Distributed and Parallel Databases 5, 405–438 (1997)
© 1997 Kluwer Academic Publishers. Manufactured in The Netherlands.

# A Parallel Scheme Using the Divide-and-Conquer Method

QI YANG AND SON DAO
*Information Science Laboratory, Hughes Research Laboratories, Malibu, CA 90265*

CLEMENT YU
*Department of EECS, University of Illinois at Chicago, Chicago, IL 60607*

NAPHTALI RISHE
*Florida International University*

*Received May 1, 1995; Accepted March 18, 1997*

**Recommended by:** Ahmed Elmagarmid

**Abstract.** A parallel scheme using the *divide-and-conquer* method is developed. This partitions the input set of a problem into subsets, computes a partial result from each subset, and finally employs a merging function to obtain the final answer. Based on a linear recursive program as a tool for formalism, a precise characterization for problems to be parallelized by the *divide-and-conquer* method is obtained. The performance of the parallel scheme is analyzed, and a necessary and sufficient condition to achieve linear speedup is obtained. The parallel scheme is generalized to include parameters, and a real application, the *fuzzy* join problem, is discussed in detail using the generalized scheme.

## 1. Introduction

The *Divide-and-Conquer* method is a common approach in designing efficient algorithms [3, 4, 8]. Intuitively, in this approach, a problem is decomposed into several subproblems of smaller sizes, each of which is similar or identical to the original problem; a partial solution is produced for each subproblem; then, the solution to the original problem is obtained by merging the partial solutions to the subproblems. An example is sorting, where the problem of sorting a set of elements can be decomposed into several subproblems, each involving the sorting of a disjoint subset of the original set, and a sorted list for the original set is obtained by combining the sorted lists for those subsets.

The *Divide-and-Conquer* method is a natural approach in parallel computing [11, 25, 37]. It is especially desirable in data intensive applications, e.g., in database systems and information retrieval systems. In *LDL* [28], which is a programming language for deductive databases, the *divide-and-conquer* method has been recognized as an important construct for potential parallel processing. In [17], it is pointed out that partitioned parallelism offers much better opportunities for speedup and scaleup than pipelined parallelism, and it

is recommended to use the *divide-and-conquer* method for exploiting partitioned parallelism. In [30], a new data model SVP is presented to capture parallelism in bulk data processing, both pipelined parallelism and partitioned parallelism, and *divide-and-conquer* mappings are formalized in the form of transducers. However, two issues have not been covered in these articles: one is the characterization of problems that can be parallelized by the *divide-and-conquer* method; another is the performance analysis for such parallel processing.

We focus on the two issues and apply the method to the fuzzy join problem. A linear recursive program is used to formalize problems to be parallelized. This program represents operations on sets and can be shown to include many problems in the database area and in other disciplines. We present a PArallel Divide-And-Conquer Scheme, or PADAC Scheme in short, for the linear recursive program. By proving a necessary and sufficient condition, we obtain a precise characterization for a problem to be parallelized by the *divide-and-conquer* method.

We analyze the performance of the PADAC Scheme. Linear speedup is the best a parallel algorithm can achieve when compared to a fastest sequential algorithm. We give a necessary and sufficient condition when the PADAC Scheme can achieve linear speedup. The result is a little surprising. Assume that a problem has time complexity $O(n^{c_1} \log^{c_2} n)$. Then linear speedup is possible only when $c_1 = 1$. For a problem of higher order complexity, i.e., when $c_1 > 1$, linear speedup cannot be achieved.

A generalization of the linear recursive program and the corresponding PADAC Scheme is also obtained. In this way, the divide-and-conquer method is applied to operations on sets with parameters, and data duplication is allowed among different processors. Some examples have shown the advantage of the generalized PADAC Scheme. We will discuss the application of this generalized PADAC Scheme to the fuzzy join problem: linear speedup is demonstrated.

The rest of the paper is organized as follows. The PADAC Scheme is formally defined in Section 2, and the characterization is discussed in Section 3. The performance of the PADAC Scheme is analyzed in Section 4. In Section 5, the generalization of the PADAC Scheme is discussed. In Section 6, we apply the generalized PADAC Scheme to solve the fuzzy join problem.

## 2. The PADAC scheme

### 2.1. A linear recursive program

We assume that a given problem can be considered as an operation on an **input set** $S$ to produce an **answer** $W$. Operations on sets are very common in database systems and other disciplines. For example, every database query language has some aggregate functions, which produce some values from a large number of tuples [27, 35]. Many other problems such as computing transitive closure and processing a join can also be considered as operations on sets. In logic programming, a predicate can be interpreted as a procedure [26]. In *LDL* [28], many operations on sets such as sorting are defined by logic programs.



We assume that a given operation on sets can be evaluated by the following linear recursive program.

Program 1

$r_1$ $p(\{x\}, W) :- sgl(x, W).$
$r_2$ $p(S, W) :- sel(x, S),$
$dif(S, x, SS),$
$p(SS, WS),$
$sgl(x, u),$
$mrg(u, WS, W).$

The predicate $sgl(x, u)$ represents a procedure that produces a result $u$ from a single element $x$ (*sgl* stands for *single*); the predicate $sel(x, S)$ represents a procedure that selects an element $x$ from a given set $S$ (*sel* stands for *select*); the predicate $dif(S, x, SS)$ represents a procedure that computes the subset $SS$ of $S$ by removing the element $x$ from $S$ (*dif* stands for *difference*); the predicate $mrg(u, WS, W)$ represents a procedure that merges $u$ and $WS$ to produce $W$ (*mrg* stands for *merge*). In the following, we will use predicate and procedure interchangeably.

The predicate $p(S, W)$ represents the operation on sets to be evaluated, where $S$ and $W$ represent the input set and the answer, respectively. Rule $r_1$ says that when the input set is singleton, i.e., when $S = \{x\}$, the answer $W$ is obtained by $sgl(x, W)$. Rule $r_2$ says that, in the general case, an element $x$ is chosen from the input set $S$ by $sel(x, S)$ a result $u$ is produced from the selected element $x$ by $sgl(x, u)$, the operation $p$ is carried out recursively on the remaining subset $SS$ to produce a partial result $WS$ by $p(SS, WS)$, and the final answer $W$ is produced from $u$ and $WS$ through $mrg(u, WS, W)$.

We point out here that Program 1 is actually a program scheme. For an operation on sets, after the specifications of those predicates are determined and the corresponding codes are written, Program 1 becomes a concrete program to evaluate the operation. Our intention is to construct a systematic approach for evaluating operations on sets in parallel using the Divide-and-Conquer Method. This would be especially beneficial for ordinary programmers who are accustomed to sequential programs but would like to have their programs parallelized.

Many operations involving sets can be defined in the form of Program 1. A query to a database can also be considered as an operation on a set (all tuples in the database involved). We will discuss two examples in Section 3.2: sorting and computing the transitive closure. More examples can be found in [40] including computing the similarity values between documents in information retrieval [32], the same generation problem, and the UP-FLAT-DOWN problem in deductive database [5, 23, 35].

To describe the execution semantics of Program 1, we introduce the following notations. Let $D$ and $R$ be two domains. We assume that the *input set* $S$ is a finite subset of $D$, and the *answer* $W$ is an element of $R$. The elements of $D$ and $R$ can be either simple values, e.g., numbers, or complex values, e.g., tuples, records, and even sets. For example, in sorting a set of records (with respect to a numerical field), the domain $D$ is the set of all records of the record type, an input set $S$ is a finite set of records, an answer $W$ is a sorted list

of records in $S$, and the domain $R$ is the set of all sorted lists of records. The predicate $sgl(x, u)$ represents a **mapping** $B$ which transforms a single element $x$ in the input domain $D$ to $u = B(x)$ in the output range $R$; the predicate $mrg(u, WS, W)$ represents a **merging function** $f$ which combines the partial results $u$ and $WS$ to form the answer $W$, and is defined as from $R \times R$ to $R$.

During an execution of Program 1, rule $r_2$ is called recursively. Each time it is called, an element $x$ is chosen by the predicate $sel(x, S)$. Let $x_i$ be the element chosen for the $i$th call, and $x_n$ the last element left. When only $x_n$ is left, rule $r_1$ is applied, and an element $u_n \in R$ is produced by the predicate $sgl(x, u)$. Then, $u_n$ is returned to the last call to $r_2$. An element $u_{n-1}$ is produced from $x_{n-1}$, $f(u_{n-1}, u_n)$ is computed by $mrg(u, WS, W)$ and returned to the previous call to rule $r_2$. This process continues until the final answer is computed.

A list $VL = [x_1, x_2, \ldots, x_n]$ with all elements taken from domain $D$ is a **valid list** for the set $\{x_i : 1 \le i \le n\}$ with respect to the predicate $sel(x, S)$ if $sel(x_i, S_i)$ is true for each $i$, where $S_i = \{x_k : i \le k \le n\}$. The predicate $sel(x, S)$ is used only when $S$ has two or more elements. For convenience, we assume that $sel(x, \{x\})$ is always true for any $x$. Conceptually, during an execution of Program 1, a valid list is generated first from the input set through the predicate $sel(x, S)$.

Let $L$ be a list with all elements taken from domain $R$ and $f$ a binary function on $R$. We define **f-back(L)** with respect to $f$ as the result by applying $f$ backward on $L$. That is

$$f\text{-}back([u_1]) = u_1, \quad \text{and}$$
$$f\text{-}back([u_1, u_2, \ldots, u_n]) = f(u_1, f\text{-}back([u_2, \ldots, u_n])), \text{ when } n > 1.$$

For example, $f\text{-}back([u_1, u_2, u_3, u_4]) = f(u_1, f(u_2, f(u_3, u_4)))$. Assume that, in an execution of Program 1, a valid list $VL = [x_1, x_2, \ldots, x_n]$ is generated from $S$. Let $u_i = B(x_i)$ for each $i$, and $VLB = [u_1, u_2, \ldots, u_n]$. The element $u_n$ is returned from rule $r_1$; from the last call to the recursive rule $r_2$, $f(u_{n-1}, u_n) = f\text{-}back([u_{n-1}, u_n])$ is returned; and so on. The final answer $W$ is $f\text{-}back(VLB)$. The execution of the linear recursive program is sequential, and the execution semantics can be described as follows:

> A valid list $VL = [x_1, x_2, \ldots, x_n]$ is generated from the input set $S$ through the predicate $sel(x, S)$;
> $VL$ is converted into $VLB = [u_1, u_2, \ldots, u_n]$ by the mapping $B$;
> The final answer $W = f\text{-}back(VLB)$.

### 2.2. *The PADAC scheme*

Our PADAC Scheme is developed to evaluate Program 1 on a shared-nothing architecture with $K$ processors for some constant $K \ge 2$. That is, each processor has its own main memory and secondary memory, and communication is accomplished in some way, say, by a bus or via a network. The number of processors or sites, $K$, is usually limited in practice, especially in comparison with the input data size. Most research that is devoted to developing parallel methods in database environment is carried out under such an assumption [1, 9, 13, 17, 21, 34, 38, 39].



The PADAC Scheme has three phases: dividing phase, processing phase, and merging phase. In the dividing phase, we require that $S$ be partitioned into $K$ non-empty subsets $S_i$. So, no element appears in multiple subsets. We assume that some rule has been specified such that for any set $S \subseteq D$ a *partial order* $\prec$ can be established on $S$. The partial order $\prec$ is required to be defined on the given set $S$, not necessarily on the entire domain $D$. For example, in sorting a set of numbers, the rule may be the comparison of numbers and the partial order is $<$, the arithmetic predicate *less than*. That is, $x_i \prec x_j$ if and only if $x_i < x_j$. On the other hand, in sorting a set of records with respect to a numerical field $nf$, the rule may be the comparison of the values of $nf$ and $x_i \prec x_j$ if and only if $nf_i < nf_j$. In both cases, the partial order $\prec$ can be defined on the entire input domain. In computing the transitive closure of a directed acyclic graph $G$, the rule may be specified according to reachability, and the partial order $\prec$ can be defined as follows. For any pair of edges $(x_1, y_1)$ and $(x_2, y_2)$, $(x_1, y_1) \prec (x_2, y_2)$ if and only if there is a path from $y_1$ to $x_2$ in the graph $G$. In this case, the partial order is associated with the input set, and not defined on the background domain.

An element $x$ of $S$ is a *minimal* (or *maximal*) element of $S$, if $S$ has no other element $y$ such that $y \prec x$ (or $x \prec y$); two different elements $x$ and $y$ are incomparable if neither $x \prec y$ nor $y \prec x$ is true; for any finite set $S$, there is a minimal (and a maximal) element; a subset $S_i \subset S$ is called an (proper) *initial segment* of $S$, if $S_i \ne \emptyset$, $S - S_i \ne \emptyset$, and for any $x \in S_i$, any $y \in S - S_i$, $y \prec x$ is not true. A partial order in which any two different elements are not ordered is called the *trivial partial order*. That is, any element is a minimal (and maximal) element, any two different elements are incomparable, and any proper subset is an initial segment.

The partition of the input set $S$ in our PADAC Scheme is done according to the partial order $\prec$ defined on $S$. The condition enforced is

> (*) For any $x \in S_i$ and any $y \in S_j$, $y \prec x$ is not true if $i < j$, i.e., no element from a later subset can be *smaller than* any element in any preceding subset.

Such a partition is called a **non-decreasing** partition. A *non-increasing* partition could be defined similarly. But, to simplify the discussion, we use *non-decreasing* partitions only.

We assume that all non-decreasing partitions for a given $S$ may be generated when different partitions are possible. That is, we only require condition (*) be enforced, but do not specify a fixed way in partitioning the input set. In most cases, $S$ should be partitioned equally such that each processor processes about the same amount of data.

In the processing phase, we require that the original Program 1 be evaluated at each site with one subset $S_i$ as input to produce a partial answer $W_i$. This is carried out independently without communicating with any other processors. So, the original problem is decomposed into $K$ identical problems. In an implementation, any other program for the same problem can be employed to compute $W_i$ at each site. That is, Program 1 is used only for the purpose of formalization and need not be used for execution.

In the merging phase, we require that the merging function $f$ represented by $mrg(u, WS, W)$ in Program 1 be applied to those $W_i$'s to form the final answer. The merging phase can also be carried out in parallel. There are different ways to apply $f$ to those $W_i$'s. To address this issue clearly, we introduce the notation of *f-terms* in the following. For

a given list $[u_1, u_2, \ldots, u_n]$, a **sublist** is a list of the form $[u_i, u_{i+1}, u_{i+2}, \ldots, u_k]$, where $1 \le i \le k \le n$. That is, a sublist contains some consecutive elements of the original list. Assume that $f$ is a function from $R \times R$ to $R$, and each $u_i$ is in $R$. An **f-term on a list** is defined inductively (but informally in logic) as follows:

$$f\text{-}term([u_1]) = u_1, \quad \text{and}$$
$$f\text{-}term([u_1, u_2, \ldots, u_n]) = f(f\text{-}term([u_1, \ldots, u_k]), f\text{-}term([u_{k+1}, \ldots, u_n])), \quad \text{where} \quad 1 \le k < n.$$

When $n > 1$, there are more than one way to form an $f$-*term* by choosing different values for $k$ in the above expression. In any case, an $f$-*term* on a list represents an element in $R$. To compute an $f$-*term* on a list $L$ with $n > 1$ elements, $L$ is partitioned into two sublists $L_1$ and $L_2$, and the result is

$$f\text{-}term(L) = f(f\text{-}term(L_1), f\text{-}term(L_2)).$$

The $f$-*back*$(L)$ defined in Section 2.1 is an $f$-*term* on $L$ by dividing $L$ in a special way. Although different $f$-*terms* on a list can be formed by dividing the list differently, the order of the elements in the list is not changed, and $f$ is applied $n - 1$ times on a list of $n$ elements. For example, $f(f(u_1, u_2), f(u_3, u_4))$ and $f(u_1, f(u_2, f(u_3, u_4)))$ are two different $f$-*terms* on the list $[u_1, u_2, u_3, u_4]$. To compute $f(u_1, f(u_2, f(u_3, u_4)))$, which is the $f$-*back* on the list, the function $f$ is applied three times, and the computation is sequential. For $f(f(u_1, u_2), f(u_3, u_4))$, $f$ is applied also three times, but the two $f$-*terms* $f(u_1, u_2)$ and $f(u_3, u_4)$ can be computed in parallel.

Different $f$-*terms* on the list $[W_1, W_2, \ldots, W_K]$ represent different ways to combine those partial answers. Assume $K$ is a power of 2, one way to combine those $W_i$'s in parallel is to combine $f(W_i, W_{i+1})$ for all odd $i$'s first, and then combine those merged results in a similar way. However, we do not specify a chosen way to combine those $W_i$'s, and only require that any two $f$-*terms* on a list be the same. That is, the final answer $W$ is an $f$-*term* on the list $[W_1, W_2, \ldots, W_K]$, but how $f$ is applied to two adjacent $f$-*terms* (with the order of $W_i$'s in the list unchanged) will not affect the final answer. This implies parallel execution in the merging phase and gives some flexibility in implementation. Our PADAC Scheme is represented by the following Algorithm 1, which reflects all these criteria on a parallel evaluation of Program 1.

**Algorithm 1**

**PHASE 1:** $S$ is partitioned non-decreasingly into $K$ non-empty subsets $S_i$;
**PHASE 2:** Each processor executes independently Program 1 with $S_i$ as the input to produce $W_i$;
**PHASE 3:** An $f$-*term* on the list $[W_1, W_2, \ldots, W_K]$ is computed as the final answer.

During an execution of Algorithm 1, $S$ is non-decreasingly partitioned into $K$ subsets $S_i$. Then, $K$ processors are working in parallel: each processor evaluates Program 1, produces a valid list $PL_i$ from $S_i$, converts it to another list $PLB_i$ by applying the mapping $B$ to all elements of $PL_i$ and obtains $W_i = f\text{-}back(PLB_i)$. The final answer $PW$ (the answer generated in parallel) is an $f$-*term* on the list $[W_1, W_2, \ldots, W_K]$. Let $PL$ (List generated in Parallel) be the concatenation of the $K$ valid lists $PL_i$ and $PLB$ the concatenation of those $K$ lists $PLB_i$. Then, $PW$ is an $f$-*term* on $PLB$, since each $W_i = f\text{-}back(PLB_i)$ is an $f$-*term* on $PLB_i$. The execution semantics of the parallel Algorithm 1 can be described as follows:



The input $S$ is non-decreasingly partitioned into $K$ non-empty subsets $S_i$;
A valid list $PL_i$ is generated from $S_i$ and converted into $PLB_i$ for each $i$, or a list $PL$, which is the concatenation of these $PL_i$'s, is generated from $S$ and converted into $PLB$, which is the concatenation of these $PLB_i$'s, in parallel;
The family of $W_i$'s, where $W_i = f\text{-}back(PLB_i)$, is computed in parallel;
The final answer $PW$ is an $f$-*term* on $[W_1, W_2, \ldots, W_K]$, which is also computed in parallel.

## 3. A necessary and sufficient condition

In this section, by proving a necessary and sufficient condition, we will give a precise characterization for a problem to be parallelized by the *divide-and-conquer* method.

### 3.1. The characterization

We now define the concept that the parallel Algorithm 1 is computationally equivalent to the sequential Program 1. In practical situations, multiple answers are possible for an operation on a given input set. For example, an *EMPLOYEE* relation is to be sorted on the attribute *AGE*, and two tuples $t_1$ and $t_2$ represent two employees with the same age. After the relation is sorted, we may exchange the positions of the two tuples $t_1$ and $t_2$ and still get a valid answer. Let $sgl(x, S)$ select a tuple with the smallest value on *AGE*. Then the predicate $sgl(x, S)$ has a choice of multiple elements, e.g., when $S = \{t_1, t_2\}$. That is, multiple valid lists for one input set are possible. On the other hand, both the mapping $B$ and the merging function $f$ are usually well defined, i.e., $B(x_1) = B(x_2)$ when $x_1 = x_2$, and $f(u_1, u_2) = f(v_1, v_2)$ when $u_1 = v_1$ and $u_2 = v_2$. Then, an answer is determined by a valid list from $S$, i.e., one *VL* gives one answer, and multiple answers are possible only when multiple valid lists exist. Thus, different executions of the sequential method can produce nondeterministically a set of valid lists, and the corresponding set of answers. We require different executions of the parallel method to produce nondeterministically the same set of valid lists and the same set of answers as the sequential method.

*Definition.* For a finite subset $S$ of $D$, let **AVL(S)** be the set of all possible valid lists from $S$ generated by Program 1, and **APL(S, K)**, the set of all possible *PL*'s from $S$ generated by Algorithm 1 with $K \ge 2$ processors. Algorithm 1 is said to be **Computationally equivalent** to Program 1 if

(1) For any input $S$ and any $K \ge 2$, $AVL(S) = APL(S, K)$; and
(2) For any $VL \in AVL(S)$, when it is generated by both Program 1 and Algorithm 1, Algorithm 1 gives the same answer as Program 1.

Recall that there is a partial order $\prec$ defined on $S$; Also the input set $S$ is very large and it is partitioned into $K$ non-empty subsets non-decreasingly with respect to $\prec$ in PHASE 1 of The PADAC Scheme, and all non-decreasing partitions for $S$ may be generated in PHASE 1 of Algorithm 1.

**Lemma 3.1.** *The following two statements are equivalent:*
1. *For any finite set $S \subset D$, and any $x \in S$, $sel(x, S)$ is true if and only if $x$ is a minimal element of $S$.*
2. *For any finite set $S \subset D$, $|S| > 1$, any $x \in S$, $sel(x, S)$ is true if and only if there is an initial segment $S_i$ of $S$ such that $x \in S_i$, and $sel(x, S_i)$ is true.*

**Proof:** Statement 1 $\Rightarrow$ Statement 2
Suppose $sel(x, S)$ is true, and $|S| > 1$. We now prove there is an initial segment $S_i$ of $S$ such that $x \in S_i$ and $sel(x, S_i)$ is true. By Statement 1, $x$ is a minimal element of $S$, i.e., for any element $y$ of $S$, $y \prec x$ is not true. Then, $\{x\}$ is an initial segment of $S$ containing $x$. Let $S_i$ be any initial segment of $S$ containing $x$. $S_i$ is a subset of $S$, and $x$ is a minimal element of $S$. So, $x$ is a minimal element of $S_i$. By Statement 1, $sel(x, S_i)$ is true.

Suppose there is an initial segment $S_i$ of $S$ such that $x \in S_i$ and $sel(x, S_i)$ is true. We now prove that $sel(x, S)$ is true. By Statement 1, $x$ is a minimal element of $S_i$, i.e., for any element $y \in S_i$, $y \prec x$ is not true. For any $y \in S - S_i$, $y \prec x$ is not true either, since $x \in S_i$ and $S_i$ is an initial segment of $S$. So, $x$ is a minimal element of $S$, and, by Statement 1, $sel(x, S)$ is true.

Statement 2 $\Rightarrow$ Statement 1
Suppose $sel(x, S)$ is true. We now prove by induction on $n$, the number of elements of $S$, that $x$ is a minimal element of $S$. When $n = 1$, $S = \{x\}$, and $x$ is a minimal element of $S$. When $n > 1$, by Statement 2, there is an initial segment $S_i$ of $S$ such that $x \in S_i$ and $sel(x, S_i)$ is true. By the definition of an initial segment, for any $y \in S - S_i$, $y \prec x$ is not true. Since $S - S_i \neq \emptyset$, we have $|S_i| < |S|$. By induction hypothesis, $x$ is a minimal element of $S_i$. Thus, $x$ is a minimal element of $S$.

Suppose $x$ is a minimal element of $S$, we now prove $sel(x, S)$ is true. Since $\{x\}$ is an initial segment of $S$, and $sel(x, \{x\})$ is true, $sel(x, S)$ is true by Statement 2. □

**Lemma 3.2.** *The following two statements are equivalent:*
1. *For any finite set $S \subset D$, and any $x \in S$, $sel(x, S)$ is true if and only if $x$ is a minimal element of $S$.*
2. *$AVL(S) = APL(S, K)$ for any $S$ and any $K \geq 2$.*

**Proof:** Statement 1 $\Rightarrow$ Statement 2
Let $S_i$, $i = 1, \ldots, K$, be a non-decreasing partition of $S$, where each $S_i$ has $n_i > 0$ elements. Suppose that $PL_i = [x_{i,1}, \ldots, x_{i,n_i}]$ is a valid list from $S_i$. Let $PL$ be the concatenation of these $PL_i$'s, that is,

$$PL = [x_{1,1}, \ldots, x_{1,n_1}, x_{2,1}, \ldots, x_{2,n_2}, x_{K,1}, \ldots, x_{K,n_K}].$$

Then, $PL \in APL(S, K)$. We now show that $PL \in AVL(S)$, and hence $APL(S, K) \subset AVL(S)$. For any $1 \leq i \leq K$, and any $1 \leq j \leq n_i$, let $S_{i,j} = \{x_{i,k} : j \leq k \leq n_i\}$. Since $PL_K$ is a valid list for $S_K$, $sel(x_{K,j}, S_{K,j})$ is true for all $j$, $1 \leq j \leq n_K$. If $sel(x_{i,j}, S_{i,j} \cup (\bigcup_{t=i+1}^{K} S_t))$ is true for all $1 \leq i < K$, and $1 \leq j \leq n_i$, then $PL$ is a valid list for $S$ and is in $AVL(S)$. It is easy to show that $S_{i,j}$ is an initial segment of $S_{i,j} \cup (\bigcup_{t=i+1}^{K} S_t)$, since $S_i$, $i = 1, \ldots, K$, is a non-decreasing partition, and $S_{i,j}$ is a subset of $S_i$. $PL_i$ is a valid list for $S_i$, so, $sel(x_{i,j}, S_{i,j})$ is true. By Statement 1 and Lemma 3.1, $sel(x_{i,j}, S_{i,j} \cup (\bigcup_{t=i+1}^{K} S_t))$ is true. That is, $PL \in AVL(S)$, and $APL(S, K) \subset AVL(S)$.

For any $VL \in AVL(S)$, we now show that $VL \in APL(S, K)$, and then $AVL(S) \subset APL(S, K)$. Let $PL = [x_1, x_2, \ldots, x_n]$ be a valid list in $AVL(S)$. By the definition of valid lists and Statement 1, $x_i$ is a minimal element of the set $\{x_j : i \leq j \leq n\}$. That is, $x_k \prec x_t$ is not true when $k > t$. $VL$ can be partitioned into $K$ non-empty sublists $VL_i$, $i = 1, \ldots, K$ (since $S$ can be partitioned into $K$ non-empty subsets). Let $S_i$ be the set containing all elements of $VL_i$. Since $x_k \prec x_t$ is not true when $k > t$, $S_1, S_2, \ldots, S_K$ is a non-decreasing partition of $S$, and $VL_i$ is a valid list for $S_i$ by Statement 1. The concatenation of these $VL_i$'s is $VL$, i.e., $VL$ can be generated by Algorithm 1 in parallel. So, $VL \in APL(S, K)$. Thus, $AVL(S) \subset APL(S, K)$, and hence $AVL(S) = APL(S, K)$.

Statement 2 $\Rightarrow$ Statement 1
We assume Statement 1 is false, and prove Statement 2 must be false too. By Lemma 3.1, we have the following cases.

*Case 1.* For some $S \subset D$, there is an element $x \in S$, such that $sel(x, S)$ is true, but for any initial segment $S_i$ containing $x$, $sel(x, S_i)$ is not true.
Suppose $S_j$, $j = 1, \ldots, K$ is a non-decreasing partition of $S$. No matter $x \in S_1$ or not, $sel(x, S_1)$ is not true. That is, no valid list $PL_1$ of $S_1$ will have $x$ as the first element, and no $PL \in APL(S, K)$ will have $x$ as the first element. However, since $sel(x, S)$ is true, there is a valid list in $AVL(S)$ with $x$ as the first element. Then, $APL(S, K) \not\subset AVL(S)$ and $APL(S, K) \neq AVL(S)$.

*Case 2.* For some $S \subset D$, there is an element $x \in S$, such that $sel(x, S)$ is not true, but $sel(x, S_i)$ is true for an initial segment $S_i$ of $S$.
Suppose $S$ is partitioned into $S_i$ and $S - S_i$ for $K = 2$. This is a non-decreasing partition of $S$. Since $sel(x, S_i)$ is true, there is a valid list for $S_i$ has $x$ as the first element, and there is a $PL \in APL(S, K)$ with $x$ as the first element. However, $sel(x, S)$ is not true, and no valid list in $AVL(S)$ will have $x$ as the first element. That is, $APL(S, K) \not\subset AVL(S)$ and $APL(S, K) \neq AVL(S)$. □

Assume that $VL = [x_1, x_2, \ldots, x_n]$ is a valid list for $S$, and $u_i = B(x_i)$ for each $i$. Let

$$VLB_1 = [u_1, \ldots, u_{k_1}], \quad VLB_2 = [u_{k_1+1}, \ldots, u_{k_2}], \quad \text{and} \quad VLB_3 = [u_{k_2+1}, \ldots, u_n],$$

where $1 \leq k_1 < k_2 < n$. Let $f$ be a function from $R \times R$ to $R$, and $t_j = f\text{-}back(VLB_j)$, $j = 1, 2, 3$. The function $f$ is called an **associative function w.r.t. valid lists**, if for any valid list $VL$, and any $t_1$, $t_2$ and $t_3$ defined above, $f(f(t_1, t_2), t_3) = f(t_1, f(t_2, t_3))$. If $f(f(u, v), w) = f(u, f(v, w))$ for any $u, v, w$ in $R$, $f$ is called an **associative function on R**. It is clear from the definitions that an associative function on $R$ is associative w.r.t. valid lists. A function which is not associative on $R$, but associative w.r.t. valid lists will be given in Section 3.2.2 when we discuss the transitive closure problem.

Recall that a list $VL = [x_1, x_2, \ldots, x_n]$ is a valid list for a set $\{x_i : 1 \leq i \leq n\}$ if $sel(x_i S_i)$ is true for each $i$, where $S_i = \{x_j : i \leq j \leq n\}$. For a sublist $VL = [x_i, x_{i+1}, \ldots, x_k]$, where $1 \leq i \leq k \leq n$, there is a subset $\{x_j : i \leq j \leq k\}$. The sublist may or may not be a valid list for the subset depending on the predicate $sel(x, S)$.

**Lemma 3.3.** *Assume that any sublist of a valid list is a valid list for the corresponding subset. Let* $VL = [x_1, x_2, \ldots, x_n]$ *be a valid list,* $u_i = B(x_i)$ *for each* $i$, *and* $VLB = [u_1, u_2, \ldots, u_n]$. *If* $f$ *is associative w.r.t. valid lists, then all* $f$*-terms on* $VLB$ *are equal.*

**Proof:** The proof is by induction on $n$, and we assume $n \geq 3$. When $n = 3$, $VL = [x_1, x_2, x_3]$, $VLB = [u_1, u_2, u_3]$. There are only two $f$-*terms* on $VLB$, i.e., $T_1 = f(f(u_1, u_2), u_3)$ and $T_2 = f(u_1, f(u_2, u_3))$. Let $t_j = f\text{-}back(u_j) = u_j$, $j = 1, 2, 3$. Then, $T_1 = f(f(t_1, t_2), t_3)$ and $T_2 = f(t_1, f(t_2, t_3))$. Since $f$ is associative w.r.t. valid lists, we have $T_1 = T_2$ by the definition.

Assume that the statement is true for any valid list with no more than $n$ elements, where $n \geq 3$. Let $VL = [x_1, x_2, \ldots, x_n, x_{n+1}]$ be a valid list of $n+1$ elements, and $VLB = [u_1, u_2, \ldots, u_n, u_{n+1}]$, where $u_i = B(x_i)$. Suppose $T$ is an $f$-*term* on $VLB$. We now prove $T = f\text{-}back(VLB)$. By definition, $T = f(T_1, T_2)$, where $T_1$ is an $f$-*term* on $[u_1, \ldots, u_k]$, $T_2$ is an $f$-term on $VLB = [u_{k+1}, \ldots, u_{n+1}]$ and $1 \leq k \leq n$.

If $k = 1$, then $T_1 = u_1$, and $T_2$ is an $f$-*term* on the last $n$ elements of $VLB$. Since a sublist of $VL$ is still valid, $T_2 = f\text{-}back([u_{k+1}, \ldots, u_{n+1}])$ by induction hypothesis. Then,

$$T = f(T_1, T_2) = f(u_1, f\text{-}back([u_{k+1}, \ldots, u_{n+1}])) = f\text{-}back(VLB).$$

If $k > 1$, then both $T_1$ and $T_2$ are $f$-*terms* on lists with no more than $n$ elements. Since a sublist of $VL$ is still a valid list, we have the following by induction hypothesis:

$$T_1 = f\text{-}back([u_1, \ldots, u_k]) = f(u_1, f\text{-}back([u_2, \ldots, u_k]));$$
$$T_2 = f\text{-}back([u_{k+1}, \ldots, u_{n+1}]).$$

Let $VLB_1 = [u_1]$, $VLB_2 = [u_2, \ldots, u_k]$, $VLB_3 = [u_{k+1}, \ldots, u_{n+1}]$, and $t_j = f\text{-}back(VLB_j)$, $j = 1, 2, 3$. Then, $T_1 = f(t_1, t_2)$ and $T_2 = t_3$. Since $f$ is associative w.r.t. valid lists, and any sublist of a valid list is still valid, we have the following by induction hypothesis:

$$\begin{aligned} T &= f(T_1, T_2) = f(f(t_1, t_2), t_3) = f(t_1, f(t_2, t_3)) \\ &= f(u_1, f\text{-}back([u_2, \ldots, u_{n+1}])) = f\text{-}back(VLB). \end{aligned}$$

□

**Theorem 3.1.** *Algorithm* 1 *is computationally equivalent to Program* 1 *if and only if*
**P1.** *There exists a partial order* $\prec$ *on* $S$ *for any finite subset* $S$ *of* $D$ *such that for any* $x \in S$, $sel(x, S)$ *is true if and only if* $x$ *is a minimal element of* $S$ *with respect to* $\prec$.
**P2.** *The merging function* $f$ *is associative w.r.t. valid lists.*

**Proof:** SUFFICIENT.
From P1 and Lemma 3.2, for a given $S$, we have $AVL(S) = APL(S, K)$ for any $K \geq 2$. Suppose that $VL \in AVL(S)$, and generated by both Program 1 and Algorithm 1. Let the corresponding answer be $W$ and $PW$ respectively. We now prove that $W = PW$. It is easy to verify that when P1 holds, any sublist of a valid list is also a valid list for the corresponding subset. Let $VLB$ be the list obtained from $VL$ by applying $B$ to all elements of $VL$. Then, as mentioned before, $W = f\text{-}back(VLB)$, and $PW$ is an $f$-*term* on $VLB$. Since $VL$ is valid, from P2 and Lemma 3.3, $PW = W$.



NECESSITY.

*Case 1.* P1 is not true.
From Lemma 3.2, when P1 is not true, $AVL(S) \neq APL(S, K)$ for some $K \geq 2$, and then Algorithm 1 is not computationally equivalent to Program 1.

*Case 2.* P1 is true, but P2 is false.
Since P1 is true, we have $AVL(S) = APL(S, K)$ for any $K \geq 2$. Let $VL = [x_1, x_2, \ldots, x_n]$ be a valid list, and partitioned into $VL_1 = [x_1, \ldots, x_{k_1}]$, $VL_2 = [u_{k_1+1}, \ldots, u_{k_2}]$, $VL_3 = [u_{k_2+1}, \ldots, u_n]$, where $1 \leq k_1 < k_2 < n$. Let $VLB_1 = [u_1, \ldots, u_{k_1}]$, $VLB_2 = [u_{k_1+1}, \ldots, u_{k_2}]$, $VLB_3 = [u_{k_2+1}, \ldots, u_n]$, where $u_1 = B(x_i)$. Let $t_j = f\text{-}back(VLB_j)$, $j = 1, 2, 3$. Assume $f(t_1, f(t_2, t_3)) \neq f(f(t_1, t_2), t_3)$. Since $VL$ is a valid list, by P1, $x_i$ is a minimal element of the set $\{x_j : i \leq j \leq n\}$. So, $x_j \prec x_i$ is not true if $j > i$. Let $S_j$ be the set containing all elements of $VL_j$, $j = 1, 2, 3$. Then, $S_1$, $S_2$ and $S_3$ is a non-decreasing partition, and $VL_j$ is a valid list from $S_j$. Thus, for the set $S$ containing all elements in $VL$, the partition $S_1$, $S_2$ and $S_3$ may be produced by Algorithm 1 for $K = 3$. When $VL$ is generated this way, $PW$ will be either $f(t_1, f(t_2, t_3))$ or $f(f(t_1, t_2), t_3)$. But the two $f$-*terms* are not equal. That is, there is an execution of Algorithm 1, the list $VL$ is generated, but the answer $PW$ is not the same as $f\text{-}back(VLB)$, the answer given by Program 1 when $VL$ is generated. Thus, Algorithm 1 is not computationally equivalent to Program 1. □

**Corollary 3.1.** *Algorithm* 1 *is computationally equivalent to Program* 1 *if*
**P1** *and*
**P2′**. *The merging function* $f$ *is associative on* $R$.

**Proof:** This is because an associative function is associative w.r.t. valid lists. □

The property **P1** guarantees $AVL(S) = APL(S, K)$, while **P2** or **P2′** ensures the same answer from a valid list. In **P1**, the statement that *x is a minimal element* can be replaced by that *x is a maximal element*. Then, the parallel algorithm will be modified accordingly. For the sake of simplicity, we use minimal only.

### 3.2. Examples

Many examples are given in [40] to illustrate the generality of the PADAC Scheme. Among them are computing the similarity values between a given query and a set of documents that have terms in common with the given query in information retrieval [32], sorting, computing the transitive closure, the same generation problem and the UP-FLAT-DOWN problem in deductive database [5, 23, 35]. In this subsection, we discuss how to apply the PADAC Scheme to parallelize the sorting problem and the transitive closure problem. The performance issue will be discussed in the next section.

*3.2.1. Sorting.* We consider sorting based on compare-exchange operations. Assume a set of numbers are to be sorted into non-decreasing order. The sorting problem can be formalized into Program 1 in two ways by defining the predicate $sel(x, S)$ and hence the predicate $mrg(u, WS, W)$ in different ways. In both cases, the predicate $sgl(x,u)$ maps an element $x$ into a singleton list $u = [x]$.

***Using the trivial order.*** When the predicate $sel(x, S)$ arbitrarily picks an element from $S$, the predicate $mrg(u, WS, W)$ represents a function which combines two sorted lists into one. In the sequential execution, $u = [x]$, and the merging function just inserts $x$ into a proper position in $WS$ to get $W$. When those predicates are defined as above, Program 1 completes the sorting task and represents the sequential algorithm of sorting by insertion. The partial order $\prec$ can be defined as the trivial partial order, i.e., any two different elements are not ordered and any element is a minimal element. So, the predicate $sel(x, S)$ selects a minimal element $x$ from $S$. It is clear that the merging function is associative. By Corollary 3.1, the sorting problem can be solved by the PADAC Scheme. In the dividing phase, the input set is arbitrarily partitioned into $K$ subset of about the same size, since the trivial partial order is used. After each processor finishes sorting a subset, these sorted lists are merged together to form a sorted list for the input set. This parallel algorithm for sorting is a parallel version of merge-sorting.

***Using the partial order*** $<$. When the the predicate $sel(x, S)$ selects the smallest number $x$ from $S$, the predicate $mrg(u, WS, W)$ represents the concatenation function, that is, it appends two lists into one. In this case, Program 1 represents the sequential algorithm of sorting by selecting. The partial order $\prec$ is the arithmetic comparison predicate $<$. Then, the predicate $sel(x, S)$ selects a minimal element $x$ from $S$ with respect to $\prec$. The merging function is clearly associative. Thus, by Corollary 3.1, the sorting problem can be solved by the PADAC Scheme using a different partial order. The dividing of the input set is not trivial any more. Suppose $K - 1$ different numbers $x_k$ are chosen and sorted. Then the dividing of $S$ can be done by comparing each element in $S$ with these $K - 1$ chosen numbers by binary search. After each subset is sorted, the concatenation of these sorted subsets is the sorted set. This parallel algorithm for sorting is a parallel version of *Quicksorting*.

*3.2.2. The transitive closure.* A binary relation $E$ represents a directed graph $G = (V, E)$, where the set of vertices $V$ contains all elements in $E$ and the set of edges is the relation $E$. Let $A$ be the transitive closure of $E$ (or $G$). Then a tuple $(x, y)$ is in $A$ if and only if there is a path from $x$ to $y$ in $G$ [35]. Computing the transitive closure of $G$ clearly represents an operation from one set to produce another set. The base relation $E$ is the input set and the relation $A$ is the answer.

In the following we assume $G$ is acyclic, i.e., $G$ has no directed cycles. Some algorithms for computing the transitive closure of acyclic graphs have been proposed [2, 14, 22, 41].

***Using an order based on the reachability.*** Assume $G$ is acyclic, a partial order $\prec$ can be defined on $E$ as follows:



For any pair of edges $(x_1, y_1)$ and $(x_2, y_2)$, $(x_1, y_1) \prec (x_2, y_2)$ if and only if there is a path from $y_1$ to $x_2$ including the case $y_1 = x_2$ (a path of length 0).

Then, an edge $(x, y)$ is a minimal element iff $x$ has no in-coming edges. We assume that $sel((x, y), E)$ takes a minimal element from $E$. When $E = \{(x, y)\}$, the transitive closure of $E$, $A$, is the same as $E$, i.e., $A = E$. So, the predicate $sgl((x, y), u)$ maps a single tuple $(x, y)$ to a singleton set $\{(x, y)\}$. The merging function $f_1$ is defined as

$$f_1(A_1, A_2) = A_1 \cup A_2 \cup (A_1 \# A_2)$$

where # represents the composition operation (e.g., $\{(x, y)\}\#\{(y, z)\} = \{(x, z)\}$).

Let $(x_0, y_0)$ be a minimal edge, and $AS$ be the transitive closure of $ES$, the subset of $E$ without $(x_0, y_0)$. For any tuple $(x, y)$ in $A$, the transitive closure of $E$, there is a path from $x$ to $y$ in $G$. Let $(x, z)$ be the first edge on the path. If $(x, z) \neq (x_0, y_0)$, i.e., either $x \neq x_0$ or $z \neq y_0$, then all edges in the path are in $ES$ and $(x, y)$ is in $AS$, since $(x_0, y_0)$ is a minimal edge and $x_0$ has no in-coming edges. If $x = x_0$ and $z = y_0$, i.e., $(x_0, y_0)$ is the first edge in the path, then either $y_0 = y$, or there is a path from $y_0$ to $y$ with all edges in $ES$, and $(y_0, y)$ is in $AS$. As a result, $(x, y)$ is in $\{(x_0, y_0)\}\#AS$. Thus, the transitive closure of $E$ can be computed as $A = f_1(\{(x_0, y_0)\}, AS)$.

Then, Program 1 computes the transitive closure of $E$. The operation # is associative [35], and the set union and join clearly satisfy the distributive law, that is,

$$A_1\#(A_2 \cup A_3) = (A_1\#A_2) \cup (A_1\#A_3)$$

and

$$(A_1 \cup A_2)\#A_3 = (A_1\#A_3) \cup (A_2\#A_3).$$

Now we show that $f_1$ is associative, i.e., for any three binary relations $A_1$, $A_2$ and $A_3$, $f_1(A_1, f_1(A_2, A_3)) = f_1(f_1(A_1, A_2), A_3)$.

$$\begin{aligned}
&f_1(A_1, f_1(A_2, A_3)) \\
&\quad = A_1 \cup f_1(A_2, A_3) \cup (A_1\#f_1(A_2, A_3)) \\
&\quad = A_1 \cup (A_2 \cup A_3 \cup (A_2\#A_3)) \cup (A_1\#(A_2 \cup A_3 \cup (A_2\#A_3))) \\
&\quad = A_1 \cup A_2 \cup A_3 \cup (A_2\#A_3) \cup (A_1\#A_2) \cup (A_1\#A_3) \cup (A_1\#A_2\#A_3) \\
&f_1(f_1(A_1, A_2), A_3) \\
&\quad = f_1(A_1, A_2) \cup A_3 \cup (f_1(A_1, A_2)\#A_3) \\
&\quad = (A_1 \cup A_2 \cup (A_1\#A_2)) \cup A_3 \cup ((A_1 \cup A_2 \cup (A_1\#A_2))\#A_3) \\
&\quad = A_1 \cup A_2 \cup (A_1\#A_2) \cup A_3 \cup (A_1\#A_3) \cup (A_2\#A_3) \cup (A_1\#A_2\#A_3) \\
&\quad = f_1(A_1, f_1(A_2, A_3))
\end{aligned}$$

Since both **P1** and **P2′** are satisfied, by Corollary 3.1, the transitive closure of $E$ can be computed in parallel by Algorithm 1.

In PHASE 1, $E$ is partitioned non-decreasingly with respect to $\prec$ into $E_i, i = 1, 2, \ldots, K$. The in-degree $d_{in}(x)$ of node $x$ is maintained for each $x$, and an edge $(x, y)$ is a minimal

edge if and only if $d_{in}(x) = 0$. When such a minimal edge is selected, the value of $d_{in}(y)$ should be decremented by 1. As a result, all out-going edges from $y$ become minimal edges if $d_{in}(y) = 0$ after the decrementing. Suppose $K$ divides $|E|$. Then, the first $|E|/K$ edges taken are in $E_1$, the second $|E|/K$ edges taken are in $E_2$, and so on.

In PHASE 2, each processor evaluates independently Program 1 with one subset $E_i$ as the input set to produce $A_i$, the transitive closure of $E_i$. (As mentioned before, any program for transitive closure can be executed.)

In PHASE 3, the function $f_1$ is applied to the list $[A_1, A_2, \ldots, A_K]$ to compute $A$. Notice that $f_1$ will be applied $K - 1$ times and only one join is required each time. This is a benefit from the non-decreasing partition using the partial order $\prec$, since join is a very expensive, if not the most expensive, operation in database systems.

***A merging function not associative but associative w.r.t. valid lists.*** If the trivial order is used, then any tuple in $E$ is a minimal edge. The predicate $sgl((x, y), u)$ represents the same mapping as mentioned earlier and maps a single tuple $(x, y)$ to a singleton set $\{(x, y)\}$. Let an edge $(x_0, y_0)$ be taken from $E$, and the transitive closure of the remaining subset be $AS$. As mentioned before, a tuple $(x, y)$ is in the transitive closure if and only if there is a path from $x$ to $y$ in $G$. Assume $(x, y) \in A$, but it is not $(x_0, y_0)$ and not in $AS$ either. Then, $(x, y)$ is in $A$ iff there is a path from $x$ to $y$ in $G$ containing edge $(x_0, y_0)$. Thus $(x, y)$ is in $\{(x_0, y_0)\}\#AS$ if $(x_0, y_0)$ is the first edge on the path (so $x = x_0$); $(x, y)$ is in $AS\#\{(x_0, y_0)\}$ if $(x_0, y_0)$ is the last edge on the edge (so $y = y_0$); and $(x, y)$ is in $AS\#\{(x_0, y_0)\}\#AS$ if $(x_0, y_0)$ is in the middle of the path. Let a merging function $f_2$ be defined as follows:

$$f_2(A_1, A_2) = A_1 \cup A_2 \cup A_1\#A_2 \cup A_2\#A_1 \cup A_2\#A_1\#A_2.$$

Then, $A$ can be computed by $f_2(\{(x_0, y_0)\}, AS)$. But function $f_2$ is not associative w.r.t. valid lists. For example, let $E = \{(1, 2), (2, 3), (3, 4), (4, 5), (5, 6)\}$. The list $[(1, 2), (4, 5), (2, 3), (5, 6), (3, 4)]$ is a valid list with respect to the trivial order. Let $VL_1 = [(1, 2), (4, 5)]$, $VL_2 = [(2, 3), (5, 6)]$ and $VL_3 = [(3, 4)]$. Then, $VLB_1 = [\{(1, 2)\}, \{(4, 5)\}]$, $VLB_2 = [\{(2, 3)\}, \{5, 6)\}]$ and $VLB_3 = [\{(3, 4)\}]$. Let $A_i$ be the transitive closure for the subgraph represented by $VL_i$, i.e., $A_i = f\text{-}back(VLB_i)$ for $i = 1, 2, 3$. It can be verified that the tuples $(2, 5)$ and $(2, 6)$ are in $f_2(A_1, f_2(A_2, A_3))$, but not in $f_2(f_2(A_1, A_2), A_3)$. So, $f_2$ is not associative with respective to valid lists when the trivial order is used. By Theorem 3.1, the transitive closure can not be parallelized by the PADAC Scheme with $f_2$ and the trivial order.

Since $f_2$ is not associative with respective to valid lists when the trivial order is used, it is not an associative function. However, it is associative with respect to valid lists when the partial order $\prec$ as defined earlier is used. Suppose that $VL = [e_1, e_2, \ldots, e_n]$ is a valid list for $E$ according to $\prec$, where each $e_i$ is an edge. Let $VL$ be partitioned into two sublists $VL_1$ and $VL_2$. Let $E_i$ be the corresponding set of edges in $VL_i$, and $A_i$ the transitive closure of $E_i$ for $i = 1, 2$. Then $E_1$ and $E_2$ form a non-decreasing partition of $E$ with respect to $\prec$, and there is no path in $G$ with an edge in $E_2$ followed by an edge in $E_1$. So, the expression $A_2\#A_1 \cup A_2\#A_1\#A_2$ will produce nothing, and $f_2$ becomes $f_1$ which has been identified as an associative function with respect to valid lists when the partial order $\prec$ is used.



## 4. Performance analysis

In this section we analyze the performance of the PADAC Scheme. In particular, we discuss the speedup attained by the PADAC Scheme. The speedup of a parallel algorithm is the ratio of the response time of a fastest sequential algorithm to that of the parallel algorithm. That is,

$$Speedup = \frac{RT_s(n)}{RT_p(n, K)}$$

where $RT_s(n)$ is the response time of a fastest sequential algorithm on an input of size $n$ and $RT_p(n, K)$ is the response time of the parallel algorithm on $K$ processors on an input of size $n$. (The subscripts $s$ and $p$ stand for *sequential* and *parallel* respectively.)

When *Speedup* is of the same order as $K$, that is, when $Speedup = C \times K$ for some constant $0 < C \leq 1$, we say that the parallel algorithm achieves linear speedup [25]. Linear speedup is the best a parallel algorithm can achieve when compared to a fastest sequential algorithm. This is because that we can use a single processor to simulate a parallel algorithm on $K$ processors and hence have a sequential algorithm of time $K \times RT_p(n, K)$. In other words, if a super-linear speedup is obtained, then we will have a sequential algorithm of time less than $RT_s(n)$. Notice that, $RT_s(n)$ should be the response time of a fastest sequential algorithm. It is often misleading and of limited value to compare a parallel algorithm to a specific sequential algorithm (instead of a fastest sequential algorithm). For more details, see, e.g., [25].

The *divide-and-conquer* method has been applied, either explicitly or implicitly, to parallelize many problems. Some experiments showed very slow or even zero speedup [31, 33], and no theoretical foundation has been given to explain the results. In this section, we will give a necessary and sufficient condition as to when the PADAC Scheme can achieve linear speedup. Our analysis shows that linear speedup is not possible unless the problem is of time complexity $O(n \log^c n)$ for a constant $c \geq 0$. Our theory explains well those previous experimental results.

### 4.1. *Notations and assumptions*

We discuss the time complexity and focus on the leading term of each expression to simplify the analysis. Some Greek letters are used to represent the orders of magnitude in such a way that, for example, $n^\alpha$ may represent $\Theta(n)$ and $n^\gamma$ may represent $\Theta(n \log n)$. In such a case, $\gamma$ represents a higher order of complexity than $\alpha$, and this is denoted as $\alpha < \gamma$. Some conventional notations are used in the analysis:

$g_1(n) = O(g_2(n))$ means that $g_1(n)$ is bounded above by $g_2(n)$, i.e., there are constants $c$ and $n_0$ such that $g_1(n) \leq cg_2(n)$ for all $n \geq n_0$;

$g_1(n) = \Theta(g_2(n))$ means $g_1(n)$ and $g_2(n)$ are of the same order of magnitude, i.e., both $g_1(n) = O(g_2(n))$ and $g_2(n) = O(g_1(n))$;

$g_1(n) = o(g_2(n))$ means that $g_1(n)$ is of lower order of magnitude than that of $g_2(n)$, i.e., for any constant $c > 0$, there is a constant $n_0$ such that $g_1(n) < cg_2(n)$ for all $n \geq n_0$.

We assume a shared-nothing architecture. Each site has its own main memory and secondary memory (disks). The input size $n$ can be very large, while the number of processors $K$ is limited. So, it is assumed that $n \gg K$. The main memory at a site is not large enough to keep all data needed, even for a subproblem with a smaller input size after partitioning. Thus, I/O time becomes the dominating factor in the response time and it is chosen as the performance metric.

The output (or the answer) $W$ may be a single element or a set. When $W$ is a set, we assume that the output size is a function of the input size, i.e., $|W| = A|S|^{\alpha}$ for some constant $A > 0$ and some order of magnitude $\alpha$. When $W$ is a single element, $|W| = 1$, i.e., $A = 1$ and $\alpha = 0$. Then, in any case, we have $|W| = A|S|^{\alpha}$.

To reach workload balance, the input set $S$ should be partitioned into $K$ subsets of the same size $\frac{n}{K}$. But, in practice, this is not guaranteed, or the partitioning procedure requires more time than we want to spend. So, we assume that any subset obtained in the dividing phase has size between $\frac{1}{\lambda}\frac{n}{K}$ and $\lambda\frac{n}{K}$ for some constant $\lambda > 1$.

We assume that the merging phase can be carried out in $\log K$ rounds in parallel according to a binary tree. The merging function $f$ takes two inputs, say $W^1$ and $W^2$, to produce an output denoted $W^{1,2}$, and the time complexity $T_f$ of the merging function should be a function of $|W^1|$ and $|W^2|$. In our analysis, each $W^j$, $j = 1, 2$, is an intermediate result produced from an original input subset $S^j$ of size $n_j$ and, by our assumption, has a size $An_j^{\alpha}$, and $W^{1,2}$ is the result produced from the original input subset $S_1 \cup S_2$ of size $n = n_1 + n_2$. Then the time complexity of the merge function is a function of the sizes of the two original subsets, $n_1$ and $n_2$, i.e.,

$$T_f(|W^1|, |W^2|) = g_1(|W^1|, |W^2|) = g_1\left(An_1^{\alpha}, An_2^{\alpha}\right) = g_2(n_1, n_2).$$

As indicated above, each subset obtained in the dividing phase has a size between $\frac{1}{\lambda}\frac{n}{K}$ and $\lambda\frac{n}{K}$ for some constant $\lambda > 1$. As a consequence, the difference among $n_1$, $n_2$ and $n$ is up to a constant factor, and $T_f$ can be expressed as a function of $n$ after replacing both $n_1$ and $n_2$ by $n$, i.e.,

$$T_f(|W^1|, |W^2|) = T_f(n).$$

So we assume that the complexity of the merging function is a function of the original input size. Let $T_f(n) = \Theta(n^{\beta})$ for some order of magnitude $\beta$. Then, $\beta \geq \alpha$, since the output of the merge function has a size $An_j^{\alpha}$.

The parallel execution need not be synchronized. To simplify the analysis, we assume the execution is synchronized, i.e., after the partitioning is done, all processors begin local processing at the same time, and each round of the merging phase starts after the previous round is finished at all involved sites. To compute the response time of the parallel scheme, we will take the maximum time for each phase and each round at all involved sites. This will worsen the response time of the parallel scheme and serves as a lower bound for our computed speedup .

We have chosen I/O cost as the performance metric, because I/O operations are much slower than main memory CPU operations. In parallel execution, communication across different processors is required. To send a package $X$ from one site to another costs at least as much as to read $X$ at the receiving site. Thus, the communication cost is not negligible and should be part of the response time of a parallel execution. The time needed to transfer a package between two sites is assumed to be proportional to the size of the package, that is, it takes time $C_{cm}|X|$, where $C_{cm}$ is a constant, to send a package $X$ between two sites.



All **assumptions** we are making in this section are

* The input size is much larger than the number of available processors, i.e., $n \gg K$.
* The output size is a function of the input size, and $|W| = A|S|^{\alpha}$.
* Each subset obtained in the dividing phase of the parallel scheme has a size between $\frac{1}{\lambda}\frac{n}{K}$ and $\lambda\frac{n}{K}$ for some constant $\lambda$.
* The merging phase takes $\log K$ rounds, and the execution is synchronized.
* The time complexity of the merging function is a function of the original input size, i.e., $T_f = T_f(n)$.
* The communication cost is proportional to the size of the package transferred.

Some **notations** used in the analysis are listed in the following.

$n$: the input size, i.e., $|S| = n$;
$K$: the number of available processors;
$\lambda$: the skew constant, i.e., each subset obtained in the dividing phase has a size between $\frac{1}{\lambda}\frac{n}{K}$ and $\lambda\frac{n}{K}$, and $\lambda > 1$;
$\alpha$: the order of magnitude of the output size in the original input size, i.e., $|W| = A|S|^{\alpha}$;
$\beta$: the order of magnitude of the merging function in the original input size, i.e., $T_f(n) = \Theta(n^{\beta})$;
$\gamma$: the order of magnitude of the time complexity of a fastest sequential algorithm (see the next sub-section).

### 4.2. *The speedup of the PADAC scheme*

Let the time complexity of a fastest sequential algorithm be $RT_s(n) = \Theta(n^{\gamma}) = \Theta(n^{c_1} \log^{c_2} n)$ for some constants $c_1$ and $c_2$. As mentioned earlier, the I/O time is the dominating factor in the response time when processing large amount of data and it is chosen as the performance metric in our analysis. It is clear that $\gamma \geq 1$, since the entire input set has to be read into the main memory at least once. This implies $c_1 \geq 1$. For the parallel PADAC Scheme, the communication cost should be included in addition to the I/O cost at each site. Let $Div(n, K)$ be the time for the dividing phase including the I/O time in partitioning the input set and the communication time in sending the partitioned input subsets to their destination sites, $Proc(n, K)$ the I/O time for the local processing phase, and $Merge(n, K)$ the time for the merging phase including the communication time to send intermediate results across site and the I/O time at those merging sites. Then, the response time of the PADAC Scheme is

$$RT_p(n, K) = Div(n, K) + Proc(n, K) + Merge(n, K).$$

The merging phase of the PADAC Scheme consists of $\log K$ rounds, and the time for the merging phase depends on the communication cost and the cost for executing the merging function. As discussed earlier, the time complexity of the merging function is a function of the original input size, i.e., $T_f(n) = \Theta(n^\beta)$. The following lemma says that the time for the merging phase is of the same order of magnitude as the complexity of the merging function, that is, $Merge(n, K) = \Theta(T_f(n))$.

**Lemma 4.1.** *When $\beta > 0$, the merging phase of the PADAC Scheme has the same time complexity as the merging function, that is, $Merge(n, K) = \Theta(T_f(n))$.*

**Proof:** The merging phase consists of $\log K$ rounds. When the merging function is applied in the last round, the entire original input is involved. That is,

$$T_f(n) = O(Merge(n, K)).$$

In the following, we prove $Merge(n, K) = O(T_f(n))$. In each round, some intermediate results are sent out from sending sites, then the merging function $f$ is applied to two intermediate results at each merging site to produce a larger intermediate result. So the time for the $i$th round consists of two parts, $CM_i(n, K)$, the time for communication, and $MG_i(n, K)$, the time for merging. The execution is assumed to be synchronized. As a consequence, $CM_i$ should be taken as that needed to transfer a largest possible intermediate result from all sending sites, and $MG_i$ as the maximum time needed to execute the merging function at all merging sites. Then,

$$Merge(n, K) = \sum_{i=1}^{\log K} \{CM_i(n, K) + MG_i(n, K)\}.$$

In the local processing phase, one processor has a subset of size at most $\lambda \frac{n}{K}$ and produces an intermediate result of size at most $A(\lambda \frac{n}{K})^\alpha$. In the first round of the merging phase, each sending site sends out a package of size at most $A(\lambda \frac{n}{K})^\alpha$; at each merging site, the merging function takes two arguments and produces an intermediate result of size at most $A(\lambda \frac{n}{K} \times 2)^\alpha$, which is the output produced from an input subset of size at most $(\lambda \frac{n}{K} \times 2)$. In general, in the $i$th round, each sending site sends a package of size at most $A(\lambda \frac{n}{K} \times 2^{i-1})^\alpha$; at a merging site, the merging function $f$ is applied to produce an intermediate result for an original input subset of size at most $\lambda \frac{n}{K} \times 2^i$. Then the time for the merging function for the $i$th round is

$$MG_i(n, K) = O\left(T_f\left(\lambda \frac{n}{K} 2^i\right)\right) = O\left(\left(\lambda \frac{n}{K} \times 2^i\right)^\beta\right).$$

The communication time for the $i$th round is

$$CM_i(n, K) = O\left(A\left(\lambda \frac{n}{K} \times 2^{i-1}\right)^\alpha\right) = O(MG_i(n, K)),$$



since the communication cost is proportional to the size of the package sent out and $\beta \geq \alpha$. The total time for the merging phase is

$$\begin{aligned} Merge(n, K) &= \sum_{i=1}^{\log K} \{CM_i(n, K) + MG_i(n, K)\} \\ &= \sum_{i=1}^{\log K} \{O(MG_i(n, K)) + MG_i(n, K)\} = \sum_{i=1}^{\log K} O\left(\left(\lambda \frac{n}{K} \times 2^i\right)^\beta\right) \\ &= O\left(\left(\lambda \frac{n}{K}\right)^\beta \sum_{i=1}^{\log K} (2^\beta)^i\right) = O\left(\lambda^\beta n^\beta \times \frac{1}{K^\beta} \frac{2^\beta (K^\beta - 1)}{(2^\beta - 1)}\right) \\ &= O\left(\lambda^\beta n^\beta \times \frac{2^\beta}{2^\beta - 1} \frac{K^\beta - 1}{K^\beta}\right) = O(n^\beta) = O(T_f(n)). \end{aligned}$$

Since $K \geq 2$, the factor $\frac{K^\beta - 1}{K^\beta}$ is less than 1 and can be dropped from the above expressions. The magnitude order $\beta$ is treated as a constant number in the above, but the result holds when it is of the form $n^{C_1} \log^{C_2} n$ for some constants $C_1 \geq 1$ and $C_2$. The $\log K$ rounds do not contribute a higher order complexity, because the summation $\frac{1}{K^\beta} \sum_{i=1}^{\log K} (2^\beta)^i$ is bounded by a constant $\frac{2^\beta}{2^\beta - 1}$. □

Unlike the merging phase, the time complexity for the dividing phase may involve both the input size $n$ and the number of processors $K$. Let the time complexity of the dividing phase be $\Theta(n^{\gamma_1} K^{\gamma_2})$. We prove in the following lemma that if the dividing phase takes time more than $\Theta(\frac{RT_s(n)}{K})$, then the PADAC Scheme can not reach linear speedup. The value of $K$ is assumed to be limited, and the value of $n$ may increase to arbitrarily large. So, both $n^{\gamma_1}$ and $n^\gamma$ are the dominating factor in the corresponding expression. That is, if $\gamma_1 < \gamma$, then $n^{\gamma_1} K^{\gamma_2} < C'(n^\gamma K^{-1})$ for constant $C'$ when $n$ is large, and vice versa. We say that $n^{\gamma_1} K^{\gamma_2}$ is of a higher order of magnitude than $n^\gamma K^{-1}$ if $\gamma_1 > \gamma$ or $\gamma_1 = \gamma$ and $\gamma_2 > -1$.

**Lemma 4.2.** *The PADAC Scheme can not reach linear speedup if the dividing phase has time complexity of a higher order than $\Theta(\frac{RT_s(n)}{K})$.*

**Proof:**

$$Speedup = \frac{RT_s(n)}{RT_p(n, K)} < \frac{RT_s(n)}{Div(n, K)} = \frac{n^\gamma}{n^{\gamma_1} K^{\gamma_2}}.$$

If $\gamma_1 > \gamma$, then the speedup will approach zero when $n$ is increasing and it is impossible to reach linear speedup. Otherwise, $\gamma_1 = \gamma$ and $\gamma_2 > -1$. Then the speedup is

$$Speedup < \frac{1}{K^{\gamma_2}} = K^{-\gamma_2} = o(K).$$

Since $-\gamma_2 < 1$, linear speedup can not be reached. □

For example, let $RT_s(n) = \Theta(n \log n)$ and $Div(n, K) = \Theta(\frac{n \log n}{\log K})$. Since $\log K = o(K)$, we have $\frac{1}{K} = o(\frac{1}{\log K})$, and $Div(n, K)$ has a higher order of magnitude than $\frac{RT_s(n)}{K}$. The speedup will be at most $\log K$.

$$Speedup = \frac{RT_s(n)}{RT_p(n, K)} < \frac{RT_s(n)}{Div(n, K)} = \frac{n \log n}{\frac{n \log n}{\log K}} = \log K.$$

The time complexity of a fastest sequential algorithm is $RT_s(n) = \Theta(n^\gamma) = \Theta(n^{c_1} \log^{c_2} n)$ for some constant $c_1 \geq 1$. The merging function in the PADAC Scheme has time complexity $T_f(n) = \Theta(n^\beta)$. The following lemma uncovers the fact that if $c_1 > 1$ then there exists no efficient merge function, i.e., for any merge function $f$, $T_f(n)$ is of at least the same time complexity as $RT_s(n)$.

**Lemma 4.3.** *Assume $RT_s(n) = \Theta(n^\gamma) = \Theta(n^{c_1} \log^{c_2} n)$ and $c_1 > 1$. Then, for any merging function $f$, $T_f(n) = \Omega(RT_s(n))$.*

**Proof:** The fact that $c_1 > 1$ implies the fastest sequential algorithm has higher than linear time complexity, i.e., $RT_s(n) = \Theta(n^\gamma)$ and $\gamma > 1$. We prove that if there exists a merging function $f$ of time complexity $T_f(n) = \Theta(n^\beta) = o(n^\gamma)$, then there is a sequential algorithm of time complexity $o(RT_s(n))$, which is faster than the given fastest algorithm.

We apply the divide-and-conquer method to construct such a faster sequential algorithm. We first sort the input set $S$ in a non-decreasing order, then compute a result $u_i$ from each single element $x_i$ in $S$, and finally apply the merging function $f$ according to a binary tree to compute the output $W$ for $S$ (see figure 1).

Since the input size $n$ is very large, external sorting is required. The I/O time becomes the dominating factor in sorting and the CPU processing time can be ignored. For a uniprocessor system, when the main memory size is moderate (contains at least as many pages as the square root of the total number of pages), the sorting can be completed in linear I/O cost (in two passes) by serial Fastsort [33]. However, as we will see later (Section 3.3), when $n$ is extremely large (or the main memory is not large enough), the I/O cost will still be $O(n \log n)$. So, the sorting cost is of $\Theta(n \log n) = o(RT_s(n))$, since $RT_s(n) = \Theta(n^\gamma)$ and $\gamma > 1$.

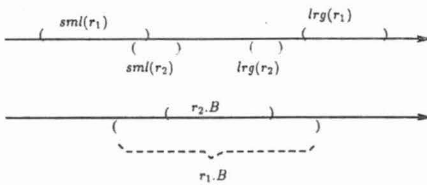


*Figure 1.* $r_1 \prec r_2$, $sml(r_1) \prec sml(r_2)$, but $lrg(r_2) \prec lrg(r_1)$.



The entire input set can not reside in the main memory. The pages of $S$ are loaded into the main memory in the sorted order. The pages residing in the main memory at the same time form a fragment, and we assume that the computation of the partial result from each fragment can be carried out in the main memory. Each fragment corresponds to a subtree of the merging tree from the bottom level. Within such a subtree, the I/O cost of the merging function can be ignored, since no I/O cost is involved. In the following we assume that the cost of merging data within a fragment is the same as $\Theta((n_1 + n_2)^\beta)$ and we will obtain an upper bound of the time complexity for the constructed algorithm.

Without loss of generality, assume $n = 2^L$ for some integer $L$. Then the binary merging tree has $L + 1$ levels. We label these levels as in figure 1. At level 0, a result $u_i$ is produced from $x_i$ for each element $x_i$ in $S$. To produce $u_i$ from $x_i$ takes constant time, since only a single element $x_i$ is involved. Then, it takes time $O(n)$ to produce all single results $u_i$ from $x_i$, including the I/O cost.

At level $l > 0$, there are $\frac{n}{2^l}$ internal nodes. At each such node, the merging function $f$ takes two intermediate results to produce a larger intermediate result. Each input to the merging function represents the result for an original input subset of size $2^{l-1}$, and the output of the merging function represents the result for an original input subset of size $2^l$. Then, the cost of the merging function at a node of level $l$ is $T_f(2^l) = \Theta((2^l)^\beta)$. Then, the total cost for all internal nodes in the entire merging tree is

$$\sum_{l=1}^{L} \left(\frac{n}{2^l}\right) \times (2^l)^\beta = n \sum_{l=1}^{L} (2^{\beta-1})^l.$$

When $\beta = 1$, the sum is $n \times L = n \log n = o(RT_s(n))$, since $RT_s(n) = \Theta(n^{c_1} \log^{c_2} n)$ and $c_1 > 1$. When $\beta \neq 1$, the sum is

$$n \times \left( \frac{(2^{\beta-1})^{L+1} - 1}{2^{\beta-1} - 1} - 1 \right) = n \times \frac{2^{\beta-1} \times (2^L)^{\beta-1} - 2^{\beta-1}}{2^{\beta-1} - 1}$$
$$= n \times \frac{2^{\beta-1} n^{\beta-1} - 2^{\beta-1}}{2^{\beta-1} - 1}.$$

When $\beta > 1$, the sum is $\frac{2^{\beta-1} n^\beta - 2^{\beta-1} n}{2^{\beta-1} - 1} = \Theta(n^\beta) = o(RT_s(n))$. When $\beta < 1$, the sum is $\frac{2^{\beta-1} n - 2^{\beta-1} n^\beta}{1 - 2^{\beta-1}} = O(n) = o(RT_s(n))$, since $RT_s(n) = \Theta(n^{c_1} \log^{c_2} n)$ and $c_1 > 1$. In any case, the cost for merging all nodes in the binary tree is of $o(RT_s(n))$.

The total cost of the constructed algorithm is the sum of the cost for sorting, the cost for computing all $u_i$ (the cost for level 0 in the merging tree) and the cost for merging all internal nodes in the tree. Since each part in the sum is of $o(RT_s(n))$, the constructed algorithm has time complexity $o(RT_s(n))$. □

The proof for Lemma 4.3 looks similar to that for Lemma 4.1. But there are $\frac{n}{2^l}$ nodes at level $l$ and the costs at all these nodes are taken in account. So the result can not be derived directly from Lemma 4.1.

**Theorem 4.1.** *Assume the fastest sequential algorithm has time complexity $RT_s(n) = \Theta(n^{c_1} \log^{c_2} n)$ for some constants $c_1 \geq 1$ and $c_2$. Then the PADAC Scheme achieves linear*

*speedup if and only if*

(1) $RT_s(n) = \Theta(n\ log^{c_2}\ n)$, *i.e.*, $c_1 = 1$;

(2) *The dividing phase takes time at most* $O(\frac{RT_s(n)}{K})$, *i.e.*, $Div(n, K) = O(\frac{n\ log^{c_2}\ n}{K})$; *and*

(3) *The merging function has time complexity of a lower order of magnitude than* $RT_s(n)$, *i.e.*, $T_f(n) = o(n\ log^{c_2}\ n)$.

**Proof:** SUFFICIENCY.

The merging function has time complexity $T_f(n) = \Theta(n^\beta)$. If $\beta = 0$ (e.g., the merging function is trivial such as list concatenation), then the total time for the merging phase will be $\log K$. As mentioned earlier, $K$ is assumed to be fixed and $n$ can increase to very large. Thus, $\log K = o(RT_s(n))$, since $RT_s(n) = \Theta(n^\gamma)$ and $\gamma \geq 1$. Otherwise, $\beta > 0$ and by Lemma 4.1, the merging phase takes time $\Theta(T_f(n))$; by condition 3, $T_f(n) = o(RT_s(n))$. In any case, we have $Merge(n, K) = o(RT_s(n))$.

Let $RT_s(n) = \Theta(Cn \log^{c_2} n)$ (i.e., the leading term has a coefficient $C$). Since $K$ is assumed to be fixed while $n$ is increasing, $\log(\lambda\frac{n}{K}) = \log n + \log\lambda - \log K = \Theta(\log n)$. Then the processing phase takes time

$$Proc(n, K) = \Theta\left(RT_s\left(\lambda\frac{n}{K}\right)\right) = \Theta\left(C\left(\frac{\lambda n}{K}\right)\log^{c_2}\left(\lambda\frac{n}{K}\right)\right) = \Theta\left(\frac{C\lambda}{K}n\log^{c_2} n\right).$$

Let $Div(n, K) = O(\frac{C_{div}}{K} n \log^{c_2} n)$ (i.e., the leading term has a coefficient $C_{div}$). Thus, the response time of the PADAC Scheme is

$$\begin{aligned} RT_p(n, K) &= Div(n, K) + Proc(n, K) + Merge(n, K) \\ &= O\left(\frac{C_{div}}{K}n\log^{c_2} n\right) + \Theta\left(\frac{C\lambda}{K}n\log^{c_2} n\right) + o(n\log^{c_2} n) \\ &= \Theta\left(\frac{C_{div} + C\lambda}{K}n\log^{c_2} n\right). \end{aligned}$$

The PADAC Scheme reaches linear speedup, because

$$Speedup = \frac{RT_s(n)}{RT_p(n, K)} = \frac{Cn\log^{c_2} n}{\frac{C_{div}+C\lambda}{K}n\log^{c_2} n} = \frac{C}{C_{div} + C\lambda} \times K.$$

NECESSITY.

By Lemma 4.2, condition (2) is necessary. If condition (1) is not true, then $c_1 > 1$. By Lemma 4.3, $T_f(n) = \Omega(RT_s(n))$ and condition (3) is not true. The merging function has time complexity $T_f(n) = \Theta(n^\beta)$, and the sequential algorithm has time complexity $RT_s(n) = \Theta(n^\gamma)$, where $\gamma \geq 1$. If (3) does not hold, then $\beta \geq \gamma > 0$. By Lemma 4.1, $Merge(n, K) = \Omega(T_f(n))$. Let $Merge(n, K) = \Theta(C_{mg}n^\beta)$ and $RT_s(n) = \Theta(Cn^\gamma)$. Then, the speedup is

$$Speedup = \frac{RT_s(n)}{RT_p(n, K)} < \frac{RT_s(n)}{Merge(n, K)} = \frac{Cn^\gamma}{C_{mg}n^\beta} \leq \frac{C}{C_{mg}}.$$



Thus the speedup is limited by a constant, since $\beta \geq \gamma$, and hence linear speedup can not be reached. □

### 4.3. An example

We use the sorting problem to illustrate the performance of the PADAC Scheme and discuss the problem in single-input-and-single-output and multi-input-and-multi-output environments. The transitive closure problem will be discussed in Section 6 with the generalized PADAC Scheme.

It has been realized that the I/O cost is the dominating factor in the response time for external sorting. Some algorithms have been proposed to reduce the I/O cost in both sequential and parallel sorting e.g. [6, 7, 10, 16, 33,]. Most of them assume that the main memory has a moderate size (contains at least as many pages as the square root of the total number of pages of the input set), and external sorting can be completed with linear I/O cost (in two passes). The requirement of a moderate main memory size is equivalent to that the input size $n$ is large but not too large. As will be seen after this paragraph, when $n$ is extremely large relative to the main memory size, the I/O cost for sorting will still be $O(n \log n)$. Our analysis shows that, under the assumption that sequential sorting can be done in linear I/O cost, linear speedup is not possible in the single-input-single-output environment. This agrees with previous results, e.g., in [16, 31, 33]. However, under the asymptotic assumption that sequential sorting takes $O(n \log n)$ I/O cost, linear speedup can be achieved in the single-input-single-output environment. It seems that this has not been observed before. In the multi-input-and-multi-output environments, linear speedup can be achieved under either assumptions.

Assume that the main memory can contain $M$ pages of data. Then $2M$ pages of data on the average can be sorted by replacement selection (based on a tournament or heapsort algorithm [24]). Assume that one page contains $P$ elements. Let $N = \frac{n}{P}$, i.e., the input set occupies $N$ pages. Suppose that $R = \frac{N}{2M}$. Then the input set can be sorted by replacement selection into $R$ runs. After all $R$ runs have been sorted, a sorted set can be computed by combining these sorted runs. If there are at most $M$ runs, i.e., $R \leq M$, then the first page of each run is loaded into the main memory and replacement selection can again be used to merge all the $R$ runs. When the first page of a run becomes empty, the second page of that run is loaded in, and so on. The sorted set is written out one page at a time. In this way, the entire input set is scanned twice: one pass for sorting all $R$ runs and one pass for merging all sorted runs. This is how serial Fastsort [33] sorts in linear I/O time.

When $M < R \leq M^2$, the original $R$ runs are divided into $M$ larger runs, each of them has at most $M$ original runs and can be sorted in two passes. After that, the $M$ large runs can be merged together in the same way. The entire input set will be scanned three times. In general, if $M^{k-1} < R \leq M^k$, the entire input set need to be scanned $k$ times before it is completely sorted. The I/O cost is

$$n\log_M R = n\log_M \frac{N}{2M} = n\log_M \frac{n}{2PM} = n(\log_M n - \log_M 2PM).$$

Both $P$ and $M$ are constants. Thus the I/O cost for sorting is still $O(n \log n)$ when $n$ is extremely large.

***Single-input-and-single-output environment.*** If the partial order $<$ is used, then a non-decreasing partitioning will generate $K$ subsets such that $x \leq y$ if $x \in S_i$, $y \in S_j$ and $i < j$. We need $K - 1$ different numbers $x_i$; then an element $x$ of $S$ is in subset $S_i$ if and only if $x_{i-1} < x \leq x_i$ ($x_0$ and $x_K$ are the possible smallest and largest numbers respectively). We call these $K - 1$ numbers partitioning numbers. A set of partitioning numbers can be obtained easily if it is known that the elements uniformly distributed within a range. Otherwise, it can be computed by sampling (e.g., [16, 18, 19]), or pre-computation. In any case, the cost for computing these partitioning numbers can be ignored. After the $K - 1$ partitioning numbers are determined and sorted, an element can be decided to belong to a unique subset $S_i$ by binary search. The entire input set has to be scanned at the starting site, and the dividing phase takes linear I/O cost, i.e., $Div(n, K) = \Theta(n)$.

There are two cases. When the input set is extremely large, $RT_s(n) = O(n \log n)$. Then linear speedup will be achieved, since the merging function (concatenation) takes at most linear I/O time and $Merge(n, K) = O(n)$. When the main memory is moderate (or that the input set is large but not too large), $RT_s(n) = O(n)$; by Lemma 4.2, linear speedup is impossible, since $Div(n, K) = \Theta(n)$.

If the trivial order is used, then the merging function has linear I/O cost, since the two sorted subsets to be merged have to be scanned to accomplish the merging. The dividing phase takes at most linear I/O cost. Assume that $RT_s(n) = O(n \log n)$, linear speedup will be achieved. On the other hand, when $RT_s(n) = O(n)$, linear speedup can not be achieved even with the dividing cost ignored, since the last step in merging takes linear I/O cost already. This is why some previous experiment results show very small or zero speedup [31, 33].

***Multi-input-and-multi-output environment.*** We assume that the partial order *less than* is used. (If the trivial order is used, the final merging step has to be carried out in a single site and multi-output does not make any sense.) In this case, linear speedup will be achieved both when $RT_s(n) = \Theta(n)$ and when $RT_s(n) = \Theta(n)$.

The input set is initially distributed across all $K$ sites, but not in a non-decreasing manner. Let $S^j$ be the fragment of $S$ at site $j$ before sorting. Then, the elements of $S^j$ are not sorted; for two elements $x$ and $y$ in two different sites, each of the three cases may be true: $x < y$, $x = y$ or $x > y$. After sorting, the entire set is still distributed across all $K$ sites, but in a non-decreasing manner. Let $S_k$ be the fragment at site $k$ after sorting. Then, all elements in $S_k$ are sorted; for two different sites $k_1 \neq k_2$, $x \leq y$ if $x \in S_{k_1}$, $y \in S_{k_2}$ and $k_1 < k_2$.

After the $K - 1$ partitioning numbers are decided, each site $j$ can partition the initial fragment $S^j$ into $K$ subsets $S^j_k$ according to these partitioning numbers, where $S^j_k = S^j \cap S_k = \{x : x \in S^j \text{ and } x_{k-1} < x \leq x_k\}$, and send $S^j_k$ to site $k$ for each $k \neq j$. The fragment after partition at site $k$ is $S_k = \bigcup_{1 \leq j \leq K} S^j_k$. After $S_k$ is sorted at site $k$ for each $k$, the entire input is sorted and resides at the $K$ sites.

Assume that the fragment at each site has a size about $\frac{n}{K}$ both before and after the partitioning. Then, the dividing can be completed in time $O(\frac{n}{K})$ at a site, including the I/O cost and the communication cost. That is, $Div(n, K) = O(\frac{n}{K})$. The merging function is trivial (concatenation of two sorted sets) and the cost can be ignored, since the output is not required to be sent to a single site. That is, $T_f(n) = O(1)$. Then by Theorem 4.1, linear speedup will be achieved for either $RT_s(n) = O(n)$ or $RT_s(n) = O(n \log n)$.



## 5. A generalization of the PADAC scheme

Some operations on sets need parameters. In some cases, it is convenient to consider part of the input set as a parameter set. This allows data duplication among different sites. The following Program 2 is a generalization of the basic linear recursive Program 1 and uses a parameter set $Z$ to produce an answer $W$ from an input set $S$.

Program 2
$p(\{x\}, Z, W) : -\, sgl(x, Z, W).$
$p(S, Z, W) \;\; : -\, sel(x, S),$
$dif(S, x, SS),$
$sgl(x, Z, u),$
$prm(x, Z, ZS),$
$p(SS, ZS, WS),$
$mrg(u, WS, W).$

The predicate $prm(x, Z, ZS)$ was not present in Program 1, each other predicate in Program 2 was present in Program 1 and has a similar meaning as in Program 1. The new predicate $prm(x, Z, ZS)$ is used to compute a parameter subset for the remaining input subset after an element is selected. When the input set is a singleton, a result is produced by the predicate $sgl(x, Z, W)$. Otherwise, an element $x$ is selected from $S$, and a result $u$ is produced from $x$ using the parameter set $Z$. To produce a result from the remaining subset $SS = S - x$, a parameter subset $ZS$ need to be computed. This is carried out by the predicate $prm(x, Z, ZS)$. After the result $WS$ is produced from $SS$ using $ZS$, the final result $W$ is formed from $u$ and $WS$ by the predicate $mrg(u, WS, W)$.

The following Algorithm 2 is our generalized PADAC Scheme which is developed to evaluate Program 2 in parallel based on the *divide-and-conquer* method. As before we assume that a partial order $\prec$ is defined on a given input set $S$. Then, in the dividing phase (PHASE 1), the input set $S$ is partitioned into $K$ subsets non-decreasingly according to $\prec$. In addition, a parameter subset $Z_k$ need to be computed for each subset $S_k$. In the processing phase (PHASE 2), a partial result $W_k$ is produced from each subset $S_k$ with the parameter subset $Z_k$. Finally, the partial results are combined to form the answer $W$ corresponding to the entire input set $S$ with the parameter set $Z$.

### Algorithm 2

**PHASE 1:** The input set $S$ is partitioned non-decreasingly into $K$ non-empty subsets $S_k$; the corresponding parameter sets $Z_k$ are computed.
**PHASE 2:** Each processor executes independently Program 2 with $S_k$ as the input set and $Z_k$ as the parameter set to produce $W_k$.
**PHASE 3:** The final answer $W$ is computed from these partial results $W_1, W_2, \ldots, W_K$ by the merging function.

Algorithm 2 is similar to Algorithm 1 except that a parameter subset $Z_k$ need to be computed for each input subset $S_k$ in the dividing phase. The intention of non-decreasing

partition of the input set is that, for any input subset $S_k$, $1 < k \leq K$, the elements in all earlier subsets, i.e., in $\bigcup_{1 \leq i < k} S_i$, might be selected before any element in $S_k$ (and in any later subset) is selected in the sequential linear program, Program 2. As a result, the parameter subset $Z_k$ for $S_k$ should be the same as that obtained in Program 2 after the elements in $\bigcup_{1 \leq i < k} S_i$ have been selected one by one and the predicate $prm(x, Z, ZS)$ has been applied accordingly. The elements in $\bigcup_{1 \leq i < k} S_i$ may be selected in different orders by the predicate $sel(x, S)$ in Program 2. To guarantee that a unique parameter subset $Z_k$ is to be computed for each input subset $S_k$, we assume that, for any input subset $S_i$, if the elements in $S_i$ can be selected consecutively in different orders by the predicate $sel(x, S)$ the parameter subset for the remaining input subset $S - S_i$ is the same. When this is true, we say that the predicate $prm(x, ZS, Z)$ is *order-independent*. Then, in the dividing phase of Algorithm 2, $Z_k$ is to be computed as the parameter subset for $S - \bigcup_{1 \leq i < k} S_i$. In the next section, we will see that in many applications it is rather simple and straightforward to compute these parameter subsets.

All concepts defined earlier for Program 1 and Algorithm 1 can be defined for Program 2 and Algorithm 2 in the same way or with minor changes, and we can prove similar results regarding the *computational equivalence* between the sequential Program 2 and the parallel Algorithm 2.

**Theorem 5.1.** *Assume the predicate $prm(x, ZS, Z)$ is order-independent. Algorithm 2 is computationally equivalent to Program 2 if and only if*

**P1.** *There exists a partial order $\prec$ on any finite subset $S$ of $D$ such that for any $x \in S$, $sel(x, S)$ is true if and only if $x$ is a minimal element of $S$ with respect to $\prec$.*

**P2.** *The function $f$ is associative with respect to valid lists.*

**Corollary 5.1.** *Assume the predicate $prm(x, ZS, Z)$ is order-independent. Algorithm 2 is computationally equivalent to Program 2 if*

**P1.** *and*

**P2′.** *The merging function $f$ is associative on $R$.*

## 6. Applications

In [41], we have shown that the transitive closure problem can be parallelized by the generalized PADAC Scheme. The set of vertices $V$ is considered as the input set, while the set of edges $E$ is treated as the parameter set. The partial order on $V$ is a topological order (We discuss acyclic graphs only.) The predicate $sgl(x, Z, u)$ produces the descendant set of vertex $x$ (i.e., the set of all tuples $(x, y)$ in the transitive closure with $x$ at the first position), and requires significant computation, while the merge function is the set union and trievial. Our simulation results show that in sequential computation our algorithm is superior to other existing algorithms in most cases, and that in parallel computation linear speedup is achieved.

In the following, we apply the generalized PADAC Scheme to the fuzzy join problem and show that linear speedup will also be obtained in this application. From the two examples, we can see that more complicated problems can be parallelized by the generalized PADAC Scheme. However, the introducing of parameter sets makes it much more difficult to analyze the performance of the scheme than in the case without parameter sets.



### 6.1. *Fuzzy joins*

In a fuzzy relation [12, 29, 20, 42–44], the value of an attribute of a tuple may represent either a single number or an interval. For example, a relation about people may have an attribute *AGE*. A tuple with a single number, say 28, as the value for the *AGE* attribute says the age of the person is 28. A tuple with "about 35" as the value for the *AGE* attribute says that the age of the person is, for instance, between 30 and 40, which represents an interval [30, 40]. Such a value "about 35" is called a fuzzy number. A tuple may have another kind of value such as "young" for the *AGE* attribute, which represents, for instance, an interval [21, 35]. Such a value "young" is called a fuzzy label. Both fuzzy number and fuzzy label represent existing but uncertain information. An attribute is called a *fuzzy attribute* if its domain contains single values as well as fuzzy numbers or/and fuzzy labels; a relation is called a *fuzzy relation* if at least one attribute is a fuzzy attribute.

Consider the join $R \bowtie_{R.B=T.B} T$, where $R$ and $T$ are two fuzzy relations and $B$ is a common fuzzy attribute. When $B$ is the attribute *AGE*, the above join requests all pairs of persons, one from $R$ and another from $T$, who have the same age. Two persons can not have the same age if one is 28 and the other is "about 35", since "about 35" means between 30 and 40. However two persons may have the same age if one is 28 and the other is "young", or one is "about 35" and the other is "young", since "young" means between 20 and 35. But we do not know for sure that they have the same age. So, unlike joins of ordinary relations, the condition $r.B = t.B$ can only be checked with some uncertainty, since the join values may represent some uncertain information. We use degrees, which are numbers between 0 and 1, to indicate the uncertainty that the join condition $r.B = t.B$ holds.

We give some notations to define fuzzy joins formally. In general, a value in the domain of a fuzzy attribute represents a (finite or infinite) subset of the background domain. For example, $AGE = 28$ represents a singleton set $\{28\}$, "about 35" represents a set of $\{x : 30 \leq x \leq 40\}$. For a tuple $r$ of $R$ and a tuple $t$ of $T$, we use $r.B$ and $t.B$ to denote either the fuzzy value or the corresponding subset, and $r.B \cap t.B$ represents the ordinary set intersection. Formally, the join $R \bowtie_{R.B=T.B} T$ is carried out as follows.

For any tuple $r$ of $R$ and any tuple $t$ of $T$,

(1) When $r.B \cap t.B = \emptyset$, no tuple is generated from $r$ and $t$.

(2) When $r.B \cap t.B \neq \emptyset$, a degree, which is a number between 0 and 1 is computed, and a tuple will be generated from $r$ and $t$ only when the degree is positive.

We do not discuss the issue in this paper how to compute the degree and assume that the degree can be computed in constant time. Interested readers are referenced to [12, 42, 44].

To compute the join $R \bowtie_{R.B=T.B} T$ by nested loops, we need to compare each tuple $r \in R$ with all tuples in $T$. There is analytical and experimental evidence that hashed join is the most effective method for joins of ordinary relations, but it is unlikely that the method can be applied to joins of fuzzy relations effectively, since a value may be an interval as

well as a single number. Sort-merge join is a good candidate for such joins. Suppose that both $R$ and $T$ are sorted according to an appropriate partial order based on the join values. Then, the join proceeds according to the partial order. It is possible that the join of the two fuzzy relations can be carried out in linear I/O cost after both relations are sorted according to the partial order (see performance in Section 5.4).

Fuzzy joins are similar to "band joins" (e.g. [15]). In "band joins", each value of the joining attribute is a simple value and represents an interval. However, all intervals for different joining values are of the same length. Fuzzy joins are much more general than "band joins", since each joining value may be an interval as well as a simple value and different intervals may have different lengths. The method for computing band joins can not be adopted directly for fuzzy joins, while our algorithm for fuzzy joins includes band joins as a special case.

### 6.2. *Formalizing fuzzy joins by Program 2*

Let the domain of the joining attribute $B$ be $D$. A value $v$ in $D$ may be either a single number or an interval. When $v$ is an interval, we use $b(v)$ and $e(v)$ to denote the two (beginning and ending) points, i.e., $v = [b(v), e(v)]$. When $v$ is a single number, we let $b(v) = e(v) = v$. Then, each value corresponds to a pair of ordinary numbers. We define a linear order on $D$ as follows.

*Definition.*

1. A linear order on $D$ is defined as follows: for two values $v_1$ and $v_2$ in $D$, $v_1 \prec v_2$ if $b(v_1) < b(v_2)$, or $b(v_1) = b(v_2)$ and $e(v_1) < e(v_2)$. For example, $[2, 5] \prec [3, 4]$ and $[3, 4] \prec [3, 5]$.
2. A partial order on $R$ (or $T$) is defined as follows: for two tuples $r_1$ and $r_2$, $r_1 \prec r_2$ if $r_1.B \prec r_2.B$.

For two values $v_1$ and $v_2$ in $D$, we use $v_1 \preceq v_2$ for $v_1 \prec v_2$ or $v_1 = v_2$. For two tuples $r_1$ and $r_2$, we use $r_1 \preceq r_2$ for $r_1 \prec r_2$ or $r_1.B = r_2.B$.

*Definition.*

3. For any tuple $r$ in $R$, $sml(r)$ is the smallest value $v$ of $D$ that appears in $T$ and $r.B \cap v \neq \emptyset$, and $lrg(r)$ is the largest value $v$ of $D$ that appears in $T$ and $r.B \cap v \neq \emptyset$.
4. The range of a tuple $r$ of $R$ is a subset of $T$ defined by $Rng(r) = \{t : t \in T$ and $sml(r) \preceq t.B \preceq lrg(r)\}$.

For two tuples of $R$, $r_1 \prec r_2$, we have $sml(r_1) \preceq sml(r_2)$, but it is possible that $lrg(r_2) \prec lrg(r_1)$ (see figure 1). This is because that $r_1 \prec r_2$ implies $b(r_1) \leq b(r_2)$, but it does not imply $e(r_1) \leq e(r_2)$.

A tuple $t$ of $T$ can not join with a tuple $r$ of $R$ if $r.B \cap t.B = \emptyset$. By the definition of $sml(r)$ and $lrg(r)$, a tuple $t$ of $T$ can not join with a tuple $r$ of $R$ if $t$ is not in $Rng(r)$, i.e., either $t.B \prec sml(r)$ or $lrg(r) \prec t.B$. However, it is possible that $t$ can not join with $r$ even $t$ is in $Rng(r)$ (see figure 2).



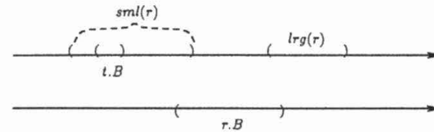


*Figure 2.* $sml(r) \prec t.B \prec lrg(r)$, but $r.B \cap t.B = \emptyset$.

We now formalize the join of two fuzzy relations $R \bowtie_{R.B=T.B} T$ by Program 2. Relation $R$ is the original input set, relation $T$ is the original parameter set, and the final answer $W$ is the set containing all tuples generated in the join. (Each tuple in $W$ has a degree of satisfaction greater than zero.) The predicate $sel(r, R)$ selects a minimal tuple $r$ from $R$ according to the partial order $\prec$ on $R$. The predicate $sgl(r, T, u)$ carries out the join between tuple $r$ and all tuples of $T$ in $Rng(r)$ and can be written as $sgl(r, Rng(r), u)$, and $u$ is the set containing all tuples generated from the join between $r$ and $Rng(r)$. The remaining input subset $RS$ is obtained by the predicate $dif(R, r, RS)$ by removing from $R$ the selected tuple $r$. The parameter set $TS$ for $RS$ is obtained by the predicate $prm(r, T, TS)$ by removing from $T$ those tuples that are not in $Rng(r)$ and precede the tuples in $Rng(r)$. These removed tuples will not be used for the join with any later tuples of $R$, since $r$ is a minimal tuple of $R$ and $sml(r) \preceq sml(r')$ for any tuple $r'$ in the remaining subset $RS$. The predicate $mrg(u, WS, W)$ forms the final answer $W$ by taking the union of $u$ and $WS$. Then the following program computes the join of two fuzzy relations.

```
Program FuzzyJoin
  p({r}, T, W) : — sgl(r, Rng(r), W).
  p(R, T, W)   : — sel(r, R),
                   dif(R, r, RS),
                   sgl(r, Rng(r), u),
                   prm(r, T, TS),
                   p(RS, TS, WS),
                   mrg(u, WS, W).
```

It can be seen that Program FuzzyJoin performs the join of two fuzzy relations in a sort-merge manner.

### 6.3. *Parallelizing fuzzy joins by the generalized PADAC scheme*

A partial order $\prec$ is defined on the input set $R$, the predicate $sel(r, R)$ selects a minimal tuple from $R$, and the predicate $mrg(u, WS, W)$ represents the set union which is apparently associative. By Theorem 5.1, the join of two fuzzy relations can be parallelized by the generalized PADAC Scheme.

We discuss the parallel scheme in a multi-input and multi-output environment. Initially, both $R$ and $T$ are distributed across the $K$ sites, but not according to the partial order $\prec$.

The partition of $R$ is carried out the same way as in sorting, and in processing phase each subset of $R$ will be sorted. We discuss the dividing of $T$ in the following. We use $R^j$ ($T^j$) to denote the subset of $R$ ($T$) at site $j$ before re-distribution, and $R_k$ ($T_k$) the subset of $R$ ($T$) at site $k$ after.

After $K-1$ *partitioning tuples* for $R$ are determined, $R$ is non-decreasingly partitioned into $K$ subsets $R_k$ and a tuple $r$ of $R$ is in $R_k$ if $x_{k-1} \prec r \preceq x_k$, $1 \le k \le K$, where $x_0$ and $x_K$ are the possible minimal and maximal tuples of $R$ respectively. For each tuple $r$ in a subset $R_k$, the tuples of $T$ in $Rng(r)$ are needed to perform the join. The parameter set $T_k$ for $R_k$ will be $\bigcup_{r\in R_k} Rng(r)$. Let $b(k) = min\{b(r.B) : r \in R_k\}$ and $e(k) = max\{e(r.B) : r \in R_k\}$. Then each subset $R_k$ of $R$ determines an interval $[b(k), e(k)]$ and the corresponding parameter subset $T_k = \{t \in T : t.B \cap [b(k), e(k)] \neq \emptyset\}$. We call these $K$ intervals *dividing intervals* for $T$. Notice that the $K$ intervals $[b(k), e(k)]$ may intersect each other, and those subsets $T_k$ may intersect too. Even the intervals $[b(k), e(k)]$ do not intersect, there still may be some overlap between different subsets $T_k$.

The computation of the dividing intervals for $T$ and the dividing of $T$ can also be carried out in parallel at all sites. Let $R_k^j = R^j \cap R_k$, i.e., $R_k^j$ is the set of tuples that will be sent from site $j$ to site $k$. Let $b^j(k) = min\{b(r.B) : r \in R_k^j\}$ and $e^j(k) = max\{e(r.B) : r \in R_k^j\}$. At each site $j$, the values of $b^j(k)$ and $e^j(k)$ for all $k$ are computed when $R^j$ is partitioned into $R_k^j$, and they are sent to all other sites after the partitioning of $R^j$ is completed. Then at each site $k$, the $K$ dividing intervals $[b(k), e(k)]$ for $T$ are computed by $b(k) = min\{b^j(k) : 1 \le j \le K\}$ and $e(k) = max\{e^j(k) : 1 \le j \le K\}$. The same $K$ intervals $[b(k), e(k)]$ are computed at all sites, because each site has the same $b^j(k)$ and $e^j(k)$. Let $T_k^j$ be defined in the same way as for $R_k^j$. Then at site $j$, $T^j$ is divided into $K$ subsets $T_k^j$. A tuple $t$ of $T_j$ is in $T_k^j$ if $t.B \cap [b(k), e(k)] \neq \emptyset$. The $K$ dividing intervals can also be sorted according to the linear order $\prec$ on the domain $D$, and a binary search can be employed on the $K$ intervals. After a tuple is determined to belong to $T_k^j$, the preceding and following intervals should be examined since the tuple may be in multiple intervals.

We give a parallel algorithm derived from our PADAC Scheme in the multi-input-and-multi-output environment. Step (1) through (6) form the dividing phase of the PADAC Scheme, Step (7) is the processing phase of the PADAC Scheme, and the merging phase of the PADAC Scheme is not shown in the algorithm. Some Steps, say Step 2 and 3, can be interleaved.

***Algorithm MIMO-fuzzy join***

(1) Decide $K-1$ partitioning tuples for $R$.
(2) At each site $j$, partition $R^j$ into $K$ subsets $R_k^j$ according to the partitioning tuples decided in (1); compute $b^j(k)$ and $e^j(k)$ for all $k$.
(3) At each site $j$, send $R_k^j$ to site $k$ for each $k$; send all $b^j(k)$ and $e^j(k)$ to all sites $k \neq j$.
(4) At each site $j$, compute the same $K$ dividing intervals $[b(k), (e(k)]$ for $T$ by $b(k) = min\{b^j(k) : 1 \le j \le K\}$ and $e(k) = max\{e^j(k) : 1 \le j \le K\}$.
(5) At each site $j$, divide $T^j$ into $K$ subsets $T_k^j$ according to the $K$ dividing intervals decided in (4).
(6) At each site $j$, send $T_k^j$ to site $k$ for each $k$.



(7) At each site $k$, perform the sort-merge join between $R_k$ and $T_k$, where $R_k = \bigcup_{1\le j\le K} R_k^j$ and $T_k = \bigcup_{1\le j\le K} T_k^j$.

### 6.4. *Performance*

After both $R$ and $T$ are sorted, the merge join can be completed in linear I/O cost (see next paragraph). If we assume that both $R$ and $T$ are very large and that sorting takes $O(n \log n)$ I/O cost, then the join of two fuzzy relations needs $O(n_1 \log n_1 + n_2 \log n_2)$ I/O cost, where $n_1 = |R|$ and $n_2 = |T|$. That is, the sorting cost is the dominating factor. Then, fuzzy join can be parallelized with linear speedup under this assumption, since sorting can be parallelized by the PADAC Scheme with linear speedup in this case (Section 3.3). In the following, we assume that the main memory has a moderate size, or both $R$ and $T$ are large but not too large. As explained earlier in Section 3.3, sorting takes linear I/O cost. Thus, the join of two fuzzy relations can be carried out in linear I/O time in sequential computation. That is, $RT_s = O(n_1 + n_2)$.

To perform the merge-join in linear I/O cost, the tuples of $R$ are loaded into the main memory one page at a time in the sorted order. Let $CR$ be the current page of $R$ in the memory, and $b(CR) = min\{b(r.B) : r \in CR\}$ and $e(CR) = max\{e(r.B) : r \in CR\}$. The value of $b(CR)$ is equal to $b(r'.B)$, where $r'$ is the first (minimal) tuple of $CR$; the value of $e(CR)$ is obtained by scanning all tuples in $CR$ and can be computed when $R$ is being sorted. The tuples of $T$ that may join with any tuples in $CR$ are in $\bigcup_{r\in CR} Rng(r)$. A tuple $t$ of $T$ precedes the tuples in $\bigcup_{r\in CR} Rng(r)$ if $e(t.B) < b(CR)$; it follows the tuples in $\bigcup_{r\in CR} Rng(r)$ if $b(t.B) > e(CR)$. The pages of $T$ are loaded into the main memory one by one in the sorted order. If a page of $T$ contains only tuples that precede those tuples in $\bigcup_{r\in CR} Rng(r)$, then the page is useless in current as well as in later joining and should be discarded. This is because $sml(r_1) \preceq sml(r_2)$ if $r_1 \preceq r_2$. If a page contains some tuples in the set $\bigcup_{r\in CR} Rng(r)$, then the join is performed and the page stays in the main memory, since some tuples in the page may join with some tuples in the next page of $R$. If a page contains one tuple that follows the tuples in $\bigcup_{r\in CR} Rng(r)$, then no later pages of $T$ are needed for the page $CR$ of $R$ and should not be loaded in. Then, the join w.r.t. the current page of $R$ is completed, and several pages of $T$ may reside in the main memory. For the next current page of $R$, the values of $b(CR)$ and $e(CR)$ are modified before the joining. The pages of $T$ in the main memory are examined first, and the first few pages may be discarded immediately if they contain only tuples preceding the tuples in $\bigcup_{r\in CR} Rng(r)$. Some later pages of $T$ may need to be loaded in. It is clear that the join can be completed in linear I/O cost as long as the main memory can keep one page of $R$ and the maximal number of pages of $T$ that cover $\bigcup_{r\in CR} Rng(r)$ for any single page of $R$.

We discuss parallel execution under assumptions similar to that in Section 3. We assume that each site $k$ does not have enough main memory to keep the entire subset $R_k$ or $T_k$, and the I/O cost is again the dominating factor for local processing at each site. The communication cost should be included. We assume that there is no contention in the network, e.g., there is a link between any pair of sites, and the communication cost at a site is proportional to the amount of data sent out from and received at that site. The cost in the merging phase is

ignored, that is, the produced tuples are scattered across all sites and not required to send to a single site. For example, another operation, say another join, is to be performed in parallel. For ordinary (non-fuzzy) relations, many parallel joining algorithms are evaluated under a uniform distribution assumption, e.g., [34, 36]. The following are similar assumptions for fuzzy relations

* Both relations $R$ and $T$ are initially distributed evenly across the $K$ sites.
* A set of $K-1$ partitioning tuples that partitions $R$ into $K$ subsets of about the same size exists or can be decided efficiently.
* The $K$ dividing intervals $[b(k), (e)]$ divide $T$ into $K$ subsets of about the same size.
* The interval of each tuple of $T$ is relative small w.r.t. the length of those dividing intervals $[(k), e(k)]$ so that one tuple of $T$ will appear in at most two subsets of $T$. This is usually true for fuzzy applications.

Under these assumptions, the cost of the dividing phase (Step 1 through 6 in Algorithm MIMO-FuzzyJoin) is $Div(n_1, n_2, K) = O(\frac{n_1+n_2}{K})$.

Step (1): The set of partitioning tuples for $R$ can be computed by sampling; the sampling cost is small and can be ignored, since the sample size is usually small. Alternatively, the set of partitioning tuples for $R$ can be pre-computed and updated periodically. Thus, the cost of the pre-computation can be amortized over many queries.
Step (2): the I/O cost is $O(\frac{n_1}{K})$ at a site;
Step (3): the communication cost is $O(\frac{n_1}{K})$ at a site;
Step (4): the computation of the $K$ intervals does not incur extra I/O cost;
Step (5): the I/O cost is $O(\frac{n_2}{K})$ at a site;
Step (6): the communication cost is $O(\frac{n_2}{K})$ at a site.

For the processing phase (Step (7)), each site has an input subset $R_k$ of size $O(\frac{n_1}{K})$ and a parameter subset $T_k$ of size $O(\frac{n_1}{K})$, and the merge-join can be carried out in parallel with $O(\frac{n_1+n_2}{K})$ I/O cost at a site provided the main memory at each site has a moderate size. Thus, the total I/O cost at each site is $O(\frac{n_1+n_2}{K})$, and Algorithm MIMO-FuzzyJoin will achieve linear speedup.

## 7. Summary

A linear recursive program is used to formalize problems to be parallelized by the *divide-and-conquer* method. A necessary and sufficient condition which characterizes problems solvable by the *divide-and-conquer* method in parallel is obtained, and a parallel scheme is developed. The performance of the parallel scheme is analyzed, and a necessary and sufficient condition is obtained as to when linear speedup can be achieved. A generalization of the PADAC Scheme is developed, and a real application, the fuzzy join problem, is parallelized by the generalized PADAC Scheme with linear speedup.



## Acknowledgments

Research supported in part by NASA(under grant NAGW-4080) and ARO(under grant DAH 04-96-1-0049, DAAH 04-96-1-0278, DAAH 04-0024/BMDO).

## References

1. R. Agrawal and H.V. Jagadish, "Multiprocessor transitive closure algorithms," Proc. Int'l. Symp. on Databases in Parallel and Distributed Systems, Austin, Tex., Dec. 1988, pp. 56–66.
2. R. Agrawal and H.V. Jagadish, "Hybrid transitive closure algorithms," Proc. 16th Int'l. Conf. Very Large Data Bases, Brisbane, Australia, Aug. 1990, pp. 326–334.
3. A.V. Aho, J.E. Hopcropt, and J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974.
4. S. Baase, Computer Algorithms: Introduction to Design and Analysis, 2nd ed., Addison-Wesley, 1988.
5. F. Bancilhon and R. Ramakrishnan, "An amateur's introduction to recursive query processing strategies," Proc. ACM-SIGMOD Intel. Conf. on the Management of Data, 1986.
6. B. Baugstø and J. Greipsland, "Parallel sorting methods for large data on a hypercube database computer," Proc. the Sixth Int'l Workshop on Database Machine, pp. 127–141, 1989.
7. M. Beck, D. Bitton, and W.K. Wilkinson, "Sorting large files on a backend multiprocessor," IEEE Trans. on Computers, vol. 37, no. 7, pp. 769–778, 1988.
8. J.L. Bentley and M.I. Shamos, "Divide-and-conquer for linear expected time," Info. Proc. Letts, pp. 87–91, Feb. 1978.
9. D. Bitton, H. Boral, D.J. DeWitt, and W.K. Wilkinson, "Parallel algorithms for the execution of relational database operations," ACM Trans. on Database Systems, vol. 8, no. 3, 1983.
10. G.E. Blelloch, C.E. Leiserson, B.M. Maggs, C.G. Plaxton, S.J. Smith, and M. Zagha, "A comparison of sorting algorithms for the connection machine CM-2," Proc. the 3rd Annual ACM SPAA, 1991.
11. S. Bondeli, "Divide and conquer: A parallel algorithm for the solution of a tridiagonal linear system of equations," Parallel Computing, vol. 17, July 1991.
12. P. Bosc, M. Galibourg, and G. Hamon, "Fuzzy querying with SQL: Extensions and implementation aspects," Fuzzy Sets and Systems, vol. 28, pp. 333–349, 1988.
13. S.S. Cosmadakis and P.C. Kanellakis, "Parallel evaluation of recursive rule queries," Proc. 5th ACM Symp. on Principles of Database Systems, 1986.
14. S. Dar and H.V. Jagadish, "A spanning tree transitive closure algorithm," Proc. 8th Int'l Conf. Data Engineering, Tempe, Arizona, Feb. 1992, pp. 2–11.
15. D. DeWitt, J. Naughton, and D.A. Schneider, "An evaluation of non-equijoin algorithms," VLDB 1991, pp. 443–452.
16. D. DeWitt, J. Naughton, and D.A. Schneider, "Parallel sorting on a shared-nothing architecture using probabilistic splitting," IEEE PDIS, pp. 280–291, 1991.
17. D. DeWitt and J. Gray, "Parallel database systems: The future of high performance database systems," CACM, pp. 85–98, July 1992.
18. D. DeWitt, J. Naughton, D.A. Schneider, and S. Seshadri, "Practical skew handling in parallel joins," VLDB, 1992, pp. 27–40.
19. W.D. Frazer and A.C. McKellar, "Samplesort: A sampling approach to minimal storage tree sorting," JACM, vol. 17, no. 3, pp. 496–507, July 1970.
20. "Fuzzy LUNA—Fuzzy database system library user's manual and fuzzy LUNA—fuzzy database system library reference manual," OMRON Corporation, 1992.
21. S. Ganguly, A. Silberschatz, and S. Tsur, "A framework for the parallel processing of datalog queries," Proc. ACM SIGMOD Intel. Conf. on the Management of Data, Atlantic City, NJ, 1990.
22. K.-C. Guh and C. Yu, "Efficient management of materialized generalized transitive closure in centralized and parallel environment," IEEE Transaction on Knowledge and Data Engineering, vol. 4, no. 4, pp. 371–381, August 1992.

23. L.J. Henschen and S.A. Nagvi, "On compiling queries in recursive first-order databases," JACM, vol. 31, no. 1, 1984.
24. D.E. Knuth, The Art of Computer Programming: Sorting and Searching, Reading, Mass., vol. 3, Addison-Wesley, 1973.
25. F.T. Leighton, Introduction to Parallel Algorithms and Architectures, Morgan Kaufmann, 1991.
26. J.W. Lloyd, Foundations of Logic Programming, 2nd edition, Springer-Verlag, 1987.
27. D. Maier, The Theory of Relational Databases, Computer Science Press: Rockvill, MD, 1983.
28. S. Nagvi and S. Tsur, A Logical Language for Data and Knowledge Bases, Computer Science Press, 1989.
29. H. Nakajima, T. Sogoh, and M. Arao, "Fuzzy database language and library: Fuzzy extension to SQL," Proc. of the Second IEEE International Conference on Fuzzy Systems, pp. 477–482, 1993.
30. D.S. Parker, E. Simon, and P. Valduriez, "SVP—a model capturing sets, streams, and parallelism," Proc. 18th VIDB, pp. 115–126, 1992.
31. M.J. Quinn, "Parallel sorting algorithms for tightly coupled multiprocessors," Parallel Computing, vol. 6, pp. 349–367, 1988.
32. G. Salton, Automatic Text Processing, Addison Wesley, 1989.
33. B. Salzberg, A. Tsukerman, J. Gray, M. Stewart, S. Uren, and B. Vaughan, "FastSort: A distributed single-input single-output external sort," SIGMOD, pp. 94–101, 1990.
34. D.A. Schneider and D.J. DeWitt, "A performance evaluation of four parallel algorithms in a shared-nothing multiprocessor environment," Proc. ACM SIGMOD, pp. 110–121, 1989.
35. J. Ullman, Database and Knowledge-Base Systems, Computer Science Press, Inc., 1988.
36. P. Valduriez and G. Gardarin, "Join and semijoin algorithms for a multiprocessor database machine," ACM TODS, vol. 9, no. 1, pp. 133–161, March 1984.
37. B.W. Wah and G. Li, "Optimal parallel evaluation of AND trees," Journal of Parallel and Distributed Computing, vol. 10, pp. 1–17, 1990.
38. O. Wolfson and A. Silberschatz, "Distributed processing of logic programs," Proc. ACM-SIGMOD Intl. Conf. on the Management of Data, 1988.
39. O. Wolfson and A. Ozeri, "A new paradigm for parallel and distributed rule-processing," Proc. ACM SIGMOD Intel. Conf. on the Management of Data, 1990.
40. Q. Yang and C. Yu, "Parallelization by the divide-and-conquer method," IEEE Systems, Man and Cybernetics Conference, Chicago, 1992, pp. 1265–1270.
41. Q. Yang, C. Yu, C. Liu, S. Dao, and T. Pham, "A hybrid transitive closure algorithm for sequential and parallel processing," IEEE Data Engineering, Houston, pp. 348–355, 1994.
42. Q. Yang, C. Liu, J. Wu, C. Yu, S. Dao, and H. Nakajima, "Efficient processing of nested fuzzy SQL queries," IEEE 11th International Conference on Data Engineering, Taiwan, 1995, pp. 131–138.
43. L.A. Zadeh, " Fuzzy logic," IEEE Computer, pp. 83–93, April, 1988.
44. W. Zhang, C. Yu, G. Wang, T. Pham, and H. Nakajima, "A relational model for imprecise queries," International Symposium on Methodologies in Intelligent Systems, Trondheim, Norway, 1993.