# Implementation and Analysis of a Parallel Collection Query Language

Dan Suciu*

AT&T Research, USA
suciu@research.att.com

## Abstract

We study implementation techniques for a parallel query language for nested collections. The language handles collections of three kinds (sets, bags, and sequences), and its expressive power is essentially that of OQL (ODMG93). From the perspective of parallel evaluation, the novelty of such a query language is that it can express *nested parallelism*, which is naturally associated to nested collections. The first implementation step is a translation into a specially designed algebra for flat sequences, having only flat parallelism: the translation "flattens" the nested parallelism, and we prove that it preserves the asymptotic parallel complexity. The second step consists in an implementation of the sequence algebra on a shared nothing architecture. Here we show that all data communications needed by the sequence algebra operators (with one exception) have a particular communication pattern, called *monotone communication*. We give a provably optimal algorithm for monotone communications on a shared nothing architecture. Here "optimal" means that for any particular initial and final data layout, its communication cost is absolute minimum (not within a constant factor). To account for the communication costs we chose as shared nothing model the recently proposed LogP model. Finally we report some preliminary experiments of our implementation techniques, on a LogP simulator.

*This work was done while the author was at the University of Pennsylvania, and was partially supported by NSF Grant CCR-90-57570, ONR Contract NOOO14-93-11284, and by a fellowship from the Institute for Research in Cognitive Science.



**Proceedings of the 22nd VLDB Conference**
**Mumbai(Bombay), India, 1996**

## 1 Introduction

In theory, relational database operations offer great potential for parallel evaluation. On a shared nothing architecture, the database relations are uniformly partitioned horizontally across nodes, and, since the relational operators are so simple, they can be implemented efficiently terms of a number of communication steps between nodes. In practice however, even for the case of relational databases, there are several barriers to good speedup, like data skew and node interference [DG92].

Object-oriented database systems [Deu90, Cat94, ABD+92] complicate the issue of parallelism. Some parallel implementations [CDF+94, GCKL93] address explicitly one dimension of object-oriented databases, namely that of being a collection of objects with links between them. A typical query will start from one object in the database and access other objects by traversing links. When a link traverses node boundaries, either the query migrates to the other node [GWLZ93] or the remote object is fetched over the network [CDF+94]. But in today's parallel object-oriented databases, the main source of scalable parallelism are still the set operations inherited from the relational algebra.

Object-oriented databases however have the potential for another kinds of parallelism: *nested parallelism* associated to nested collections, and parallelism associated to the other kinds of collections, like arrays or sequences (these are still arrays, but with a richer set of operations). We discuss here implementation techniques for nested parallelism on three kinds of collections: sets, bags and sequences. We start from a high level query language for nested collections (like OQL [Cat94]), and end on a shared nothing architecture. Nested collections, and their associated nested parallelism, are more vulnerable to data skew than flat relations: therefore we discuss an implementation technique which guarantees good load balance. We give a complete analysis of the complexity of these techniques.

We start with an example showing the main ideas.

**An example** Consider $Stores(name, sales)$, a relation where *name* is a string, and *sales* is a bag (multiset) consisting of all sales at that store: $sales(price, item)$. The Object Definition Language ODL of the ODMG-93 proposal [Cat94] allows us to describe this as:

interface *sale*
{ attribute float *price*;
attribute string *item*;
};
interface *store*
{ attribute string *name*;
attribute Bag< *sale* > *sales*
};
Bag< *store* > *stores*;

Now consider the following OQL [Cat94] query $Q$:

select $f(x.name, x.sales)$ from $x$ in *stores* where $p(x.name)$

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

select $g(n, y)$ from $y$ in $s$

Then $Q$ has *nested parallelism*. Note that $g(n, y)$ has access both to the name $n$ of a store, and to some sales $y$ at that store.

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

Our solution is based on a flattening technique using *segment descriptors* [BS90, Ble90]. It only works for sequences, not for sets or bags, so a part of this paper deals with encodings of (nested) sets and bags in terms of sequences. To illustrate the technique, suppose that our database has 6 stores, with $5, 6, 4, 0, 1, 2$ sales respectively (for a total of 18). First we split *stores* into a flat sequence *names*, and a nested sequence *sales*:

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

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

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

The numbers in $ss$ are the lengths of the subsequences in *sales* and are called segment descriptors. Finally, once the whole database is represented only with flat sequences, we can distribute each of the sequences uniformly on the nodes. Figure 1 illustrates this for 3 nodes.

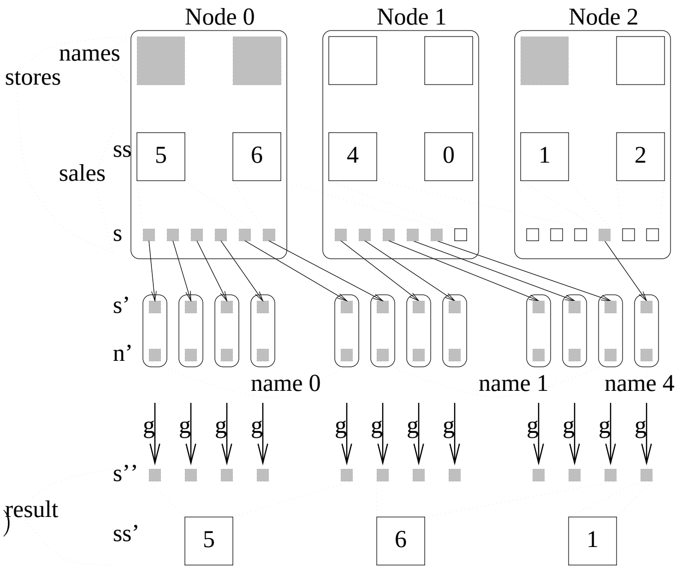


Figure 1: Implementation of the query $Q$.

Now we can explain how the query $Q$ can be computed on this data representation, step by step (see also Figure 1):

1. Apply the predicate $p$ to each store. Assume that $p$ returns true on stores $0, 1, 4$ (those shaded in Figure 1).

2. Send a 0 or 1 flag to every element in $s$, saying whether its store has been selected or not: this step involves data communication. The resulting subsequence of $s$, call it $s'$, has 12 elements (which are shaded in Figure 1).

3. Redistribute $s'$ uniformly on the 3 nodes.

4. We cannot apply $g$ on the elements in $s'$ yet, because $g$ needs access to the name of the store too. For that, replicate the names of the selected stores: the resulting sequence $n'$ contains the names of stores 0, 1, and 4, with multiplicities 5, 6, and 1 respectively.

5. Now apply $g$ on each name-sale pair from $n'$ and $s'$. This is again done locally, with perfect load balance. Call $s''$ the sequence of 12 results.

6. Let $ss'$ be the subsequence of $ss$ corresponding to the selected stores: $ss' = [5, 6, 1]$. Distribute $ss'$ uniformly on the nodes.

7. The final result is a nested sequence represented by the two sequences $s''$ and $ss'$.

Each step is a perfectly balanced parallel computation, but the price paid for that is that of frequent data communications: one wonders whether this price is not too high. Our *first main contribution* in this paper consists in observing that all these communications are of a particular, simple form, and to describe a provably optimal algorithm for computing them.

On the other hand one wonders how the example above generalizes to arbitrary queries on nested collections (like OQL queries). In this example, a key step consists in expressing nested collections in terms of flat sequences. Our *second contribution* consists in identifying a list of flat sequence operations which are simple enough to parallelize, and yet sufficient to implement "efficiently" all queries on nested collections (like OQL queries).

We comment more on these two issues next.

**Communication cost** We implement the flat sequence operations on a shared nothing architecture in the obvious way: by partitioning sequences uniformly on the nodes, doing all communications required to compute that operation and having the result partitioned uniformly too. Thus every operation has perfect processor load balance, but at the cost of frequent data communications. This may saturate the network, leading to poor performance. We show however that all flat sequence operations (with one exception) require only a simple communication pattern, called *monotone communication* (definition in Section 4): communication steps 2, 3, 4, 6 above are monotone. More, we describe a provably optimal algorithm for monotone, one-to-one communications on a shared nothing architecture (Steps 3, 6 are one-to-one, the other two are one-to-many). Our optimality statement is as strong as it can get: we prove that for any given initial and final distribution of the data on the nodes, our algorithm achieves best possible running time. For many-to-one monotone communications, our algorithm does most of the job too, but needs to be followed by a parallel broadcast step: here we cannot prove absolute optimality, but it is easy to see that the running time does not exceed the optimal one by more than the time for broadcasting one data item, which is independent on the total size of the data (depends only on the number of processors). Hence, for all practical purposes, monotone communications admit highly efficient implementations.

We base our communication cost analysis on the LogP model, recently proposed by Culler et al. [CKP+93]. While precise enough to model virtually all modern multiprocessors [CKP+93, AISS95, Ble94, TM94], the LogP model is simple enough to serve as a framework for the design and analysis of parallel algorithms [KSSS93]. Our optimal algorithm for monotone communication is for the LogP model.

**Sequence operations** In order to choose the right sequence operations which can implement all nested collection queries (OQL), we start at a high level, by describing a *nested collection calculus*, $\mathcal{NCC}$, much in the spirit of [AB88, BBW92, VD91, FM95]. $\mathcal{NCC}$ is expressive enough to express all queries in OQL, but it is original in its choice of operations for sequences: we show that all sets and bag operations in $\mathcal{NCC}$ (and, hence, OQL) can be expressed "efficiently" (explained below) in terms of the sequence operations in $\mathcal{NCC}$. Having done that, we identify next which operations on flat sequences suffice to further translate all $\mathcal{NCC}$ sequence operations "efficiently" in terms of flat sequence operations. In short, we get an efficient translation of $\mathcal{NCC}$ (OQL) into flat sequences.

**Parallel complexities** Traditionally, declarative query languages do not carry explicit information about the "complexity" of a query: the optimizer is expected to provide the best execution plan for a query. But since our translation from $\mathcal{NCC}$ to nested sequences and then to flat sequences involves complex query rewritings, we want to prove that these translations do not increase the complexity of the original query. Therefore we choose to associate a formal, abstract parallel complexity with $\mathcal{NCC}$: it corresponds to an ideal, but intuitive model of parallel query execution, assuming arbitrary many processors. One can easily infer the parallel complexity of an $\mathcal{NCC}$ query, based on this intuition. Then, we can make our claims about the efficiency of the query translations precise, by proving that the translations preserve the asymptotic parallel complexities. As explained above, the flat sequence operations are further implemented on parallel shared nothing architectures: their actual running time is the same as their abstract parallel complexity. In short we have shown that the real parallel running time can be deduced from the high-level, abstract, parallel complexity.

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

Bancilhon et al. [BBKV87] describe a database language FAD, designed for a parallel database machine Bubba. FAD is a functional language, featuring nested sets with object identifiers (no bags or sequences) and two main parallel constructs: a parallel map (called **filter**), and a parallel divide and conquer construct (called **pump**). Parker et al. [PSV92] describe a value-based model of parallelism in bulk data processing, SVP, in which both sets and streams are represented as trees. It has a complex parallel operator, called *transducer*, generalizing parallel map, parallel divide-and-conquer, and parallel stream processing. Neither FAD nor SVP have explicit sequence operations. Our language $\mathcal{NCC}$ is value-based, like SVP, but its implementation technique, which relies on flat sequences, differs from that of FAD and SVP: also, we focus on the parallel complexities and on the communication costs, which are not considered in FAD and SVP. Although we do not discuss divide-and-conquer here, this form of parallelism can be integrated in our techniques, using the ideas in [ST94].

This paper is organized as follows. In Section 2 we introduce the nested collection calculus, $\mathcal{NCC}$, its high-level parallel complexities, and show that it can express all OQL queries. In Section 3 we describe the first part of the implementation, namely the translation of $\mathcal{NCC}$ queries into flat sequences; we prove that the translation preserves the asymptotic complexity. Section 4 describes the second part of the implementation: here we start by describing the LogP model, show that all operations on flat sequences (with one exception) have monotone data communications, present a provably optimal algorithm for monotone, one-to-one communications, and show how it can be extended to an efficient algorithm for any monotone communication. Finally we discuss some preliminary experiments on a LogP simulator in Section 5.

## 2 The Nested Collection Calculus

We start by defining the high level query calculus $\mathcal{NCC}$ (Nested Collection Calculus).

### 2.1 Data Model

The Nested Collection Calculus, $\mathcal{NCC}$, operates on three collection types: sets, bags, and sequences. We denote **sets** as usual, e.g. $\{a, b, c\}$. **Bags**, sometimes called multisets, may have duplicates: e.g. $\{\!|a, a, b, c|\!\}$ has two occurrences of $a$, and is the same as $\{\!|a, b, a, c|\!\}$. **Sequences** are ordered collections, like arrays and lists: e.g. $[a, b, a, c]$ is a sequence of length 4, different from $[a, a, b, c]$. All collections are homogeneous. To simplify the presentation we use **tuples** instead of records: e.g. we write $\langle a, b, c\rangle$ for the 3-tuple containing $a, b$, and $c$, and $\langle\rangle$ for the empty tuple. Also, there are **base types**, which include *string*, *int*, *bool*, *real*, as well as an unspecified number of user defined base types. E.g. any C++ class can be an $\mathcal{NCC}$ base type.

Each literal in $\mathcal{NCC}$ has a precisely defined type. E.g. the pair $\langle 3, \text{“}abc''\rangle$ is of type $int \times string$, the set $\{3, 5, 8\}$ is of type $\{int\}$, while $\{\!|\langle 3.14, \{2, 5\}\rangle, \langle 2.71, \{3, 5, 2, 7\}\rangle|\!\}$ is a bag of type $\{\!|real \times \{int\}|\!\}$.

$\mathcal{NCC}$ includes a number of **external functions**: e.g. arithmetic operators on integers $(+, -, *, /$, etc), string operators, logical operators (not, and, or), as well as an unspecified number of user defined functions and predicates. For each base type $t$ we assume to have a canonical order relation denoted $\leq_t$.

Following sound design principles [BBW92, VD91], we organize the operators in $\mathcal{NCC}$ around the types: there will be independent operators for sets, for bags, for sequences, for product types, and for base types.

### 2.2 $\mathcal{NCC}$

$\mathcal{NCC}$ has basically the same set and bag operations as OQL, but with a more concise abstract syntax; we briefly sketch them here. $\mathcal{NCC}$ is original however in its choice of operations on sequences, which we discuss in the next subsection. A complete list of $\mathcal{NCC}$ operations is given in [Suc95].

The central operation in $\mathcal{NCC}$ is the **parallel map**. There are three kinds of map, one for sets, one for bags, and one for sequences. Let $f$ be some function expressible in $\mathcal{NCC}$: $\mathsf{map}_{sequence}(f)$ is a function mapping sequences to sequences, defined by:

$$\mathsf{map}_{sequence}(f)([x_0, \ldots, x_{n-1}]) \stackrel{\text{def}}{=} [f(x_0), \ldots, f(x_{n-1})]$$

Similarly, we have a $\mathsf{map}_{bag}$ for bags, and one for sets; we drop indexes when no confusion arises. $\mathsf{map}_{set}(f)(x)$ is a particular OQL select construct, namely:

$$\mathsf{select\ distinct}\ f(u)\ \mathsf{from}\ u\ \mathsf{in}\ x$$

An interesting case is when the function $f$ itself contains another map: in that case we obtain **nested parallelism**, as in the example of Section 1.

Other operations are: set union $(\cup)$, bag addition $(\oplus)$ and sequence concatenation $(@)$; flatten, which takes a set of sets and removes one level of parentheses, e.g. $\mathsf{flatten}(\{\{a, b\}, \{\}, \{b, c\}\}) = \{a, b, c\}$; similarly there exists a flatten for bags of bags, and one for sequences of sequence. $\mathcal{NCC}$ has a conditional, if then else , and an unspecified list of external functions, like primitive operations $(+, -, *, /$, etc.), and user-defined functions and predicates.

$\mathcal{NCC}$ derives lots of flexibility from nesting map constructs and mixing them with the flatten operator. E.g. the query select distinct $u$ from $x$ where $p(x)$ is expressed in $\mathcal{NCC}$ as $\mathsf{flatten}(\mathsf{map}(f)(x))$, where $f$ is: $f(u) = \mathsf{if}\ p(u)\ \mathsf{then}\ \{u\}\ \mathsf{else}\ \{\}$.

Cartesian product and join are expressed as nested map's. E.g. $x \times y$ is $\mathsf{flatten}(\mathsf{map}(f)(x))$ where $f(u) = \mathsf{map}(g)(y)$, and where $g(v) = \langle u, v\rangle$. In OQL syntax this becomes:

$$\mathsf{flatten}(\mathsf{select}\ (\mathsf{select}\ \langle u, v\rangle\ \mathsf{from}\ v\ \mathsf{in}\ y)\ \mathsf{from}\ u\ \mathsf{in}\ x)$$

### 2.3 Operators for sequences

Besides the operators mentioned in the previous subsection, $\mathcal{NCC}$ has sequence operators specially designed to allow both (1) an efficient implementation of sets and bags in terms of sequences, and (2) further efficient parallel implementation of the sequence operations. They are: enumerate, zip, and split. For $x = [x_0, \ldots, x_{n-1}]$ and $y = [y_0, \ldots, y_{n-1}]$, enumerate and zip are defined as:

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

Example: $\mathsf{zip}\langle x, \mathsf{enumerate}(x)\rangle$ pairs every element in $x$ with its index, $[\langle x_0, 0\rangle, \langle x_1, 1\rangle, \ldots, \langle x_{n-1}, n-1\rangle]$. zip produces an error when $x$ and $y$ have different length.

$\mathsf{split}\langle x, y\rangle$ is an inverse to flatten: it returns a sequence $z$ such that $\mathsf{flatten}(z) = x$. Here $y$ is a

$$T(\mathsf{map}(f),[x_0,\ldots,x_{n-1}]) \stackrel{\text{def}}{=} 1+\max_{i=0,n-1} T(f,x_i)$$

$$W(\mathsf{map}(f),[x_0,\ldots,x_{n-1}]) \stackrel{\text{def}}{=} \mathit{size}([x_0,\ldots,x_{n-1}]) + \sum_{i=0,n-1} W(f,x_i)$$

Figure 2: The definition of $T$ and $W$ for $\mathsf{map}(f)$.

sequence of integers dictating the lengths of subsequences in $z$, i.e. $z$ will be such that $\mathsf{map}(\mathsf{count})(z) = y$ ($\mathsf{count}$ computes the length of a sequence). E.g. $\mathsf{split}\langle[a,b,c,d,e],[2,0,3]\rangle = [[a,b],[],[c,d,e]]$. Of course, the sum of numbers in $y$ must be equal to the length of $x$, or else $\mathsf{split}\langle x,y\rangle$ is undefined.

We illustrate with an example which we will use in the sequel.

**Example 2.1** *Consider the query* $\mathsf{select}\langle x,y\rangle$ *taking two sequences $x$ and $y$ of the same length, and returning the sequence of those elements $y_i$ for which $x_i \neq 0$. E.g. when $x = [1,0,0,1,1,0]$ and $y = [a,b,c,d,e,f]$, then* $\mathsf{select}\langle x,y\rangle = [a,d,e]$. *We express* $\mathsf{select}\langle x,y\rangle$ *as* $\mathsf{flatten}(\mathsf{map}(f)(\mathsf{zip}\langle x,y\rangle))$*, where* $f\langle u,v\rangle = \mathsf{if}\ u \neq 0\ \mathsf{then}\ [v]\ \mathsf{else}\ []$. *That is* $\mathsf{map}(f)$ *returns a nested sequence: for $x,y$ above, it returns* $[[a],,[],[],[d],[e],[]]$. *Finally* $\mathsf{flatten}$ *will do the job.*

$\mathcal{NCC}$ is an expressive calculus. It can express set difference, nest, unnest, etc. More generally:

**Proposition 2.2** *$\mathcal{NCC}$ can express all OQL queries.*

### 2.4 Parallel Complexities

As promised, we formally define a high-level parallel complexity for $\mathcal{NCC}$. To every $\mathcal{NCC}$ query we associated two numbers, $T$ and $W$, called the **parallel time complexity**, and the **work complexity** respectively. Intuitively $T$ stands for an idealized parallel running time, assuming arbitrary many processors and zero-cost communication; $W$ stands for the total number of operations done during the computation (same as the sequential complexity). E.g. for $\mathsf{map}(f)$ (say, on sequences; for sets and bags, $T$ and $W$ are defined similarly), the rules defining $T$ and $W$ are given in Figure 2. For all other operations in $\mathcal{NCC}$, by definition $T \stackrel{\text{def}}{=} 1$, while $W \stackrel{\text{def}}{=}$ the sum of the size of the inputs and the outputs. It is with respect to these parallel complexities that we will prove our query translations into flat sequences to be efficient (Section 3). Some explanations are in order.

**$T, W$ for nested collection queries** The definitions of $T$ and $W$ may seem arbitrary at a first look, but they correspond to an intuitive, abstract parallel model of computation. E.g. to compute $\mathsf{map}(f)([x_0,\ldots,x_{n-1}])$, one computes $f(x_0),\ldots,f(x_{n-1})$ in parallel, then assembles the results in a sequence: hence $T$ is $1+\max T(f,x_i)$. As another example, consider $x \cup y$. Imagine that the sets $x$ and $y$ are given as sorted sequences: then, using enough processors, we can merge $x$ and $y$ in a constant number of parallel steps, and with linear work complexity, so $T \stackrel{\text{def}}{=} 1, W \stackrel{\text{def}}{=} \mathit{size}(x)+\mathit{size}(y)$. Although formal rules for $T$ and $W$ accompany every $\mathcal{NCC}$ language construct [Suc95], one can "read" $T$ and $W$ directly from the query, based on this idealized, but intuitive parallel model of computation.

**$T, W$ for flat sequences** For queries on flat sequences however, $T$ and $W$ are "real", in the following sense. We show in Section 4 that any flat sequence operation of work complexity $W$ (and $T = 1$) runs on a shared nothing architecture with $P$ processors in time $T_P = O(\lceil \frac{W}{P} \rceil)$. Therefore any flat query of complexities $T, W$ runs in time[1]:

$$T_P = O(T + \frac{W}{P}) \tag{1}$$

We care about the asymptotic behaviors of $T$ and $W$, not about their absolute values. E.g. $W = O(n^{1.5})$ is better than $W = O(n^2)$. Notice that, in absence of external functions, every query in $\mathcal{NCC}$ has $T = O(1)$. But external functions $f$ are considered to be sequential, hence $T \stackrel{\text{def}}{=} W \stackrel{\text{def}}{=}$ sequential running time of $f$. So queries with external functions may have $T$ larger than a constant. E.g. assume $f(z)$ sequentially processes the sequence $z$ in time $T = (\mathit{size}(z))^2$. Then the parallel time complexity of $\mathsf{map}(f)(x)$ applied to a matrix $x = [[x_{11},\ldots,x_{1n}],\ldots,[x_{n1},\ldots,x_{n1}]]$ is $T = O(n^2)$.

## 3 Translation into Flat Sequences

Here we show how to translate nested sets, bags and sequences into flat sequences. The main result in this section is:

**Theorem 3.1** *For any $\varepsilon > 0$ and any query $Q$ in $\mathcal{NCC}$ with complexities $T$ and $W$, there exists an equivalent query $Q'_\varepsilon$ using only flat sequences, with $T' = O(T)$ and $W' = O(W^{1+\varepsilon})$.*

We pay the $\varepsilon$ penalty in the work complexity in order to sort sequences representing sets and bags. We do not discuss the constants behind the $O(\ldots)$ in this section: while they are not out of control (they are $\approx 4$ in Example 3.2 below), Theorem 3.1 should be taken as existence proof. Additional optimization techniques are further required in order to reduce the constants behind the $O$'s. This is the topic of future work. We discuss now the two steps leading to Theorem 3.1.

We start by translating $\mathcal{NCC}$ queries involving sets and bags into sorted sequences (without duplicates, in the case of sets). We use the order relations at base types, which we lift at arbitrary types. The key step in the translation is a fast sort function [Suc95]: for every $\varepsilon > 0$ there exists an $\mathcal{NCC}$ expression which sorts[2] a

[1] Because $T_P = \sum_{i=1,T} O(\lceil \frac{W_i}{P} \rceil)$ where $W_i$ is the work complexity of the $i$'s operation [Ble90].

[2] We use a specialized form of distribution sort; algorithms like quicksort, with an $O(n \log n)$ sequential running time, typically lead to $T = O(\log n)$ parallel steps [Ble90], while here we need $O(1)$.

sequence $x$ of size $n$ in $T = O(1), W = O(n^{1+\varepsilon})$. Using this, we translate a $Q$ with $T, W$, into some $\mathcal{NCC}$ query $Q'_\varepsilon$, implementing $Q$ in terms of sequences (no sets or bags), with $T' = O(T)$ and $W' = O(W^{1+\varepsilon})$. Of course, we sort only sparingly, namely after those set or bag operations which require duplicate elimination. When sorting is unnecessary, the translation achieves a much better complexity: $T' = O(T)$ and $W' = O(W)$, with very low ($\approx 1$) constants under the $O$'s. E.g. the query $Q$ in the example of Section 1 can be translated in terms of sequences without sorting.

Next, we flatten the sequences in $\mathcal{NCC}$, and, in the process, flatten the nested parallelism. For this to work we need to choose the right primitive operations on flat sequences: those in $\mathcal{NCC}$ are not sufficient, because $\mathcal{NCC}$ can express some operations on flat sequences only by using nested sequences as intermediate results, like the select query in Example 2.1. The complete list of operations is given in [Suc95]. Most of them are trivial, or directly inherited from $\mathcal{NCC}$, except for three: $\mathsf{select}\langle x, y\rangle$ (Example 2.1), $\mathsf{bmRoute}\langle x, y, z\rangle$, and $\mathsf{sbmRoute}\langle x, y, z, u\rangle$. Here $\mathsf{bmRoute}\langle x, y, z\rangle$ ("bounded monotone routing") expects three input sequences, $x, y, z$, with $y$ and $z$ having the same length, and $y$ a sequence of integers. It produces a sequence in which each element in $z$ is replicated a number of times dictated by the corresponding element in $y$. The first parameter, $x$, is ignored, except for its length, which must match in length exactly the length of the output of bmRoute: we call $x$ a "bound". E.g. $\mathsf{bmRoute}\langle[x_0, x_1, x_2, x_3, x_4], [3, 0, 2], [a, b, c]\rangle = [a, a, a, c, c]$, i.e. $a$ has to be replicated 3 times, $b$ 0 times, and $c$ 2 times. The "bound" $x$ has length 5, matching the expected result. One can check that bmRoute can be expressed in $\mathcal{NCC}$, by using nested sequences as intermediate results, but one can prove that without the bound $x$ bmRoute is not expressible. Finally $\mathsf{sbmRoute}\langle x, y, z, u\rangle$, the most complex operation in $\mathcal{SA}$, is a "segmented" or "nested" version of bmRoute, defined to be $\mathsf{bmRoute}\langle x, y, \mathsf{split}\langle z, u\rangle\rangle$.

We call the collection of all flat sequence operations the **sequence algebra**, $\mathcal{SA}$. Next we encode nested sequences as flat sequences, using **segment descriptors** technique [BS90, Ble90] (Section 1). By adapting techniques from [ST94] we show that every $\mathcal{NCC}$ query on (nested) sequences with $T, W$ can be translated into an equivalent $\mathcal{SA}$ query with $O(T), O(W)$. Some operations are trivial to translate, like $\mathsf{split}\langle x, y\rangle$ when $x, y$ are flat sequences (it becomes a no-op). But other operations are really tricky, e.g. $Q\langle x, y\rangle = \mathsf{map}(@)(\mathsf{zip}\langle x, y\rangle)$.

This completes the proof of Theorem 3.1.

**Example 3.2** *Consider the query $Q$ in Section 1 with complexities $T, W$, and its implementation in 7 parallel steps. Theorem 3.1 applied to $Q$ produces the following $\mathcal{SA}$ query $Q'$, having*[3] *$T' = 4T$, $W' = 4W$:*

| | | | |
|---|---|---|---|
| *Step 1* | $f$ | $\leftarrow$ | $\mathsf{map}(p)(names)$ |
| *Step 2* | $f'$ | $\leftarrow$ | $\mathsf{bmRoute}\langle s, ss, f\rangle$ |
| *Step 3* | $s'$ | $\leftarrow$ | $\mathsf{select}\langle f', s\rangle$ |
| *Step 6* | $ss'$ | $\leftarrow$ | $\mathsf{select}\langle f, ss\rangle$ |
| *Step 4* | $n'$ | $\leftarrow$ | $\mathsf{bmRoute}\langle s', ss', \mathsf{select}\langle f, names\rangle\rangle$ |
| *Step 5* | $s''$ | $\leftarrow$ | $\mathsf{map}(g)\langle n', s'\rangle$ |

## 4 Implementing the Flat Sequences on the LogP Model

We show in this section that the flat sequence operations in $\mathcal{SA}$ can be implemented on a shared nothing architecture with guaranteed uniform processor balance. This is achieved by constantly rebalancing after each $\mathcal{SA}$ operation, with the potential risk of high communication cost. Our main result in this section consists in giving a provably optimal algorithm for the kind of communications needed in all $\mathcal{SA}$ operations (except sbmRoute).

### 4.1 The LogP model

To account for the communication cost we use the LogP model introduced by Culler et al. in [CKP+93]. The model is essentially a shared-nothing architecture, in which $P$ independent nodes are connected by a communication network enabling point-to-point communications. The focus of the LogP model are the parameters of this network, and its goal is to provide a realistic framework for the design and analysis of parallel algorithms. Moreover, the resulting algorithms can be fine-tuned to the parameters of a particular network.

The model describes the network in terms of the *latency* $L$, the *gap* $g$, and the *overhead* $o$. $L$ is an upper bound on the time needed for one byte of a message to traverse the network from the sender node to the receiver. The overhead, $o$, is the time a processor is busy preparing a message for send or for receive. The gap, $g$, is the minimum time interval between two consecutive bytes sent or received by a node, and is the inverse of the network's bandwidth. Moreover, to avoid network saturation, no more than $L/g$ bytes are allowed to be in transit between any two processors, at any given time.

In the original LogP model, messages have a fixed size, which is assumed to be a small number of bytes. We slightly extend it by allowing variable length messages (but without adding a fourth parameter as [AISS95]). Figure 3 illustrates the cost of sending a message of size $s$, starting at time $t$. The receiver fully receives the message at time $t + 2o + L + (s-1)g$. Except for the overhead, there may be overlap of computation and communication at any given node: e.g. the sender can initiate sending the next message at time $t + (s-1)g$. Also, we assume a dual port model: a node can both send and receive a message at the same time.

[3] E.g. look at *names*: $Q$ apparently touches it only once (by mapping $p$ on it), while $Q'$ does 4 operations on *names* and $f$, which has the same size.

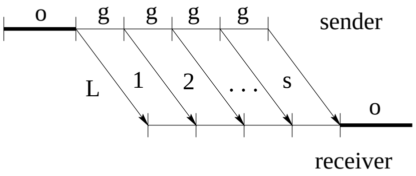


Figure 3: A message of size $s$ sent on the LogP model.

Poor message orchestration can significantly slow down the communications because of:

- Receive/receive conflict: when several nodes send simultaneously to the same destination, all but one will stall.
- Send/receive conflict: although on our dual port model a message can be sent while another is received at the same node, the two processor overheads $o$ cannot overlap.

We will focus on efficient orchestration of messages for the communications in our flat sequence operations. We start with data partitioning.

### 4.2 Data partitioning

We distribute flat sequences using *block partitioning.* To illustrate, assume first that the length $n$ of some sequence $x = [x_0, x_1, \ldots, x_{n-1}]$ is divisible by $P$, i.e. $n = Pn_0$. Then we store the subsequence $x^i = [x_{in_0}, x_{in_0+1}, \ldots, x_{in_0+n_0-1}]$ at node $i$, $i = 0, P-1$. When $n$ is not divisible by $P$, then we store an additional item in the first ($n$ *mod* $P$) nodes. Always $x^0 @ x^1 @ \ldots @ x^{P-1} = x$, and the lengths of any two $x^i, x^j$ differ by at most one. We also store the total length, $n$, at each node.

Sequences of user-defined types may have large data skew, i.e. $size(x_i)$ and $size(x_j)$ may differ considerably. To guarantee good load balance in this case, we represent such objects as sequences of bytes: then $x = [x_0, \ldots, x_{n-1}]$ becomes a nested sequences of bytes, which we first unnest into two sequences using segment descriptors, then partition both sequences using block partitioning. In consequence objects may cross node boundaries: we explain how to implement a $\mathsf{map}(f_0)(x)$ operator in $\mathcal{SA}$, with $f_0$ a user-defined function. First we *cluster* $x$, i.e. repartition it such that no object crosses node boundaries, and that the distribution is still uniform, only then apply $f_0$ in parallel. More precisely, let $s \stackrel{\text{def}}{=} (\sum_{i=0,n-1} size(x_i))$. Scanning from left to right, *cluster* assigns object $x_i$, $i = 0, n-1$, to node $j$, $j = 0, P-1$ as follows. If the total size currently assigned to $j$ is $< s/P$, then assign $x_i$ to $j$; else increase $j$. This ensures that each processor's load is $\leq s/P + \max_{i=0,n-1} size(x_i)$. Of course, *cluster* will run in parallel, using a parallel prefix sum and the monotone communication algorithm described next. Assuming that the complexity of the external function $f_0$ is at least linear ($T(f_0, x_i) \geq c \cdot size(x_i)$, for some $c > 0$), then the running time of *cluster* (ignoring the prefix sum) is $O(T + \frac{s}{P})$, since each node has to perform a work of cost $O(s/P + \max_{i=0,n-1} size(x_i)) = O(s/P + \max_{i=0,n-1} T(f_0, x_i)) = O(s/P + T)$.

### 4.3 An example: $\mathsf{select}\langle x, y\rangle$

We take now a close look at the steps needed to implement some $\mathcal{SA}$ operations, say $z = \mathsf{select}\langle x, y\rangle$. To begin with, each node $i$ holds the subsequences $x^i$ and $y^i$ of $x$ and $y$ respectively. Then $\mathsf{select}\langle x, y\rangle$ requires three communication steps:

1. Each node $i$, $i = 0, P-1$, starts by counting locally the number of nonzero elements occurring in $x^i$: call this number $n_i$. In a global communication step, we compute a **prefix sum** on the $n_i$'s: after that, node $i$ will hold the value $m_i \stackrel{\text{def}}{=} n_0 + n_1 + \ldots + n_i$.
2. The last node **broadcasts** $m$ $(= m_{P-1})$ to all other nodes: this will be $\mathsf{length}(z)$.
3. Having $m_i$ and $m$, each node will be able to determine for each of its local elements $y_j$, what position they take in the final result $z$, and, sends it to the corresponding node.

The first two communications are **control communications**, and the last one **data communication**. The only control communications needed in $\mathcal{SA}$ operations are broadcasts and/or prefix sums, for which provably optimal implementations exists [KSSS93]. Still, optimality here is not crucial, since their cost depends only on $P$ (they are $O(\log P)$), and not on the database size. We focus on data communications next.

### 4.4 Monotone communications

It turns out that our data communications have special patterns. Formally, let a **communication problem** for the LogP model be given by a number of *items*, a number of *cells*, both distributed on the nodes, and a one-to-many relation from items to cells, saying which item has to be sent to which cell(s). The problem asks to orchestrate the messages such that all items arrive at their destination in a minimum global time.

Let $x = [x_0, \ldots, x_{n-1}]$ be the sequence of all items held by all $P$ nodes, and $x^i$ the subsequence of $n_i$ items stored at node $i$: $x = x^0 @ x^1 @ \ldots @ x^{P-1}$. Similarly, let $y = [y_0, \ldots, y_{m-1}]$ be the sequence of all destination cells, and $y^i$ the subsequence stored at node $i$. In a communication problem, each item $x_k$ has to be sent to a number of cells $y_{d_{k1}}, y_{d_{k2}}, \ldots$, possibly residing on several nodes. When each item $x_k$ has exactly one destination, $d_{k1}$, then we call the communication problem **one-to-one**, otherwise we call it **one-to-many**. Figure 4 contains three examples of communication problems: (b) is one-to-one, while (a), (c) are one-to-many. Next, we call a communication problem **monotone** if for any two distinct items $x_k, x_{k'}$, with $k < k'$, any two destination cells of these items $y_{d_{kl}}, y_{d_{k'l'}}$ satisfy: $d_{kl} < d_{k'l'}$. Examples (b) and (c) of Figure 4 are monotone, while (a) is not. Intuitively a communication problem is monotone iff the arrows drawn as in Figure 4 do not intersect each other.

Our central observation is that all data communications in the $\mathcal{SA}$ operations (except $\mathsf{sbmRoute}$), are

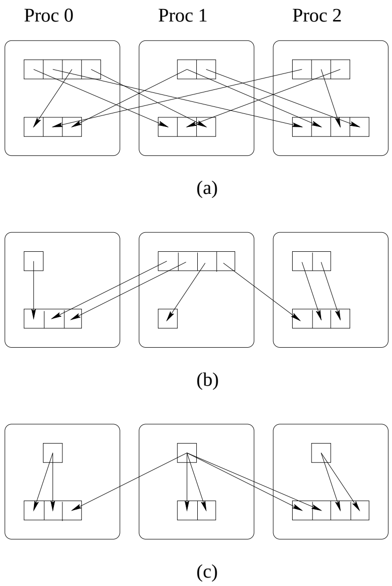


Figure 4: Communication problems, for $P = 3$ nodes.

monotone: in fact they are even one-to-one, except for bmRoute which is monotone, one-to-many. Indeed: for sequence concatenation, $x@y$, we perform two monotone, one-to-one communications (the first moves $x$ to the left, and the second moves $y$ to the right); for select$\langle x, y\rangle$ we need a monotone, one-to-one communication with some elements of $y$; for map$(f_0)$ we need to cluster, then apply $f_0$ locally, then decluster the result, and both cluster and decluster are monotone, one-to-one. We focus next on the implementation of monotone communication problems.

### 4.5 A provably optimal algorithm for monotone, one-to-one communications

We give here a simple algorithm for the monotone, one-to-one communication problem, an prove it is optimal whenever $sg > 3o$, where $s$ is the size of the items $x_k$, $g$ and $o$ are the gap and overhead respectively. Since items do not need to be replicated, we basically obtain optimality by avoiding receive/receive and send/receive conflicts.

Recall that each item $x_k$ on node $i$ has to be sent to exactly one destination cell $y_{d_{k1}}$, say on node $j$. We will assume[4] that the node $i$ knows the relative position of cell $y_{d_{k1}}$ on node $j$.

The algorithm consists of two threads at each node: a *send-thread* and a *receive-thread*, with the *send-thread* of higher priority. The *send-thread* at node $i$ starts by partitioning its items into blocks, according to their destination node, Figure 5. Each block will be sent in a single, possibly long message to its destination node. Because the communication is monotone, only blocks 0 and $\lambda$ (first and last) may generate receive/receive conflicts at their destinations. To avoid them, we send the blocks in the order $\lambda, \lambda - 1, \ldots, 0$: hence, each node $j$ will receive its messages in left-to-right order. Then, the only possible receive/receive

```
procedure send-thread
  for b = λ to 1 do
      send block b to processor j_b

  let t be the time needed to send the λ blocks above
  let δ = the relative position of x_{k_0} in node j_0
  let t' = δg
  if t' > t then wait t' − t time units else continue
  send block 0 to processor j_0
procedure receive-thread
  for b' = 1 to λ' do
      let t = time left to next activation of send-thread
      if t < o then wait t + o time units else continue
      receive block b'
```

Figure 6: Optimal communication algorithm.

conflict remains only for block 0: there may be other nodes, say $i-1, i-2, \ldots$ sending to the same node $j_0$. Hence the *send-thread* waits for all processors to the left to complete their sends. Since it knows the cell number at node $j_0$ where item $x_{k_0}$ goes, it can compute exactly how much time it needs to wait before it is its turn. The *send-thread* is described in Figure 6.

The *receive-thread* receives messages and stores them in their destination cell, at full speed. The only problem is that it has to avoid send/receive conflicts, Figure 7. The black boxes represent time intervals in which a thread needs to acquire the processor: two black boxes are not allowed to overlap. Before starting to accept a message, the *receive-thread* checks the state of the *send-thread*: if the time $t$ left before the *send-thread* needs to acquire the processor is $< o$, then the *receive-thread* waits, for at most $t + o$ time units. Even in this case, the delayed black box will not interfere with the next black box, provided that $3o \leq sg$. We can prove [Suc95]:

**Theorem 4.1** *When $3o \leq sg$, then the algorithm in Figure 6 is optimal for the monotone one-to-one communication problem.*

When applied to an $\mathcal{SA}$ operation with work complexity $W$, the running time is $\approx g\frac{W}{P} = O(\frac{W}{P})$.

### 4.6 An efficient algorithm for monotone, one-to-many communications

For the case of one-to-many monotone communications, we proceed in two steps: (1) a one-to-one communication step, in which each item is sent only to its rightmost destination, (2) a number of local broadcasts (group communications), in which each item is replicated to its left. The cost of the latter is independent on the data size, and depends only on $P$. We don't know whether the two steps combined form an optimal algorithm (the optimal algorithm may overlap the two steps), but the running time $t$ is obviously bounded by $t \leq t_{optimal} + t_{broadcast}$, where $t_{opt}$ is

[4] This assumption is true for all our $\mathcal{SA}$ operations.

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

Figure 5: The *send*-blocks in the optimal monotone, one-to-one algorithm.

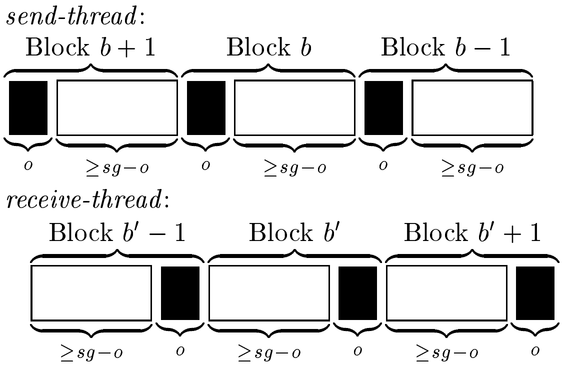


Figure 7: Possible send/receive conflicts.

the optimal running time, and $t_{broadcast}$ is the broadcast time. The latter is $O(\log P)$, and is independent of the data size.

### 4.7 Running time for $\mathcal{SA}$ operations.

Hence every $\mathcal{SA}$ instruction (except sbmRoute) is implemented in time $T_P = O(\frac{W}{P}) + O(\log P)$, the first term being the cost of data communication, and the last one the cost of control messages. We will argue in the next section that, under realistic assumptions, the total cost is dominated by the first term.

## 5 Preliminary results on a LogP simulator

In order to test the feasibility of our LogP implementation techniques, we run some preliminary experiments for two simple benchmarks on a LogP simulator. We fed the simulator with the $L$, $o$, and $g$ parameters for the Connection Machine 5 ($L = 60, o = 105, g = 100$, measured in clock units) and the SP1 multiprocessors ($L = 70, o = 120, g = 55$) [Ble94]: we observed no changes in overall behavior, so we report here only the experiments with the parameters for the CM5. Our primary goal was to compare the cost of communications to that of the "useful" computation, and the cost of control communications and to that of data communications.

We ran two benchmarks: a simple *merge* algorithm, and an object-oriented flavored benchmark, inspired from the OO7 benchmark of [CDN93]. For lack of space we report only the results of the first one: both are fully described in [Suc95].

The *merge* function is a pragmatic adaptation of Valiant's parallel merge algorithm [Val75, Jaj92]. Two sorted sequences $x = [x_0, x_1, \ldots, x_{m-1}]$ and $y = [y_0, y_1, \ldots, y_{n-1}]$ are given as inputs. In order to merge them, our algorithm starts by dividing $x$ into $\sqrt{m}$ subsequences of length $\sqrt{m}$: $x = x^0 @ x^1 @ \ldots @ x^{\sqrt{m}-1}$. In parallel for each subsequence $x^i$, it finds the corresponding subsequence $y^i$ of $y$ with which $x^i$ has to be merged. Then $x^i$ with $y^i$ are merged using an external, sequential *sequential-merge* function, in parallel for $i = 0, \sqrt{m} - 1$. (Valiant's algorithm continues recursively, for a total of $O(\log \log m)$ parallel steps.) For $m$ sufficiently large, the number of $\sqrt{m}$ subsequences which have to be merged in parallel is large enough to allow a reasonably good processor load balance.

The data sets for the *merge* benchmark were generated such that $x$ is uniformly distributed in the interval $[x_0, x_{m-1}]$, while $y$ is uniformly distributed in the middle third of $x$. While these data sets were easy to generate, they may not cover the worst case, when one subsequence $y^i$ is equal to $y$: a second pass of the algorithm however can guarantee perfect load balance in this case too [Suc95].

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

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

We tested two performance metrics *speedup* and *scaleup* [DG92]. In the speedup experiments we kept the data size constant, and varied $P$ from 1 to 64: both $x$ and $y$ have length 100,000. In the scaleup experiments we varied $P$ again from 1 to 64, and chose the length of $x$ and $y$ to be $10,000 \cdot P$.

The speedup of *merge* in Figure 8 reveals a nice surprise: the communication time decreases with $P$. For $P = 1$ there are no communications at all. When $P$ increases, the time for the control communications is $O(\log P)$, forcing the total communication time to increase. But at the same time each processor has to send less data messages, because the input data size is constant. What the graph in Figure 8 says is that for a large input size $N = 100,000$ and not too many processors, 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 the communication time, because it starts being dominated by the control communications. However for larger data sets, the data communications would be still dominant for even larger $P$'s.

Once we account for the computation time too, the graph in Figure 8 shows a good overall speedup.

The total number of words sent plotted in Figure 8 reinforces the observation about the communication time being dominated by the data communications. Indeed the total number of words in the data messages is constant (because the input size is constant), while that in the control messages 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. The other two plots in Figure 8 reveal a negative phenomenon of our implementation on the LogP model. As $P$ grows, more and smaller data messages are exchanged: the total number of messages grows linearly with $P$.

In the scaleup experiment, Figure 9, the communication time grows faster than in the speedup experiment, because the cost of data communications per node remains roughly constant, while that of control communications increases. Still the growth is not significant as long as the communication phase is dominated by the data messages, as in the range $P = 4 \ldots 32$. For $P = 64$ however, the communication time starts getting dominated by the control messages, and it grows sharply. This upper bound for $P$ increases as the data size at each processor increases: we expect to see good scaleup beyond 64 processors for, say, $40,000$ elements per processor. The maximum message size shown in Figure 9 stabilizes at $10,000$, which means that, at some point, a processor sends all its $x$ or $y$ subsequence to another processor. Unlike in the speedup experiment, the total number of words sent increases linearly both for the data messages and for the control messages.

## 6 Conclusions and Future Work

We have discussed parallel implementation techniques for databases organized as nested sets, bags, and sequences. At a high level we have shown how nested parallelism can be flattened in an efficient way w.r.t. the asymptotic parallel complexities. At a low level we have discussed implementations on the LogP model, showing absolute optimality for data communications. Several issues remain to be addressed:

**Optimizations** The translation of $\mathcal{NCC}$ queries into $\mathcal{SA}$ queries (Theorem 3.1) preserves only the *asymptotic* parallel complexities: this is OK for an existence proof, but for practical purposes the constant behind the $O$'s need to be decreased. This can be done by algebraic transformations of the resulting $\mathcal{SA}$ expression.

**Joins** $\mathcal{NCC}$ has no special join operation, but can express it with 2 nested map's (like in Subsection 2.2). Written like this, its complexity is $T = O(1), W = O(n^2)$ (assuming both relations of size $n$). We can do better, by representing relations as sequences, sorting them, then using merge join: this gives us $T = O(1), W = O(n^{1+\varepsilon})$, for arbitrarily small $\varepsilon > 0$, which leads to a $O(W^{1+\varepsilon}/P)$ parallel running time (equation 1). But of course, we would do much better by using one of the existing specialized parallel join algorithm [SD89]. It is of interest to integrate our techniques for nested parallelism with existing specialized parallel algorithms for flat relations, by treating flat relations as special cases.

## References


[AB88] 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.

[ABD+92] 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.

[AISS95] Albert Alexandrov, Mihai Ionescu, Klaus Schauser, and Chris Scheiman. LogGP: Incorporating long messages into the LogP model. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 95–105, 1995.

[BBKV87] 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.

[BBW92] 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.

[Ble90] Guy E Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, Cambridge, Massachusetts, 1990.

[Ble94] 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.

[BS90] Guy E Blelloch and Gary Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8:119–134, 1990.

[Cat94] R. G. G. Cattell, editor. *The Object Database Standard: ODMG-93.* Morgan Kaufmann, San Mateo, California, 1994.

[CDF+94] 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.

[CDN93] 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.

[CKP+93] 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.

[Deu90] O. Deux. The story of $O_2$. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, March 1990.

[DG92] David DeWitt and Jim Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

[FM95] Leonidas Fegaras and David Maier. Towards an effective calculus of object query languages. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 47–58, June 1995.

[GCKL93] Shahram Ghandeharizadeh, Vera Choi, Clifford Ker, and Kai-Ming Lin. Design and implementation of the Omega object-based system. In *Proceedings the Fourth Australian Database Conference*, pages 198–209, 1993.

[GWLZ93] Shahram Ghandeharizadeh, David Wilhite, Kaiming Lin, and Xiaoming Zhao. Object placement in parallel object-oriented database systems. In *Proceedings of the Tenth International Conference on Data Engineering*, pages 198–209, February 1993.

[Jaj92] Joseph Jaja. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[KSSS93] 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.

[PSV92] 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.

[SD89] Donovan Schneider and David DeWitt. A performance evaluation of four parallel join algorithms in a shared nothing multiprocessor environment. In *Proceedings of the 1989 SIGMOD Conference*, Portland, 1989.

[ST94] 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.

[Suc95] Dan Suciu. *Parallel programming languages for collections.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, August 1995. Available as University of Pennsylvania IRCS Report 95-18.

[TM94] L.W. Tucker and A. Mainwaring. CMMD: Active messages on the CM-5. *Parallel Computing*, 20(4), April 1994.

[Val75] Leslie G. Valiant. Parallelism in comparison problems. *SIAM Journal of Computing*, 4(3):348–355, 1975.

[VD91] Scott Vandenberg and David DeWitt. Algebraic support for compolex objects with arrays, identity, and inheritance. In *Proceedings of the 1991 SIGMOD Conference on Management of Data*, 1991.

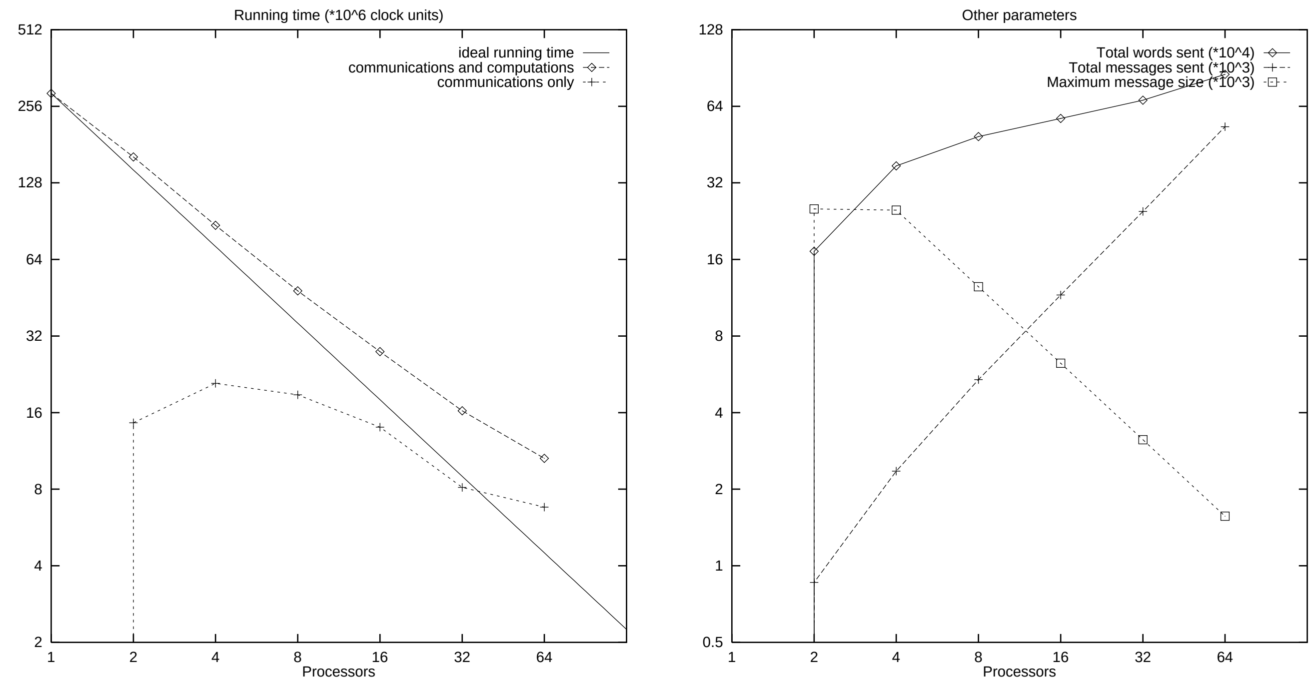


Figure 8: Speedup of $merge(x, y)$.

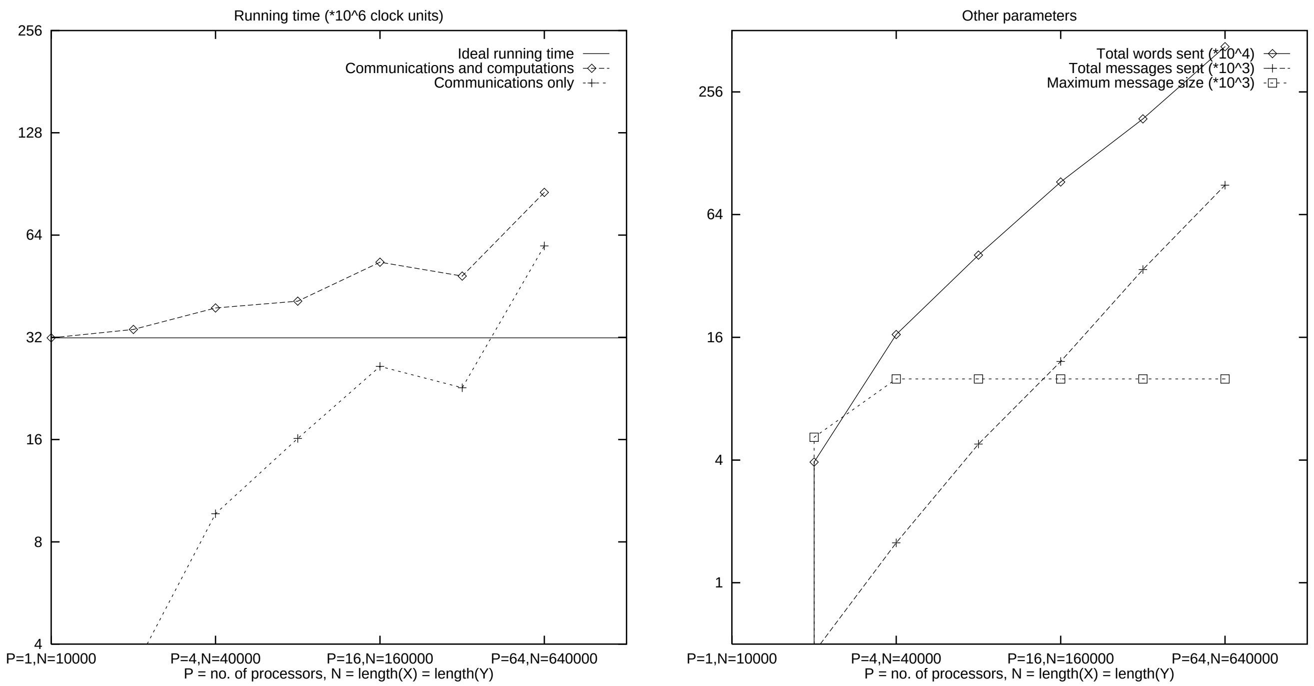


Figure 9: Scaleup of $merge(x, y)$.