] \to [t]$$

with the restriction that the sequence of indexes be sorted. More precisely:

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

provided that $0 \leq i_0 \leq i_1 \leq \ldots \leq i_{m-1} < n$. Here $m = \mathsf{length}(I)$, $n = \mathsf{length}(X)$.

Knowing that I is sorted allows us to reduce the work complexity to $W = O(m+n)$. For that we reduce *m_Index* to a *bmRoute* problem. More precisely, we compute how many times each element x_i of X has to occur in the final result; call J the sequence of these occurrences. To compute J, start by computing for each element x_i, the position p_i following the last occurrence of x_i in the final result; then $j_i = \Delta p_i \stackrel{\text{def}}{=} p_i \dot{-} p_{i-1}$. Finally, to compute $P = [p_0, p_1, \ldots, p_{n-1}]$, take each of the final positions $0, 1, 2, \ldots, m$, and replicate them ΔI times. E.g. when $n = 9, X = [x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8], I = [2, 4, 4, 7]$, then:

$$\begin{aligned}
\Delta I &\stackrel{\text{def}}{=} [2, 2, 0, 3, 2] \\
P &\stackrel{\text{def}}{=} [\underbrace{0, 0}_{2}, \underbrace{1, 1}_{2}, \underbrace{}_{0} \underbrace{3, 3, 3}_{3}, \underbrace{4, 4}_{2}] \\
J &\stackrel{\text{def}}{=} [0, 0, 1, 0, 2, 0, 0, 1, 0] \\
Y &\stackrel{\text{def}}{=} [\underbrace{}_{0}\underbrace{}_{0}\underbrace{x_2}_{1}, \underbrace{}_{0}\underbrace{x_4, x_4}_{2}, \underbrace{}_{0}\underbrace{}_{0}\underbrace{x_7}_{1}, \underbrace{}_{0}]
\end{aligned}$$

The algorithm is given in Figure 4.15, and has complexity $T = O(1), W = O(m+n)$.

Sorting n small integers with radix_sort: $T = O(1), W = O(m^{\varepsilon} n)$ The function

$$radix\_sort : [t \times \mathbb{N}] \times \mathbb{N} \to [t \times \mathbb{N}]$$

has the same meaning as *small_sort*, namely

$$radix\_sort\langle X, m\rangle = Y$$

s.t. (1) X and Y contain the same elements, and (2) Y is sorted according to its second component. Here $n = \mathsf{length}(X)$ and $\forall\langle x, k\rangle \in X$, k must be in $\{0, 1, \ldots, m-1\}$, and

$$
\begin{array}{l}
\textsf{fun}\ m\_Index\langle X, I\rangle = \\
\quad \textsf{letval}\ n = \textsf{length}(X) \\
\quad\quad \textsf{val}\ m = \textsf{length}(I) \\
\quad\quad \textsf{val}\ zero\_to\_m = \textsf{enumerate}(I)@[m] \\
\quad\quad \textsf{val}\ delta\_I = \textsf{map}(\dot{-})(\textsf{zip}\langle I@[n], [0]@I\rangle) \\
\quad\quad \textsf{val}\ P = bmRoute\langle X, delta\_I, zero\_to\_m\rangle \\
\quad\quad \textsf{val}\ J = \textsf{map}(\dot{-})(\textsf{zip}\langle P, remove\_last([0]@P)\rangle) \\
\quad \textsf{in}\ bmRoute\langle I, J, X\rangle \\
\quad \textsf{end}
\end{array}
$$

See Example 2.3.5 for the function *remove_last*.

Figure 4.15: The function *m_Index* expressed in $\mathcal{MAP}$

$m \leq n$. In contrast with *small_sort*, *radix_sort* has a smaller complexity: $T = O(1)$, and $W = O(m^\varepsilon n)$, where $\varepsilon > 0$ can be chosen arbitrarily small.

To achieve the lower work complexity we use radix sort. Namely given $\varepsilon > 0$, start by expressing m in the base $b = m^\varepsilon$ using $\frac{1}{\varepsilon}$ digits. Next process each digit, for a total of $\frac{1}{\varepsilon}$ steps, starting with the least significant one. At each digit k, sort the elements in X according to their k's digit, using *small_sort*, with $m' = m^\varepsilon$. See the algorithm in Figure 4.16. The complexities are $T = O(1/\varepsilon) = O(1)$, $W = O(m^\varepsilon n)$. Note that this gives us an $T = O(1), W = O(n^{1+\varepsilon})$ algorithm for computing an arbitrary permutation.

4.6.2 Merging and Sorting

The examples in this Subsection prove that sorting can be expressed in $\mathcal{MAP}$ in constant parallel time, with a work complexity which is arbitrarily close to linear.

Throughout this Subsection we assume that some order relation is given on the type t, $<: t \times t \to t$, which can be computed in constant time and linear work complexity.

Direct ranking, $T = O(1), W = O(mn)$ The function

$$direct\_rank : [t] \times [t] \to [\mathbb{N}]$$

```
fun radix_sort⟨X, m⟩ =
    let val n = length(X)
        val b = m^ε
        fun one_step⟨Y, k⟩ =
            let valZ = map(λ⟨x, i⟩.⟨⟨x, i⟩, digit⟨i, k, b⟩⟩)(Y)
            in ⟨Π₁(small_sort⟨Z, b⟩), k + 1⟩
            end
        fun p⟨Y, k⟩ = (k ≤ 1/ε)
        val ⟨Y, _⟩ = while(p, one_step)⟨X, 0⟩
    in Π₁(Y)
    end
```

Here $digit(i, k, b)$ returns the digit no. k of the number i expressed in base b.

Figure 4.16: The function *radix_sort* expressed in $\mathcal{MAP}$

```
fun rank_one⟨x, Y⟩ = length(filter(λy.y ≤ x)(Y))
fun direct_rank⟨X, Y⟩ = map(λx.rank_one⟨x, Y⟩)(X)
```

Figure 4.17: The function *direct_rank* expressed in $\mathcal{MAP}$

has the meaning:

$$direct\_rank\langle X, Y\rangle = [z_0, \ldots, z_{m-1}]$$

where $z_i = |\{y_j \mid y_j \leq x_i\}|$, for $i = 0, m-1$ (here $m = \mathsf{length}(X)$).

Example: suppose $a < b < c < d < e$, then $direct\_rank\langle [b, d, c], [b, a, d, c, e]\rangle = [2, 4, 3]$.

Figure 4.17 shows *direct_rank* expressed in $\mathcal{MAP}$. Its complexity is $T = O(1), W = O(mn)$. For sake of clarity we gave the code for the case when the keys are integers and $\leq$ is the order relation on numbers. Obviously any other order predicate could be used.

Direct merging, $T = O(1), W = O(mn)$ The function

$$direct\_merge : [t] \times [t] \rightarrow [t]$$

$$
\begin{array}{l}
\textsf{fun } \mathit{index\_split}\langle X, I\rangle = \\
\quad \textsf{let val } n = \textsf{length}(X) \\
\quad \textsf{in split}(X, \textsf{map}(\dot{-})(\textsf{zip}\langle I@[n], [0]@I\rangle)) \\
\quad \textsf{end} \\
\textsf{fun } \mathit{direct\_merge}\langle A, B\rangle = \\
\quad \textsf{let val } R = \mathit{direct\_rank}\langle A, B\rangle \\
\quad\quad \textsf{val } BB = \mathit{index\_split}\langle B, R\rangle \\
\quad \textsf{in } \mathit{first}(BB)@ \\
\quad\quad \textsf{flatten}(\textsf{map}(\lambda\langle a, B\rangle.[a]@B)(\textsf{zip}\langle A, \mathit{tail}(BB)\rangle)) \\
\quad \textsf{end}
\end{array}
$$

Figure 4.18: The function *direct_merge* expressed in $\mathcal{MAP}$

expects two sorted input sequences and merges them. See Figure 4.18. Its complexity is $T = O(1), W = O(mn)$.

Note: We shall use the function *index_split* : $[t] \times [\mathbb{N}] \to [[t]]$ in the sequel. Its meaning is:

$$\mathit{index\_split}([x_0, \ldots, x_{n-1}], [i_0, \ldots, i_{m-1}]) =$$

$$[[x_0, \ldots, x_{i_0-1}], [x_{i_0}, \ldots, x_{i_1-1}], \ldots, [x_{i_{m-2}}, \ldots, x_{i_{m-1}-1}], [x_{i_{m-1}}, \ldots, x_{n-1}]]$$

It has $T = O(1), W = O(m + n)$.

Direct sorting, $T = O(1), W = O(n^2)$ Our first sorting function:

$$\mathit{direct\_sort} : [t] \to [t]$$

sorts and input sequence X of length n in constant parallel time, with $W = O(n^2)$. For that, we replicate X n times, to compare each element with each other elements. Next we count how many elements are less than each X_i, then use *small_sort*. For this method to work, X needs to have distinct elements: otherwise we include the position of each element in X in the key and enhance the order predicate. See the algorithm in Figure 4.19.

```
fun direct_sort(X) =
    let fun less_than(u) = length(filter(λ(v).v < u)(X))
        val P = map(less_than)X
    in small_sort(zip⟨X, P⟩)
    end
```

Figure 4.19: The function *direct_sort* expressed in $\mathcal{MAP}$

Valiant's algorithm for merging in $T = O(\log\log m), W = O((m+n)\log\log m)$ We can reduce the $W = O(mn)$ work complexity of our first merging algorithm *direct_merge* by increasing the time complexity T. Note that a standard divide-and-conquer algorithm in which X is divided in two equal halves requires $T = O(\log m)$ steps. Instead, Valiant's algorithm [103, 67] divides X into $\sqrt{m}$ subsequences of length $\sqrt{m}$.

$$valiant\_merge : [t] \times [t] \to [t]$$

Let $m = \mathsf{length}(X), n = \mathsf{length}(Y)$ in $valiant\_merge(X, Y)$, where X and Y are two sorted sequences. Then we divide X into $\sqrt{m}$ subsequences $X_0, \ldots, X_{\sqrt{m}-1}$ of length $\sqrt{m}$ each. For each subsequence X_i, we extract the subsequence Y_i of Y with which X_i has to be merge: we do this in parallel for $i = 0, \sqrt{m}-1$. Then we call the function recursively, in parallel, on all pairs $\langle X_i, Y_i\rangle$. See Figures 4.20, and 4.21. The complexities are $T = O(\log\log m)$, $W = O((m+n)\log\log m)$.

Mergesort: $T = O(\log n \log\log n), W = O(n\log n\log\log n)$ (Valiant's sorting algorithm) Valiant's *merge* algorithm gives us an algorithm for sorting a sequence X in $T = O(\log n\log\log n), W = O(n\log n\log\log n)$. The function is

$$merge\_sort : [t] \to [t]$$

and its code is given in Figure 4.22.

```
fun sqrt_positions(X) =
    let val n = length(X)
        val I = filter(λi.i mod √n = 0)(enumerate(X))
    in index⟨X, I⟩
    end

fun sqrt_split(X) =
    index_split⟨X, sqrt_positions(enumerate(X))⟩
```

Figure 4.20: Auxiliary functions needed in Valiant's merge algorithm

```
fun divide⟨X, Y⟩ =
    letval m = length(X)
      val n = length(Y)
      val X' = sqrt_positions(X)
      val Y' = sqrt_positions(Y) (* X', Y' have lengths √m and √n respectively *)
      val R' = direct_rank⟨X', Y'⟩
      val YY1 = sqrt_split(Y) (* split Y into √n blocks *)
      val x_Y = zip⟨X', index⟨YY1, R'⟩⟩ (* group each x' with its block *)
      val RR' = map(rank_one)(x_Y) (* rank each x' in its block *)
      val R = map(λ⟨i, j⟩.(i ∸ 1) * √n + j)(zip⟨R', RR'⟩)
      val XX = sqrt_splitX
      val YY = index_split⟨Y, R⟩
    in zip⟨XX, YY⟩
    end

fun valiant_merge⟨X, Y⟩ =
    if length(X) ≤ 2 then direct_merge⟨X, Y⟩
    else
      in flatten(map(valiant_merge)(divide⟨X, Y⟩))
      end
```

Figure 4.21: Valiant's *merge* algorithm and the function *divide*

```
fun merge_sort(X) =
      if length(X) ≤ 1 then X
      else let val n = length(X)
                val XX = split(X, [n − n/2, n/2])
           in valiant_merge⟨mergesort(first(XX),
                              mergesort(last(XX)))⟩
           end
```

Figure 4.22: A *sort* function based on Valiant's *merge* algorithm

Merging two sequences: $T = O(1), W = O(m + m^{\varepsilon}n)$ Can we merge two sorted sequences in $T = O(1)$? Or can we sort a sequence in $T = O(1)$? In both cases we want an almost linear work complexity. The answer is yes for both questions. We start by showing how to merge in constant parallel time.

$$\textit{fast\_merge} : [t] \times [t] \to [t]$$

We use Valiant's merging algorithm of Figure 4.21, with the following change: stop the recursion when $\mathsf{length}(X) \leq m^{\varepsilon}$, instead of $\mathsf{length}(X) \leq 2$. Thus, *fast_merge* will call itself recursively on subsequences of X having lengths $m, m^{\frac{1}{2}}, m^{\frac{1}{2^2}}, m^{\frac{1}{2^3}}, \ldots, m^{\varepsilon}$. The total number of iterations will be $T = \lceil \log \frac{1}{\varepsilon} \rceil = O(1)$. The work complexity for *direct_merge*(X, Y) is $m^{\varepsilon}n_i$, where n_i is the length of Y (a subsequence of the initial Y). The total work complexity for all calls to *direct_merge* will therefore be $O(m^{\varepsilon}n)$.

Merging k sequences of total length n, $k = O(n^{\varepsilon})$: $T = O(1), W = O(n^{1+2\varepsilon})$ This function will be useful for us in designing a *sort* procedure running in a constant number of parallel steps and with almost linear work complexity. Its type is:

$$\textit{multi\_merge} : [[t]] \to [t]$$

The function *multi_merge*(XX) expects a nested sequence $XX = [X_0, \ldots, X_{k-1}]$. Each X_i, $i = 0, k-1$, is a sorted sequence. *multi_merge* will merge the k sequences. Here

$n = \mathsf{length}(\mathsf{flatten}(XX))$, and k is subject to the restriction $k = O(n^\varepsilon)$.

The method we use is related to the techniques described in [74] (Section 3.5.3) for an $O(\log n \log\log n)$ sorting algorithm on a hypercube, attributed to Cypher and Plaxton. First we choose a set of *splitters* S_i from X_i, $i = 0, k-1$. Namely S_i will contain exactly n^ε splitters, dividing X_i into n^ε subsequences of equal size. Sort the total $n^{2\varepsilon}$ splitters using *direct_sort*, in time $T = O(1)$, and with work complexity $W = O(n^{4\varepsilon})$ (here we assume $\varepsilon \leq 1/4$). Call S the resulting sequence. Split each sequence X_i into $n^{2\varepsilon}$ subsequences $X_{i0}, X_{i1}, X_{i2}, \ldots$, according to the splitters. This requires a direct merge of S and X_i, with $T = O(1), W_i = O(n^{2\varepsilon} n_i)$, where $n_i = \mathsf{length}(X_i)$. The total work complexity is $W = O(n^{1+2\varepsilon})$. Now transpose the nested sequence $[[X_{00}, X_{01}, X_{02}, \ldots], [X_{10}, X_{11}, \ldots], \ldots]$, to get $P = [[X_{00}, X_{10}, X_{20}, \ldots], [X_{01}, X_{11}, X_{21}, \ldots], \ldots]$; we transpose using *radix_sort*, in time $O(1)$ and work complexity $O(n^{1+\varepsilon})$. Finally apply *multi_merge* recursively to each sequence $XX_j \stackrel{\text{def}}{=} [X_{0j}, X_{1j}, \ldots, X_{(k-1)j}]$, $j = 0, n^{2\varepsilon} - 1$. 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 $n, n^{1-\varepsilon}, n^{1-2\varepsilon}, \ldots$. Thus, after $T = 1/\varepsilon = O(1)$ steps, we have to merge nested sequences of total length $\leq n^\varepsilon$. We do this using direct-sort, for a total work complexity of $O(n^{1+\varepsilon})$. The code is given in Figure 4.23.

For the complexity, note that the function f has $1/\varepsilon = O(1)$ recursive calls. All operations have a linear work complexity, except *direct_sort* and *transpose*, which have an additional factor $n^{2\varepsilon}$. Hence, $T = O(1), W = O(n^{1+2\varepsilon})$.

Constant time sorting: $T = O(1), W = O(n^{1+\varepsilon})$ Finally we can describe the function:

$$const\_sort : [t] \to [t]$$

which sorts in a constant number of parallel steps, with almost linear work complexity.

For that we split the input sequence X into n^ε subsequences of equal size. Sort these sequence recursively, then merge them using *multi_merge*. Thus *const_sort* will be called on sequences of lengths $n, n^{(1-\varepsilon)}, n^{(1-\varepsilon)^2}, n^{(1-\varepsilon)^3}, \ldots$. After $T = \log \frac{1}{\varepsilon} / \log \frac{1}{1-\varepsilon} = O(1)$ recursive

```
fun ne_indexes⟨n, X⟩ = filter(λi.i mod n^ε = 0)(enumerate(X))
fun ne_positions⟨n, X⟩ =
    m_Index⟨X, ne_indexes⟨n, X⟩⟩
fun merge_split⟨X, Y⟩ =
    let val R = direct_rank⟨X, Y⟩
    in index_split⟨Y, R⟩
    end
fun multi_merge⟨XX⟩ =
    let val n = length(flatten(XX))
        fun f(XX) =
            if length(flatten(XX)) ≤ n^ε then direct_sort(flatten(XX))
            else let val S = direct_sort(flatten(map(ne_positions)(distribute_left⟨n, XX⟩)))
                     val Q = map(merge_split)(distribute_left⟨S, XX⟩)
                 in flatten(map(f)(transpose(Q)))
                 end
    in f(XX)
    end
```

Figure 4.23: Merging k sorted sequences in $\mathcal{MAP}$

```
fun ne_split⟨n, X⟩ =
    index_split⟨X, ne_indexes⟨n, X⟩⟩
fun const_sort(X) =
    let val n = length(X)
       fun f(X) =
          if length(X) ≤ n^ε then direct_sort(X)
          else multi_merge(map(f)(ne_split⟨length(X), X⟩))
    in f(X)
    end
```

Figure 4.24: Sorting in constant parallel time, with almost linear work complexity

calls, *const_sort* will have to sort sequences of length $\leq n^{\varepsilon}$. Here we use *direct_sort*. Thus, the complexities are $T = O(1), W = O(n^{1+\varepsilon})$. Figure 4.24 contains the code.

4.6.3 Operations on Sets and Bags

Using the algorithms described so far, we have enough tools to design efficient implementations in $\mathcal{MAP}$ of the parallel operations on sets and bags described in Chapter 2.

Sets We implement a set $X : \{t\}$ as a sorted sequence. When t is a base type, we take some standard order relation $\leq: t \times t \to \mathbb{B}$ at that type. For a composed type t we lift the order relation. E.g. when $t = [t']$, then we use the lexicographic ordering on t, that is, for $X = [x_0, \ldots, x_{m-1}]$ and $Y = [y_0, \ldots, y_{n-1}]$, $leq\langle X, Y\rangle = \mathsf{false}$ iff $\exists i$, $i = 0, m-1$, such that $\forall j < \min(i, n)$, $x_j = y_j$, and ($i > n$ or $x_i > y_i$). The function *leq* can be expressed in $\mathcal{MAP}$, with $T = O(1)$ and $W = O(m + n)$. Then the operations on sets are implemented as follows.

We implement **union**, $X \cup Y$, by adapting Valiant's *merge* algorithm, with $T = O(\log\log m)$, $W = O((m+n)\log\log m)$. Namely, after repeatedly dividing the first sequence X until $\mathsf{length}(X) \leq 2$, instead of calling $direct\_merge\langle X, Y\rangle$, we *union* X and Y using a direct method.

To implement $\mathsf{flatten}(X)$, we have to "union" k sorted sequences, where $k = \mathsf{length}(X)$. For that we apply $\log k$ times the union algorithm sketched above. This gives us an $T = O(\log k \log\log n)$ algorithm for $\mathsf{flatten}(X)$, where $n = \mathsf{length}(\mathsf{flatten}(X))$.

Finally $\mathsf{map}(f)$ on sets becomes simply $\mathsf{map}(f)$ on sequences, followed by a *sort*.

Note however that, in practice, more work is necessary than that. Queries for *flat databases* are often expressed in terms of the *Relational Algebra*, i.e. using the operations *union* $\cup$, *difference* $-$, *join* $\bowtie$, *selections* σ, and *database projections* Π_i. Powerful optimization and efficient implementation techniques exist for these operations. It would more efficient for a compiler to generate specialized code for them, rather than to to translate them into $\mathcal{NRA}$ and then implement them on sequences. We mention here only that Wong [107] has shown that all $\mathcal{NRA}$ expressions mapping flat relations to flat relations can be normalized and translated into equivalent relational algebra expressions. Those techniques could be combined with the implementation techniques described here.

Bags We implement a bag $X : \{\!|t|\!\}$ as a sequence of type $[t \times \mathbb{N}]$, sorted on its first component. A pair $\langle x, k\rangle$ in X denotes the fact that x occurs in X k times. E.g. the bag $\{\!|a, b, b, c, b, a|\!\}$ would be implemented as $[\langle a, 2\rangle, \langle b, 3\rangle, \langle c, 1\rangle]$, provided that $a < b < c$.

We implement **bag addition**, $X \oplus Y$, by, again, adapting Valiant's *merge* algorithm. Here, after repeatedly dividing X until $\mathsf{length}(X) \leq 2$, we merge X and Y and add their occurrence numbers. We implement $\mathsf{flatten}(X)$ as for sets, namely by repeating bag addition $\log k$ times, where $k = \mathsf{length} X$.

Next, we implement the specialized functions on bags unique and monus, see Section 2.3, straightforwardly: $\mathsf{unique}(X) \stackrel{\text{def}}{=} \mathsf{map}(\lambda\langle x, k\rangle.\langle x, 1\rangle)(X)$, while for $\mathsf{monus}\langle X, Y\rangle$ we use the same idea from Valiant's *merge* algorithm: divide repeatedly X and Y until X is small, then compute monus directly.

Chapter 5

Complexity-Preserving Compilation

We have argued in the previous Chapter that $\mathcal{MAP}$, although a seemingly simple language, is expressive enough to allow us to express high-level parallel algorithms. Here we capitalize on its simplicity and show that it admits a complexity-preserving compilation.

Our *first main result* in this Chapter consists in showing that map-recursion can be translated into while-loops, while asymptotically preserving the time complexity and adding an arbitrarily small overhead to the work complexity (Theorem 5.2.1). The major benefit however is that we can compile $\mathcal{MAP}$ without the need for an unbounded stack of vectors, as general recursion would require. Avoiding the stack is advantageous both for an implementation on a SIMD architecture, and on a MIMD architecture. SIMD architectures associate a relatively small memory with each processor: a program that generates many entries in its vector stack will run out of memory even if the vectors are very short and hence much of the *total* amount of memory of the machine remains unused. Memory management on a MIMD architecture becomes simpler, when the number of vector registers is bound, see Chapter 6.

Following Blelloch, we define a simple parallel vector model in order to describe abstractly the class of target architectures for our compilation method (Section 5.3). Our BVRAM (Bounded Vector Random Access Machine) differs from the VRAM [12] in two ways. First it has a finite number of vector registers. This emphasizes the absence of a runtime vector stack. Of course the number of registers needed depends on the source program being compiled. Secondly, the BVRAM instructions require less powerful communication primitives. We exploit the simplicity of the BVRAM in two ways. First we show that all BVRAM instructions can be implemented on a butterfly network with $n \log n$ nodes in $O(\log n)$ steps. Secondly, we will show in Chapter 6 how the simplified communication primitives allow us to design an efficient implementation on a specific shared-nothing architecture, the LogP model.

Our *second main result* is a technique that compiles any $\mathcal{MAP}$ program into a BVRAM program again while asymptotically preserving the time complexity and adding an arbitrarily small overhead to the work complexity (Theorem 5.4.1).

As a consequence of the compilation result of $\mathcal{MAP}$ for the BVRAM, and of the implementation of the BVRAM on a butterfly network, one can derive efficient algorithms for the butterfly network from algorithms expressed in $\mathcal{MAP}$. In particular we show how to derive an $O(\log n)$ time sorting algorithm on a butterfly network with $n^{1+\varepsilon}$ nodes, matching a result by Preparata [86]).

One of the weaknesses of some of the traditional models of parallel computations is that they assume 0-cost communication, and infinite bandwidth [39]. This holds both for the PRAM model, and for the VRAM of [12]. Here we show that the BVRAM model of parallel computation, and hence the $\mathcal{MAP}$ model, cannot express a general permutation at "no cost", i.e. with $T = O(1)$ and $W = O(n)$. Thus, both models of parallel computation, $\mathcal{MAP}$ and BVRAM, partially account for the cost of communication.

Some of the results in this Chapter have been reported in [98].

5.1 The Major Steps in the Implementation of $\mathcal{MAP}$

We have shown so far that $\mathcal{MAP}$ is a powerful, high-level parallel language, with a clean definition of the parallel complexity, and which can express high-level parallel algorithms needed to implement declarative, high-level parallel query languages. We now show that $\mathcal{MAP}$ is still simple enough to admit an efficient implementation on shared-nothing architectures. The major implementation steps are shown in Figure 5.1.

We start by compiling away the map-recursion from $\mathcal{MAP}$, and replacing recursion with while-loops: this is described in Section 5.2. This compilation step preserves the parallel time complexity T, but adds an arbitrarily small overhead to the work complexity W, to $W^{1+\varepsilon}$.

Next we compile $\mathcal{MAP}$ without recursion into a simple vector parallel machine model, called *Bounded Vector Random Access Machine*, BVRAM. This step is described in Section 5.4.

The BVRAM, described in Section 5.3, has an instruction set which is simple enough to admit efficient implementation on a variety of parallel machines. We prove this claim in two ways. First we prove that the BVRAM admits and efficient implementation on a butterfly network, Proposition 5.3.1. Next, in Chapter 6, we describe a practical implementation of the BVRAM on a shared-nothing architecture.

5.2 Elimination of map-Recursion in $\mathcal{MAP}$

The first of our two main results states that *map*-recursion can be translated (in a source-to-source manner) into a $\mathcal{MAP}$ expression, while preserving its time complexity and "almost" preserving its work complexity.

Theorem 5.2.1 *Consider some function f defined in $\mathcal{MAP}$, with time and step complexity T, W. Then, for any $\varepsilon > 0$, we can construct a function f' in $\mathcal{MAP}$ without* map*-recursion, which is equivalent to f and which has time and work complexity $T' = O(T)$ and $W' = O(W^{1+\varepsilon})$ respectively.*

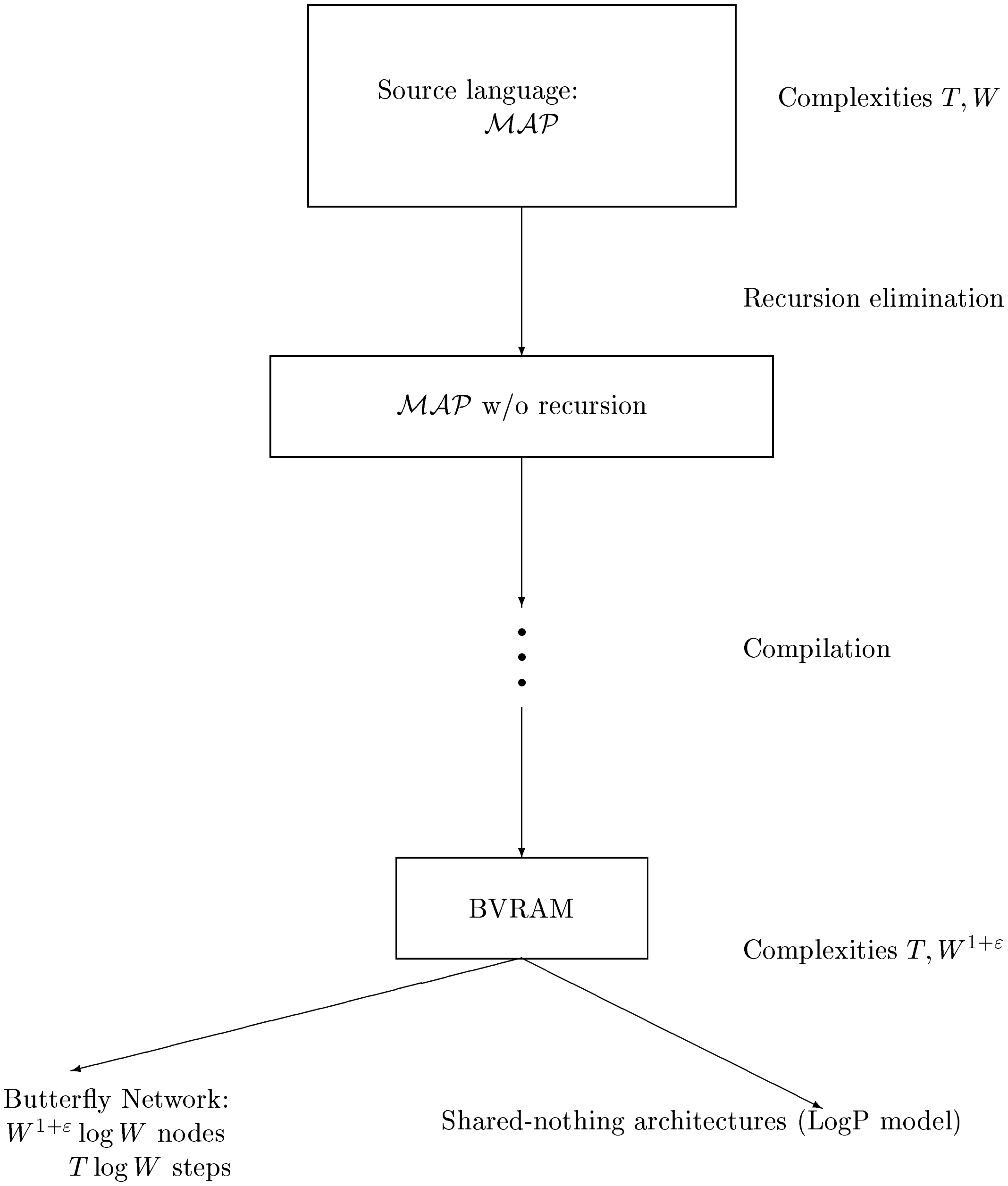


Figure 5.1: Implementation of $\mathcal{MAP}$: major steps

Proof. For illustration, we consider only the function g in Figure 4.2. Suppose the types are: $g : s \to t$, $d_1, d_2 : s \to s$, and $c : t \times t \to t$. Not surprisingly, g can be expressed in $\mathcal{MAP}$, without recursion, in two steps, called *divide phase* and *combine phase* in [80]:

Divide Phase Start with the singleton sequence $y = [x]$ of type $[s]$, and apply repeatedly the function $\mathsf{flatten} \circ \mathsf{map}(\lambda x.\mathsf{if}\ p(x)\ \mathsf{then}\ [x]\ \mathsf{else}\ [d_1(x), d_2(x)])$ 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 s 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 is also preserved, in the case in which the divide an conquer tree for the computation of $g(x)$ is perfectly balanced. When the tree is unbalanced, the leaves which are reached sooner have to coexists in the same sequence with those nodes which need more divide steps, thus adding to the total work complexity. Let ν be the number of different levels in the divide and conquer tree which contain leaves E.g. in an almost perfectly balanced tree, $\nu = 1$ or $\nu = 2$, while in a total "unbalanced" tree, ν can be equal to the total number of leaves, but still $\nu \leq W(g, x)$. We can compute ν 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 $\mathcal{MAP}$ 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 ν^ε 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 ν^ε times, at which point, we empty z_1, by moving everything into z_2. In general, we allow z_i to accumulate only ν^ε 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 ν levels. To compute the total additional work complexity, observe that each leave travels exactly once through $z_0, z_1, \ldots, z_{\frac{1}{\varepsilon}}$, and in each z_i is "touched" exactly ν^ε times. Thus, the total work complexity is bounded by $(\frac{1}{\varepsilon}+1)\nu^\varepsilon W = O(\nu^\varepsilon W) \leq O(W^{1+\varepsilon})$. Of course, rather complicated bookkeeping is necessary to keep all elements in z_i sorted. The *combine phase* is done similarly, but in reverse.

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 leave. Thus, in the general case ν is equal to the height of the divide and conquer tree. $\Box$

5.3 The Bounded Vector Random Access Machines

To compile $\mathcal{MAP}$ only a very simple vector parallel model is needed. The Bounded Vector Random Access Machine, BVRAM, is a restriction of the VRAM introduced in [12], in that it only admits a fixed number of registers, and has only particular communication primitives, not a general permutation. These restrictions further allow more efficient implementations of the BVRAM on existing parallel architectures.

As all our previous languages, the BVRAM is parameterized by a set Σ of base types and external functions. For simplicity we shall assume that all functions $p \in \Sigma$ are *scalar*, i.e. are of the form $p : \mathbb{D}_1 \times \ldots \mathbb{D}_k \to \mathbb{D}$. Also, we assume that $\mathbb{N}$ is among the base types, and that Σ contains a reasonable collection of arithmetic functions, like $+, -, *, /$. More complex base types and external functions will be needed in Chapter 6.

A BVRAM, M, consists of a fixed number of **vector registers** $V_1, \ldots, V_r$. Each V_i can hold a sequence (a vector) of natural numbers of arbitrary, but finite length. To keep the model simple, we do not include *scalar* registers: a number is represented by a sequence of length 1. A *program* for M is a sequence of labeled instructions, from the following instruction set. For some of the instructions below, it is convenient to view a pair of registers V_i, V_j in which the length of the first equals the sum of the numbers in the second as a *nested sequence*. E.g., intuitively we view $[x_0, x_1, z_0, z_1, z_2], [2, 0, 3]$ as standing for the nested sequence $[[x_0, x_1], [], [z_0, z_1, z_2]]$.

- Move instruction: $V_i \leftarrow V_j$.

- External functions, of the form $V_i \leftarrow p(V_{i_1}, \ldots, V_{i_k})$, with $p \in \Sigma$. $V_{i_1}, \ldots, V_{i_k}$ must be arrays of the same length, and the operation p is applied simultaneously on all all elements of $V_{i_1}, \ldots, V_{i_k}$ from the same positions, and the result is stored in V_i. In general we leave Σ unspecified, but mention here that for Theorems 5.2.1 and 5.4.1 Σ has to contain at least $+, \dot{-}, *, /,$ *right-shift*, $\log_2$, while for Proposition 5.7.5 we require that all operations in Σ be in NC.

- Sequence oriented operations: $V_i \leftarrow []$ loads the empty sequence in V_i. $V_i \leftarrow [n]$, where $n \in \mathbb{N}$ loads the singleton sequence $[n]$ into V_i. $V_i \leftarrow V_j @ V_k$ appends V_j and V_k and stores the result in V_i. $V_i \leftarrow [\mathsf{length}(V_j)]$ computes the length of V_j. $V_i \leftarrow \mathsf{enumerate}(V_j)$ loads the sequence $[0, 1, \ldots, n-1]$ into V_i, where n is the length of V_j.

- Bounded monotone routing $V_i \leftarrow \mathsf{bmRoute}(V_j, V_k, V_l)$; here V_k and V_l must have the same length. The effect is that each element in V_l is replicated a number of times equal to the corresponding number in V_k. In addition, it is required that the result matches in length the sequence V_j (i.e. initially V_j, V_k represent a nested sequence). E.g. if $V_j = [x_0, x_1, z_0, z_1, z_2]$, $V_k = [2, 0, 3]$ and $V_l = [a, b, c]$, then the instruction $V_i \leftarrow \mathsf{bmRoute}(V_j, V_k, V_l)$ stores $[a, a, c, c, c]$ into V_i. An unbounded version of $\mathsf{bmRoute}$ can be found in [87] under the name `dist`.

- Segmented bounded monotone routing $V_i \leftarrow \mathsf{sbmRoute}\ (V_j, V_k, V_l, V_m)$. Here, V_j, V_k and V_l, V_m must be nested sequences, and $\mathsf{length}(V_k) = \mathsf{length}(V_m)$. Then, the subsequences of V_l are replicated according to the numbers in V_k and the result is stored in V_i. E.g., suppose $V_j = [x_0, x_1, z_0, z_1, z_2]$, $V_k = [2, 0, 3]$, $V_l = [a_0, a_1, b_0, b_1, b_2, c_0, c_1, c_2]$ and $V_m = [2, 3, 3]$. Then, after $V_i \leftarrow \mathsf{sbmRoute}(V_j, V_k, V_l, V_m)$, V_i will hold the value

$[a_0, a_1,\ a_0, a_1, c_0, c_1, c_2, c_0, c_1, c_2, c_0, c_1, c_2]$. In the particular case when V_k, V_m have length one, this computes the *cartesian product* of V_j and V_l. Note that the length of the output is $\leq \mathsf{length}(V_j) * \mathsf{length}(V_l)$ and that bmRoute can be expressed with two sbmRoute instructions.

- Selection $V_i \leftarrow \mathsf{select}(V_j, V_k)$. Here V_j and V_k must have the same length. The values from V_k corresponding to the non-zero values in V_j are packed and stored in V_i. E.g. if $V_j = [1, 0, 1, 0, 0, 1]$ and $V_k = [a, b, c, d, e, f]$, then $[a, c, f]$ is stored in V_i.

- The unconditional jump goto l and the conditional jump if empty(V_i) then goto l, where l is a label of some instruction. The conditional jump is taken iff V_i currently holds the empty sequence.

- halt, stops the program.

We associate with each BVRAM program P two numbers: r_i, r_o, the number of *input* and *output* registers. P expects r_i inputs in the registers $V_1, \ldots, V_{r_i}$, and returns r_o outputs, in $V_1, \ldots, V_{r_o}$. 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 **parallel time complexity** T to be the total number of instruction executed by P, i.e. each instruction is considered to have parallel time complexity 1. Similarly, we define the **work complexity** W as the sum of the work complexities of all instructions executed by P, where the work complexity of some instruction is defined to be the sum of the lengths of its input and output registers.

As opposed to VRAMs [12] there is no general permutation instruction on a BVRAM (but one can be computed with an increase in the time or work complexity). This may lead to more efficient implementations on fixed-connection networks, as exemplified by the following Proposition.

Proposition 5.3.1 *Any BVRAM instruction of work complexity W can be implemented in time $O(\log n)$ on a butterfly network with $n \log n$ nodes, where $n = O(W)$, using only* oblivious *routing algorithms.*

Proof. (Sketch) The arithmetic operations involve no communication at all, thus can be implemented in $O(1)$ steps. The append operation $V_i \leftarrow V_j @ V_k$ only requires a *monotone routing* of the values in V_k. This can be done in $O(\log n)$ steps, using the greedy routing algorithm, see [74], pp. 534. $\mathsf{bmRoute}$ is implemented by a monotone routing, and takes $O(\log n)$ steps with the greedy algorithm. For $\mathsf{sbmRoute}$, suppose first that $\mathsf{length}(V_j) = \mathsf{length}(V_l) = 1$, i.e. $\mathsf{sbmRoute}$ computes the cartesian product of V_i and V_k. Also, suppose that the length of V_i and V_k are powers of 2, namely 2^p and 2^q respectively. Take $n = 2^{p+q}$; then we have 2^p packets residing in the first 2^p rows of a butterfly with 2^{p+q} rows, and we have to route the packet with address $00\ldots0u_{p-1}\ldots u_1u_0$ to all addresses of the form $v_{q-1}\ldots v_1v_0u_{p-1}\ldots u_1u_0$. This is done in q stages, starting with the higher dimension, using the greedy algorithm. In the general case of $\mathsf{sbmRoute}$, we have to replicate a number of smaller sequences. First, round upwards to the closest power of 2 the length of each such subsequence, and spread the sequences such that each sequence of length m starts at an address divisible by m. Next, perform in parallel all replications, as described above. □

When the number n of available processors is less than the number W of elements in an array, then we group $\frac{W}{n}$ adjacent elements of the array in the same processor. The above Proposition can be extended to this case: some instruction of complexity W can be implemented on a butterfly network in $O(\frac{W}{n}\log n)$ steps.

5.4 Efficient Compilation of $\mathcal{MAP}$ without Recursion on the BVRAM

We will deal only with $\mathcal{MAP}$ *without recursion* throughout this Section: for conciseness we shall drop the qualifier, and simply call it $\mathcal{MAP}$, when it is understood from the context.

Theorem 5.4.1 (Compilation Theorem) *For every function expression f in $\mathcal{MAP}$ with time and work complexity T, W, there is a BVRAM, M, such that: $\forall \varepsilon > 0$, there is some program P for M, equivalent to f, having time complexity $T' = O(T)$ and $W' = O(W^{1+\varepsilon})$. Moreover, when $T = O(1)$ then $W' = O(W)$.*

Note that, in contrast to Theorem 5.2.1, the number of registers only depends on f and not on ε.

Before proving the Theorem we make some comments relating it with the compilation technique of [12]. A while-construct can be rewritten as a tail recursive function, hence is *contained*, according to the definition in [12], and therefore the compilation technique described there (for a VRAM, with unbounded many vector registers) preserves its step and work complexity. However, we cannot apply that compilation technique here, because when viewed as tail recursive function, the work complexity of while may increases significantly: we will illustrate this in an example below. This discrepancy between the two forms of iterations is due to the fact that in the tail-recursive version, the result of the iteration is touched n additional times, as the tail recursive function returns from its calls, where n is the number of iteration steps. In the definition of the work complexity for while, these n additional touches are not counted (see Figure 4.7).

To illustrate this, consider the definition of the function $f : \mathbb{N} \to [\mathbb{N}]$ given in Figure 5.2, with the meaning:

$$f(n) = \underbrace{[n, n, \ldots, n]}_{2^{\lceil \log n \rceil} \text{ times}}$$

That is, $\mathsf{length}(f(n)) = 2^{\lceil \log n \rceil} = O(n)$. It first makes n useless iterations, in which each step has work complexity 1, and finally makes $\log n$ iterations in which it doubles the length of the sequence at each step. Hence $T = n + \log n = O(n)$, $W = n + (1+2+4+8+\ldots+n) = O(n)$. However, its tail-recursive counterpart f' has $W = [n+(1+2+4+\ldots+n)] + [(\log n+n)*n] = O(n^2)$, because it has to touch the final result n additional times, corresponding to the n useless recursive calls. Since the result has size n, the total work complexity is $O(n^2)$.

So the tail recursive translation has a higher work complexity than the original while construct. This suggests that while-iterations are in some sense more powerful than recursion, and that they should be included in a parallel programming language. But this certainly proves that we need a stronger compilation technique for $\mathcal{MAP}$ than that described in [12], in order to stay within the lower work complexity. Moreover, we also have to stay within a bounded number of vector registers.

$$
\begin{array}{l}
\textsf{fun } f(n) = \\
\quad \textsf{let fun } p\langle i,j,x\rangle = (i \neq 0 \textsf{ or } j \neq 0) \\
\quad\quad \textsf{fun } g\langle 0,j,x\rangle = \langle 0, j/2, x@x\rangle \;\; | \\
\quad\quad\quad g\langle i,j,x\rangle = \langle i-1, j, x\rangle \\
\quad\quad \textsf{val } \langle \_,\_,x\rangle = \textsf{while}(p,g)\langle n,n,[n]\rangle \\
\quad \textsf{in } x \\
\quad \textsf{end} \\
\textsf{fun } f'(n) = \\
\quad \textsf{let fun } g\langle 0,0,x\rangle = x \;\; | \\
\quad\quad\quad g\langle 0,j,x\rangle = g\langle 0, j/2, x@x\rangle \; | \\
\quad\quad\quad g\langle i,j,x\rangle = g\langle i-1, j, x\rangle \\
\quad\quad \textsf{val } \langle \_,\_,x\rangle = g\langle n,n,[n]\rangle \\
\quad \textsf{in } x \\
\quad \textsf{end}
\end{array}
$$

Figure 5.2: A while-function, and its tail-recursive counterpart

The proof goes through the following steps:

- *Variable Elimination.* We translate $\mathcal{MAP}$ into a rather similar, but variable-free language called *Nested Sequence Algebra*, $\mathcal{NSA}$. The new language only contains *function expressions* $f : s \to t$, i.e. no term expressions. Some term expression e in $\mathcal{MAP}$, of type t and with free variables $x_1 : s_1, \ldots, x_n : s_n$, will be translated into a function expression $g_e : s_1 \times \ldots \times s_n \to t$ in $\mathcal{NSA}$, while some function expression $f : t_1 \to t_2$ in $\mathcal{MAP}$ with the same free variables will be translated into a function $g_f : s_1 \times \ldots \times s_n \times t_1 \to t_2$ in $\mathcal{NSA}$. The primitive functions and the constructs in $\mathcal{NSA}$ correspond roughly to those in $\mathcal{MAP}$, with only two additional primitives: the function $\textsf{distribute}_{left} : s \times [t] \to [s \times t]$, and a function δ, see Figure 5.6. The step and work complexity of functions expressed in $\mathcal{MAP}$ and $\mathcal{NSA}$ are the same.

- *Flattening.* We define a language for flat sequences, called Sequence Algebra $\mathcal{SA}$, and translate $\mathcal{NSA}$ into $\mathcal{SA}$. Namely, for any $\varepsilon > 0$, we show how to translate a function f of $\mathcal{NSA}$ with time and work complexity T, W into an equivalent function in $\mathcal{SA}$ (thus using only flat types), with time and work complexity $O(T)$ and $O(W^{1+\varepsilon})$. Moreover, we can preserve the work complexity too, when f has no loops, i.e. $T = O(1)$. Of course, any function in $\mathcal{SA}$ can be expressed in $\mathcal{NSA}$ with the same time and work complexity.

- We show that $\mathcal{SA}$ and BVRAM are equivalent, in the sense that any function in $\mathcal{SA}$ can be simulated by a BVRAM with the same time and work complexity, and conversely. One direction of this equivalence helps us completing the compilation, while the other direction allows us to perform optimizations at the level of the language $\mathcal{SA}$, instead of BVRAM.

5.4.1 The Nested Sequence Algebra, $\mathcal{NSA}$

The **Nested Sequence Algebra** $\mathcal{NSA}$ is a variable-free version of $\mathcal{MAP}$ without recursion, in the same spirit as FP [8, 106]. It has the same types as $\mathcal{MAP}$, and consists only of *function terms*, $f : s \to t$ (recall that $\mathcal{MAP}$ has both term expressions and function expressions). $\mathcal{NSA}$ is an *algebra of functions*: it consists of some primitive functions, and rules combining simpler function expressions into more complex ones.

There are two notable additional functions in $\mathcal{NSA}$ which where not in $\mathcal{MAP}$: the first is

$$\mathsf{distribute}_{left} : s \times [t] \to [s \times t]$$

with the meaning:

$$\mathsf{distribute}_{left}\langle x, [y_0, \ldots, y_{n-1}]\rangle \stackrel{\text{def}}{=} [\langle x, y_0\rangle, \ldots, \langle x, y_{n-1}\rangle]$$

The second is the function $\delta : (t_1 + \ldots + t_n) \times t \to t_1 \times t + \ldots t_n \times t$, with the meaning $\delta(in_i(x), y) \stackrel{\text{def}}{=} in_i\langle x, y\rangle$, for $i = 1, n$. The language is described by the rules in Figures 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9, and 5.10.

For a function expression $f : s \to t$ in $\mathcal{NSA}$ and S-objects $C : s$ and $C' : t$, we define the ternary evaluation relation $f(C) \Downarrow C'$ in a way similar to the definition for $\mathcal{MAP}$. This time however, the definition is even simpler, because functions in $\mathcal{NSA}$ do not have free variables, hence there is no need for an environment. The time and work complexity $T(f, C)$

$$\frac{}{\Omega^t : unit \to t} \qquad \frac{n \in \mathbb{N}}{n : unit \to \mathbb{N}}$$

$$\frac{}{\Gamma \rhd p : d_p \to c_p}\,(p \in \Sigma) \qquad \frac{}{=: \mathbb{N} \times \mathbb{N} \to \mathbb{B}}$$

Figure 5.3: The Nested Sequence Algebra: constants, external functions

$$\frac{}{\mathsf{id}_t : t \to t} \qquad \frac{f : r \to s \quad g : s \to t}{g \circ f : r \to t}$$

Figure 5.4: The Nested Sequence Algebra: function identity and composition

$$\frac{f_1 : s \to t_1, \ldots, f_n : s \to t_n}{\langle f_1, \ldots, f_n \rangle : s \to t_1 \times \ldots \times t_n} \qquad \frac{}{\pi_i^{t_1 \times \ldots \times t_n} : t_1 \times \ldots \times t_n \to t_i}(i = 1, n)$$

Figure 5.5: The Nested Sequence Algebra: product types

$$\frac{}{in_i^{t_1 + \ldots + t_n} : t_i \to t_1 + \ldots + t_n}(i = 1, n) \qquad \frac{f_1 : s_1 \to t, \ldots, f_n : s_n \to t}{f_1 + \ldots + f_n : s_1 + \ldots + s_n \to t}$$

$$\frac{}{\delta : (t_1 + \ldots + t_n) \times t \to t_1 \times t + \ldots + t_n \times t}$$

Figure 5.6: The Nested Sequence Algebra: sum types

$$\frac{p : t \to \mathbb{B} \quad f : t \to t}{\mathsf{while}(p, f) : t \to t}$$

Figure 5.7: The Nested Sequence Algebra: Iteration

$$\frac{}{[] : unit \to [t]} \qquad \frac{}{\mathsf{singleton} : t \to [t]} \qquad \frac{}{@ : [t] \times [t] \to [t]} \qquad \frac{}{\mathsf{flatten} : [[t]] \to [t]}$$

$$\frac{}{\mathsf{length} : [t] \to \mathbb{N}} \qquad \frac{}{\mathsf{get} : [t] \to t} \qquad \frac{f : t_1 \to t_2}{\mathsf{map}(f) : [t_1] \to [t_2]}$$

Figure 5.8: The Nested Sequence Algebra: Collections

$$\frac{}{\mathsf{zip} : [s] \times [t] \to [s \times t]} \qquad \frac{}{\mathsf{enumerate} : [t] \to \mathbb{N}} \qquad \frac{}{\mathsf{split} : [t] \times [\mathbb{N}] \to [[t]]}$$

Figure 5.9: The Nested Sequence Algebra: Sequences

$$\frac{}{\mathsf{distribute}_{left} : s \times [t] \to [s \times t]}$$

Figure 5.10: The Nested Sequence Algebra: broadcast

and $W(f, C)$ are defined accordingly. The relationship between $\mathcal{MAP}$ without recursion and $\mathcal{NSA}$ is given by the following lemma:

Lemma 5.4.2 *$\mathcal{MAP}$ without recursion and $\mathcal{NSA}$ are equivalent, in the following strong sense:*

1. *For every type context $\Gamma = \{x_1 : s_1, \ldots, x_n : s_n\}$, and term expression e and function expression f in $\mathcal{MAP}$ without recursion and any type derivation $\Gamma \rhd e : t$ and $\Gamma \rhd f : t_1 \to t_2$, there are function expressions $g_e : s_1 \times \ldots \times s_n \to t$ and $g_f : s_1 \times \ldots \times s_n \times t_1 \to t_2$ in $\mathcal{NSA}$ which are "equivalent" to e and f respectively, in the following strong sense. For every environment $\rho = \{x_1 = C_1, \ldots, x_n = C_n\}$:*

   - *$\rho \bullet e \Downarrow C$ iff $g_e\langle C_1, \ldots, C_n\rangle \Downarrow C$, and:*

   $$\begin{aligned} T(g_e, \langle C_1, \ldots, C_n\rangle) &= O(T(\rho, e)) \\ W(g_e, \langle C_1, \ldots, C_n\rangle) &= O(W(\rho, e)) \end{aligned}$$

   - *$\rho \bullet f(C) \Downarrow C'$ iff $g_f\langle C_1, \ldots, C_n, C\rangle \Downarrow C'$, and:*

   $$\begin{aligned} T(g_f, \langle C_1, \ldots, C_n, C\rangle) &= O(T(\rho, f, C)) \\ W(g_f, \langle C_1, \ldots, C_n, C\rangle) &= O(W(\rho, f, C)) \end{aligned}$$

2. *For every function expression $g : t_1 \to t_2$ in $\mathcal{NSA}$ there is some close function expression $\emptyset \rhd f : t_1 \to t_2$ in $\mathcal{MAP}$ having the same meaning, and the same time and work complexity.*

Proof. (Sketch) We only discuss 1, the other statement being simpler. Here, we use induction on the type derivation of $\Gamma \triangleright e : t$ (and $\Gamma \triangleright f : t_1 \to t_2$). We illustrate some of the cases:

Variable

$$\frac{}{x_1 : s_1, \Gamma \triangleright x : s_1}$$

We translate this to $\pi_1 : s_1 \times \ldots \times s_n \to s_1$. Certainly, one needs to keep track of the order of variables in the type context introduces, but this can be done easily.

Tuple

$$\frac{\Gamma \triangleright e_1 : t_1, \ldots, \Gamma \triangleright e_n : t_n}{\Gamma \triangleright \langle e_1, \ldots, e_n \rangle : t_1 \times \ldots \times t_n}$$

Let us abbreviate $s = s_1 \times \ldots \times s_n$. Here we use induction to translate e_i, $i = 1, n$, and get the function expressions $g_{e_i} : s \to t_i$, $i = 1, n$. The the translation of $\langle e_1, \ldots, e_n \rangle$ is $g_{\langle e_1, \ldots, e_n \rangle} \stackrel{\text{def}}{=} \langle g_{e_1}, \ldots, g_{e_n} \rangle$.

Projection

$$\frac{\Gamma \triangleright e : t_1 \times \ldots \times t_n}{\Gamma \triangleright \pi_i^{t_1 \times \ldots \times t_n}(e) : t_i}(i = 1, n)$$

First we use induction to translate e to $g_e : s \to t_1 \times \ldots \times t_n$, next we translate $\pi_i(e)$ to $\pi_i \circ g_e$.

Case

$$\frac{\Gamma \triangleright e : t_1 + \ldots + t_n \quad x_1 : t_1, \Gamma \triangleright e_1 : t \ldots x_n : t_n, \Gamma \triangleright e_n : t}{\Gamma \triangleright \mathsf{case}\ e\ \mathsf{of}\ in_1^{t_1+\ldots+t_n}(x_1) \Rightarrow e_1 \mid \ldots \mid in_n^{t_1+\ldots+t_n}(x_n) \Rightarrow e_n : t}$$

Here we need the function δ. Namely we translate first $e, e_1, \ldots, e_n$ to $g_e : s \to t_1 + \ldots + t_n$, $g_{e_i} : t_i \times s \to t$, for $i = 1, n$. Note that $g_{e_1} + \ldots + g_{e_n}$ has type $t_1 \times s + \ldots + t_n \times s \to t$. We need to compose it with $\delta : (t_1 + \ldots + t_n) \times s \to t_1 \times s + \ldots + t_n \times s$ in order to get $g \stackrel{\text{def}}{=} (g_{e_1} + \ldots + g_{e_n}) \circ \delta$, $g : (t_1 + \ldots + t_n) \times s \to t$. Finally the case-expression will be translated to $g \circ \langle g_e, \mathsf{id}_s \rangle : s \to t$.

Map

$$\frac{\Gamma \rhd f : t_1 \to t_2}{\Gamma \rhd \mathsf{map}(f) : [t_1] \to [t_2]}$$

Here we need the $\mathsf{distribute}_{left}$ function. Namely we first translate f to $g_f : s \times t_1 \to t_2$; then $\mathsf{map}(g_f)$ will have type $[s \times t_1] \to [t_2]$. So the translation of $\mathsf{map}(f)$ is $\mathsf{map}(g_f) \circ \mathsf{distribute}_{left}$, of type $s \times [t_1] \to [t_2]$.

$\square$

This proves:

Theorem 5.4.3 *Any closed function $f \in \mathcal{MAP}$ without recursion, with time and work complexity T, W is expressible in $\mathcal{NSA}$ by some function g with time and work complexity $O(T), O(W)$, and vice versa. Thus, $\mathcal{MAP}$ without recursion and $\mathcal{NSA}$ have the same expressive power.*

5.4.2 The Sequence Algebra, $\mathcal{SA}$

The Sequence Algebra, $\mathcal{SA}$, only has *flat types*. It is essentially $\mathcal{MAP}$ with nested sequences and sum types eliminated. The challenge in designing $\mathcal{SA}$ consists in finding the right set of primitives over flat types which, by combination, can express everything $\mathcal{MAP}$ can. We want $\mathcal{SA}$ to be defined in an inductive way, to enables us to prove, by induction, properties about the functions expressible in $\mathcal{SA}$, e.g. Lemma 5.4.7. $\mathcal{SA}$ stands in the same relationship to $\mathcal{NSA}$ as the Relational Algebra stands to the Nested Relational Algebra [2].

$\mathcal{SA}$ is an algebra of functions, like $\mathcal{NSA}$, i.e. a variable-free language with some primitive functions and some rules for combining them in order to get more complex functions. It is a 2-tiered language, having *scalar types and functions* and *flat types and functions.* It does not have sum types.

Definition 5.4.4 **Scalar types** *are given by the grammar:* $s ::= \mathbb{N} \mid \mathbb{B} \mid \mathbb{D} \mid s \times \ldots \times s$. **Scalar functions** $\varphi : s \to s'$ *are given by the rules of figure 5.11.*

$$\frac{n \in \mathbb{N}}{n : unit \to \mathbb{N}} \qquad \frac{}{\Gamma \rhd p : d_p \to c_p}\,(p \in \Sigma)$$

$$\frac{}{=: \mathbb{D} \times \mathbb{D} \to \mathbb{B}}\text{(for any base type } \mathbb{D}\text{, including } \mathbb{N})$$

$$\frac{}{\mathsf{id}_s : s \to s} \qquad \frac{\varphi : s \to s' \quad \psi : s' \to s''}{\psi \circ \varphi : s \to s''}$$

$$\frac{}{\pi_i : s_1 \times \ldots \times s_n \to s_i}(i = 1, n) \qquad \frac{\varphi_1 : s \to s_1, \ldots, \varphi_n : s \to s_n}{\langle \varphi_1, \ldots, \varphi_n \rangle : s \to s_1 \times \ldots \times s_n}$$

$$\frac{\varphi : s_1 \to \mathbb{B} \quad \psi_1 : s_1 \to s_2 \quad \psi_2 : s_1 \to s_2}{\mathsf{if}\ \varphi\ \mathsf{then}\ \psi_1\ \mathsf{else}\ \psi_2 : s_1 \to s_2}$$

Figure 5.11: Sequence Algebra: scalar functions

For simplicity we assume that all external functions in Σ are over scalar types.

Definition 5.4.5 Flat types *are given by the grammar:* $t ::= [s] \mid t \times \ldots \times t$*, where* s *is a scalar type. The* **Sequence Algebra** $\mathcal{SA}$ *is defined as the collection of expressions* $f : t \to t'$ *described by the rules in figure 5.12.*

Note that there is no boolean type in $\mathcal{SA}$, and our convention $\mathbb{B} \stackrel{\text{def}}{=} unit + unit$ from Chapter 2 does not work, because we don't have disjoint sum types either. Here we adopt a different convention for the $\mathsf{if} - \mathsf{then} - \mathsf{else}$ and while constructs in $\mathcal{SA}$: the booleans will be encoded by $[\mathbb{N}]$, with the convention $\mathsf{true} = [0]$, and false is anything else. This will be consistent with the fact that in $\mathcal{NSA}$ true is encoded by in_1 and false by in_2: hence true becomes $[0]$ and false becomes $[1]$.

The database selection select retains only those elements having a true on their first position, see Example 2.2.6. We abbreviate with $\mathsf{select}' : [\mathbb{B} \times s] \to [s]$ the function selecting the elements with a false:

$$\mathsf{select}'(x) = \mathsf{select}(\mathsf{map}(\lambda\langle b, z\rangle.\langle \mathsf{not}\ b, z\rangle)(x))$$

Example 5.4.6 *Informally we show how to compute combine* $: [\mathbb{B}] \times [s] \times [s] \to [s]$*, where combine*$\langle b, x, y\rangle$ *shuffles the sequences* x *and* y*, according to the flags given by* b*. More precisely* $\mathsf{length}(b) = \mathsf{length}(x) + \mathsf{length}(y) = \mathsf{length}(z)$*, where* z *is the result. Moreover* z_k

$$\frac{}{\Omega^t : unit \to t} \qquad \frac{\varphi : s \to s' \text{ a scalar function}}{\mathsf{map}(\varphi) : [s] \to [s']}$$

$$\frac{}{\mathsf{id}_t : t \to t} \qquad \frac{f : t \to t' \quad g : t' \to t''}{g \circ f : t \to t''}$$

$$\frac{}{\pi_i : t_1 \times \ldots \times t_n \to t_i}(i = 1, n) \qquad \frac{f_1 : t \to t_1 \quad \ldots \quad f_n : t \to t_n}{\langle f_1, \ldots, f_n \rangle : t \to t_1 \times \ldots \times t_n}$$

$$\frac{f : t_1 \to [\mathbb{N}] \quad g_1 : t_1 \to t_2 \quad g_2 : t_1 \to t_2}{\mathsf{if}\ f\ \mathsf{then}\ g_1\ \mathsf{else}\ g_2 : t_1 \to t_2}$$

$$\frac{p : t \to [\mathbb{N}] \quad f : t \to t}{\mathsf{while}(p, f) : t \to t}$$

Figure 5.12: The Sequence Algebra: scalar functions, product types, conditionals, iterations

$$\frac{}{[] : unit \to [s]} \qquad \frac{}{\mathsf{singleton} : unit \to [unit]} \qquad \frac{}{@ : [s] \times [s] \to [s]}$$

$$\frac{}{\mathsf{length} : [s] \to [\mathbb{N}]} \qquad \frac{}{\mathsf{select} : [\mathbb{B} \times s] \to [s]}$$

$$\frac{}{\mathsf{zip} : [s] \times [s'] \to [s \times s']} \qquad \frac{}{\mathsf{enumerate} : [s] \to [\mathbb{N}]}$$

$$\frac{}{\mathsf{bmRoute} : [s] \times [\mathbb{N}] \times [s'] \to [s']} \qquad \frac{}{\mathsf{sbmRoute} : [s] \times [\mathbb{N}] \times [s'] \times [\mathbb{N}] \to [s']}$$

Figure 5.13: The Sequence Algebra: sequences

will be some x_i *when* $b_k = \mathsf{true}$*, and some* y_j *when* $b_k = \mathsf{false}$*. That is combine is a kind of inverse to* select*:*

$$\begin{aligned} \mathsf{select}(\mathsf{zip}\langle b, combine\langle b, x, y\rangle\rangle) &= x \\ \mathsf{select}'(\mathsf{zip}\langle b, combine\langle b, x, y\rangle\rangle) &= y \end{aligned}$$

E.g. 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:*

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

To compute combine in $\mathcal{SA}$*, start by computing* $e = \mathsf{enumerate}(b)$*: in our example* $e = [0, 1, 2, 3, 4, 5, 6]$*. Next we compute:*

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

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* $\mathsf{length}(b)$ *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'*. In our example we get:*

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

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

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

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* $\lambda\langle b, x, y\rangle.\mathsf{if}\ b\ \mathsf{then}\ x\ \mathsf{else}\ y$*.*

As for $\mathcal{MAP}$ and $\mathcal{NSA}$ we define the the time and work complexity for some evaluation $f(C) \Downarrow C'$, where f is a function in $\mathcal{SA}$ and C, C' are flat S-objects. Note that in the absence of a general map there is no nested parallelism in $\mathcal{SA}$.

Although $\mathcal{SA}$ does not contain nested types, like $[\mathbb{N} \times [\mathbb{N} \times \mathbb{N}]]$, it is strong enough to allow such types to be encoded into flat types. The key technical tool for that is to encode some non-flat type $[t]$, where t is a flat type, by some flat type $SEQ(t)$. For this we use *segment descriptors*, as in [12]. Formally, we transform some *flat type* t into another flat type $SEQ(t)$, defined by induction on t:

$$\begin{aligned} SEQ([s]) &\stackrel{\text{def}}{=} [\mathbb{N}] \times [s] \\ SEQ(t_1 \times \ldots \times t_k) &\stackrel{\text{def}}{=} (SEQ(t_1) \times \ldots \times SEQ(t_k)) \times [\mathbb{N}] \end{aligned}$$

Then $SEQ(t)$, although a flat type, can encode values of type $[t]$. Note that in particular $SEQ(unit) = [\mathbb{N}]$: the idea is that a sequence $[\langle\rangle, \ldots, \langle\rangle]$ is encoded by $[n]$, where n is its length.

To see how the encoding works we define by induction on the flat type t the injective function $seq_t : [t] \to SEQ(t)$. It turns out that seq_t is definable in $\mathcal{NSA}$ with time complexity $O(1)$ and linear work complexity. We show below seq_t for the case when t is a sequence type, or a binary product. The case when t is a n-ary product is similar.

$$\begin{aligned} seq_{[s]}([x_0, \ldots, x_{n-1}]) &\stackrel{\text{def}}{=} \langle [s_0, \ldots, s_{n-1}], x_0 @ \ldots @ x_{n-1} \rangle \\ seq_{t \times t'}[\langle x_0, x'_0 \rangle, \ldots, \langle x_{n-1}, x'_{n-1} \rangle] &\stackrel{\text{def}}{=} \langle \langle seq_t[x_0, \ldots, x_{n-1}], seq_{t'}[x'_0, \ldots, x'_{n-1}] \rangle, [n] \rangle \end{aligned}$$

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

The key technical tool for proving the compilation theorem is the following map lemma.

Lemma 5.4.7 (The Map Lemma) . *Let $f : t \to t'$ be some function in $\mathcal{SA}$, and let T, W be the time and work complexity of* map(f) *(recall that* map(f) *is in $\mathcal{MAP}$, but not in $\mathcal{SA}$). Then, for every $\varepsilon > 0$, there exists some function* flatmap$(f) : SEQ(t) \to SEQ(t')$

in $\mathcal{SA}$, of time complexity $T' = O(T)$ and work complexity $W' = O(W^{1+\varepsilon})$, which "simulates" $\mathsf{map}(f) : [t] \to [t']$. *More precisely, $\forall x_0, \ldots, x_{n-1} \in t$, $seq_{t'}(\mathsf{map}(f)[x_0, \ldots, x_{n-1}]) = \mathsf{flatmap}(f)(seq_t([x_0, \ldots, x_{n-1}]))$. Moreover, the structure of the expression $\mathsf{flatmap}(f)$ is independent of ε: the value ε occurs in $\mathsf{flatmap}(f)$ only as the integer constant $\lceil \frac{1}{\varepsilon} \rceil$.*

In addition, when $T = O(1)$, then $W' = O(W)$.

Proof. This is done by induction on the structure of f. When f is map of a scalar function, $\mathsf{flatmap}(f)$ is essentially the same map. When f is some operation on a sequence, then $\mathsf{flatmap}(\mathsf{select})$ is essentially select, $\mathsf{flatmap}(\mathsf{bmRoute})$ is a $\mathsf{bmRoute}$ too, $\mathsf{flatmap}(\rho_2)$ is a $\mathsf{sbmRoute}$, 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 $x = [x_0, \ldots, x_{n-1}]$, on a BVRAM. Note that we could use the same idea as in Theorem 5.2.1, but then the structure of $\mathsf{flatmap}(\mathsf{while}(p, g))$ would depend on ε. Suppose x is in register V_0. We will use only two additional registers, V_1 and V_2, which are initially empty. Let t_i be the number of iterations of $\mathsf{while}(p, g)(x_i)$, and assume without loss of generality that $t_0 < t_1 < \ldots < t_{n-1}$ (we conceptually group all x_i's having the same t_i), which implies $t_i \geq i$. Let $\delta = n^\varepsilon$, $w_i = W(\mathsf{while}(p, g), x_i)$ and $r = \lceil \frac{1}{\varepsilon} \rceil - 1$. It follows that $\delta^r \geq \frac{n}{n^\varepsilon}$. For the moment, assume that in the sequence $x_i, g(x_i), g^{(2)}(x_i), \ldots$, the last value (on position t_i) has the smallest size, denoted by s_i, so $s_i t_i \leq w_i$. The simulation proceeds in $r+1$ stages, numbered $0, 1, \ldots, r$. During stage 0, we start by repeatedly applying $\mathsf{flatmap}(g)$ 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 V_1. Stop this stage when the first $\frac{n}{\delta^r} (\leq n^\varepsilon)$ values have been moved from V_0 into V_1. Due to our assumption $t_0 < t_1 < \ldots$, these are exactly $x_i, i = 0, \frac{n}{\delta^r} - 1$. During this stage we have done some redundant work on V_1, because we have touched the values stored there several times. But this redundant work is less than $O(\frac{n}{\delta^r} W) \leq O(n^\varepsilon W)$. After stage 0 is completed, we move the entire V_1 into V_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 V_1. More precisely, during stage k, all elements x_i, $i = \frac{n}{\delta^{r-k+1}}, \frac{n}{\delta^{r-k}} - 1$ will be moved from V_0 to V_1. At the end of stage k, we move the entire V_1 into V_2. Again we have some redundant work during stage k, due to repeatedly touching the elements already stored in V_1: namely each x_i stored in V_1 could be additionally touched for a total work which is $\leq s_i(\frac{n}{\delta^{r-k}} - \frac{n}{\delta^{r-k+1}}) \leq s_i \frac{n}{\delta^{r-k}}$. But since $i \geq \frac{n}{\delta^{r-k+1}}$, we have that $t_i \geq i \geq \frac{n}{\delta^{r-k}} \frac{1}{\delta}$, 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 V_0 to V_1 at stage k, accounts for only $O(n^\varepsilon W)$. Now by adding up over all r stages, we find that the total additional work done on V_1 is at most $O(\frac{1}{\varepsilon} n^\varepsilon W) = O(n^\varepsilon W)$. There is also some redundant work done on V_2: during all r stages, V_2 is touched only r times, for an additional $O(rW) = O(W)$ work complexity (recall $r = \lceil \frac{1}{\varepsilon} \rceil - 1$). At the end of the last stage, all x_i's ($i = 1, n$) end up in V_2, so V_2 contains the result of $\mathsf{flatmap}(\mathsf{while}(p, g))(x)$.

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 V_2, and progressively moved to V_1, and from here to V_0. The main iteration is done on V_0, i.e. here are the elements on which we apply $\mathsf{flatmap}(g)$ successively. Elements x_i are moved from V_2 to V_1 and V_0 in decreasing order of their $t_i - m_i$. The moments when these elements are moved to V_0 are chosen such that all iterations in V_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.

When $T = O(1)$ then $\mathsf{while}(p, g)$ makes a bounded number of iterations, hence it can be rewritten as an $\mathcal{NSA}$ expression without while. $\square$

Now we can describe the translation of $\mathcal{NSA}$ into $\mathcal{SA}$. We start by flattening the types. For every type t in $\mathcal{NSA}$, we define $COMPILE(t)$ to be a flat type, which encodes t. Namely:

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

Next, for every type t we define the functions $encode_t : t \to COMPILE(t)$ and $decode_t : COMPILE(t) \to t$ in $\mathcal{NSA}$, with time complexity $O(1)$ and work complexity linear in the size of the input, with the property $decode_t(encode_t(x)) = x$, for every $x \in t$. Namely:

$$\begin{aligned}
encode_b(x) &\stackrel{\text{def}}{=} [x] \qquad b = \mathbb{N} \text{ or } b = \mathbb{D} \\
encode_{t_1 \times \ldots \times t_n}\langle x_1, \ldots, x_n\rangle &\stackrel{\text{def}}{=} \langle encode_{t_1}(x_1), \ldots, encode_{t_n}(x_n)\rangle \\
encode_{t_1 + \ldots + t_n}(in_i(x)) &\stackrel{\text{def}}{=} \langle [i], [], \ldots, [], encode_{t_i}(x), [], \ldots, []\rangle \\
encode_{[t]}([x_0, \ldots, x_{n-1}]) &\stackrel{\text{def}}{=} seq_t[encode_t(x_0), \ldots, encode_t(x_{n-1})]
\end{aligned}$$

We leave out the straightforward definition of $decode_t$.

Finally, we can prove:

Proposition 5.4.8 *Let $f : t \to t'$ be some function in $\mathcal{NSA}$ with time and work complexity T, W. Then, for every $\varepsilon > 0$, there is some function $COMPILE(f) : COMPILE(t) \to COMPILE(t')$ in $\mathcal{SA}$ "equivalent" to f, i.e. such that $\forall x \in t$:*

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

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

Proof. (Sketch) By induction on the structure of f. All cases are straightforward, except for the case when $f = \mathsf{map}(g)$, where we use the Map lemma. $\square$

5.4.3 Equivalence of $\mathcal{SA}$ and BVRAM

Proposition 5.4.9 *$\mathcal{SA}$ and BVRAM are equivalent, i.e. any function f in $\mathcal{SA}$ with time and work complexity T, W can be simulated on a BVRAM with the same time and work complexity, and conversely.*

Proof. Simulating some function of $\mathcal{SA}$ by a BVRAM program is easily done by induction on the structure of that function. The converse is slightly more involved. Indeed, let r be the number of registers of a BVRAM M, and h some function in $\mathcal{SA}$ of type $[\mathbb{N}] \times ([\mathbb{N}])^r \to [\mathbb{N}] \times ([\mathbb{N}])^r$ performing *one step* of the program of M (where the program counter is encoded by a singleton sequence, on the first position). By iterating h we indeed achieve the desired time complexity, but not the work complexity, since at each step, the function h touches all r registers. 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 5.4.9 is actually needed for the Compilation Theorem 5.4.1, 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 $\mathcal{SA}$ language.

5.5 An Application

It is not known whether n items can be sorted on an n-node hypercube in $O(\log n)$ parallel steps [74]. However, n items can be sorted in $O(\log n)$ parallel steps on an N-node hypercube, when $N = n^{1+\varepsilon}$ (see [74] Section 3.5.2; and Preparata [86]). We can derive an algorithm with the same complexity as follows:

1. First consider the sorting algorithm *const_sort* of Section 4.6. It sorts in $T = O(1), W = O(n^{1+\varepsilon})$.

2. Compile it using Theorem 5.4.1 to get a BVRAM program with the same time and work complexities.

3. Use Proposition 5.3.1 to derive a sorting algorithm for the butterfly network. The algorithm runs in time $T' = O(T \log W) = O(\log n)$ on a butterfly network with $W \log W = n^{1+\varepsilon} \log n = n^{1+\varepsilon'}$ nodes, with $\varepsilon' > \varepsilon$.

Hence we can use $\mathcal{MAP}$ to design efficient algorithms on a butterfly network.

5.6 A Lower Bound for Permutations on BVRAM

We prove in this Section that a general permutation of a sequence of length n *cannot* be computed on a BVRAM or in $\mathcal{MAP}$ with $T = O(1)$ and $W = O(n)$. In particular this proves that $\mathcal{MAP}$ is not a "trivial" language in which everything can be computed in constant parallel time with no additional cost: many examples of $\mathcal{MAP}$ functions with constant parallel time are given in Section 4.6 but some of them come at the cost of higher-than-linear work complexity.

More significantly, this result shows that the BVRAM, as a model of parallel computation, accounts at least in part for the cost of communication: we cannot compute cheaply an arbitrary permutation on a BVRAM. We will capitalize on this fact in Chapter 6.

To be precise, we will show that two particular permutations cannot be computed "uniformly" with $T = O(1)$ and $W = O(n)$. The two permutations are defined over a base type $\mathbb{D}$:

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

with the following meanings:

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

Both functions can be computed in $T = O(1), W = O(n^{1+\varepsilon})$ using *const_sort* of Section 4.6. We show that they cannot be done *uniformly* in $T = O(1), W = O(n)$. By "uniformly" we mean with no particular knowledge about the elements of type $\mathbb{D}$. Note that 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 will make the uniformity condition precise in the following way. Namely we say a BVRAM is **uniform** in the type $\mathbb{D}$ if no external function in Σ has type $p : t_1 \times \ldots \times t_n \to \mathbb{D}$. When $\mathbb{D} = \mathbb{N}$, then no BVRAM would be uniform in $\mathbb{D}$, since we always assume the existence of external functions like $\dot{-}: \mathbb{N} \times \mathbb{N} \to \mathbb{N}$. Hence we shall assume $\mathbb{D} \neq \mathbb{N}$ in the sequel.

Theorem 5.6.1 *Let M be a BVRAM which is uniform in $\mathbb{D}$. 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.*

Proof. We may assume without loss of generality that the BVRAM M has only two kinds of vector registers: those of type $[\mathbb{D}]$ and those of type $[\mathbb{N}]$. For each moment t, $t = \overline{0,T}$, let v^t be the sequence of type $[\mathbb{D}]$ obtained by concatenating all values of the $[\mathbb{D}]$ 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 **$\mathbb{D}$-trace** of the BVRAM M. We introduce now a definition:

Definition 5.6.2 *A* **computation sequence** *of length T, with input $a : [\mathbb{D}]$, output $b : [\mathbb{D}]$, and constant $c \geq 1$ is a sequence $x^1, x^2, \ldots, x^T$, where $x^t \in [\mathbb{D}]$, $t = 1, T$, 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:*

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

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

3. *There is a constant c such that $\mathsf{length}(x^t) \leq cn$, $\forall t = 0, T$.*

Next we prove that the $\mathbb{D}$-trace can be transformed into a computation sequence by at most doubling its length.

Lemma 5.6.3 *Let $v^0, v^1, \ldots, v^T$ be the $\mathbb{D}$-trace of the BVRAM M with linear work complexity $W = O(n)$. Then there exists a computation sequence $x^0, \ldots, x^{2T}$ 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 $[\mathbb{D}]$ is restricted, due to the uniformity condition. It can be only one of: $[]$, append, select, bmRoute, 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:

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

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} is at most 4 times larger than $size(x^{2t-1})$.

- $V_i \leftarrow \mathsf{select}(V_j, V_k)$. Here V_i and V_k are of type $[\mathbb{D}]$ while V_j is of type $[\mathbb{N}]$. 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 $size(v^t) \leq c'n$, for some constant $c' \geq 0$. It suffices than to choose $c = 4c'$. □

Let $[a_0, \ldots, a_{n-1}]$ be some sequence of type $[\mathbb{D}]$, 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. $0 \leq i_0 < i_1 < \ldots < i_{L-1} < n$ 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 5.6.4 *Let $[a_0, \ldots, a_{n-1}]$ be the output of a computation sequence 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^1 *of the computation sequence.*

Proof. We make induction on T. For $T = 0$ we have $\frac{1}{c^{2^T-1}}n = n$ and the input equals the output. Obviously the whole sequence $a_0, \ldots, a_{n-1}$ is a 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 subsequence of length $\geq dn$ of x^1 is also a subsequece of x^0, and it is certainly $\geq \frac{1}{c^{2^T-1}}n$.

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

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

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 $n_i = \mathsf{length}(y_i)$. Then $\mathsf{length}(y) = \sum_{i=1,k} n_i \geq dn$.

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:

$$\sum_{i=1,k} n_i \geq dn \text{ (see above)} \tag{5.1}$$

$$\sum_{i=1,k} m_i n_i \leq cn \text{ (because } n_i \leq \mathsf{length}(x_i) \text{ and } \mathsf{length}(x^1) \leq cn) \tag{5.2}$$

$$m_i p_i \geq n_i \text{ (because } \textstyle\sum_j n_{ij} = n_i \text{ and } p_i \text{ is max)} \tag{5.3}$$

Then we have the following:

$$\begin{aligned}
(\sum_i p_i)(cn) &\geq (\sum_i p_i)(\sum_i m_i n_i) \text{ (by 5.2)} \\
&\geq (\sum_i \sqrt{p_i m_i n_i})^2 \text{ (by Schwartz's inequality)} \\
&= (\sum_i \sqrt{n_i n_i})^2 \text{ (by 5.3)} \\
&\geq d^2 n^2 \text{ (see above)}
\end{aligned}$$

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

□

Finally we can prove Theorem 5.6.1 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 5.6.3 there exists a computation sequence 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.

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

□

Finally, using the Compilation Theorem 5.4.1 we get:

Corollary 5.6.5 *Neither reverse nor tr can be expressed uniformly in $\mathcal{MAP}$ 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. □

As a final comment, we note that one could contemplate extending $\mathcal{MAP}$ with a primitive *permute* : $[t \times \mathbb{N}] \to [t]$ with the meaning:

$$permute([\langle x_0, i_0\rangle, \ldots, \langle x_{n-1}, i_{n-1}\rangle]) = [x_{i_0}, \ldots, x_{i_{n-1}}]$$

provided that $i_0, i_1, \ldots, i_{n-1}$ is a permutation of $0, 1, \ldots, n-1$. Note that this is the function *Index* of Section 4.6, with the additional assumptions that $i_0, i_1, \ldots, i_{n-1}$ form a permutation. Moreover, we define the time and work complexity of the new primitive *permute* to be:

$$\begin{aligned} T &\stackrel{\text{def}}{=} 1 \\ W &\stackrel{\text{def}}{=} size([\langle x_0, i_0\rangle, \ldots, \langle x_{n-1}, i_{n-1}\rangle]) \end{aligned}$$

The Compilation Theorem 5.4.1 is robust enough to hold in the presence of the *permute* primitive, provided that we extend the BVRAM model with a permutation primitive as well. But in its present form Theorem 5.4.1 is stronger, because it proves a general permutation is not necessary in a BVRAM in order to compile $\mathcal{MAP}$, as long as $\mathcal{MAP}$ doesn't have a permutation primitive itself. This is of importance in view of the high cost of implementing a general permutation on existing massively parallel architectures [70].

5.7 Theoretical Expressive Power

$\mathcal{MAP}$ is a model of parallel computation in its own right. In this Section we establish connections between $\mathcal{MAP}$ and other models of parallel computation, in particular with a CRCW PRAM. A tight connection is not possible, in light of Theorem 5.6.1: a BVRAM *does* account in part for the cost of a permutation, while a PRAM does not. More precisely we prove that, under reasonable assumptions, $\mathcal{MAP}$ can simulate any CRCW PRAM with the same time complexity and with only a polynomial increase in the work complexity. Let $\mathit{CRCW\text{-}TIME\text{-}PROC}(T(n), P(n))$ be the set of functions computable on a CRCW PRAM in time $T(n)$ using $P(n)$ processors, and $\mathcal{MAP}\text{-}\mathit{TIME\text{-}WORK}(T(n), W(n))$ the set of functions expressible in $\mathcal{MAP}$ with time and work complexity $T(n), W(n)$. We will show that

CRCW-TIME-PROC$(T(n), W(n)) \subseteq \mathcal{MAP}$-*TIME-WORK*$(O(T(n)), (W(n))^{O(1)})$. For this we will relate $\mathcal{MAP}$ functions with Alternating Turing Machines and then rely on the relationship between Alternating Turing Machines and CRCW PRAM due to Ruzzo and Tompa. As a consequence $\mathcal{MAP}$ can express all functions in NC in time $T = O(\log^k n)$ and with work complexity $W = O(n^{O(1)})$.

Recall that an *Alternating Turing Machine* [29] ATM is a Turing Machine with four kinds of states: an *accepting* state q_a, a *rejecting* state q_r, *existential* and *universal* states. We call a configuration existential or universal, if its state is existential or universal. Following [90], we view a computation on a ATM as a tree. The nodes are labeled by configurations, subject to the following conditions: (1) each node labeled with an *existential* configuration α has exactly one successor, and the label of the successor is β, with $\alpha \vdash \beta$. (2) for each node labeled with a *universal* configuration α, the number of successors is equal to the cardinality of the set $s = \{\beta \mid \alpha \vdash \beta\}$; moreover, each successor is labeled by a distinct configuration $\beta \in s$. As in [90], we use a *random access input* variation in treating the inputs of the ATM. Namely we assume a special "index" tape in addition to the work tape, and a fifth kind of states called *read states*. Whenever M enters a read state with a, i written on the index tape, it halts and accepts iff the symbol i of the input is the character a; else it rejects. We say that some ATM M *accepts* some word $w \in \{0,1\}^*$ if there is some computation tree whose root is labeled by the initial configuration on w, and whose leaves are all accepting configurations. We say that M accepts an input w in space $S(n)$ and with $T(n)$ alternation (where n is the length of w) if there is some accepting computation tree for w whose nodes are labeled with configurations using at most $\leq S(n)$ tape space, and which has at most $T(n)$ alternations (changes from existential to universal states and vice versa) on each path from the root to some leave. *ATM-ALT-SPACE*$(T(n), S(n))$ denotes the class of functions acceptable by some ATM with $T(n)$ alternations, in space $S(n)$.

As models for parallel computations, the ATM's are interesting in the light of the following theorem from [93], where it is credited to Ruzzo and Tompa. A pair of functions $(T(n), S(n))$ is called *suitable* in [93] iff $S(n) \geq \log n$, $\log T(n) \leq S(n) \leq T(n)$, and some rather technical constructibility conditions are fulfilled.

Theorem 5.7.1 (Ruzzo and Tompa) *Let $(T(n), S(n))$ be suitable. Then:*

$$\textit{ATM-ALT-SPACE}(O(T(n))), O(S(n)) = \textit{CRCW-TIME-PROC}(O(T(n)), 2^{O(S(n))})$$

Here, the arithmetic operations on the PRAM are restricted to $+$ and $-$, i.e. $\Sigma = \{+, -\}$.

Here we prove:

Proposition 5.7.2 *Let $(T(n), S(n))$ be suitable. Then the following holds:*

$$\textit{ATM-ALT-SPACE}(O(T(n)), O(S(n))) \subseteq \mathcal{MAP}\textit{-TIME-WORK}(O(T(n)), 2^{O(S(n))})$$

Proof. (Sketch) Let $f \in \textit{ATM-ALT-SPACE}(O(T(n)), O(S(n)))$. Let M be the ATM M computing f. M has $W(n) = 2^{(S(n))}$ states, for some input of length n. We compute all these states in $\mathcal{MAP}$, as well as the successor relation, $\alpha \vdash \beta$, using objects of size $O(W(n)^2)$. Next, we compute the transitive closure of $\vdash$ in time $O(\log W(n))$ and with work complexity $W(n)^{O(1)}$, and only retain those pairs of states $\alpha \vdash^* \beta$ for which the last transition is an alternation: call z the sequence of these pairs. Now we start constructing the set x of all accepting configurations, from the leaves to the root. We initialize x with the sequence of all final accepting configurations: these will have the number of alternations $T(n)$. Next, we perform for $T(n)$ steps the following: at the odd steps, we add to x the sequence $[\alpha \mid \exists\beta \in x.(\alpha, \beta) \in z]$. At the even steps, we add $[\alpha \mid \forall\beta \in x.(\alpha, \beta) \in z]$. After $T(n)$ steps we stop, and check whether the initial configuration is in x. Thus, the total time complexity is $\log W(n) + T(n) = O(T(n))$. For the work complexity, note that each step has work complexity $W(n)^{O(1)}$, hence the total work complexity is $T(n)W(n)^{O(1)} = O(W(n)^{O(1)})$. □

Combining theorem 5.7.1 with proposition 5.7.2 we immediately obtain:

Theorem 5.7.3 *For $T(n), W(n)$, that are* suitable *(in the sense of [93]), we have:*

$$\textit{CRCW-TIME-PROC}(T(n), W(n)) \subseteq \mathcal{MAP}\textit{-TIME-WORK}(O(T(n)), W(n)^{O(1)})$$

For the converse, we first use the Compilation Theorem 5.4.1 to compile a $\mathcal{MAP}$ program on a BVRAM. Next we use the following lemma, which is a minor variation of Theorem 4, pp. 196 in [12] (the BVRAM's version of Brent's scheduling principle, as it were):

Lemma 5.7.4 *Any function expressible on a BVRAM with complexities T, W can be simulated on a CREW PRAM* with plus-scan primitives *using P processors with asymptotic complexity $O(T + W/P)$.*

Proof. (Sketch) Use the simulation of the BVRAM on a CREW with plus-scan primitives, in the spirit of [12]. We need a CREW instead of a EREW in order to simulate bmRoute and sbmRoute: a bmRoute could replicate a single value many times, in one parallel steps, so we need concurrent read to simulate this on a PRAM. □

Implementing the plus-scan primitives introduces an additional logarithmic slow-down factor. Hence we get:

Theorem 5.7.5 *Suppose all arithmetic operations in Σ are in NC and include $+, \dot{-}$. Then:*

$$NC = \mathcal{MAP}\text{-}TIME\text{-}WORK(\log^{O(1)} n, n^{O(1)})$$

That is, the functions in NC can be expressed in $\mathcal{MAP}$ in a natural way.

Chapter 6

Implementation on the LogP Model

6.1 The LogP Model

The LogP model of parallel computation was defined by Culler et al. [39]. It is intended to capture 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 of computations, like zero communication delay or infinite bandwidth. Karp, Sahay, Santos, and Schauser [69] present provably optimal algorithms for the LogP model, for simple problems like broadcast and summation.

The model consists of a fixed number, P, of independent nodes, each with a processor and local memory, connected by a network allowing point-to-point communications: each processor i may send messages to each other processor j. The L, o, and g in the name LogP stand for the following parameters of the model:

- L is the *latency*: this is an upperbound of the delay of a word of a message to be delivered from the source processor to the destination processor. That is, a word sent by the source processor at time t will arrive at the destination processor no later than

$t + L$.

- g is the *gap*, defined to be the amount of time which a node needs to wait between sending or receiving two consecutive words of a message. That is, if a processor has sent a word at time t, he may not send the next word until $t + g$: it takes $(s-1)g$ time to send a message of size s. We assume here that the node has both an input and an output port, that is, that a processor may send and receive simultaneously.

- o is the *overhead*, defined to be the amount of time a processor is engaged in preparing the communication network for sending or receiving one message. For each message, the overhead is spent both at the source and at the destination processor, and is independent on the size of the message.

In addition the LogP model assumes that the network is of *finite capacity*, in that for any two processors i, j, at most $\lceil L/g \rceil$ words can be in transit from i to j: if processor i attempts to send another word exceeding this capacity he will stall.

We have modified slightly the original model in a couple of ways. First we allow variable-length messages instead of fixed-length ones, and, following [14] we measure message lengths in words rather than bytes. Secondly we assume that the network interface at each node has a local processor and DMA. Each message will contain enough information in its header to allow the network interface to compute the destination address in the node's local address space. The processor will be notified about the incoming message only *after* the entire message has been placed in memory, that is $(s-1)g$ units of time after the first word of the message has arrived, where s is the size of the message. The original model is recaptured for messages of size $s = 1$.

To see how the L, o, and g parameters contribute to the total communication time, assume that some source processor decides at time t to send a message of s words to some destination processor. Figure 6.1 shows the activities performed by the sender and receiver. Note that both the sender and the receiver processor will each spend o units of time for this communication, between $t, t+o$, and $t+o+L+(s-1)g, t+o+L+(s-1)g+o$ respectively: both are free to perform other computations otherwise. The sender will be able to send the

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 processor busy
$t+o+L+(s-1)g+o$	ready

Figure 6.1: Sending a message of size s in the LogP model

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 model becomes the *postal model* of [10]. Moreover, every LogP model with $o=0$ is equivalent to a postal model, because we can normalize g to 1 [69].

6.2 Data Partitioning

The basic principle underlying the implementation of BVRAM M on the LogP model is *data partitioning* [46, 45, 102]. That means that each flat sequence x stored in one of the vector registers of a BVRAM M is distributed over the P nodes of the LogP model. Data partitioning is sometimes called *declustering*, as opposed to *clustering*, see Subsection 6.5.8.

DeWitt and Gray [46] describe several data partitioning techniques currently used in parallel databases systems. *Round robin* consists in placing the elements $x_0, x_P, x_{2P}, \ldots$ at processor 0, the elements $x_1, x_{P+1}, x_{2P+1}, \ldots$ at processor 1, etc. This method guarantees equal load of the processors. In *range partitioning* and *hash partitioning* the processor in which some element x_i is stored is dependent on the value of x_i. In absence of domain-specific knowledge, these methods cannot guarantee uniform load balance of the processors. Moreover, these techniques seem more suitable for sets, and less for sequences, where the order of the elements matters.

We choose here a different data partitioning method: *block partitioning*. The idea is to

Processor	Elements
0	$x_0, x_1, \ldots, x_{n_1-1}$
1	$x_{n_1}, \ldots, x_{2n_1-1}$
$\ldots$	$\ldots$
$m-1$	$x_{(m-1)n_1}, \ldots, x_{mn_1-1}$
m	$x_{mn_1} \ldots x_{mn_1+n_0-1}$
$m+1$	$x_{mn_1+n_0}, \ldots, x_{mn_1+2n_0-1}$
$m+2$	$x_{mn_1+2n_0}, \ldots, x_{mn_1+3n_0-1}$
$\ldots$	$\ldots$
$P-1$	$x_{mn_1+(P-m-1)n_0} \cdots x_{mn_1+(P-m)n_0-1}$

Figure 6.2: Block Partitioning

distribute the elements of some sequence $x = [x_0, x_1, \ldots, x_{n-1}]$ evenly among the processors. Let $n_0 \stackrel{\text{def}}{=} \lfloor \frac{n}{P} \rfloor$, $n_1 \stackrel{\text{def}}{=} \lceil \frac{n}{P} \rceil$, $m \stackrel{\text{def}}{=} (n \ mod \ P)$. Then under block partitioning the elements of the sequence x will be distributed over the processors as shown in Figure 6.2. Note that $mn_1 + (P-m)n_0 = P$.

As round-robin, block partitioning guarantees equal distribution of the elements over the processors. But, as we explain below, the communication patterns needed to implement the BVRAM instructions for block partitioning have nicer properties than those for round-robin. For that reason, we choose block partitioning as our data partitioning method.

Thus, processor i will hold the following information about the sequence x stored in some vector register V_j:

- n (= $\mathsf{length}(x)$), the total length of x, and
-
    - $[x_{in_1}, \ldots, x_{in_1+n_1-1}]$, if $i < m$, or
    - $[x_{mn_1+(i-m)n_0}, \ldots, x_{x_{mn_1+(i-m)n_0+n_0-1}}]$, if $i \geq m$.

The problem of implementing the BVRAM instructions on the LogP model consists in orchestrating efficiently the messages which need to be exchanged by the P nodes in order to implement each instruction in turn. A key concept in our search for an efficient implementation is that of a *monotone communication*, discussed next.

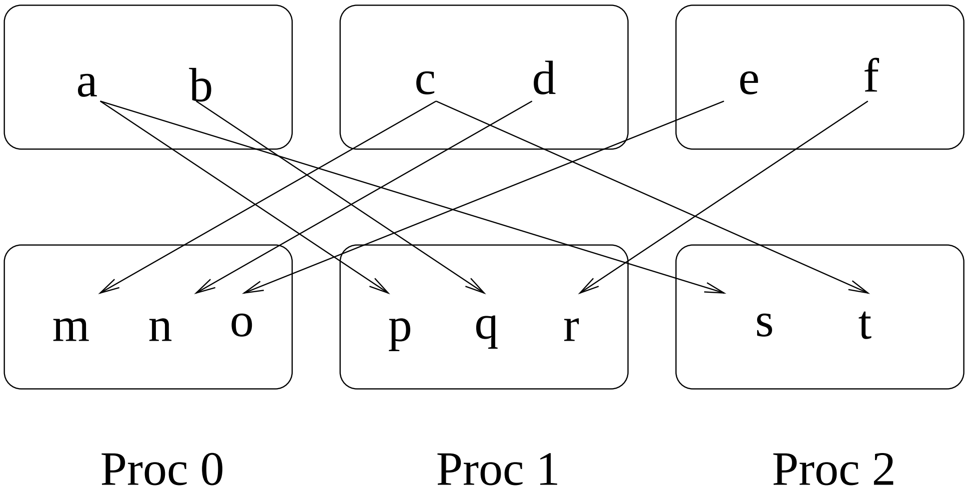
6.3 Monotone Communications on the LogP Model

Assume each node of a LogP model holds a number of data items, we call them *packets*, each of which needs to be send to one of several other nodes. We call this a *communication problem.*

Definition 6.3.1 *A* **communication problem** (X, Y, π, σ) *for the LogP model consists of two disjoint sets of* source packets X *and* destination packets Y, *a* processor *function* $\pi : X \cup Y \to \{0, 1, \ldots, P-1\}$, *and a function* $\sigma : Y \to X$.

The intended meaning is that a source packet x has to be sent to a destination packet y iff $\sigma(y) = x$. When $\pi(x) = i$ and $\pi(y) = j$, this means that a message has to be sent from processor i to j; but the message does not need to travel directly from i to j, it could take a longer path with more intermediate nodes. In the case when $i = j$ then no communication is needed for the destination packet y, and we can simply drop y from Y: without loss of generality we will assume that for any communication problem $\pi(\sigma(y)) \neq \pi(y)$, $\forall y \in Y$. Also, source messages x which are not in the range of σ need not be sent at all: again without loss of generality we shall assume that σ is surjective. Finally, note that when $y \neq y'$, $i = \pi(y) = \pi(y')$ and $\sigma(y) = \sigma(y') = x$, we have a packet x which has to be sent to two packets y, y' in the same destination processor i: obviously a single message suffices to be sent to i, and i will copy the incoming packet in both y and y'. Therefore without loss of generality we will assume that $y \neq y' \wedge \pi(y) = \pi(y') \Longrightarrow \sigma(y) \neq \sigma(y')$.

Example 6.3.2 *To illustrate the concept, we give a trivial example. Let* $P = 3$, $X = \{a, b, c, d, e, f\}$, *and* $Y = \{m, n, o, p, q, r, s, t\}$. *Assume* $\pi(a) = \pi(b) = 0$, $\pi(c) = \pi(d) = 1$, $\pi(e) = \pi(f) = 2$ *and* $\pi(m) = \pi(n) = \pi(o) = 0$, $\pi(p) = \pi(q) = \pi(r) = 1$, $\pi(s) = \pi(t) = 2$. *Let* $\sigma(m) = c, \sigma(n) = d, \sigma(o) = e, \sigma(p) = a, \sigma(q) = b, \sigma(r) = f, \sigma(s) = c, \sigma(t) = c$, *see Figure 6.3. Then* (X, Y, π, σ) *is a communication problem. In a solution to it, we could*



Each processor is represented twice: once with its source packets, and once with its destination packets. The arrows represent the function σ, and not necessarily the messages sent.

Figure 6.3: A simple communication problem

send packet a from node 0 to node 1, and then from node 0 to node 2. Alternatively, we could send it from node 0 to node 2, and then from node 2 to node 1.

We are interested in solutions to the comunication problem which minimize the time needed by all packets to reach their destinations. One obvious way of finding an optimum solution is to consider all possible schedules and to take the one with minimum running time. This is not acceptable for us, because we want solutions which are "local", i.e. for which the decisions at processor i are based only on the packets available at processor i.

For simplicity we assume that all packets have a fixed size s.

Definition 6.3.3 *We say that a communication problem (X, Y, π, σ) is* **monotone** *iff there exists total order relations on X and Y, such that $\pi : (X, \leq) \to (\{0, 1, \ldots, P-1\}, \leq)$, $\pi : (Y, \leq) \to (\{0, 1, \ldots, P-1\}, \leq)$, and $\sigma : (X, \leq) \to (Y, \leq)$ are monotone functions.*

The communication problem in Example 6.3.2 is not monotone: intuitively, it is monotone when it can be drawn such that the arrows do not intersect, see Figure 6.4 below.

Definition 6.3.4 *A communication problem is* **one-to-one** *if σ is injective. Otherwise, we call the communication problem* **one-to-many**.

In a one-to-one communication problem packets are not replicated, and intuitively is should be more efficient to send them from the source directly to the destination without any detour. Theorem 6.3.7 below proves that this is actually the case, for monotone, one-to-one communication problems.

Note that in a one-to-one communication problem there could still be several packets being sent from some source processor i to the same destination processor j.

Monotone one-to-one communication problems are easier to solve efficiently on the LogP model than arbitrary communication problems, as we show in Theorem 6.3.7. This suggests that, in general, monotone communication problems admit efficient implementations on the LogP model.

Example 6.3.5 *The* single-item *broadcast problem [69] is a monotone communication problem. Here the processor 0 has to send one item to each other processor. Formally $X = \{x\}$, $Y = \{y_0, \ldots, y_{P-1}\}$ with the order relation $y_0 < y_1 < \ldots < y_{P-1}$, $\pi(x) = 0$, $\pi(y_i) = i$, and $\sigma(y_i) = x$ for $i = 0, P-1$. This is not a one-to-one communication problem problem.*

Note that the k-item broadcast problem [69], in which k data items residing in processor 0 have to be broadcast to all other processors is *not* a monotone communication problem. The single-item broadcast problem admits a simple, provably optimal solution on the LogP model [69]. By contrast, the solution presented in [69] to the k-item broadcast problem is far more complex, and only within an additive constant from an optimal solution.

We call $(\{x\}, Y, \sigma)$ a **broadcast problem**. Note that if memory copying time is negligible, then the broadcast problem can be solved optimally by (1) performing the single-item broadcast algorithm of [69] (see also Subsection 6.5.3 below), and (2) performing a local replication of the single item at each processor holding more than one destination packet.

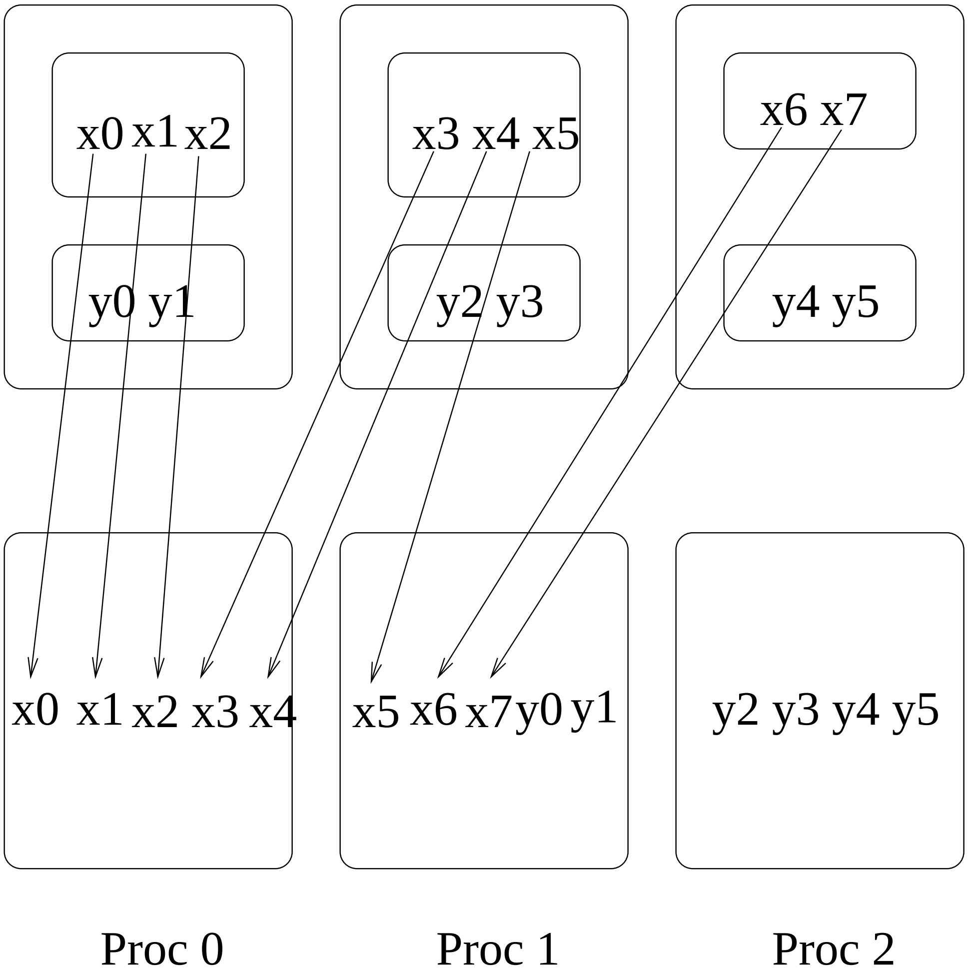
Example 6.3.6 *Suppose the sequences* $x = [x_0, \ldots, x_7]$, *and* $y = [y_0, \ldots, y_5]$ *are block-partitioned on* $P = 3$ *nodes, see Figure 6.4, and suppose we want to compute* $z = \mathsf{append}\langle x, y\rangle$, $z = [z_1, z_1, \ldots, z_{14}]$. *This problem can be decomposed into two monotone communication problems. The first has* $X = [x_0, \ldots, x_7]$, $Y = [z_0, \ldots, z_7]$, *with* $\sigma(z_i) = x_i$, $i = 0, 7$. *The second has* $X = [y_0, \ldots, y_5]$, $Y = [z_8, \ldots, z_{14}]$, *with* $\sigma(z_i) = y_{i-6}$, $i = 8, 14$. *In fact both communication problems are simpler, because some of their source packets reside in the same nodes as the destination packets, therefore have to be removed.*

We shall present an optimal algorithm for the one-to-one, monotone communication problem (X, Y, π, σ). First we establish a lower bound for it. For some processor i, let S_i be the total size of all source packets at processor i and S'_i be the total size of all destination packets at processor i. Define:

$$\begin{aligned}
T_i^{send} &\stackrel{\text{def}}{=} o + (S_i - 1)g + L + o \\
T^{send} &\stackrel{\text{def}}{=} \max(\{T_i^{send} \mid i = 0, P-1\}) \\
T_i^{receive} &\stackrel{\text{def}}{=} o + L + (S'_i - 1)g + o \\
T^{receive} &\stackrel{\text{def}}{=} \max(\{T_i^{receive} \mid i = 0, P-1\}) \\
T^{opt} &\stackrel{\text{def}}{=} \max(T^{send}, T^{receive})
\end{aligned}$$

Obviously T^{send} is a lowerbound for the communication problem, because if processor i were to spend all its time only sending out messages, the last word send would arrive at the destination node at time T_i^{send}. Similarly, $T^{receive}$ is a lowerbound too, because if processor i would spend all its time receiving, it would need $T_i^{receive}$ time. Hence T^{opt} is a lowerbound for our problem: we give below an algorithm which runs time T^{opt}, under reasonable assumptions, hence is optimal. We start by describing these assumptions.

In an optimal solution to a communication problem we expect a processor to interleave its sends and receives. Recall that each packet has size s. If a processor initiates sending a message at time t, it will actually send it between $t + o$ and $t + o + (s-1)g$. Since the next message would be actually sent no earlier than $t + o + sg$, the processor is idle between $t + o$



This figure illustrates the first of the two monotone communications needed to implement append$\langle [x_0, x_1, \ldots, x_7], [y_0, y_1, \ldots, y_5] \rangle$. The arrows having the same processor both as source and as target are represented only for illustration, and are not actually part of the communication problem.

Figure 6.4: Appending two block-partitioned sequences

and $t + sg$: in the meantime it could *receive* a message which requires another overhead o, provided that $2o \leq sg$. If $2o > sg$, then the processor is not fast enough, or sends too small messages, and cannot keep both input and output ports of the network busy at their full capacity: we shall assume for the sequel that $2o \leq sg$. For our optimal algorithm however we need a slightly stronger assumption: $3o \leq g$. We leave open the question whether an optimal communication algorithm exists for $2o \leq sg < 3o$.

The condition $3o \leq sg$ is not too restrictive. For existing multiprocessor architectures, o and g are roughly equal, at worst $o \cong 2g$, so it suffice to assume that the packets have size ≥ 6 words.

Theorem 6.3.7 *Suppose $3o \leq sg$, where s is the size of the packets. Then there exists an optimal, local algorithm for the monotone one-to-one communication problem.*

Proof. Let $X = \{x_0, \ldots, x_m\}$, $Y = \{y_0, \ldots, y_n\}$ be the set of outgoing and incoming packets respectively. Suppose they are ordered by $x_0 < x_1 < \ldots < x_{m-1}$ and $y_0 < y_1 < \ldots < y_{n-1}$. Since the communication problem is one-to-one, $\sigma : Y \to X$ has an inverse $\rho : X \to Y$: $\rho(x_k) = y_l$ means that the packet x_k has to be sent to y_l. Let $i = \pi(x_k)$ and $j = \pi(y_l)$ be the processors holding x_k and y_l respectively. Let $y_{l_0}, y_{l_0+1}, \ldots, y_l, \ldots, y_{l_1}$ be all the incoming packets at processor j. For each x_k, let $\delta(x_k)$ be the offset of $\rho(x_k)$ at the destination processor, which is $\pi(\rho(x_k))$: in our notation, $\delta(x_k) = l - l_0$. We make the additional assumption that processor i knows not only that it has to send x_k to processor j, but also the offset $\delta(x_k)$.

We will describe the algorithm at processor i. Let x_{k_0} be the first outgoing packet at processor i. Divide all outgoing packets at i into blocks according to their destination processor, see Figure 6.5.

Because the communication is monotone, the processors $j_1, j_2, \ldots, j_{\lambda-1}$ will not receive any other messages except from processor i: hence there will be no message contention at these destinations, no matter in which order processor i sends these messages. Only the first and last block should be send out with care, to avoid contention at j_0 and j_λ with messages

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

The source packets at processor i are partitioned into blocks, according to their destination processor. Since the communication problem is monotone, only the first and the last block may have destination processors which coincide with packets sent by other processors.

Figure 6.5: The *send*-blocks in the optimal monotone, one-to-one algorithm

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

  let t be the time needed to send the m blocks above
  let $d = \delta(x_{k_0})$
  let $t' = dg$
  W: if $t' > t$ then wait $t' - t$ seconds else continue
  send block 0 to processor j_0
end

Figure 6.6: The *send*-thread in the optimal monotone, one-to-one algorithm

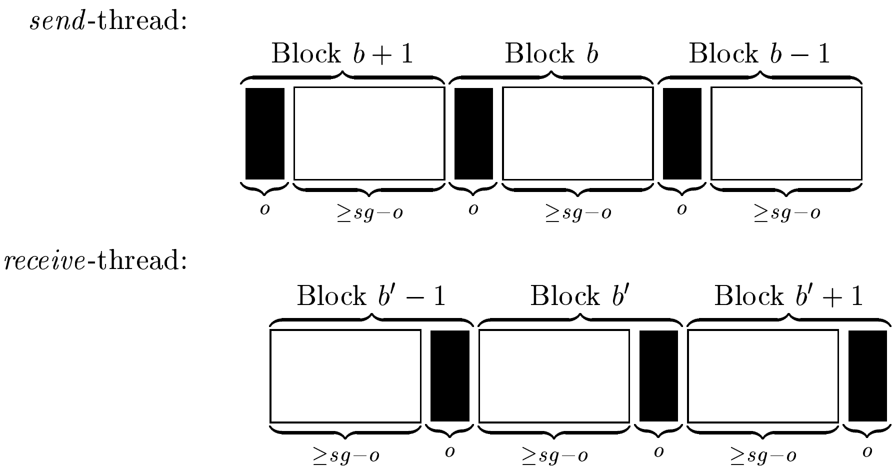
coming from processors $i-1$ and $i+1$ respectively. Note that when $\lambda = 0$, then both processors $i-1$ and $i+1$ will send messages to j_0. In fact, an arbitrary large number of processors may send messages to the same j_0; but at most two of them (the first and the last one) may send messages to other processors besides 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 processor i to send its blocks in the right-to-left order, i.e. to processors $j_\lambda, j_{\lambda-1}, \ldots, j_1, j_0$. Moreover, i may proceed at full speed, except for the last block j_0: here it needs to wait first for processor $i-1$ to finish sending its messages to j_0. We explain the details below.

The algorithm consists of two threads: a send-thread and a receive-thread. The send-thread is described in Figure 6.6.

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

First we will analyze this algorithm for the postal model ($o = 0$), where the messages don't need to be received explicitly, hence there is no need for a receive-thread. For this case we prove that our algorithm is optimal, as follows. Let i be the process sending out the last word of the last message, at time $T - L - 0$ (which is $T - L$, because $o = 0$), where T is the running time of the algorithm. There are two cases: (1) processor i didn't have to wait at line W (that is $t' \leq t$). Then processor i has sent out its messages at full speed, and therefore $T = T_i^{send}$ is the optimal running time. (2) processor 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 $T = T_j^{receive}$ is again the optimal running time.

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 for processor time with the send-thread. Here 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. See Figure 6.7 Let t_w be the waiting time, $t_w < 2o$. As argued earlier, the receive-thread will have enough time to receive at least one message in the time interval in which the sent-thread is idle, because $sg \geq 2o$. Now we argue that the t_w delay of the receive-thread will not affect in any way the next incoming message, that is, that the receive thread, although delayed by t_w, will be ready to receive the next message in time. Indeed, suppose the last word of the current incoming message y arrived at time t. Ideally the receive-thread would process y between t and $t + o$. Instead he is delayed and processes it between $t + t_w$ and $t + t_w + o$. The s words of the next incoming message arrive at the moments $t + g, t + 2g, \ldots, t + sg$. So it suffices to observe that $t_w + o \leq sg$ (here we use the fact that $3o \leq sg$), hence the receive-thread will finish with y in time to handle the next incoming message between $t + sg$ and $t + sg + o$.



The send-thread and the receive-thread of some processor represented independently. The black boxes represent the processor overhead: two black boxes cannot overlap. In the *send*-thread, a white and a black box together (in this order) represents the total time in which a message arrives: the processor however will start preparing the next message before the current one has been completely sent. The receive-thread will have to delay some of its black boxes by $t_w < 2o$ in order not to delay the *send*-thread. Even if a black box is delayed, there is still time for the *receive*-thread to handle the next black box in time, when $3o \leq sg$.

Figure 6.7: Interference between the *send*-thread and the *receive*-thread

We analyze the running time using a similar argument to that used in the case of the postal model. Namely let i be the processor sending out the last word of the last message at, say, time t. As before, we distinguish two cases. (1) The send-thread of processor i didn't wait at all. Then processor i has sent out its messages at full speed. Moreover, the destination processor j of the last message will receive its last word at time $t + L$: the send-thread on processor j has finished before that, so the receive-thread will handle this message without delay, and the algorithms ends at time $t + L + o$, which is T_i^{send}, hence the algorithm is optimal. (2) The send-thread of processor i had to wait before sending out its last message. Let j be the destination processor 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 processor j have been sent already. So the running time is $T_j^{receive}$, which is optimal.

□

Using the result in Theorem 6.3.7, a one-to-many monotone communication problem (X, Y, σ) (where $(X, \leq)$ and $(Y, \leq)$ are ordered) admits the following solution: (1) perform a one-to-one monotone communication (X, Y_0, σ), where $Y_0 \stackrel{\text{def}}{=} \{y_0 \mid y_0 \in Y \wedge y_0 = \min(\{y \mid \sigma(y_0) = \sigma(y)\})\}$, and (2) perform in parallel the family of broadcast problems $\{(\{y_0\}, \{y \mid \sigma(y_0) = \sigma(y)\}, \sigma) \mid y_0 \in Y_0\}$. Note that each processor may participate in at most two parallel broadcasts (and in this case, for one of them it will be the source of the broadcast), so there will be no conflict between the parallel broadcasts. While we do not know whether this algorithm is optimal, it is reasonably efficient, simple and relatively easy to implement. We consider it to be evidence for our claim that monotone communications can be implemented more efficiently on the LogP model than arbitrary communications.

6.4 Implementation Principles

Existing massively parallel multiprocessors have networks which are a 1-2 order of magnitudes slower than their processors: typical values for L, o, and g are 80-100 processor instructions [14]. In order to overlap communications and computations, we borrow from the idea of *active messages* communication mechanism introduced by T. von Eicken, D. Culler, S. Goldstein, and K. Schauser [105].

Originally, active messages are defined as messages in which the control information at the head of a message is the address of a user-level procedure that will extract the message from the network and integrate it in the ongoing computation. This reduces operating system overhead and eliminates the need for the message to be copied into an operating system buffer before being copied it into the user space.

In our implementation of the BVRAM on the LogP model we observe that many of the messages arriving at a node do not need immediate attention of the processor. Therefore we adopt the convention that each message arriving at some node i will contain in its header enough information for the network interface to determine the address in the user space where the message has to be stored. The DMA of the network interface can store the message without interrupting the processor. We assume the network interface to have a local processor giving it enough computing power to compute the destination address from the message header, and to synchronize with the main processor for access to common datastructures: e.g. the second-generation ATM network interface SBA-200 produced by Fore Systems consists of an Intel i960 processor, a 256Kbytes of memory, and a DMA. We will explain in the sequel the computations which the network interface needs to perform in order to determine the destination address in the user space.

A typical BVRAM instruction I, say $V_1 \leftarrow f(V_2, V_3)$, is implemented at node i in two phases: a *send*-phase of data residing in V_2 and V_3 to other processors, and a *receive*-phase of data from other processors, to be stored locally in V_1. After initiating all the *send*s, processor i may proceed executing subsequent BVRAM instructions without waiting for the *receive*s. As messages for the BVRAM instruction I arrive from other processors, they are being placed in their corresponding memory location without interrupting the processor. When processor i reaches a subsequent BVRAM instruction J using V_1 as one of its input registers, then it *waits* for all incoming messages for V_1 to be completed. This technique hides the network's latency when instructions I and J are apart. Well understood compilation techniques like *instruction scheduling* can be used to reorder the BVRAM instructions such as to keep dependent instructions far apart.

This technique requires more subtle memory management techniques. Recall that processor i only *initiates* sending a message and does not actually wait for it to be sent: it only places a request in a FIFO queue managed by the network interface. Thus, in our example, the memory allocated to the registers V_2 and V_3 at node i has to be kept untouched until all *send*s are completed. However, between instructions I and J another BVRAM instruction may use one of them, say V_2, as output register: the incoming messages for *that* instruction may destroy the content of V_2 before all the *send*s for instruction I have been completed. We solve this problem with *time-stamps*. Namely we qualify every BVRAM vector register V_j with (1) its number j, and (2) the number t of the BVRAM instruction which computed V_i. We call t a **time-stamp**. It is a number between 0 and the total number of BVRAM instructions executed in that particular run. Note that t can be larger that the size of the BVRAM program, since the program may contain loops. Since all P processors execute the same stream of BVRAM instructions, all will generate the same time-stamps.

Thus instead of holding a fixed number r of vector registers $V_1, \ldots, V_r$, each node of the LogP model will store several lives for each vector register V_j, distinguished by different time-stamps t. A local dictionary will associate pairs (j, t) with addresses of the actual register.

In practice however, the implementation of every BVRAM instructions involving communication requires some global computations like *scan* or *broadcast*: before the *send*-phase, processor i needs to participate in these global computations to determine where to send each item[1]. This prevents the P processors to get too much out of sync[2], and we can prove

[1] The only exception is *append*, where the destination processor for every send can be computed locally.
[2] Latency can still be hidden when the instructions requiring global computations are far apart.

that the number of different active lives for one vector register is bound and relatively small. Hence the dictionary can be efficiently organized as an array V of size r (the total number of vector registers), in which each entry $V[i]$ is a list of pairs (t, a), with t a time-stamp and a the address of the actual register.

Not all messages are *data messages*, i.e. messages containing chunks of the BVRAM's sequences. Some are *control messages*, e.g. parts of a *scan*, or a *broadcast*. We will associate the control sequences with the output vector register V_j of the BVRAM instruction which initiated them. But the data coming with these messages is stored in other memory locations than the sequence V_j: we call this memory location a **subregister**. Thus a BVRAM register V_j at node i will consists of:

- a *vector subregister* holding the fragment of the sequence V_j stored at node i, and
- a fixed number of named *control subregisters*, each of a fixed size, typically of 1-2 words. We will mention some control subregisters in Section 6.5.

In consequence, the header of a message will consists of the following fields: (1) j, the register number, (2) t, the timestamp, (3) the subregister name, and (4) an offset Δ inside the subregister. When the message arrives, the processor in the network interface accesses the dictionary with the key (j, t), computes the address a of the register, adds to a the offset corresponding to the subregister and the offset Δ, and stores there the message content.

The main processor will synchronize its access to the subregisters with the network interface as follows. Each subregister contains an additional field with the total number of words it has to receive. After storing a message, the network interface decrements this field with the size of the message. When the main processor needs to inspect the subregister it will wait until this field becomes 0.

```
procedure top-loop(𝒫)
    t ← 0
    pc ← 0
    loop
        case 𝒫[pc] of
            "V_{j0} ← f(V_{j1}, V_{j2}, ...)":
                t1 ← currentTimeStamp[j1]
                t2 ← currentTimeStamp[j2]
                ...
                execute(f, (j0, t), (j1, t1), (j2, t2), ...)
                currentTimeStamp[j] ← t
                pc ← pc + 1
            "a branch instruction":
                compute the new pc
            "HALT" : exit
        end-case
        t ← t + 1
    end-loop
end-procedure
```

Notations: pc is the program counter, t is the current time-stamp, $currentTimeStamp[j]$ gives the time-stamp of the last assignment to the register V_j, $execute(f, \ldots)$ implements the BVRAM instruction f.

Figure 6.8: Main loop of the implementation of the BVRAM on the LogP model

6.5 Description of the Implementation

We describe in this Section the implementation of a BVRAM program $\mathcal{P}$ on the LogP model. The main loop of the simulation is given in Figure 6.8. The array *currentTimeStamp* holds the current time-stamp for each register j.

6.5.1 Scalar Operations

The BVRAM instructions include a number of unary, binary, and ternary operations. As a typical example, consider the binary operation:

```
procedure execute-plus((j, t), (j1, t1), (j2, t2))
        let v1 be the vector subregister of (j1, t1)
        let v2 be the vector subregister of (j2, t2)
        wait(v1)
        wait(v2)
        allocate a new vector subregister v for Vj with time-stamp t
        for k = 0 to length(v1) - 1 do
            v[k] <- v1[k] + v2[k]
        mark v as full
end-procedure
```

Here t, t_1, t_2 are the time-stamps for the registers V_j, V_{j_1} and V_{j_2} respectively. Note that $\mathsf{length}(V_{j_1}) = \mathsf{length}(V_{j_2})$. The procedure $wait(v_1)$ waits until the vector subregister v_1 is full.

Figure 6.9: Implementation of the binary operation $V_j \leftarrow V_{j_1} + V_{j_2}$ on the LogP model

$$V_j \leftarrow V_{j_1} + V_{j_2}$$

This is implemented at node i by the algorithm in Figure 6.9.

6.5.2 Append

Unlike the scalar operations, the implementation of *append* on the LogP model requires communication. We implement it with two monotone one-to-one communications, like in the example of Figure 6.4. Hence we use the algorithm described in the proof of Theorem 6.3.7 without change. Notice that in each of the two monotone one-to-one communications, a processor i has at most two blocks to send.

6.5.3 Broadcast

Implementing some BVRAM instructions requires a broadcast and/or a scan operation. The optimal broadcast algorithm for the LogP model is fully described in [69] for the postal model ($L > 0, o = 0, g = 1$) and for items of size $s = 1$. Namely denoting with f_t the total number of processors which can receive an item stored initially at one processor, we have:

- $f_t = 1$ for $0 \leq t < L$, because there is no time to send anything.
- $f_t = 1 + f_{t-L} + f_{t-L-1} + f_{t-L-2} + \ldots$ for $t > L$, because the root processor sends its item at moments $0, 1, 2, \ldots$, and these are received at moments $L, L+1, L+2, \ldots$ by processors having left only $t-L, t-L-1, t-L-2, \ldots$ time to broadcast that item.

The second relation becomes: $f_t = f_{t-1} + f_{t-L}$, for $t \geq L$.

We adapt this algorithm to an arbitrary LogP model and messages of size s. Let $L' = 2o + L + (s-1)g$ be the "total" latency for a message, and let $g' = \max(sg, o + (s-1)g + 1)$ be the total gap between two messages sent by one processor. Then f_t with the same meaning as above is given by:

- $f_t = 1$ for $t < L'$, and
- $f_t = 1 + f_{t-L'} + f_{t-L'-g'} + f_{t-L'-2g'} + \ldots$, for $t \geq L'$.

The second relation becomes now[3]: $f_t = f_{t-g'} + f_{t-L'}$ for $t \geq L'$. Obviously for the postal model and for $s = 1$ we have $L' = L$ and $g' = 1$, hence we recover the relations above. The algorithm runs in T steps, where T is the minimum value for which $f_T \geq P$. Thus $T = O(\log P)$, and the total number of messages is $P - 1$ (each processor receives exactly one message).

We use a special subregister for the unique message arriving at a node during a broadcast operation. The processor waits until that subregister becomes full, then sends out the item to all its children.

[3]Note that, when $g' > L'$ we may have $t - g' < 0$. By convention, $f_{t-g'} = 1$ in this case.

6.5.4 Scan

In the *all-to-all broadcast problem with combining* in [69] each processor i holds an item x_i, and each processor needs to receive the value $\sum_{i=0,P-1} x_i$. An optimal algorithm for the postal model and for the case where each x_i has size 1 is given in [69]. This can adapted to an optimal *scan* algorithm (in which processor i needs to hold in the end the value $\sum_{i'=0,i} x_{i'}$). Under this algorithm, at time t every processor i will hold the value $\sum_{i'=i-f_t+1,i} x_{i'}$, assuming $x_{i'} = 0$ for $i' < 0$ or $i' \geq P$. Here f_t is given by:

- $f_t = 1$ for $0 \leq t < L$, because there is no time for processor i to receive any other item besides its own x_i.
- $f_t = f_{t-1} + f_{t-L}$ for $t \geq L$, because processor i receives at time t the sum sent by processor $i - f_{t-1}$ at time $t - L$ and combines it with its own sum.

We adapt this idea to arbitrary parameters L, g, o and message size s. Let $L' \stackrel{\text{def}}{=} 2o + L + (s-1)g$ be the total latency of a message, and $g' = \max(sg, o+(s-1)g+1)$ be the total gap between two messages received by one processor. That is a processor processor will receive a message at times $t = L', L' + g', L' + 2g', L' + 3g', \ldots$. It also needs to send messages at times $0, g', 2g', 3g', \ldots$. Assuming the processor overheads for the *send*s do not overlap with those for the *receive*s, then in an optimal scan algorithm processor i will hold at time t the value $\sum_{i'=i-f_t+1,i} x_{i'}$, where f_t is given by:

- $f_t = 1$ for $t < L'$, and
- $f_t = f_{t-g'} + f_{t-L'}$ for $t \geq L'$ and $(t - L') = 0 \ mod \ g'$, and
- $f_t = f_{t-1}$ for $t \geq L'$ and $(t - L') \neq 0 \ mod \ g'$, because no message is received at time t.

Notice the difference between the expressions of f_t for the broadcast and for scan. This is due to the fact that in the implementation of *scan* the *send*s and the *receive*s are interleaved.

As for *broadcast*, the total number of steps is the smallest T for which $f_T \geq P$, hence $T = O(\log P)$. The total number of messages however, is larger: $O(P \log P)$.

There is a final technical twist for the implementation of the *scan* primitive on the LogP model: the fact that the messages can arrive out of order. Indeed, processor i needs to receive messages (in that order) from processors $i - f_{L'}, i - f_{L'+g'}, i - f_{L'+2g'}, i - f_{L'+3g'}, \ldots$. In practice, lower-numbered processors have fewer messages to receive themselves: after they finish receiving, they can send out messages at a higher rate. Therefore processor i may receive its messages out of order. We solve this problem by keeping an array of size P of subregisters S for the broadcast problem: a message from processor i' to processor i will be stored by the network interface DMA of the node i in the subregister $S[i']$. The main processor i inspects these messages in the right order and waits when the next message has not arrived yet, even if others are already available.

6.5.5 Select

Recall that the selection operation $V_j \leftarrow \mathsf{select}(V_{j_1}, V_{j_2})$ stores in V_j all the elements in V_{j_2} corresponding to the non-zero values of V_{j_1}. We implement it in three steps: (1) a *scan* operation, which enumerates the non-zero elements of V_{j_1}, (2) a *broadcast* operation, in which the last processor sends to all other processors the total number of non-zero elements in V_{j_1}, and (3) a monotone one-to-one communication with the data in V_{j_2}. The algorithm is given in Figure 6.10.

6.5.6 Bounded Monotone Routing - bmRoute

The bounded monotone routing $V_j \leftarrow \mathsf{bmRoute}(V_{j_1}, V_{j_2}, V_{j_3})$ is implemented in two steps: (1) a scan operation on V_{j_2} to compute the destination positions of the elements of V_{j_3}, and (2) a monotone, one-to-many communication in which the elements of V_{j_3} are sent to other processors. In our current implementation however, the bound V_{j_1} is discarded: this makes another step necessary, namely to broadcast the total length of the result (which is $\mathsf{length}(V_{j_1})$) from the last processor to all others.

```
procedure execute-select((j,t), (j1,t1), (j2,t2))
        let v1 be the vector subregister of (j1,t1)
        let v2 be the vector subregister of (j2,t2)
        wait(v1)
        wait(v2)
        allocate v1' of length length(v1)
        for i = 0 to length(v1) do
            if v1[i] = 0 then v1'[i] ← 0
                         else v1'[i] ← 1

        s ← scan(v1') // This involves communications

        broadcast n, the size of the result, from the last
            element of s in the last processor
            to all processors

        allocate v (its size can be computed from n)

        do a monotone, one-to-one communication
            sending the data in v2
            corresponding to non-zero values in v1
            to the destinations dictated by s
end-procedure
```

Figure 6.10: The implementation of select $V_j \leftarrow \mathsf{select}(V_{j_1}, V_{j_2})$ on the BVRAM

6.5.7 Segmented Bounded Monotone Routing - sbmRoute

This primitive cannot be decomposed into monotone communications. We use an ad-hoc algorithm. Our experiments have shown that this instruction is by far the most expensive to run.

6.5.8 Applying an External Function: Cluster and Decluster

For a flat sequence of integers $x = [x_0, x_1, \ldots, x_{n-1}]$, we can map an external function f on x by applying f locally at each processor. Otherwise, if the elements of x are not integers, then we need to *cluster* x first, then apply f, and finally *decluster* it. Clustering is the operation by which a sequence is partitioned on the P nodes according to an application-specific criteria rather than the block-partitioning of Section 6.2. To illustrate, consider the case when x is a nested sequence of integers, i.e. $\forall i,\ x_i = [x_{i0}, x_{i1}, x_{i2}, \ldots, x_{i(m_i-1)}]$. Recall that x is represented on the BVRAM by two sequences: a segment sequence $m \stackrel{\text{def}}{=} [m_0, m_1, \ldots, m_{n-1}]$, and a data sequence, $d \stackrel{\text{def}}{=} \mathsf{flatten}(x)$. The trouble is that, under block-partitioning, m_i and the elements $x_{i0}, x_{i1}, \ldots$ in general end up on different nodes, so we cannot simply apply f locally. Instead we proceed as follows:

1. Compute the cost c_i of computing $f(x_i)$: e.g. take this to be $c_i \stackrel{\text{def}}{=} m_i + 1$, i.e. the size of the sequence x_i (we need the cost c_i to be > 0 in the sequel).

2. Let $s_i \stackrel{\text{def}}{=} \sum_{i'=0,i-1} c_{i'}$, $s \stackrel{\text{def}}{=} \sum_{i'=0,P-1} c_{i'}$, and $p_i \stackrel{\text{def}}{=} \lfloor \frac{s_i P}{s} \rfloor$. The idea is that we want to perform the computation $f(x_i)$ on processor p_i. This will balance the total cost as good as possible among the P processors. To compute s_i and s we need to perform a scan and a broadcast.

3. **Cluster** the sequence $m_0, \ldots, m_{n-1}$ according to $p \stackrel{\text{def}}{=} [p_0, \ldots, p_{n-1}]$, i.e. send m_i to processor p_i: $\mathtt{m} \leftarrow \mathit{cluster}(m, p)$.

4. Compute $p' \leftarrow \mathsf{bmRoute}(d, m, p)$, where $p = [p_0, p_1, \ldots, p_{n-1}]$.

5. Cluster the sequence d according to p': $\mathfrak{d} \leftarrow \mathit{cluster}(d, p')$. Now m_i and $x_{i0}, x_{i1}, \ldots$ are all on the same node.

6. Apply f locally: $\mathfrak{m}$ and d will contain a description of a nested sequence stored locally.

7. **Decluster** the result.

Here $\mathfrak{d} \leftarrow \mathit{cluster}(d, p)$ has the following meaning: assume $\mathsf{length}(d) = \mathsf{length}(p) = n$ and $p_0 \leq p_1 \leq \ldots \leq p_{n-1}$, then send d_i to processor p_i. The result $\mathfrak{d}$ is the same sequence as d, but partitioned on the processors under a different convention.

cluster is a monotone, one-to-one communication. There are however some additional technical complications. Recall that the algorithm in the proof of Theorem 6.3.7 requires each processor to know for each element d_i not only its destination processor p_i, but also $\delta(d_i)$, i.e. how many other elements will be sent to processor p_i *before* d_i. For all other BVRAM operations (append, bmRoute, select) this information is readily available, or becomes available after the *scan* and *broadcast* operation. Here however we need to send additional control messages in order to make this information available.

decluster is the opposite of cluster, i.e. when $\mathfrak{d} = \mathit{cluster}(d, p)$, then $\mathit{decluster}(\mathfrak{d}) = d$. It is computed in three steps: (1) a scan, notifying each processor about the position of its chunk $\mathfrak{d}$ in the sequence, (2) a broadcast, which makes the total length of $\mathfrak{d}$ available to every processor, and (3) a monotone, one-to-one communication.

Finally note that the cluster algorithm guarantees that the external work is uniformly distributed among the processors, within a certain tolerance. Namely ideally we would like to have no more than $\lceil \frac{s}{P} \rceil$ work done at each processor. The clustering algorithm guarantees that the work performed by each processor is bounded by:

$$\lfloor \frac{s}{P} \rfloor + \max_{i=0,n-1} c_i \tag{6.1}$$

Indeed, let $x_i, x_{i+1}, \ldots, x_j$ be all elements processed by some processor p, i.e. $p = \lfloor \frac{s_i P}{s} \rfloor = \lfloor \frac{s_i P}{s} \rfloor = \lfloor \frac{s_{i+1} P}{s} \rfloor = \ldots = \lfloor \frac{s_j P}{s} \rfloor < \lfloor \frac{s_{j+1} P}{s} \rfloor$. Obviously $\frac{(c_{i+1} + \ldots + c_j)P}{s} < 1$, hence $c_{i+1} + \ldots + c_j < \frac{s}{P}$. So the total work done at processor p is $< c_i + \frac{s}{P}$, and therefore it is $\leq \max_i \lfloor c_i \rfloor + \frac{s}{P}$.

6.6 Benchmarks

6.6.1 Goal of the Benchmarks

The goal of the benchmarks is to test the feasibility of the implementation techniques of BVRAM on LogP described above. The experiments are done for a simple, no-frills prototype implementation, on simply generated data sets. Although the goal is limited and is not a full validation of the model, the results of the experiments are encouraging and offer a useful guide for the design of further experiments.

Our benchmarks test two widely accepted performance metrics for parallel database systems [46]: *speedup* and *scaleup*. The **speedup** is defined to be the speed of an implementation as a function of the number of processors, as the size of the database is kept constant: T_1/T_P: ideally this is a linear function in P. The speedup measures the ability of a given application to improve its performance when more equipment is bought. The **scaleup** is defined to be the running time of an implementation when both the size of the database and the number of processors are increased at the same ratio: ideally the running time should remain constant. This performance metric addresses an important issue of database applications which have a fixed amount of time for computation: as the database grows, the companies would like to buy additional processors and keep the same response time.

The kind of algorithms which can be sped up by our techniques are those for which there is enough work to be done at each processor to justify the communication effort. Roughly, such applications will have (1) a communication phase, which distributes the data evenly on the processors, followed by (2) a computation phase, in which each processor performs computations locally. This alternation could be repeated. Thus the total running time is

$$T \quad \stackrel{\text{def}}{=} \quad T_{comm} + \delta \frac{T_{comp}}{P} \tag{6.2}$$

where T_{comm} is the communication time spent in phase (1), T_{comp} is the total computation time for a sequential computation, and δ is a factor measuring the load-imbalance, i.e. the ratio between the largest work to be done by a processor in phase (2), and the ideal work obtained by dividing the total work equally among the processors. It is interesting to

$$\text{fun } merge\langle X, Y\rangle = \text{map}(sequentialMerge)(divide)\langle X, Y\rangle$$

A pragmatic version of Valiant's merging algorithm: *sequentialMerge* is an external sequential function which merges two sequences, *divide* is defined in Figure 4.21.

Figure 6.11: The *merge* benchmark

compare this formula with formula 2.2, Section 2.6. In order for the parallelism to be useful there should be enough work left to be done in phase (2), and it should be evenly enough distributed on the processors, to compensate for the cost paid in phase (1).

Our benchmarks measure the ability of the BVRAM implementation to keep the cost of phase (1) low while achieving reasonable load-balance for phase (2). For this we measure T_{comm} and δ, both by increasing P and keeping the same input size, and by increasing both P and the input size.

6.6.2 Description of the Benchmarks

Our first benchmark is a *merge* algorithm obtained by modifying Valiant's *merge* algorithm of Subsection 4.6.2. Recall that Valiant's *merge* algorithm repeatedly divides the first sequence X into subsequences of size $\sqrt{n}$, where $n = \text{length}(X)$, until its length becomes ≤ 2. In practice, for n sufficiently large, say $n \geq 10{,}000$, a single division suffices to create more subproblems than the number of processors, which is typically $P \leq 32$ or $P \leq 64$. Hence the idea of our modified *merge* algorithm: apply the divide phase only once, then merge the $\sqrt{n}$ pairs of sequences sequentially, by mapping an external function. See Figure 6.11. Only *divide* involves communication, *sequentialMerge* is purely sequential. Other useful algorithms in databases have the same structure as *merge* of Figure 6.11, but with a different external function replacing *sequentialMerge*. Namely:

- Set operations, $\cup, \cap, -$, etc, on sets represented as sorted sequences. Simply replace *sequentialMerge* with a sequential function to compute the union, intersection, difference, etc. of two sorted sequences.

- Database join operations. Simply replace *sequentialMerge* with *sequentialJoin*. Note that parallelism is especially rewarding when the join operation is not equality but rather a computation-intensive operation, like those used in content-based image retrieval.

Our second benchmark has an object-oriented flavor and is inspired from the OO7 benchmark of [26]. It consists of an *assembly hierarchy* and a collection of *parts*. Each assembly could be a *composite assemblies* or a *base assemblies*. 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. See Figure 6.12.

The benchmark has two parts. First we traverse the assembly hierarchy an apply an external predicate *inspect* to the binary data stored in each assembly. The predicate "inspects" the assembly and returns 0 if it is "bad" and 1 if it is "good". In the second phase we apply another external function *adjust* to all parts which are direct or indirect members of a "bad" assembly.

We ran the benchmarks on a LogP simulator, with the L, o, and g parameters set for the CM5 and SP1 [14]. Our measurements reveal no qualitative differences between these two machines.

6.6.3 The LogP Simulator

We wrote a LogP simulator in ML to run the benchmarks. The simulator is parameterized by L, o, g, and P, and is based on a queue of events. It allows several threads at one processor. The ML signature of the simulator is shown in Figure 6.13.

6.6.4 The $\mathcal{NSA}$ to BVRAM Translator

Our benchmarks are intended to test the feasibility of the BVRAM implementation on the LogP model, and not that of a compiler of $\mathcal{MAP}$ into the BVRAM. However writing code

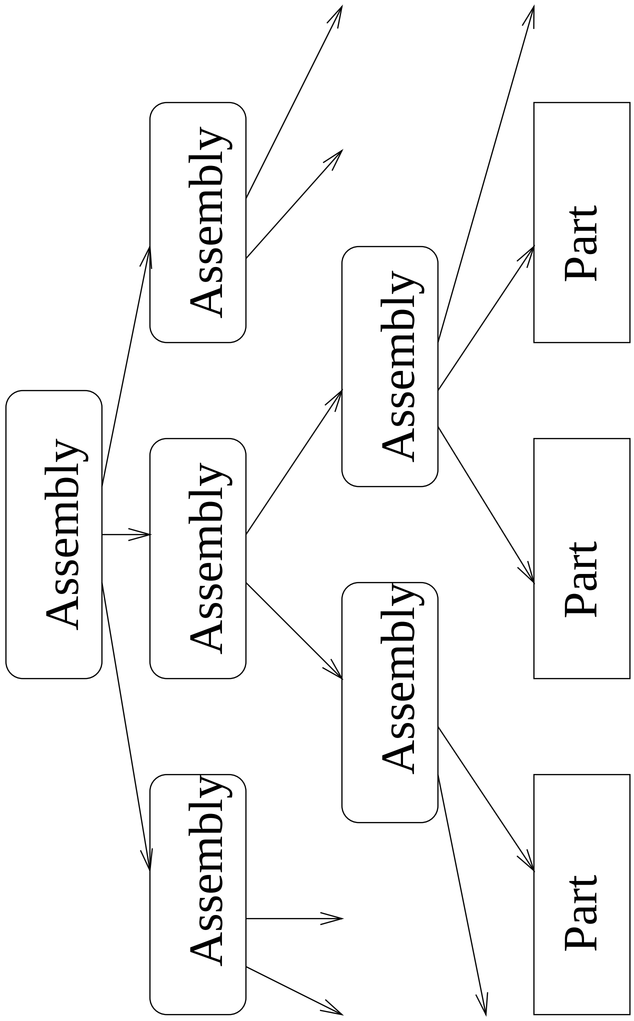


Figure 6.12: An example of an assembly hierarchy of depth 3 and its parts

```
signature LOGP_PARAMETERS =
    sig val L : int  val o : int  val g : int  val P : int
        type message
    end

signature LOGP_SIMULATOR =
    sig structure Parameters : LOGP_PARAMETERS
        datatype proc = PROC of int
        datatype time = TIME of int
        exception LOGP
        val send : proc * Parameters.message*
            (proc * Parameters.message-> unit)-> unit
        val delay : time-> unit
        val wait : unit-> unit
        val simulate : (proc-> unit)->
            {time:time, messagesSent:int, bytesSent:int, maxMessageSize:int}
    end
functor LogPSimulator(Parameters : LOGP_PARAMETERS) : LOGP_SIMULATOR =
    ...
```

message is the type of of messages sent, typically a datastructure containing a header and a body. $send(p, m, h)$ sends message m to processor p: on deliver, it will invoke on processor p the message handler $h(p', m)$, as a new thread, where p' is the sender processor. $simulate(f)$ will simulate f on P processors. *delay* is used to simulate an expensive computation at the local node. $wait()$ suspends the thread until a message arrives at the current processor.

Figure 6.13: The ML signature of the LogP simulator

directly in BVRAM is difficult and error-prone. Therefore we wrote a minimal translator (compiler) from $\mathcal{NSA}$— the variable-free version of $\mathcal{MAP}$ — to BVRAM, consisting of two steps: first it translates a $\mathcal{NSA}$ program into a $\mathcal{SA}$ program, next it translates the $\mathcal{SA}$ program into the BVRAM. Essentially the compiler follows the lines of Section 5.4. It does not handle sum types, conditionals, and loops: sum types and conditionals where not needed for our benchmarks, while the loops for the second benchmark were unrolled. This small compiler is unoptimized, except for common-subexpression elimination. We expect further optimizations, like instruction scheduling, to improve the speedup and scaleup of our benchmarks.

6.7 Experimental Methodology

Choice of data sets 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 among the middle third of X, i.e. in the interval $[x_{\frac{m}{3}}, x_{\frac{2m}{3}-1}]$. This ensures that among the $\sqrt{m}$ subproblems $\langle X_i, Y_i \rangle, i = 0, \sqrt{m} - 1$, both interesting cases are represented: when Y_i is empty, and when it is nonempty. While these data sets were easy to generate, they may not cover the worst case: when X and Y fall into disjoint intervals, say when all elements in Y are greater than those in X. In that case, although all X_i's of the subproblems have the same size, all but one Y_i will be empty, and in consequence the residual work at the processors will be unevenly balanced. While this case is unlikely in practice, it can be still dealt with as follows. Perform the *divide* phase a second time, with the roles of X and Y interchanged. This will split Y into $\sqrt{n}$ equal sized subsequences, and X into $\sqrt{n}$ unequal sequences. After combining the results of the two *divide* phase, we obtain at most $\sqrt{m} + \sqrt{n}$ subproblems, each of size at most $\sqrt{m} + \sqrt{n}$. The upper bound on the size of the subproblems guarantees a good balance of the residual work, see fromula 6.1 in Subsection 6.5.8.

The data sets for the object-oriented benchmark were also chosen in a way which makes them easy to generate. First 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. Next all base assemblies (tree leaves) are at the same depth in the hierarchy, i.e. the assembly tree is perfectly balanced: this kept the BVRAM benchmark simpler, and we expect to obtain similar results on data sets with unbalanced trees. The flawed assemblies represent 25% of all assemblies and are scattered uniformly among all the assemblies listed in prefix order of the 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. We do not expect different distributions of the flawed assemblies among the assembly hierarchy to affect the results of our experiments in a significant way, but they would complicate the design of scalable data sets, because we need to ensure that the same fraction of parts is processed in the second half of the benchmark. Finally the distribution of the size of the binary data associated to each assembly and each part is constant: 10 words. Changing this constant would not affect our results, but choosing a different distribution would put a heavier burden on load-balancing the computations of the external functions *inspect* and *adjust*. Formula 6.1 at the end of Subsection 6.5.8 gives an upper bound on the resulting imbalance: more significant load imbalance should be expected only when some binary data has a size which is far grater than the total data size divided by P.

No I/O For simplicity, our simulations ignore the I/O time. Taking into account I/O time may in fact improve our results. Indeed the I/O time is part of the "useful" computation, T_{comp}, hence its contribution to the total running time decreases as P increases. In short, we would benefit from doing I/O in parallel on all P nodes.

Single run Every point is the graphs represents a single run. Since we use a simulator, use a single data set for each experiment, and ignore I/O time, subsequent runs produce exactly the same results.

"Communications only" and "Communications and Computation" mode We ran both benchmarks twice: first measuring only communication time, secondly measuring both communication and computation time. In the "communications-only" mode, the LogP simulator only takes into account the communication time, as computed based on the L, o, and g parameters: in this mode, it is assumed that, except for the overhead o in sending and receiving a message, the P processors are infinitely fast. In particular, the main benefit of parallelism, the $\frac{T_{comp}}{P}$ term, does not show up, hence we do not expect to see any speedup in this mode.

In the "communications-and-computations" mode, the LogP simulator takes into account the computation time too. This includes both "useful" computations making up the $\frac{T_{comp}}{P}$ term, and computation time spent in the implementation of the BVRAM instructions: therefore even the T_{comm} term will be higher than in the "communication-only" mode.

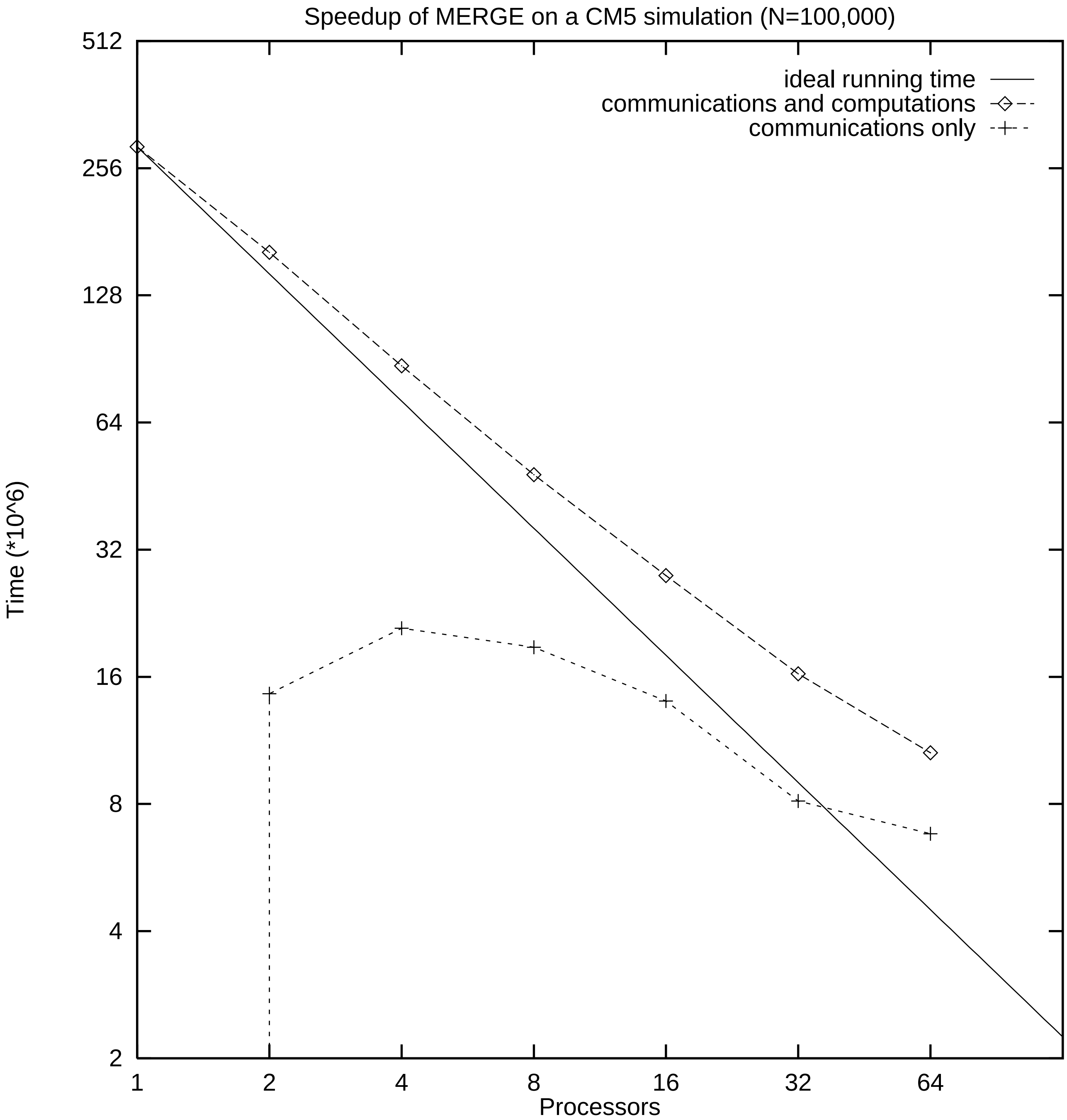
The computation times are simulated using the *delay* function, see Figure 6.13, and are only estimations of the real computation times on a multiprocessor.

6.8 Results

Figure 6.14 contains the time measurements for the speedup of the *merge* benchmark on a simulated Connection Machine 5 ($L = 60$, $o = 105$, $g = 100$). Here both sequences are of a fixed size $N = 100,000$, sequence Y covers approximatively the middle third of sequence X. We ran the same benchmark with the LogP parameters for a different machine (the SP1, $L = 70$, $o = 120$, $g = 55$), and observed no qualitative differences, see Figure 6.15: therefore we ran the remaining benchmarks for the CM5 only. The variation between the minimum and maximum residual work at each processor (which gives an upper bound for δ) is shown in Figure 6.16.

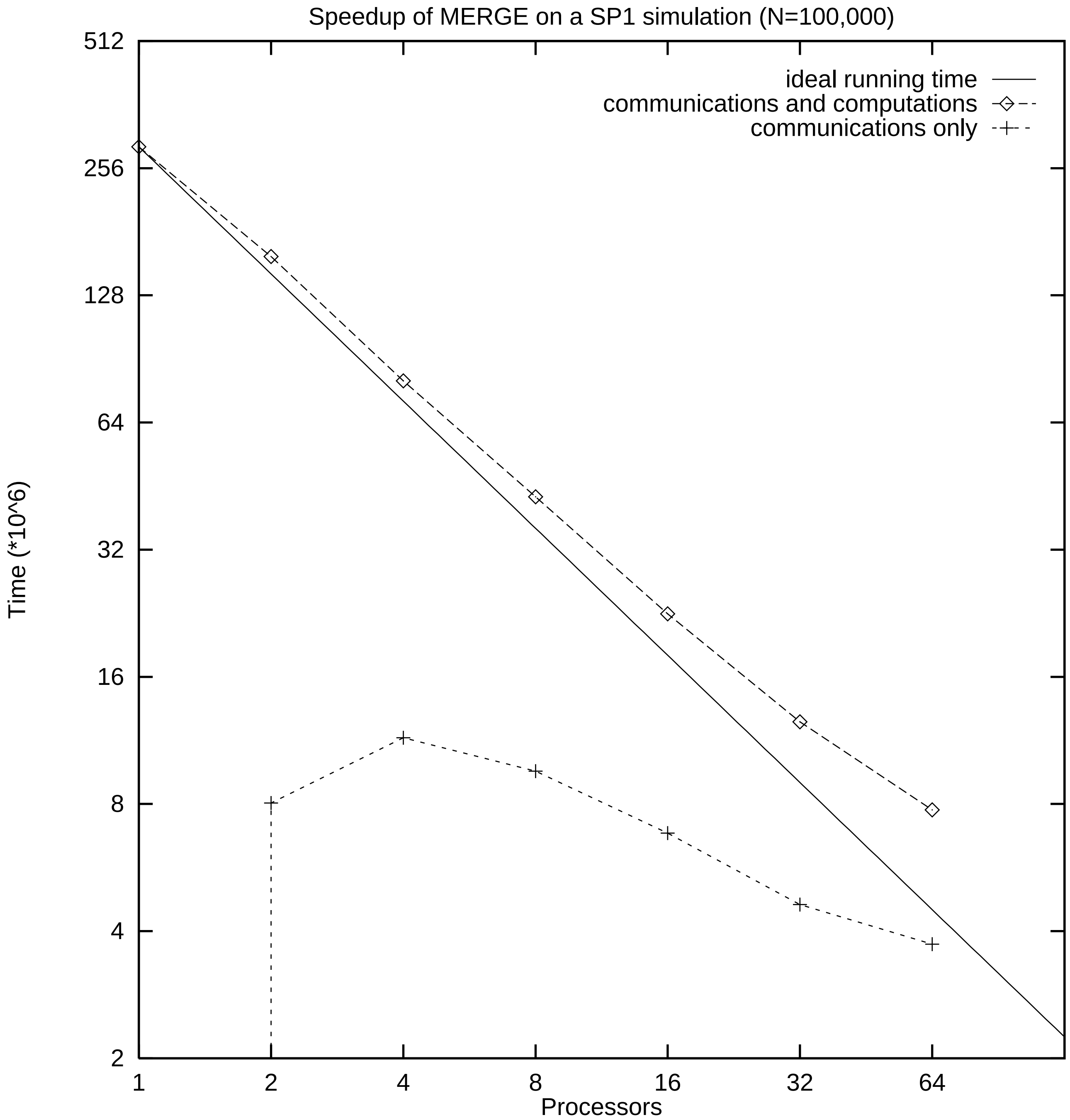
Besides total running time, we also measured the total number of messages sent, the total number of words sent, and the maximum message length. For the speedup experiment, they are reported in Figure 6.17. These parameters are independent of L, o, and g: they were the same for the speedup experiment on the SP1. L, o, and g *will* affect the shape of the broadcast tree, but, for example, the total number of messages for a broadcast[4] is $P - 1$

[4]Large variations in L, o, g could influence the constant behind the $O(P \log P)$ total number of messages and, hence, the total number of words sent for *scan*: there were no differences between the CM5 and SP1.



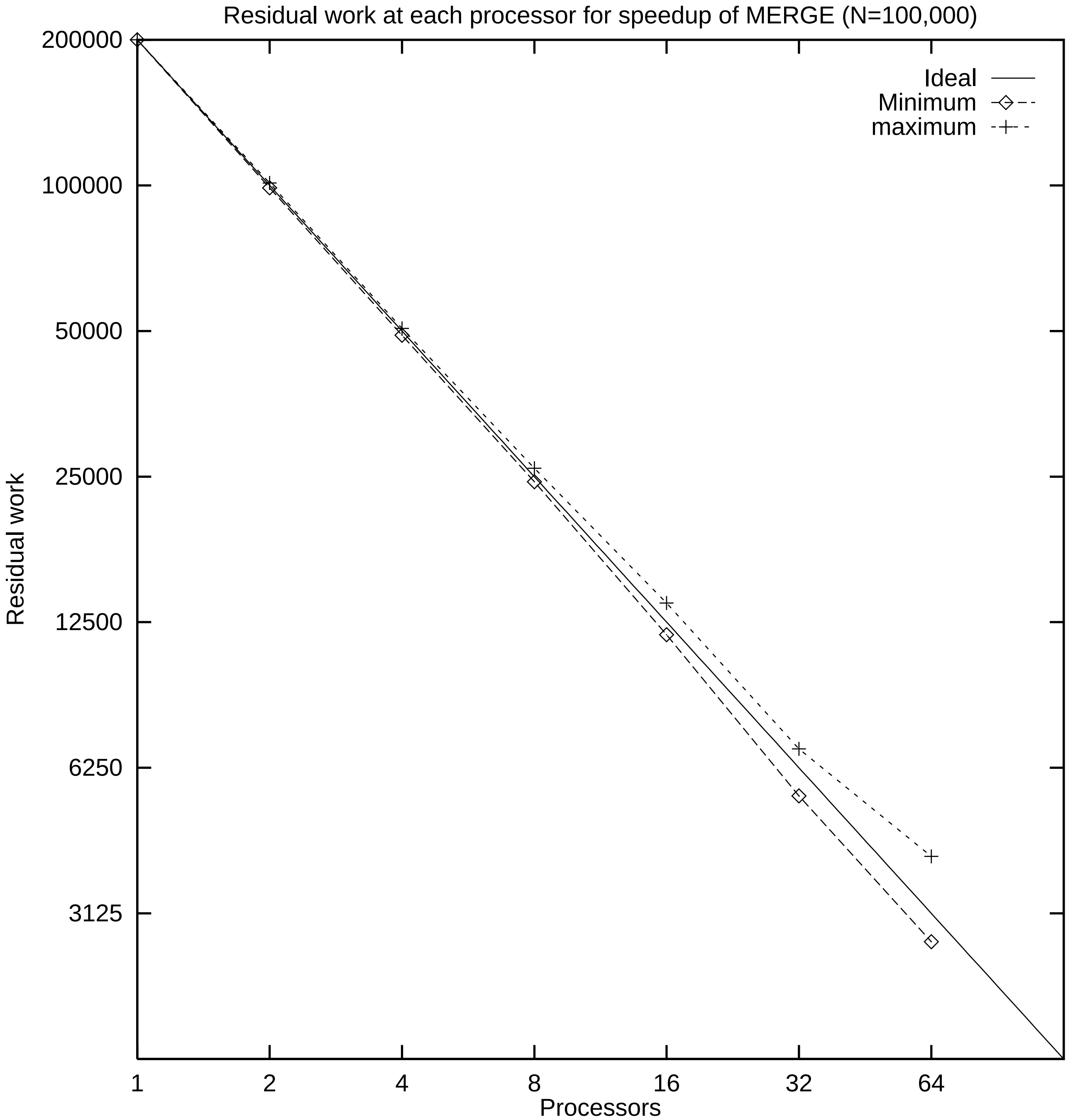
The CM5 has $L = 60$, $o = 105$, $g = 100$. Here $\mathsf{length}(X) = \mathsf{length}(Y) = 100,000$. The ideal running time is the communication-and-computation running time for $P = 1$, divided by P.

Figure 6.14: Speedup of $merge(X, Y)$ on a CM5



The SP1 has $L = 70$, $o = 120$, $g = 55$. Here $\mathsf{length}(X) = \mathsf{length}(Y) = 100,000$. The ideal running time is the communication-and-computation running time for $P = 1$, divided by P.

Figure 6.15: Speedup of $merge(X, Y)$ on a SP1



length(X) = length$(Y) = 100,000$, hence the ideal residual work is $200,000/P$. δ is the maximum residual work divided by the ideal residual work.

Figure 6.16: The residual work for the speedup of $merge(X, Y)$

and is independent of L, o, g.

Figure 6.18 contains the scaleup experiments for *merge* on the CM5. Here we varied both the number of processors and the size of the database: $\mathsf{length}(X)/P = \mathsf{length}(Y)/P = 10,000$. The minimum and maximum residual work is shown in Figure 6.19. The other parameters (total number of messages sent, total number of words, and the size of the longest message) are shown in Figure 6.20.

The speedup experiments for the object-oriented benchmark are shown in Figures 6.21, and 6.22. The residual work was uniformly distributed among the processors, and is not shown here. The scaleup experiments for the object-oriented benchmark are shown in Figures 6.23, 6.24, and 6.25.

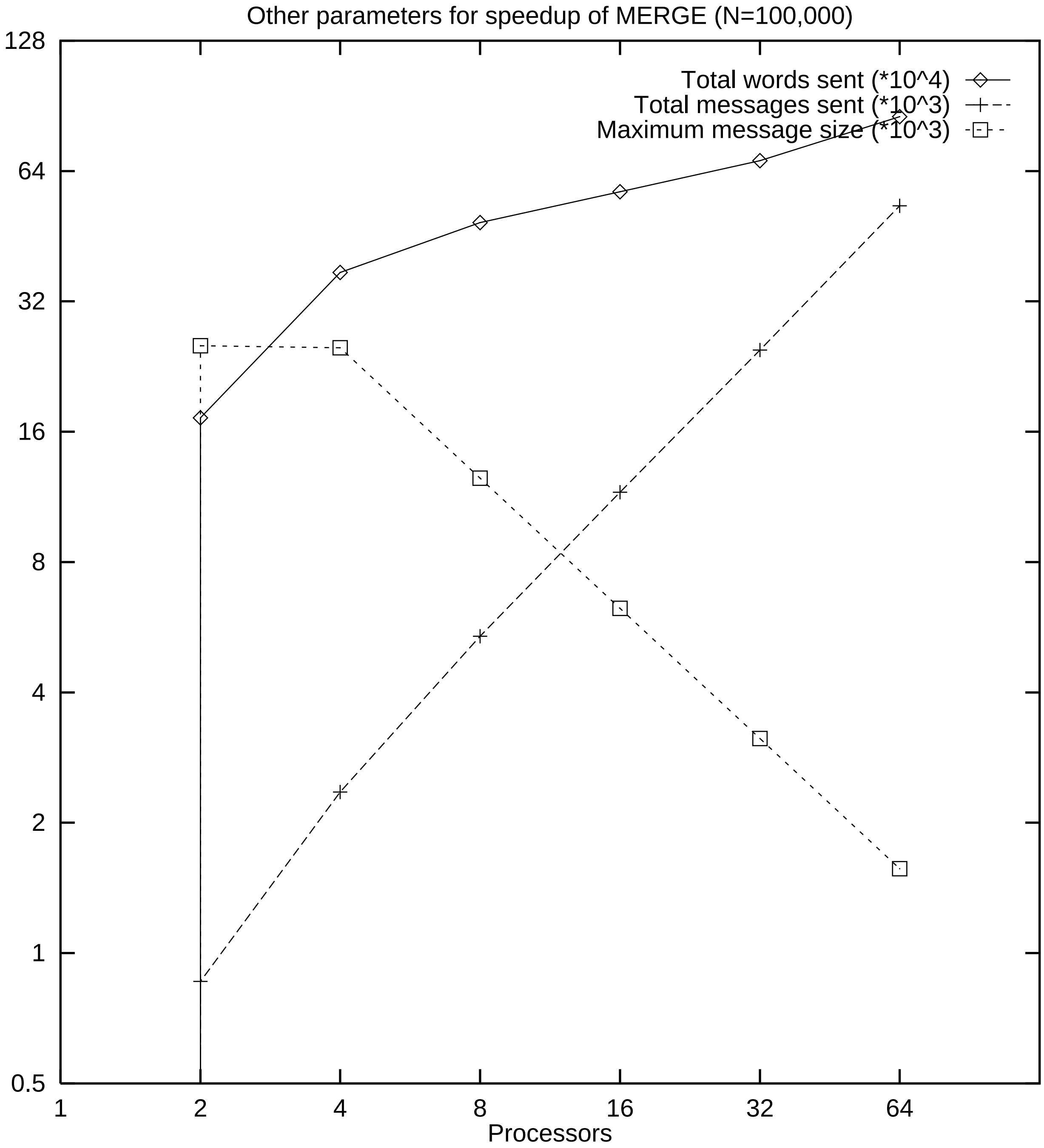
6.9 Interpretation of the Results

The running times of the experiments are largely influenced by two factors:

1. The communication time, T_{comm}. We perceive this time as a necessary evil in order to load balance the work among the processors.

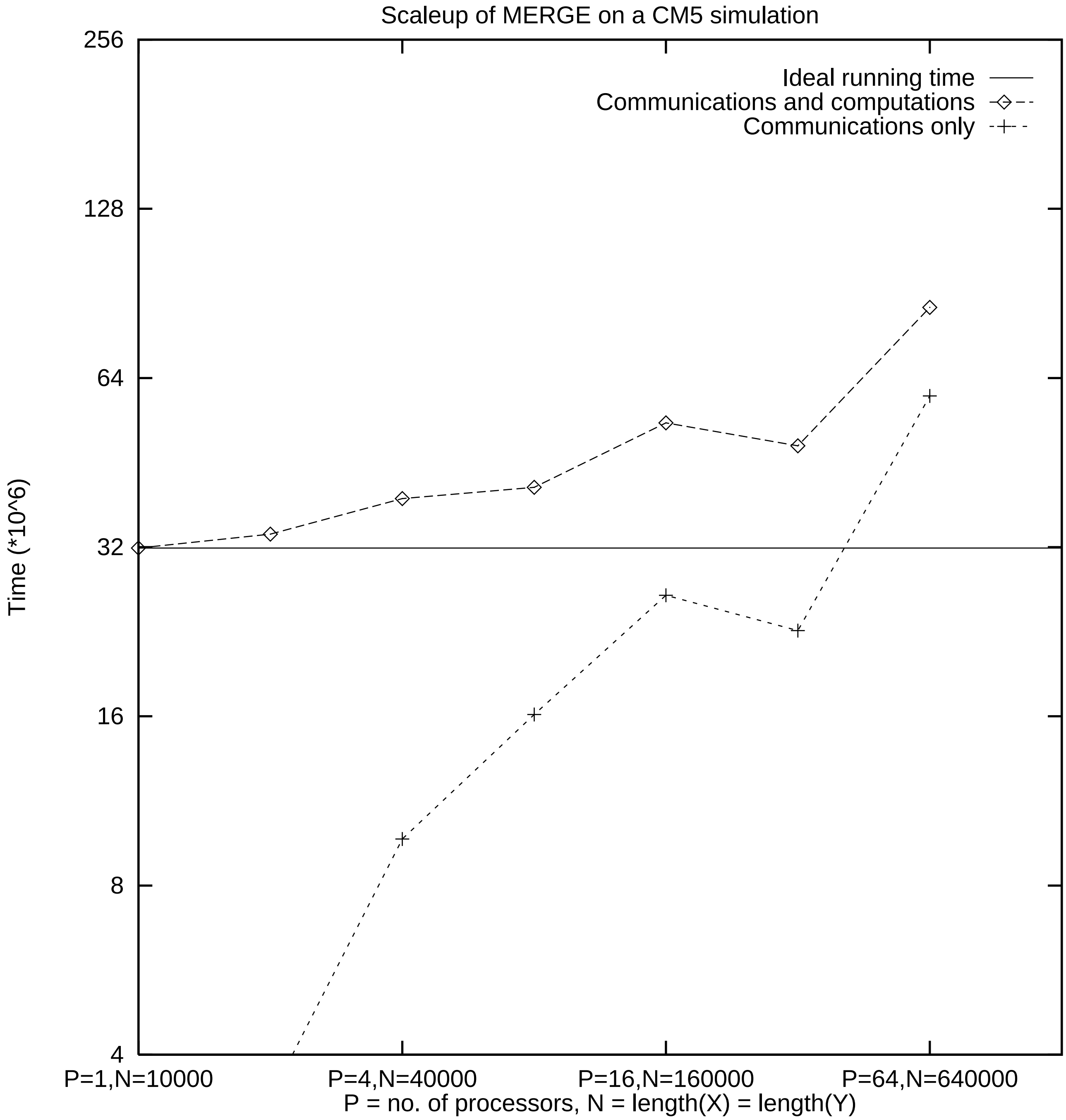
2. The load balance of the "useful" computation, i.e. δ. The compilation techniques described in Chapter 5 and the implementation techniques described earlier in this Chapter enable us to guarantee a good load balance, however at the cost of the communication time. The graphs in Figures 6.16, 6.19, and 6.24 show indeed that δ stays reasonably close to 1.

The speedup of $merge(X, Y)$ in Figure 6.14 revealed a nice surprise: the communication time T_{comm} decreases with P ! For $P = 1$ there are no communications at all, hence the communication time is zero, $T_{comm} = 0$. When P increases, the time for the control messages *broadcast* and *scan* is $O(\log P)$: this forces T_{comm} to increase. But at the same time each processor has to send less data messages, because the input size N is constant. What the graph in Figure 6.14 revealed is that for a large input size $N = 100,000$ and not



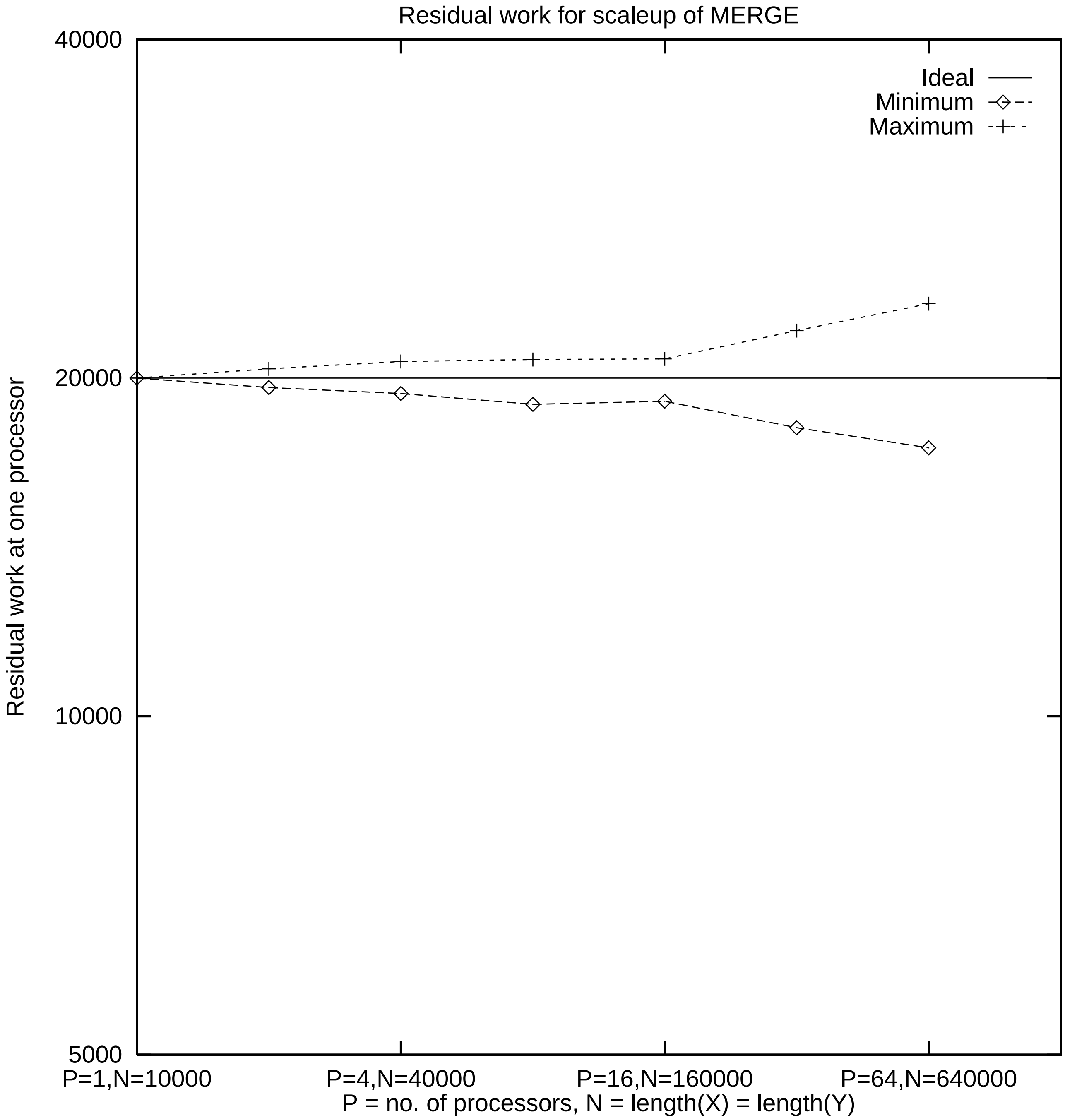
Total size of all messages sent (in words), the number of messages sent, and their maximum size (in words), for speedup experiments for $merge(X, Y)$. For $P = 1$ all of them are 0.

Figure 6.17: Other parameters for the speedup of $merge(X, Y)$



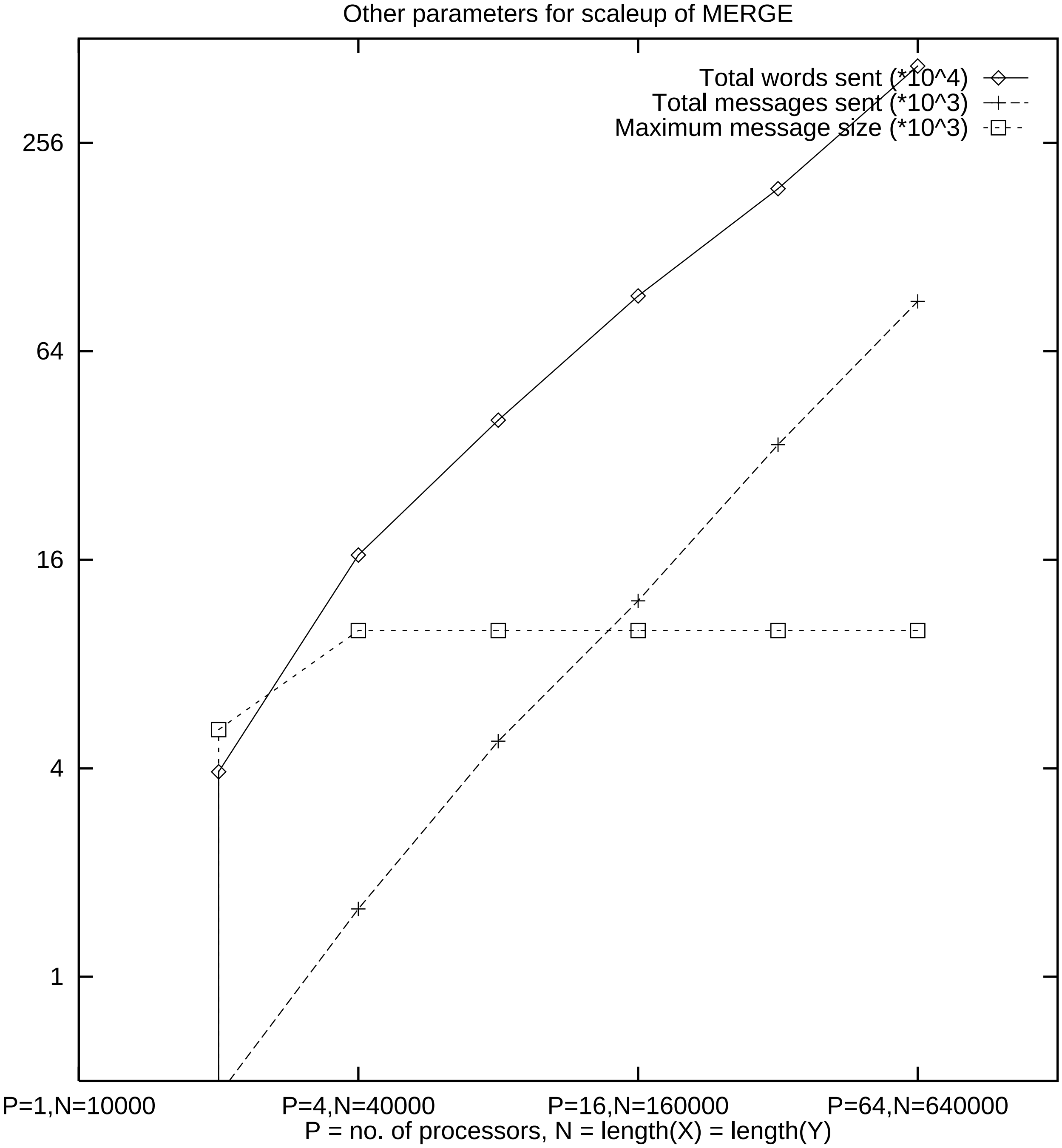
The CM5 has $L = 60$, $o = 105$, $g = 100$. Here $\mathsf{length}(X) = \mathsf{length}(Y) = 10,000P$. The ideal running time is the communication-and-computation time for $P = 1$.

Figure 6.18: Scaleup of $merge(X, Y)$ on a CM5



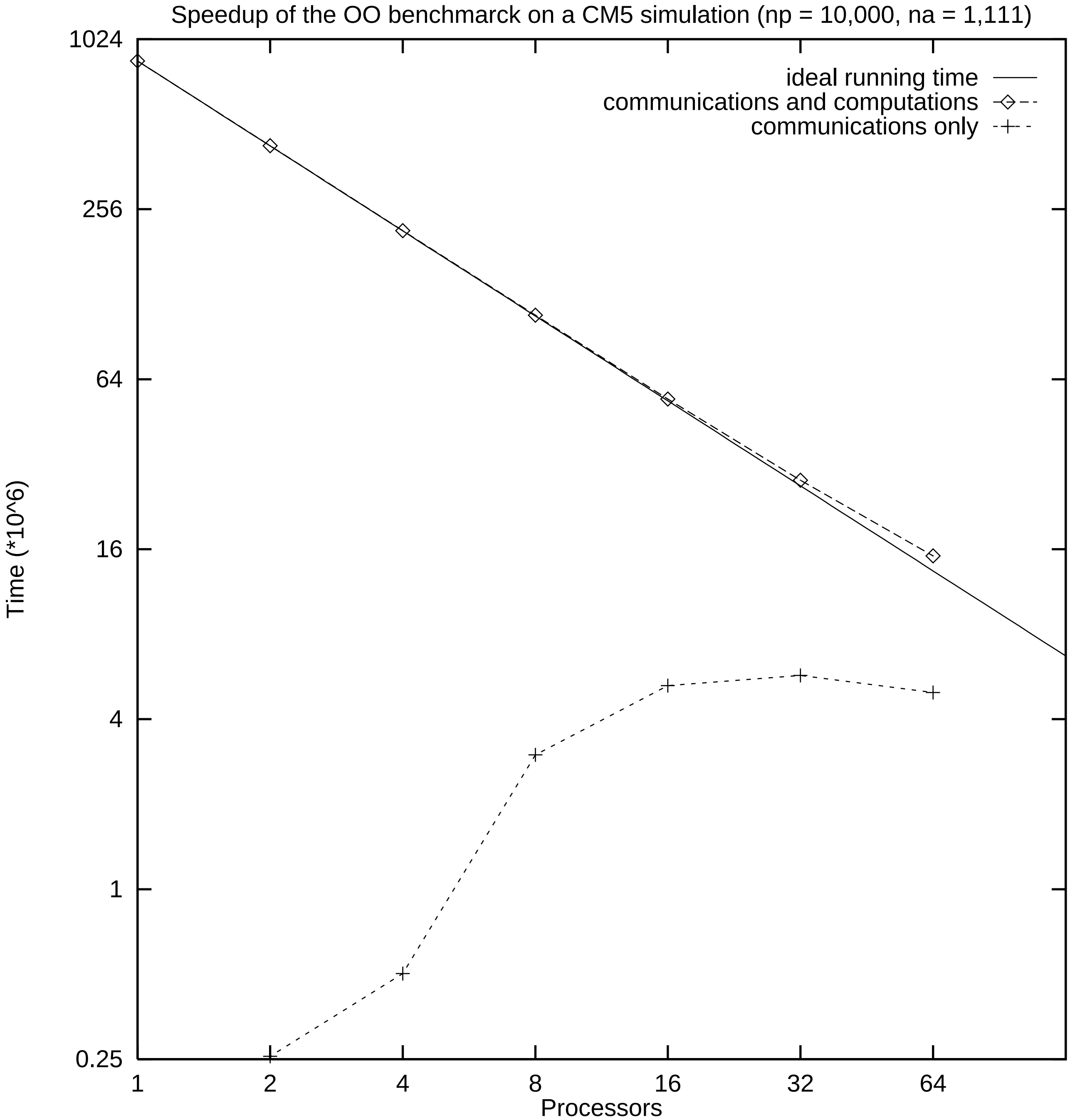
$\mathsf{length}(X) = \mathsf{length}(Y) = 10,000P$, hence the ideal residual work at each processor is $20,000$. δ is the maximum residual work divided by the ideal residual work.

Figure 6.19: The residual work for the scaleup of $merge(X, Y)$



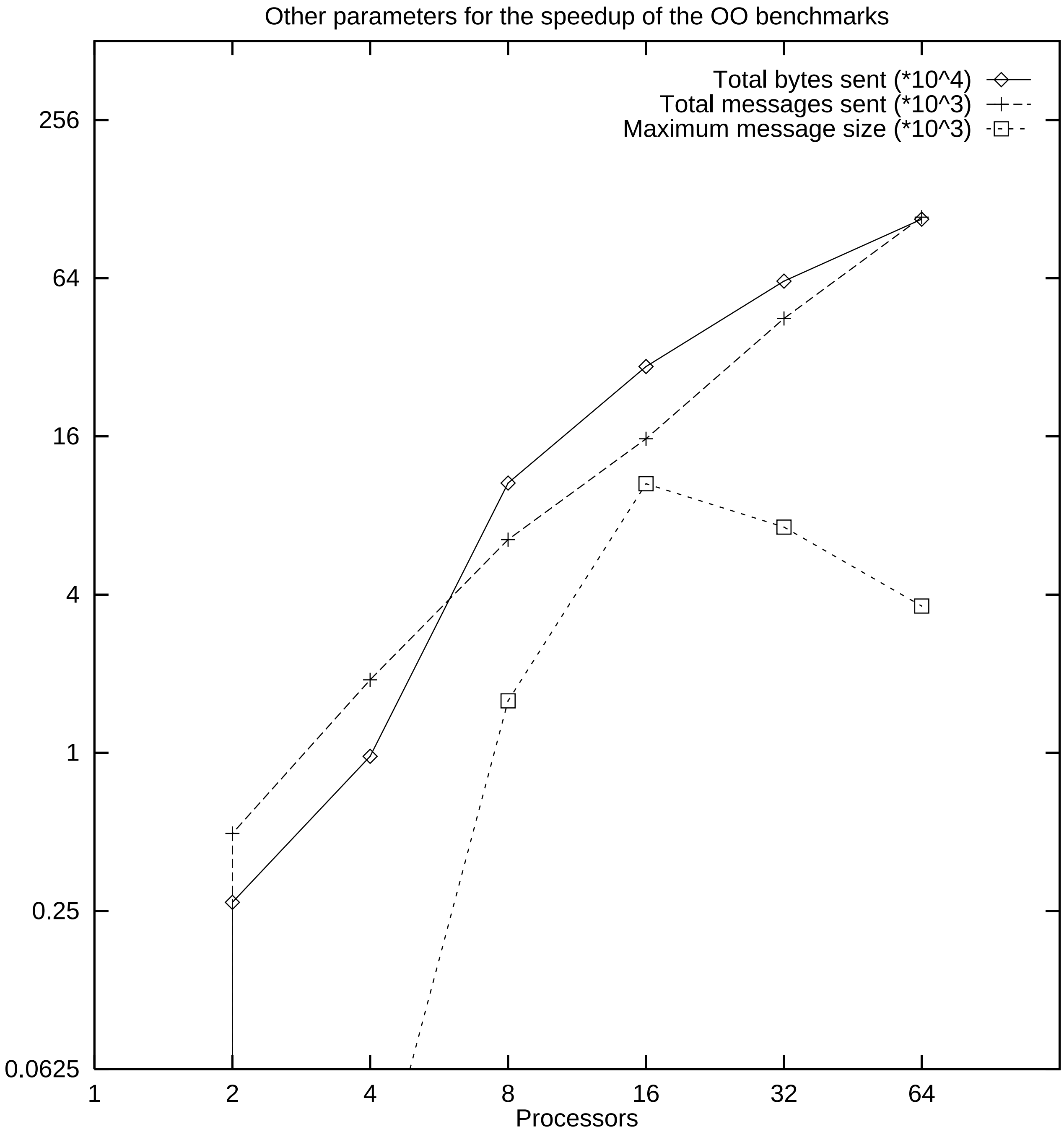
Total size of all messages sent (in words), the number of messages sent, and their maximum size (in words), for scaleup experiments for $merge(X, Y)$. Here $\mathsf{length}(X) = \mathsf{length}(Y) = 10,000P$.

Figure 6.20: Other parameters for scaleup of $merge(X, Y)$



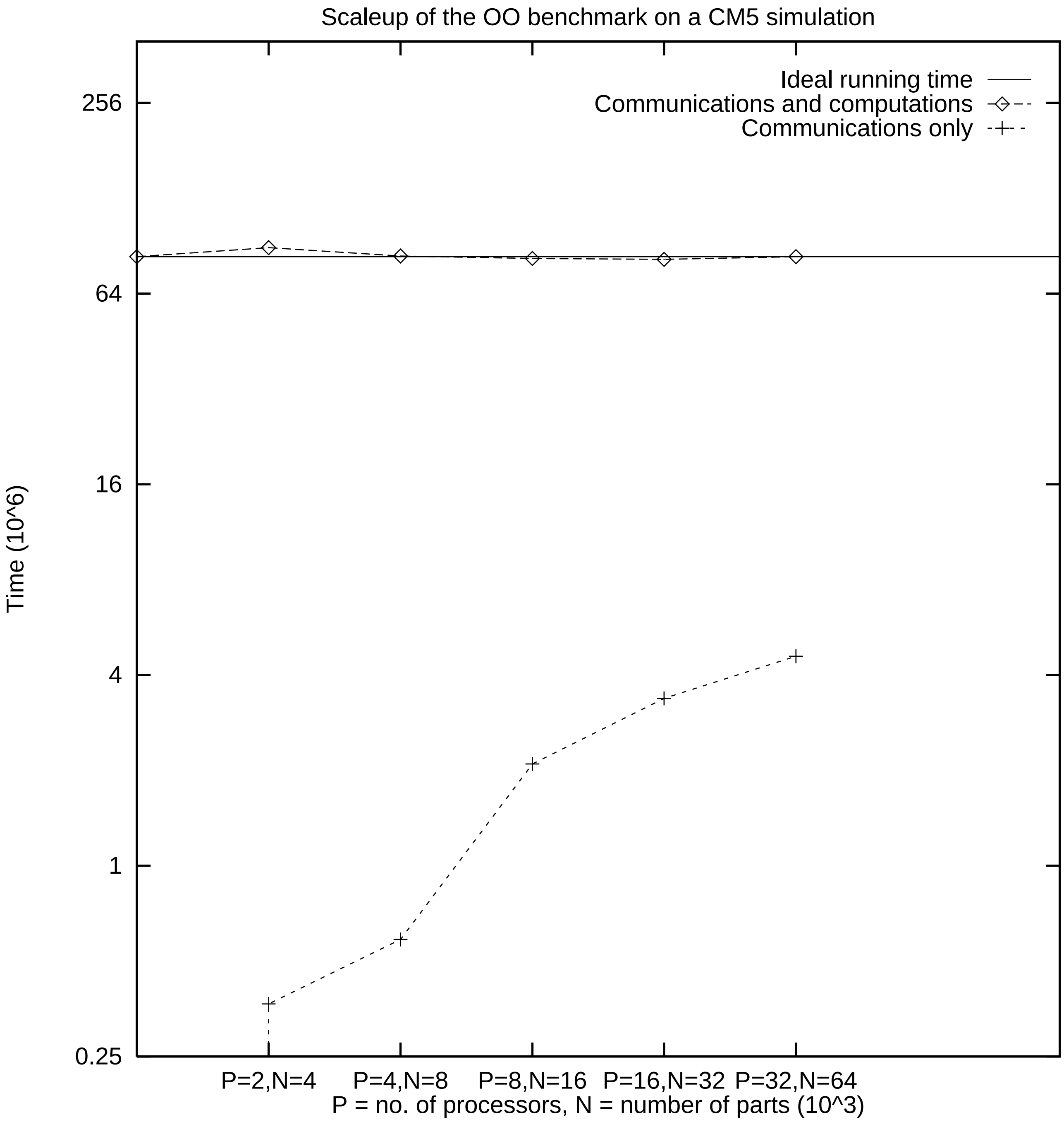
The CM5 has $L = 60$, $o = 105$, $g = 100$. The depth of the assembly hierarchy is $h = 4$, the fan-out is $d = 16$, for a total of $na = 4,369$ assemblies. The number of parts is $np = 40,000$. The binary data associated to each assembly and each part has 10 words. 25% of all assemblies are "bad".

Figure 6.21: Speedup of the OO benchmark on a CM5



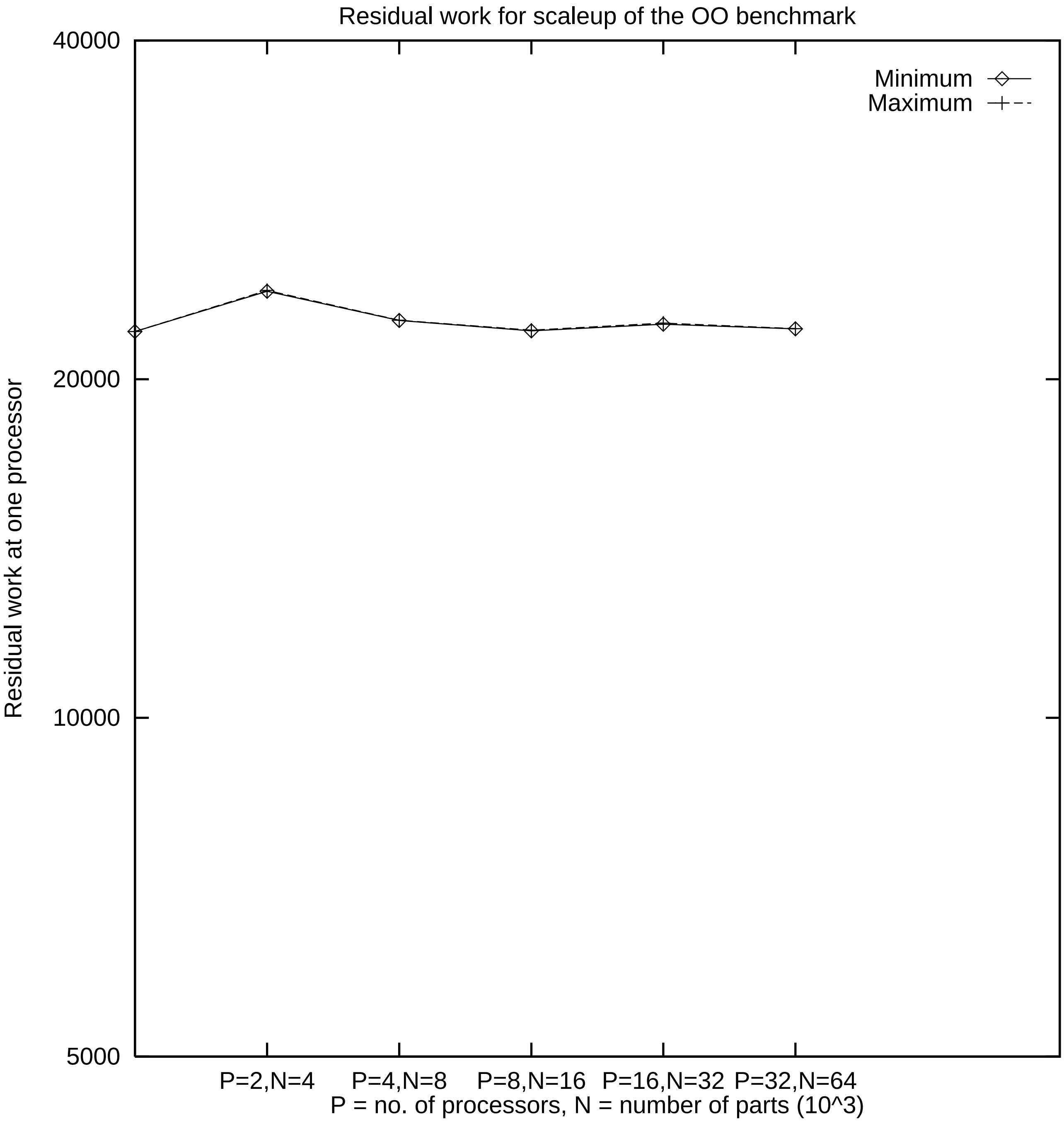
Total size of all messages sent (in words), the number of messages sent, and their maximum size (in words), for speedup experiments for the OO benchmark.

Figure 6.22: Other parameters for the speedup of the OO benchmark



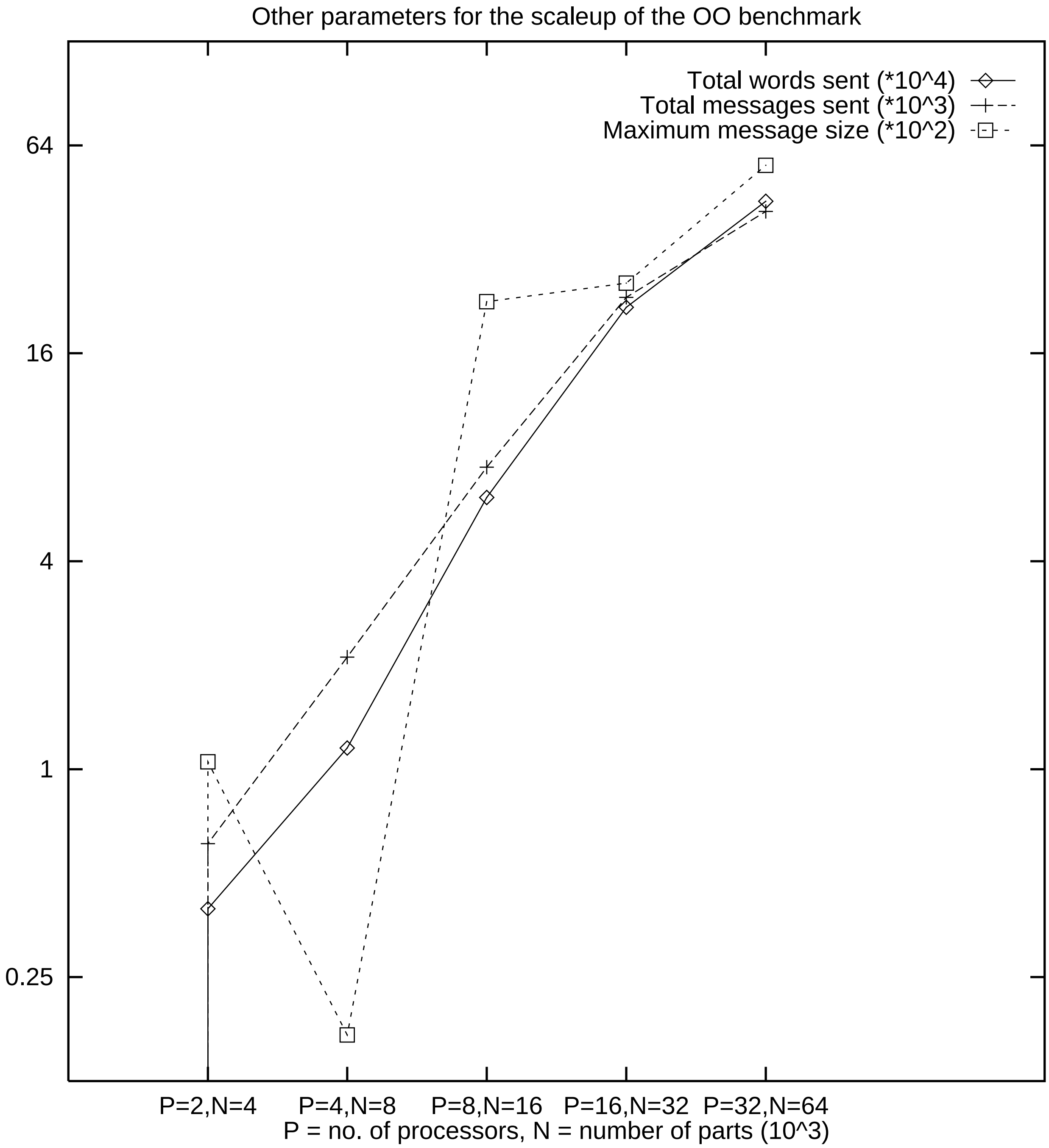
The number of parts is $np = 2,000P$, the assembly hierarchy depth is constant $h = 4$. The fan-out d varies such that the number of assemblies na grows approximatively linear, $na \approx 1,100P$. 25% of all assemblies are faulty.

Figure 6.23: Scaleup of the OO benchmark on a CM5



The residual work is uniformly distributed among the processors. The slight variation with P is due to the way we construct our benchmark.

Figure 6.24: Residual work for the scaleup of the OO benchmark



Total size of all messages sent (in words), the number of messages sent, and their maximum size (in words), for scaleup experiments for the OO benchmark.

Figure 6.25: Other parameters for the OO benchmark

too many processors (P in the range $4 \ldots 64$ in our experiments), the communication time is dominated by the data messages. Hence it decreases when P increases. Had we increased P even further (say $P = 128$) we would have observed an increase of T_{comm}, because as the size of the local data decreases, T_{comm} becomes dominated by the control messages, which take time $O(\log P)$.

In order for T_{comm} to decrease as in Figure 6.14 it is essential for the network to allow simultaneous point-to-point communications, as in the case in the LogP model. Naturally, we expect T_{comm} to increase on a shared medium like the Ethernet or a common bus, due to network congestion.

Recall that the communication-and-computation time T_{comp} consists of T_{comp} plus the computation time for the communication part, plus the "useful" computation time. Since the useful work (the residual work) is evenly distributed among the P processors (Figure 6.16), the communication-and-computation time in Figure 6.14 speeds up nicely with P.

Overlapping of the communications and the "useful" computations does not occur in our experiments. Recall that both benchmarks have a very simple structure, namely a communication phase followed by a computation phase: since the two phases are dependent, the computation phase has to wait for the communication phase to complete. An optimized compiler could interleave the communication and computation phases of two or more independent tasks and thus overlap communications with computations.

There exists however overlap of communication and computations within the communication phase, namely between the communications and the "unuseful" computations, need during the communication time. We call these computations "unuseful" because they are not done in a sequential version of the benchmark. They consists in splitting sequences into blocks before sending, preparing the control messages, etc. The speedup graphs in Figure 6.14 and the scaleup graph in Figure 6.18 reveal that the communication-and-computation time (T) is roughly equal the communication-only time (T_{comm}) plus the ideal running time ($\frac{T_{comp}}{P}$). The "unuseful" computation time, needed to prepare the communications did not increase the total running time.

The total number of words sent plotted in Figure 6.17 reinforces the observation that the communication phase is dominated by the data messages. Indeed the total number of words in the data messages is constant (because N is constant), while that in the control messages varies ($P-1$ 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. In short, the communication time is dominated by the data messages. The other two plots in Figure 6.17 reveal a negative phenomenon of the BVRAM implementation on the LogP model: data fragmentation. As P grows, more data messages are exchanged. Indeed the graphs shows that the total number of messages grows linearly with P. Data fragmentation also means that the size of the data messages decreases, as the experiments confirm.

In the scaleup experiment, Figure 6.18, the communication time grows more quickly than in the speedup experiment, because the total number of elements per processor does not decrease: in our experiment, it is a constant $10,000$ for each sequence. This implies that the communication time never decreases with P (the drop of T_{comm} at $P = 32$ is due to message scheduling, and not to the total size and number of the messages). 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 communication time gets dominated by the control messages, and T_{comm} grows sharply.

For $P \leq 32$ the communication time is low enough to keep the total running time close to the ideal constant. As P grows beyond a certain point (64 in our case), the communication time is dominated by the control messages, and the total running time increases. This upper bound for P increases as the data size at each processor increases: we expect to get reasonable speedup beyond 64 processors for, say, $40,000$ elements per processor. As expected, the residual work shown in Figure 6.19 disperses as P grows. The maximum message size shown in Figure 6.20 stabilizes at $10,000$, which means that, at some point, a processor has to send its whole X or Y sequence to another processor. The total number of words sent increases linearly, because the total size of both the data messages and control messages increase linearly. This is unlike the speedup experiment, where the total size of data messages was constant.

The speedup experiment for the OO benchmark, Figure 6.21 shows the same behavior as in the speedup experiment for *merge*. Here the communication-only time decreases too, but less sharply. We did not show the minimum and maximum residual work, because they were uniformly distributed.

The larger gap between the communication time and the ideal computation time shown in the scaleup experiment of Figure 6.23 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. The residual work at each processor shown in Figure 6.24 is uniformly distributed among the processors.

For the scaleup experiment, 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

There is a stronger variation in the residual work per processor in the scaleup experiment (Figure 6.24) than in other experiments, due to the way we generated our data sets. Namely 25% of all assemblies are flawed, and these are uniformly distributed among the total na assemblies: for each such flawed assembly, all its parts have to be processed during the residual work. E.g. if the root assembly were flawed, then all parts would have to be processed during the residual work, but we avoided making the root assembly flawed. However for small outdegrees ($d = 10 \ldots 16$), an increase by only one flawed assembly on the second level of the hierarchy increases the residual work by $6\% - 10\%$.

While the experiments altogether suggest the feasibility of our model, they are limited by the following:

1. The choice of the LogP model and of the simulator. While the LogP model specifies a maximum latency and deals with network congestion, it ignores the fact that the latency may be actually a function of the network load. In consequence our simulator assumes a maximum latency for each message. On a real network however, some messages may be delivered faster, e.g. when the network is small, or not very loaded.

2. The impact of I/O. For simplicity, we chose to ignore the time for input and output in our experiments. Moreover, I/O caches found in today's modern storage systems (see e.g. [25]) would add another dimension to the complexity of the model. But we expect I/O time to benefit from parallelism in our implementation techniques. Since sequences are evenly partitioned among the P nodes, the total I/O time should decrease as P increases, because more I/O's are done in parallel. Also as P increases, the total cache of all processors increases, thus improving the performance of the system.

3. The choice of the data sets. While we tried to chose representative data, we only considered a single data set for each benchmark. Different data sets could lead to different results especially by generating a load imbalance ($\delta \gg 1$). While our implementation techniques enable us to keep the load imbalance under control, experiments on a larger collection of data sets are needed to validate this.

Chapter 7

Conclusions and Further Work

7.1 Summary

We have described high-level parallel programming constructs for collections like sets, bags, and sequences (lists). Although we did not emphasize it, most of the constructs are inspired from the fundamental mathematical principles governing the collection types: sets, bags, and sequences, each form a *monad* in the category of sets and most operations in $\mathcal{NCC}$ are derived from this observation [21, 22]. The two alternative presentations of $\mathcal{NCC}$, the ext and the map-and-flatten presentations (Section 2.2), correspond to the presentation of a monad in *extension form*, and in *monoid form* respectively [78]. And, finally, the various forms of recursions considered here are derived from the simple observation that each of the three collections considered is a particular kind of a *freely generated monoid* [19].

Next we have shown that a particular query language for sets centered around a construct called *divide and conquer recursion on sets* can express exactly the queries which are in NC. This result extends previously known results connecting various query languages for sets with sequential complexity classes like $PTIME$, $PSPACE$, NP. Connected to this result, we also discussed related forms of recursion on sets, and encoding methods for flat and nested relations.

Implementing efficiently high-level parallel constructs on collections on existing parallel multiprocessors is a major challenge. We addressed it by designing an implementation language, called $\mathcal{MAP}$, which meets two conflicting demands: first, it is powerful enough to express naturally high-level parallel algorithms needed in the implementation of the parallel languages for collections, and, second, it is cut down enough to admit an efficient implementation on parallel multiprocessors. We gave evidence showing that $\mathcal{MAP}$ satisfies both demands. First we showed how to express in $\mathcal{MAP}$ a variety of parallel algorithms needed to implement high-level parallel query languages, and secondly we have shown that $\mathcal{MAP}$ can be compiled into a simple vector model, the BVRAM.

Finally we addressed the problem of implementing the BVRAM on a shared-nothing architecture. We used the LogP model as our implementation target. Due to their simplicity, most of the BVRAM instructions require only *monotone* communications among the processors, as opposed to arbitrary communications. Central to our implementation of the BVRAM on the LogP model is an efficient implementation of the monotone communications. We run experiments on two simple database benchmarks: a flat database benchmark and an object-oriented database benchmark. For both benchmarks we run both in *speedup* experiments, in which the database size is kept constant and the number of processor grows, and in *scaleup* experiments, in which both the database size and the number of processors grow simultaneously. Our results showed good performance when the database size at each node is large enough, and the local computation expensive enough, to justify the communication cost payed during the load-balance. Lantz, Nowicki, and Theimer [73] study the the effect of varying different parameters on the performance of a distributed application, and show that that the highest impact on performance is given by processor speed. The same remains true for the kind of applications which we found to benefit mostly from our parallel implementation techniques.

7.2 Further Work

A number of possible projects emerge from this thesis. We mention some of them here.

Complexity of other Languages for Collections Ordering seems to play a crucial role in capturing complexity classes below NP with query languages for sets, and our characterization of NC is no exception. Indeed, it follows from Theorem 7.8 in [66] that in the absence of ordering $FO + \mathsf{dcr}$ cannot express the lower bound in [24] which is in AC^0 plus parity gates ([24] Remark 7.2). As with $PTIME$, $DLOGSPACE$, etc., it remains an important open question whether there exists an r.e. set of "programs" that express exactly the NC-computable queries over arbitrary relational databases.

On the other hand, studying the expressiveness of the various forms of recursion on sets in the absence of ordering is quite relevant to query language design. It may also be relevant to complexity theory, if an analog to the surprising result of Abiteboul and Vianu [4] holds. They have shown that $PTIME \neq PSPACE$ iff first-order least fixpoint queries $\neq$ first-order *while* queries. (Vardi had shown that in the presence of order the $FO+while$ captures $PSPACE$ [104].) Dawar, Lindell, and Weinstein [42] give a machine-independent proof of the Abiteboul and Vianu result making use of properties of bounded variable logics. Abiteboul, Vardi and Vianu [1] give evidence for the robustness of the idea with several such results for other pairs of complexity classes. In our case, the analog would be: $NC \neq PTIME$ iff $FO + \mathsf{dcr} \neq FO + \mathsf{sri}$ (in our formalism, $\mathcal{NRC}^1(\mathsf{dcr}) \neq \mathcal{NRC}^1(\mathsf{sri})$). By setting aside the ordering, with its potential for tricky encodings, this would strengthen the observation (Section 3.7) that the difference between tractable sequential and tractable parallel computation can be characterized as the difference between two ways of recurring on sets.

While low complexity classes have been nicely captured with languages for sets, there has been less work done for characterizing these classes with languages for other collections, like bags or sequences. There is an inherent difficulty in doing that: unlike sets, we can build arbitrarily large bags or sequences with a small number of atomic objects. This makes it hard to design a *natural* programming language which stays within a low complexity class. Recently however Grumbach and Milo [52] have made progress on this issue.

Object IDs and Object Encapsulations Object id's, and object encapsulation are among the key features of today's object-oriented databases [27]. We have partially addressed the integration of these features in parallel languages, but much work still remains to be done.

As mentioned earlier, the database adaptation of the concept of object encapsulation is that objects should encapsulate both *data* and *methods*. The external functions in Σ considered here have some of the functionality of the *methods* found in object-oriented databases. But methods add more complexity, in that they are not confined to a fixed, finite set: unlike external functions, methods can be added to and deleted from an object-oriented databases. In the context of a parallel language, methods offer a nice interface between the parallel and the sequential world. The benchmarks of Section 6.6 are examples of how sequential external functions can be integrated in a parallel algorithm. More work has to be done in order to achieve good load balance for arbitrary external functions and/or methods. For example [60] discuss how runtime information can be gathered and used to guide parallel thread creation. Similar techniques can be adapted to guide the distribution of work among the processors.

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 As mentioned earlier, several successful parallel database servers are already available on the market, see [46] for a review. Their draw their success from well engineered implementations of the relational algebra operations: but this also implies their limitation to flat relations. For more advance operators, Valduriez and Khoshafiain [102] describe experiments with a parallel implementation of transitive closure on a shared-nothing architecture. None of this work studies the cost of the communications to the level of detail enabled by the recent LogP model.

Our approach of studying the communication cost of database operations in the framework of the LogP model is new. But we only addressed the implementation of the BVRAM instructions. We believe that optimal, or near-optimal parallel implementations can be found for some database operations, like join, set union, set difference, nest, and unnest on the LogP model. For sequences, we leave open the problem whether bmRoute admits a provably optimal implementation on the LogP model; also we would like to find a more efficient implementation of sbmRoute, the most complex of the BVRAM instructions.

Optimizations An important research topic which we have not addressed is that of optimizations. All elements for database-like optimizations are in place. The high-level parallel query languages discussed here admit a clean equational theory, and the target LogP model offers a detailed cost model. Our hope is that one could use the equational theory of the high-level language to transform a given query into an equivalent one which minimizes the cost of the implementation on the LogP model. Of course, lot of work has to be done, and we plan to address this sometimes in the future.

Bibliography

[1] S. Abiteboul, M. Vardi, and V. Vianu. Fixpoint logics, relational machines, and computational complexity. In *Structure and Complexity*, 1992.

[2] Serge Abiteboul and Catriel Beeri. On the power of languages for the manipulation of complex objects. In *Proceedings of International Workshop on Theory and Applications of Nested Relations and Complex Objects*, Darmstadt, 1988. Also available as INRIA Technical Report 846.

[3] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison Wesley Publishing Co, 1995.

[4] Serge Abiteboul and Victor Vianu. Generic computation and its complexity. In *Proceedings of 23rd ACM Symposium on the Theory of Computing*, 1991.

[5] 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.

[6] 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.

[7] Joshua S. Auerbach, Ajei S. Gopal, Mark T. Kennedy, and James R. Russell. Concert/C: Supporting distributed programming with language extensions and a portable multiprotocol runtime. In *Proceedings of ICDCS*, 1994.

[8] John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[9] 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.

[10] 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.

[11] David Mix Barrington, Neil Immerman, and Howard Straubing. On uniformity within NC^1. *Journal of Computer and System Sciences*, 41:274–306, 1990.

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

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

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] V. Breazu-Tannen and R. Subrahmanyam. Logical and computational aspects of programming with Sets/Bags/Lists. In *LNCS 510: Proceedings of 18th International Colloquium on Automata, Languages, and Programming, Madrid, Spain, July 1991*, pages 60–75. Springer Verlag, 1991.

[20] Val Breazu-Tannen. Generalized structural recursion and sets vs. bags, January 1992. Manuscript available from `val@saul.cis.upenn.edu`.

[21] 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.

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

[23] Paul Butterworth, Allen Otis, and Jacob Stein. The GemStone object database management system. *Communications of the ACM*, 34(10):64–77, October 1991.

[24] Jin-Yi Cai, Martin Furer, and Neil Immerman. An optimal lower bound on the number of variables for graph identification. *Combinatorica*, 12(4):389–410, 1992.

[25] 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.

[26] 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.

[27] R. G. G. Cattell, editor. *The Object Database Standard: ODMG-93.* Morgan Kaufmann, San Mateo, California, 1994.

[28] Ashok Chandra and David Harel. Computable queries for relational databases. *Journal of Computer and System Sciences*, 21(2):156–178, 1980.

[29] Ashok Chandra, Dexter Kozen, and Larry Stockmeyer. Alternation. *Journal of the ACM*, 28:114–133, 1981.

[30] Rohit Chandra, Anoop Gupta, and John L. Hennesy. COOL: a language for parallel programming. In David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 126–148. The MIT Press, Cambridge, Massachusetts, 1990.

[31] 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.

[32] P. Clote. Sequential, machine-independent characterizations of the parallel complexity classes $AlogTime$, AC^k, NC^k, and NC. In Samuel R. Buss and Philip J. Scot, editors, *Feasible Mathematics.* Birkhäuser, Boston, 1990.

[33] E. F. Codd. A relational model for large shared databank. *Communications of the ACM*, 13(6):377–387, June 1970.

[34] Latha S. Colby. *Query Languages and a Unifying Framework for Non-traditional Data Models.* PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana 47405-4101, May 1993. Available as Indiana University Computer Science Technical Report 381.

[35] Kevin L. Compton and Claude Laflamme. An algebra and a logic for NC. *Information and Computation*, 87(1/2):240–262, 1990.

[36] S. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64:2–22, 1985.

[37] Thomas H. Cormen, Charels E Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[38] David Culler, Andrea Dusseau, Seth Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in Split-C. In *Proceedings of Supercomputing*, 1993.

[39] 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.

[40] 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.

[41] 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.

[42] Anuj Dawar, Steven Lindell, and Scott Weinstein. Infinitary logic and inductive definability over finite structures. *Information and Computation*, 1993. To appear. Available as UPenn Technical Report MS-CIS-91-97.

[43] K. Denninghof and V. Vianu. The power of methods with parallel semantics. In *Proceedings of 17th International Conference on Very Large Databases*, 1991.

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

[45] 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.

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

[47] R. Fagin. Finite model theory — a personal perspective. *Theoretical Computer Science*, 116(1):3–32, August 1993.

[48] 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.

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

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

[51] 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.

[52] Stephane Grumbach and Tova Milo. An algebra for pomsets. In G. Gottlob and M. Vardi, editors, *Database Theory — ICDT'95*, pages 191–208, Prague, Czech Republic, January 1995. Springer Verlag.

[53] Stephane Grumbach and Victor Vianu. Expressiveness and complexity of restricted languages for complex objects. In *Proceedings of 3rd International Workshop on Database Programming Languages, Naphlion, Greece*, pages 191–202. Morgan Kaufmann, August 1991.

[54] Stephane Grumbach and Victor Vianu. Tractable query languages for complex object databases. Technical Report 1573, INRIA, Rocquencourt BP 105, 78153 Le Chesnay, France, December 1991. Extended abstract appeared in PODS 91.

[55] Stephane Grumbach and Victor Vianu. Tractable query languages for complex object databases. In *Proceedings of 10th ACM Symposium on Principles of Database Systems*, 1991.

[56] Y. Gurevich. Algebra of feasible functions. In *Proceedings of 24th IEEE Symposium on Foundations of Computer Science*, pages 210–214. IEEE Computer Society Press, 1983.

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

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

[59] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, CA, January 1995.

[60] Lorenz Huelsbergen, James Larus, and Alexander Aiken. Using the run-time sizes of data structures to guide parallel-thread creation. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, June 1994.

[61] Neil Immerman. Upper and lower bounds for first-order expressibility. *Journal of Computer and Systems Sciences*, 25:76–98, 1982.

[62] Neil Immerman. Relational queries computable in polynomial time. *Information and Control*, 68:86–104, 1986.

[63] Neil Immerman. Expressibility as a complexity measure: Results and directions. In *Proceedings of 2nd Conference on Structure in Complexity Theory*, pages 194–202, 1987.

[64] Neil Immerman. Languages that capture complexity classes. *SIAM Journal of Computing*, 16:760–778, 1987.

[65] Neil Immerman. Expressibility and parallel complexity. *SIAM Journal of Computing*, 18:625–638, 1989.

[66] Neil Immerman, Sushant Patnaik, and David Stemple. The expressiveness of a family of finite set languages. In *Proceedings of 10th ACM Symposium on Principles of Database Systems*, pages 37–52, 1991.

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

[68] Laxmikant Kale and Sanjeev Krishnan. CHARM++, a portable concurrent object oriented system based on C++. In *OOPSLA '93, Eights Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 91–108, 1993.

[69] 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.

[70] Kathleen Knobe, Joan D. Lukas, and Guy L. Steele Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.

[71] 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.

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

[73] Keith A. Lantz, William I. Nowicki, and Marvin M. Theimer. An empirical study of distributed application performance. *IEEE Transactions on Software Engineering*, SE-11(10):1162–1174, October 1985.

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

[75] Leonid Libkin and Limsoon Wong. Aggregate functions, conservative extension, and linear orders. In Catriel Beeri, Atsushi Ohori, and Dennis E. Shasha, editors, *Proceedings of 4th International Workshop on Database Programming Languages, New York, August 1993*, pages 282–294. Springer-Verlag, January 1994. See also UPenn Technical Report MS-CIS-93-36.

[76] Leonid Libkin and Limsoon Wong. New techniques for studying set languages, bag languages, and aggregate functions. In *Proceedings of 13th ACM Symposium on Principles of Database Systems*, pages 155–166, Minneapolis, Minnesota, May 1994. See also UPenn Technical Report MS-CIS-93-95.

[77] 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.

[78] Ernest G. Manes. *Algebraic Theories*, volume 26 of *Graduate Texts in Mathematics*. Springer-Verlag, Berlin, 1976.

[79] Y. N. Moschovakis. *Elementary Induction on Abstract Structures*. North Holland, 1974.

[80] Zhijing Mou and Paul Hudak. An algebraic model for divide-and-conquer and its parallelism. *Jounal of Supercomputing*, 2:257–278, 1988.

[81] 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.

[82] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database programming in Machiavelli, a polymorphic language with static type inference. In James Clifford, Bruce Lindsay, and David Maier, editors, *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 46–57, Portland, Oregon, June 1989.

[83] Jan Paredaens and Dirk Van Gucht. Possibilities and limitations of using flat operators in nested algebra expressions. In *Proceedings of 7th ACM Symposium on Principles of Database Systems,Austin, Texas*, pages 29–38, 1988.

[84] 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.

[85] 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.

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

[87] 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.

[88] M. Sipser R. Boppana. The complexity of finite functions. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science. Vol A: Algorithms and Complexity.* MIT Press, 1990.

[89] Pushpa Rao and Clifford Walinsky. An equational language for data-parallelism. In *Proceedings of 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 112–118. ACM Press, 1993.

[90] W. Ruzzo. On uniform circuit complexity. *Journal of Computer and System Sciences*, 22:365–383, 1981.

[91] H.-J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, 1986.

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

[93] Larry Stockmeyer and Uzi Vishkin. Simulation of parallel random access machines by circuits. *SIAM Journal of Computing*, 13:409–422, May 1984.

[94] Michael Stonebraker. The case for shared nothing. *Database Engineering*, 9(1), March 1986.

[95] Dan Suciu. Fixpoints and bounded fixpoints for complex objects. In Catriel Beeri, Atsushi Ohori, and Dennis Shasha, editors, *Proceedings of 4th International Workshop on Database Programming Languages, New York, August 1993*, pages 263–281. Springer-Verlag, January 1994.

[96] Dan Suciu. Domain-independent queries on databases with external functions. 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 177–190. Springer Verlag, January 1995.

[97] Dan Suciu and Val Breazu-Tannen. A query language for NC. In *Proceedings of 13th ACM Symposium on Principles of Database Systems*, pages 167–178, Minneapolis, Minnesota, May 1994. See also UPenn Technical Report MS-CIS-94-05.

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

[99] Dan Suciu and Limsoon Wong. On two forms of structural recursion, January 1995.

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

[101] S. J. Thomas and P. C. Fischer. Nested relational structures. In P. C. Kanellakis and F. P. Preparata, editors, *Advances in Computing Research: The Theory of Databases*, pages 269–307, London, England, 1986. JAI Press.

[102] 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.

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

[104] M. Y. Vardi. The complexity of relational query languages. In *Proceedings of 14th ACM SIGACT Symposium on the Theory of Computing*, pages 137–146, San Francisco, California, 1982.

[105] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Acive messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992.

[106] Clifford Walinsky and Deb Banerjee. A functional programming language compiler for massively parallel computers. In *Proceedings of ACM Conference on Lisp and Functional Programming*, Nice, June 1990.

[107] Limsoon Wong. Normal forms and conservative properties for query languages over collection types. In *Proceedings of 12th ACM Symposium on Principles of Database Systems*, pages 26–36, Washington, D. C., May 1993. See also UPenn Technical Report MS-CIS-92-59.

[108] 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.