Distributed Computation of Transitive Closure

by

Christopher P. Almstrom

B.A., Simon Fraser University, 1985

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
August 1988

Approval

Name: Christopher P. Almstrom

Degree: Master of Science

Title of Thesis: Distributed Computation of Transitive Closure

Examining Committee:

Chairman: Dr. J. Peters

Dr. Wosiun Luk,
Senior Supervisor

Dr. JiaWei Han
Supervisory Committee Member

Dr. Stella Atkins,
External Committee Member

8 August 1988
Date of Approval

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

Distributed Computation of Transitive Closure

Author: [redacted]
(signature)

Christopher P. Almstrom
(name)

Aug. 23, 1988
(date)

Acknowledgements

I would like especially to thank Wo Shun Luk for his invaluable guidance and support. His consistent willingness to devote time and energy to this research was greatly appreciated.

I am grateful to Jia Wei Han for his useful suggestions, and for bringing pertinent literature to my attention.

I would also like to thank Stella Atkins for her careful reading and constructive criticism of the thesis.

Garnik Haftevani's explanations of the V-System's communication facilities were indispensable and much appreciated.

I thank Sanjeev Mahajan for his enlightening discussion of the combinatorics of the join operation.

Abstract

In the relational database field, a solution to the Transitive Closure (TC) problem is fundamental to supporting recursive queries. This thesis is concerned with the *Total* Closure problem, rather than with the more restricted Query Closure one.

There are two broad classes of serial TC algorithms. The "iterative" algorithms are defined in terms of relational operations: the solution is computed by a sequence of joins, unions and differences. The "direct" algorithm and its variants are adapted from the matrix- based "all pairs shortest path" algorithm. All of the distributed algorithms that I consider are adapted from serial versions falling into one or the other of these classes. I suggest a systematic approach to classifying and enumerating the distributed alternatives. Four algorithms are simulated, and analyzed in detail. The algorithms are written for a loosely coupled group of workstations connected by a LAN.

Cost estimates for serial algorithms have been based mainly on disk access time. This study is based on a main memory model - the cost components estimated are for local processing (cpu time) and communication. A major emphasis of our work is to investigate the effects of trading increased communication for decreased local processing. In the serial case, disk traffic can be reduced by judicious partitioning of the data. While not *directly* germane to this study, some of those partitioning strategies can be used to distribute the data between machines. Our results indicate that performance is in large part a function of the number of duplicate tuples that an algorithm has to handle. Some partitioning strategies that work well for minimizing disk traffic exacerbate the duplicate problem in the distributed case.

Recent work has suggested that, in the serial case, the direct algorithm outperforms even the best iterative ones. Our work indicates that those results, based on disk access costs, do not apply to the main memory model, especially when the algorithm is distributed.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

The general aim of this thesis is to investigate the relative efficiency of different approaches to distributed computing on a local area network. More specifically, this study falls within the field of distributed deductive relational database operations. The study proceeds from previous work done at Simon Fraser University, on distributed sorting [12] and on distributed join operations [16]. The particular problem investigated is the computation of the *total transitive closure* (TC) of a relation. In the relational database context, transitive closure falls under the heading of "recursive query computation". We have developed, simulated and analyzed four alternative distributed TC algorithms.

In the LAN environment, both local processing and communication costs are significant. The relationship between those cost components cannot realistically be determined a priori. It will depend on the technologies used for processing and communication. We were interested in producing finer-grained predictions than could be had from an asymptotic analysis. Our predictions are for the *expected* case. While the *framework* of the study is analytical, the results are obtained empirically. Time and resource limitations precluded our actually implementing the distributed algorithms developed. Empirical results derive from simulations of those algorithms, and from the measurement of fundamental processing and communication costs.

The *serial* computation of transitive closure has been much written-about of late. Much attention has been paid to the optimization of disk access. Since disk access is so much slower than cpu activity, the cost estimates from such studies are dominated by disk access time. We have adopted a *main memory model* - all relations handled are assumed to reside entirely within main memory. (With increasingly inexpensive hardware, the distributed approach becomes more attractive, and the main memory model more realistic.) Thus disk i/o costs are irrelevant to us. Our estimates are based entirely on expected cpu and communication times. I suggest that these parameters are especially revealing of relative complexities inherent in the *structures of* the algorithms studied.

We hoped that from this rather narrow study, some general principles of coarse-grained distribution would emerge. A few such are summarized in the conclusion. Perhaps our most important contribution is

methodological. An analysis should be fine-grained enough to differentiate, but not so fine-grained that incidental details obscure those features which dominate their complexity. When one intends the analysis to guide an empirical study, there is every incentive to state that analysis in terms of entities which can be easily measured. A happy consequence is that the results will tend to be easy to understand!

The transitive closure problem is defined in the following section. In section 3, I discuss applications for a TC operator in a deductive database. The hardware and software environments for the study are described in section 4. Chapter 2 provides the theoretical context in which our distributed algorithms can be placed. In chapter 3, particulars of implementation are discussed. Chapter 4 is the heart of the thesis. The analytical framework is laid down, and each algorithm analyzed according to it. Then the empirical results are applied to those analyses, yielding expected running-time estimates for each of the algorithms. Conclusions and suggestions for further research are offered in chapter 5. Pseudo-code for the algorithms developed and tabulations of results can be found in the appendix. A detailed cost comparison between our algorithms is also included therein.

1.1. The Transitive Closure Problem

The TC problem has its roots in graph theory. Given a graph $G = (V,E)$, where V is the set of vertices and E the set of edges, the solution is $G' = (V,E')$, where E' is defined as follows:

> For each pair of vertices (i,j), $i,j \in V$ the edge (i,j) is in E' iff there is a path from i to j in the input graph G.

Solutions to the problem proceed from the hypothetical syllogism:

$((i,k) \in E' \text{ and } (k,j) \in E') \rightarrow (i,j) \in E'$

The original TC algorithm (Warshall's algorithm) works directly on the problem graph, represented by a boolean adjacency matrix. Each iteration of the algorithm transforms that graph. It can be adapted to the relational model by interpreting the input relation as a graph. Each (binary) tuple represents an edge, and each element of such a tuple a vertex. The specifics are detailed in chapter 2.

The other algorithms which we consider, collectively called "iterative", proceed more directly from the relational database model. TC can be computed on a relation, say R, that has at least two attributes of the same domain, say A and B. The transitive closure of R would consist simply of R unioned with all the tuples that could be derived from R by transitivity on the attributes A and B. Suppose, for instance, that R contained the tuples (a,b) and (b,c). Those two would derive (a,c). [15]

In a deductive database, TC queries are supported by adding TC as an *operation* to the database manipulation mechanism. The problem (one could also say the database itself) is modelled in *two* ways, extensionally and intensionally.

Extensionally, TC takes as input a *relation*, ie. a table of values. It can be implemented as a series of joins, projections and unions, where the i^{th} iteration takes as input the output from the $i\text{-}1^{th}$. My implementation makes use of hash-based join techniques due to Wang [16].

The *intensional* model consists of a set of rules from which the extension of the TC can be derived. This definition of TC is exemplified by the "Ancestor Relation", which we can express in Prolog-like notation as follows:

Ancestor(x,y) :- Parent(x,y).

Ancestor(x,y) :- Parent(x,u), Ancestor(u,y).

where Parent would be the original relation, and Ancestor its transitive closure.

A TC query may be either for *query closure*, where selection is performed on the original relation before TC is computed, or for *total closure*, where no selection is done. The same rule is used for both query types. To get the total closure, the query is "?Ancestor(x,y)", where x and y are variables. The output from the query will be an unordered list of pairs, of the form $\{(a_1,b_{1,1})\ (a_1,b_{1,2}\ \ldots\ (a_1,b_{1,m})\ (a_2,b_{2,1})\ \ldots\ (a_n,b_{n,m})\}$, where each subscripted a and b is a different constant. For query closure, the input is of the form (a,y), where a is a constant. The output will be of the form $\{(a,b_1)\ \ldots\ (a,b_n)\}$. Another query type of the same intensional form is the *existence check*. This query simply detects whether or not there is a path between a given pair of vertices. Here, *both* the variables in the rule are replaced by constants, and the output is just "yes" or "no". For a discussion of TC in both intensional and extensional databases, see [5].

In this study, only the *total* closure problem is investigated. When "transitive closure" or "TC" are used, they denote total closure.

1.2. Applications for the TC Operation

For most user requests involving TC, *query closure* or *existence checking* will suffice.[i] Looking at the query closure's output form, shown above, you will see that all pairs in the output have the same constant as the first element value. This allows us to "drop" the first column not only from the *answer*, but from the intermediate relations which drive the computation. This "unary" approach is computationally less expensive than the "binary" one, which retains both columns of the relation throughout.

A unary answer to a *total* closure query would be meaningless - pairs are required to specify who is an ancestor of whom. Although this makes solving the total closure problem more expensive, the extra expense may be justifiable, even if only query closures are asked for. The query closure results are not resusable, unless the same query is repeated. Once the *total* closure has been computed, any particular query closure can simply be looked up in the result relation. (It helps if it is sorted, or stored in a hash table.) If a total closure algorithm is used for query closure queries, the *first* query will be costly, but each subsequent one on that relation can be implemented as a simple selection. It should, however, be kept in mind that the TC result may be very large - unless main memory is plentiful, selection from that relation will likely involve disk access. Another option for improving response time is to *precompute* the total closure.

Many of the methods used to compute total closure can also be applied to the query closure problem. The Delta-Wavefront varieties (see chapter 2) can be used almost unchanged. Generally, however, query closure is not a very expensive proposition. Accordingly, there is little point in distributing it. When the fan-out of the underlying graph is high, though, the size of the query closure can explode. In such a case, the distributed algorithms presented in this thesis might prove attractive. This is particularly true of the Wavefront algorithms, since they are easily adaptable to the less costly unary form.

Transitive closure is the simplest of the recursive queries. Its computation can play an important role in the processing of many more complicated recursive queries. [6]

---

[i] Users are more likely to want to know about the characteristics of a single entity than of every entity in a relation. I, for example, am more interested in finding out who *my* ancestors are than in knowing the ancestors of everyone in my home town.

1.3. Research Environment

The experiments were performed on a Sun3 workstation, using Sun System Release 3.0, a heavily enhanced version of the 4.2 BSD UNIX operating system. The Sun3 has an MC68020 processor. The results shown in the appendix assume that all participating machines are Sun3's, each with at least 4Mb of main memory. The test relations were generated in main memory, by means of the UNIX "random()" function.

Point-to-point communication costs are based on tests by Ling and Wang [12], using Sun3 workstations connected by a 10Mb Ethernet local area network. They used the V-System, a distributed operating system developed at Stanford University. Broadcast costs are based on tests by Haftevani. All of these tests were conducted at Simon Fraser University. V supports very efficient communications operations. Our analyses assume that the V-System is used for communication.

The V multi-cast operation has receivers wait a random period before acknowledging, in order to prevent acknowledgements overrunning the originator's buffer capacity. Such overrun is unlikely to occur with Sun-3's, since they have ample buffer space. Accordingly, we assume that the random waits have been removed from the V-System multicast. This improves the performance of that operation by an order of magnitude. The broadcast is used only by the Direct algorithm. Since that algorithm incurs very high communication costs, we wished to assume as optimistic a cost estimate as was realistic.

Chapter 2
The Algorithms and their Theoretical Context

All four of the TC algorithms studied are adapted from well-known serial ones. Figure 2-1 depicts the context in which our algorithms lie.

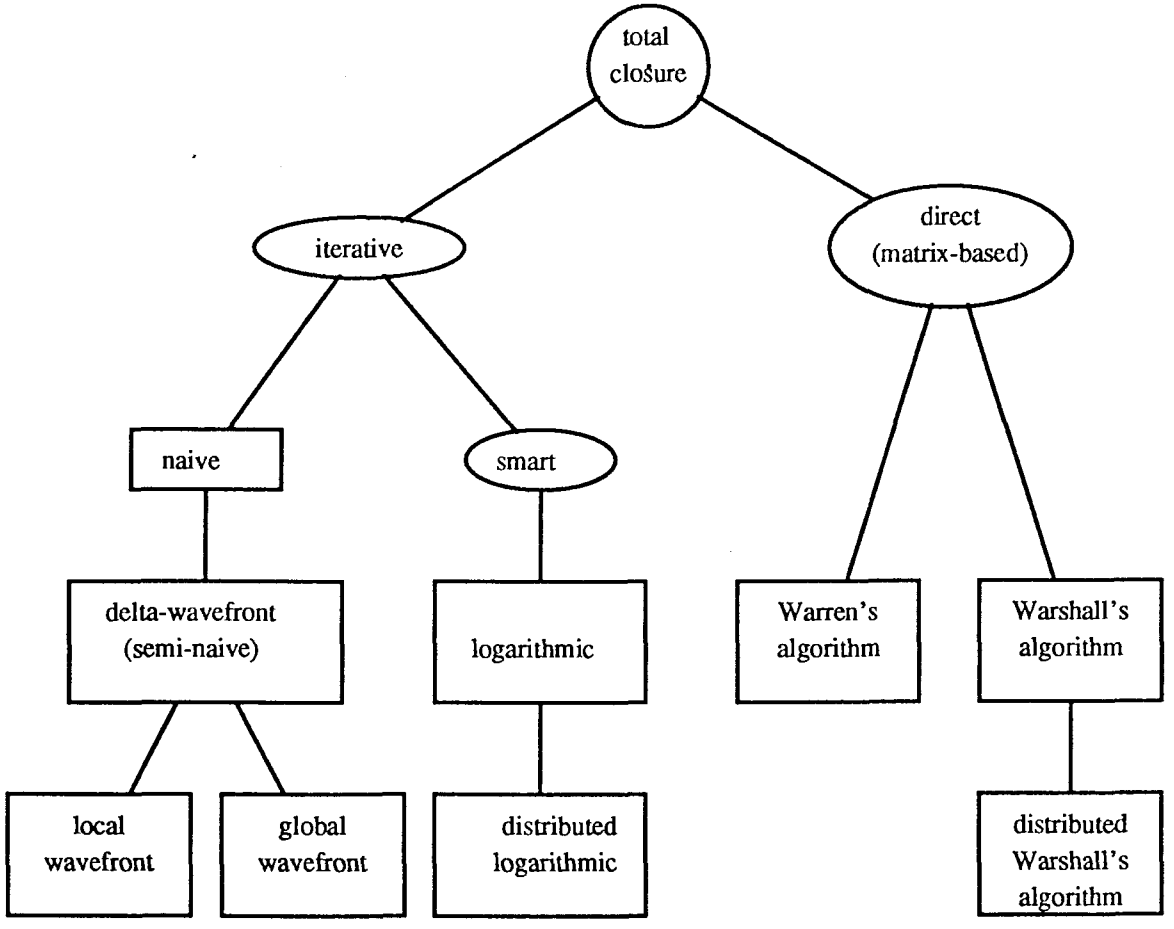

Figure 2-1: Hierarchy of TC Algorithms

Problems are indicated by circles, algorithms by ellipses, and algorithm instances by rectangles. Children of classes are special cases; children of algorithms are refinements of their parents. The four distributed

algorithms studied are shown on the bottom level of the graph. Their serial progenitors are shown on the next lowest level. Of course, there are TC algorithms which do not appear in figure 2-1 - only those directly relevant to this study are shown.[i] In the first section of this chapter, I introduce the serial bases for our distributed algorithms, and discuss refinements of them which may or may not be applicable to the distributed case. The second section concerns theoretical issues particular to the distributed case. That section offers the motivation for our choosing the distributed versions that we did, and places them in the context of other distributed alternatives.

2.1. The Serial Background

Lu distinguishes between two classes of TC algorithms, the "iterative" and the "recursive". [11] I have adopted the terminology used by [1], and refer to the "recursive" class as the "direct" algorithms.[ii] In the latter approach, the underlying graph is represented as a boolean adjacency matrix - the problem is then solved by finding the transitive closure of that matrix (as in the familiar dynamic programming solution to the "all pairs shortest path" problem [8]). The central operation is bit oring. To adapt this method to the relational database context, the graph (relation) is represented by successor lists. The effect of bit oring two matrix rows is then captured by merging one successor list into another. Some versions also maintain a set of predecessor lists, to facilitate finding those successor lists which have to be merged. Warshall's Algorithm, and Warren's improvement on it [18], are instances of the direct class.

The iterative approach is based on relational algebra. The main data structures are relations (flat tables), and the algorithms use relational operations - joins, unions and differences.

I will first discuss the iterative algorithms, then the direct ones.

---

[i]Ioannidis and Ramakrishnan suggest a TC algorithm based on depth-first search. [10] Raschid and Su [14] propose a parallel TC algorithm employing pipelining.

[ii]Agrawal and Jagadish explain their choice of terminology thus: "Unlike iterative algorithms, such as the semi-naive and the logarithmic algorithms, the termination of our algorithms does not depend on the length of paths in the underlying graph (hence, the name *direct* algorithms)." [1]

2.1.1. The Iterative Algorithms

2.1.1.1. The Naive Algorithm and its Derivatives

To find the transitive closure T of a database relation R, the iterative algorithms compute the least fixed point of the following relation:

$T = R \cup \pi_{1,4}$ (T.2 join R.1) [11]

The Naive Algorithm is as follows:

```
OldT := R;
REPEAT
        T := OldT ∪ (OldT join R);
        ΔT := T - OldT;
        OldT := T
UNTIL (ΔT = Φ) [13]
```

where $_{\Delta}T$ comprises newly-added tuples.

The form in which I have presented the Naive Algorithm is taken from [13].

During each iteration, the Naive Algorithm uses as the driving relation OldT, which comprises *all tuples found so far*. Since at each iteration the Naive Algorithm uses *the entire result found so far*, it involves a lot of redundant processing. It would obviously be more efficient to use only *newly-generated* tuples. The Delta-Wavefront Algorithm[iii] reduces redundant processing by using only newly-generated tuples - $_{\Delta}T$ - to drive the join on the next iteration. The difference operation that removes redundant tuples, ($R_{\Delta} := R_{\delta} - T$), does incur some cost. However, the Naive Algorithm must *also* perform a difference operation ($_{\Delta}T := T -$ OldT) in order to check for the termination condition. T in the Naive method must be at least as large as R_{Δ} in the Delta-Wavefront Algorithm. I conclude that it is improbable that the Naive Algorithm will outperform the Delta-Wavefront. Accordingly, I will only investigate the latter. The Delta-Wavefront Algorithm is:

```
T := R;
R_Δ := R;
WHILE (R_Δ ≠ Φ)
        R_Δ := R_Δ join R;
        R_Δ := R_Δ - T;
        T := R_Δ ∪ T
END WHILE
```

where R_{Δ} consists of new tuples during the current

[iii] also referred to variously as the "Henshen-Naqvi", "Brute Force" or "Semi-Naive" Algorithm.

iteration.

This algorithm originates with Henshen and Naqvi [7]. The form and notation used are from [13].

2.1.1.2. Smart Iterative Algorithms

Ioannidis has formalized what he calls "smart algorithms" as follows:

$$R^{+}=\prod_{k=0}^{\infty}\sum_{j=0}^{m-1}R_0^{jm^k}$$

[9]

If the depth of the transitive closure (longest path) is n, then the Delta-Wavefront approach must iterate n times. (Level relaxation, discussed below, may decrease this number.) The smart algorithms *always* require $log_m\ n$ iterations. I shall only test the Logarithmic Algorithm, which is the special case of smart ones, with $m = 2$. It can be expressed as follows:

```
T := R;
R_Δ := R;
D_R := R;
WHILE (D_P ≠ Φ)
        R_Δ := R_Δ join R_Δ;
        D_R := T join R_Δ;
        T := T ∪ D_R ∪ R_Δ
END WHILE
```

[13]

The algorithm is from Valduriez and Boral. [15] The particular presentation form is again from [13].

Note that R_Δ is always R^{2i} - ie. a power of 2.

2.1.1.3. Delta-Wavefront Refinements

The algorithm above is really just the *skeleton* of the delta-wavefront method. Various optimizing refinements have been suggested. It is not always clear whether a given refinement should be considered to yield a new algorithm, or whether to regard it as simply an implementation detail of the refined algorithm. Those refinements which modify the *basic flow of control* I consider in this chapter. By "modifying flow of control", I mean that the refinement has such effects as changing the number of iterations required to complete the computation, or changing the amount of data processed. Choices as to how particular relational operations are implemented are discussed in the chapter 3, "Implementation Issues".

I first discuss (see "Clustering") the effect of the *physical* ordering of the data on performance. Then I discuss two refinements which, in the serial case, seem almost guaranteed to outperform a "naive" implementation. These are *level relaxation* and *source reduction*. Given the importance of these refinements in the serial case, the possibility of carrying them over to the distributed versions seems well worth investigating.

Clustering

Methods used to optimize disk traffic might also help us to optimize intermachine communication. The basic thrust of such methods is to take advantage of clustering to decrease the number of page faults. The relations are partitioned into small enough pieces so that two of them (one from each relation being joined) can fit in main memory. After joining these partitions, the result is repartitioned and written back to disk, and the next input partitions are read in. The clustering of data allows it to be dealt with (input and output) a *partition* at a time, rather than tuple-by-tuple.[iv] Of course, it is just this amenability to partitioning that makes coarse-grained load sharing possible. Where partitioning allows a serial approach to deal with large chunks of data *consecutively*, it may allow a parallel one to deal with them *concurrently*. Since all of my distributed versions derive from data distribution (as opposed to, say, task distribution), all take advantage of clustering.

Level Relaxation

In the basic version, no tuple drives more than *once* per iteration. All new tuples found during a given iteration are on paths of the same length (ie. they are at the same level). This scheme can be improved on through the use of *level relaxation*. Newly-generated drivers which can drive tuples in the partition currently in memory are allowed to do so. The current partition is swapped back to disk only after it has no drivers left.

The situation is best understood by considering the graph of the relation, where each tuple is an edge, and each distinct attribute value a node. Each partition is then that subgraph whose nodes hash into the appropriate range. Any path which lies *entirely within* the subgraph corresponding to the current partition is completely explored before the disk is written to. This saves reads as well as writes, since those tuples do not have to be read in later. Note that tuples *not* in the current partition cannot drive any tuples currently in memory, and therefore *have to be* written to disk - they are then read back in when their partition is

[iv] Serial techniques based on clustering have been widely discussed in the literature. See [5], [3] and [15].

processed. (To some extent, "foreign" tuples can be buffered before writing to their disk partitions.)[v] It is fairly easy to see that if level relaxation is used, the serial algorithm might well finish in fewer iterations than without it. (For an example, see [5].) This is expecially likely if there are few partitions - the fewer the partitions , the higher the probability that a given path will not cross partition boundaries. In the serial case, level relaxation is used to reduce disk access. In the distributed case, it can reduce inter-machine communication. The advantages gained for disk access, however, cannot all be carried over to communication optimization. This matter is discussed further in the next section.

Source Relation Reduction

Another serial refinement, suggested by Lu [11], is to dynamically reduce the size of the *source* relation R. This may speed up subsequent join operations. But the main advantage is that less tuples have to be written to disk when the current partition is swapped out. (Of course, there will also be less disk reads when that partition is swapped in again.) After each iteration, tuples which can never in future be matched with any driver are removed. How are such tuples detected? The key observation is that if a tuple is not driven during iteration *i*, then it cannot be driven in any iteration *j* later than *i* either. (If a node has no children, then it has no grandchildren either.)

Our implementations all use hash-based joins. The method involves looping through the wavefront relation R_{Δ}, hashing each tuple there into the source relation R. Of course any benefit to be had from reducing the size of a hash table can also be had simply by decreasing its loading factor (increasing the number of buckets). However, if the key domain is realistically large, the table will inevitably be too small to preclude collisions. We resolve collisions by chaining. Search time can be reduced by deleting redundant links (tuples) from the chains.

Just doing the join (without looking for redundant tuples) involves traversing a certain portion of the hash table, implemented as linked lists (chains). If a particular bucket is not used during a given iteration, then it will not be hashed into on any subsequent iteration either. All tuples in that bucket are redundant - there is no need to delete them, since they will never contribute to lookup cost. In fact, any redundant tuples which *could* contribute to future join (table lookup) costs can be detected *without doing any more traversals than are required for the join anyway*.! When traversing a chain in the hash table, just mark those which are used - ie. matched with a driver. Any unmarked links traversed during a subsequent iteration can be deleted.

---

[v]For further discussion of level relaxation, see [4] and [5].

Though this method of reducing the size of the source relation does not involve extending the search, it *does* add to the cost of visiting any given tuple. The relative benefit of implementing the method depends, of course, on being able to implement the marking, mark checking and deletion operations very efficiently. The tradeoff is between these *added* processing costs, and the processing costs saved in subsequent table lookups.

Our results indicate that table lookup for the join operation is not an especially large component of the algorithms' overall costs. The cost of reducing the source relation's size will most likely outweigh the benefit.

2.1.2. The Direct Algorithms

The direct algorithms most discussed in the literature are versions either of Warshall's algorithm or of Warren's modification of it. In describing these, I use the notation of Agrawal and Jagadish [1].

The direct algorithms were originally designed to compute the TC of graphs represented by boolean matrices. A graph on v vertices is represented by a v x v matrix. Each element a_{ij} of the matrix is set to 1 if the edge (i,j) is in the graph, 0 otherwise. Warshall's Algorithm is as follows:

```
For k = 1 to v
        For i = 1 to v
                For j = 1 to v
                        a_ij = a_ij OR a_ik AND a_kj
```

[19]

This algorithm can be adapted for relational use as follows:

> Each arc in the graph is represented by a tuple (a pair). Successor lists can be determined by sorting the relation on its first column. (In our distributed version, sorting is avoided. Successor lists are formed by inserting the relation into a two-tiered hash table, as described in chapter 3, "Implementation Issues".) For each unique column 1 value n, fetch its successor list. Then for each predecessor p of n, fetch p's successor list, and add to it all elements of n's successor list (removing any duplicates). Finding the predecessors can be expedited by maintaining predecessor lists. (Whenever an element q is added to p's list, q's predecessor list must also be updated.) [1]

Warshall's algorithm scans by *columns*, so that it can OR by rows. This makes it very inefficient for a paging environment - the *entire matrix* must be paged through during each iteration of the middle loop! If we could scan *and* OR by rows, then more work could be done before paging became necessary. [18][vi]This motivates Warren's Algorithm, which is as follows:

[vi]We can take advantage of clustering. Assuming that rows i and $i+1$ are contiguous with each other, there is a good chance that they will fall within the same page. After scanning row i, there is thus a good chance that row $i+1$ will already be memory-resident.

```
For i = 1 to v
        For k = 1 to i-1
                For j = 1 to v
                        a_ij = a_ij OR (a_ik AND a_kj)

For i = 1 to v
        For k = i+1 to v
                For j = 1 to v
                        a_ij = a_ij OR (a_ik AND a_kj)
```

[18]

Warren's algorithm can be adapted to the relational context in a manner very like that for Warshall's:

> For every vertex, fetch its successor list. Then for every member of that list, fetch *its* successor list and add it to the list for the original vertex. [1]

Two loops are required to ensure that the algorithm is complete (ie. that it finds all edges in the TC).

Our distributed direct algorithm is adapted from Warshall's algorithm. For reasons which will be discussed in section 2 of this chapter, we have not considered a distributed version of Warren's algorithm. It is mentioned only because it is so much discussed in the literature.

2.2. The Distributed Context

For any given serial algorithm, there will be numerous distributed versions of it, depending on the degree to which it is distributed. It is standard to consider this "degree of distribution" along three dimensions: the interconnections of the physical elements available, the granularity of control distribution, and the granularity of data distribution.

2.2.1. Physical Distribution

As discussed in the Introduction, under "Research Environment", the different machines are connected by a local area network (Ethernet). This loose coupling encourages a coarse-grained approach to both control and data distribution. The granularity of control that will be appropriate depends in large part on the granularity of data distribution. I discuss *temporal* matters (eg. when or how often data are exchanged) under "control distribution" and *spatial* ones (eg. sizes of messages and local fragments of the data) under "data distribution".

2.2.2. Control Distribution

Each machine runs a copy of the same program. Individual machines proceed largely independently. I call one of the machines the "coordinator", although this is perhaps a misnomer. All that is special about this coordinator is that the source relation originates with it, and the result is gathered there. Data are transmitted by either a blocking point-to-point send or a blocking broadcast. (The reasons for this are discussed in chapter 3, "Implementation Issues".) As a result, when data are exchanged the machines are synchronized. Thus they run in lockstep, though the steps may be very large. In the Local Wavefront Algorithm, for example, data are shared (and synchronization thus done) only at the beginning and the end.

A decision must be made regarding *when* data will be shared. The options can be envisioned as lying on a continuum between very "eager" and very "lazy" communicators. If the very eager approach is taken, each machine will transmit each datum not in its own partition as soon as that datum is generated. The atom of local processing is the *tuple*. If the very eager option were chosen, messages would consist of single tuples. The bound on laziness is imposed by the *problem*. At the start of the algorithm, the source relation must be distributed. At the end, local results must sent to the controller.

The very eager approach is not suitable for a loosely-coupled system - the cost per tuple of short messages is too high. The grain of communication that we chose is the *relation fragment*. Given that data granularity,

there is still a range of options available. The degree of laziness that can be tolerated (subject to the bound mentioned above) depends on how the data are intitially distributed. As explained in the next section, "Methods of Distribution", the Local Wavefront algorithm provides each machine with enough state (the entire source relation) to complete all local processing without the exchange of intermediate results. In the Global Wavefront method, less global state is initially replicated locally. Consequently the machines need to exchange intermediate results. In the Logarithmic and Direct algorithms, intermediate results must *always* be exchanged - possession of initial state is never sufficient to complete local processing. The Direct algorithm is enough of a special case to warrant special consideration - see below.

In the iterative algorithms, new tuples are generated by the join operation. Thus concerns of correctness demand only that data be exchanged once per join. There is no advantage to be gained from sending them more often, since this will only result in more smaller messages being transmitted. This leaves virtually no choice for the Logarithmic algorithm - data are exchanged twice per iteration (there are two joins per iteration). For Delta-Wavefront adaptations, the viable options can be placed on a continuum between exchanging once per iteration at the eager extreme and exchanging only for result collection at the lazy end.

Distributed Level Relaxation

What the aforementioned continuum describes is the degree of *level relaxation* employed. The eager extreme, instantiated by the Global Wavefront algorithm, employs none, while the Local algorithm is as lazy as can be. If level relaxation is employed, each machine will be allowed to continue processing for more than one iteration between synchronization points. Only once it has exhausted all locally available drivers will it exchange with the other machines.

In the serial case, the main benefit of level relaxation is that it reduces disk access to the *current* partition. The current partition in that case corresponds to the *local* one in the distributed case. But avoiding communication with the *local* machine is non-issue - the tuples are already there! We thus expect level relaxation to have less of an impact on the performance of distributed versions. What use of that technique *will* reduce is the amount of communication with *other* machines. (This is analogous to the disk i/o benefits the serial method gains from buffering writes to foreign partitions.)

A version intermediate between the Global and Local algorithms can be obtained as follows:

> The data are distributed as in the Global method. But each machine is allowed to continue local processing until it exhausts all drivers either resident or generated there. Only then does it send to the other machines.

I would expect this intermediate algorithm to be especially susceptible to the sorts of statistical anomolies in the data that lead to uneven load sharing. For example, the local presence of a single edge could result in a much longer path's being locally traversable. While this path is being explored, the other machines may be waiting for results from the explorer. Note also that the method does not scale well. As more machines are used, their partitions will be small. The lengths of paths within the local partition will tend to decrease, while the problem of uneven load sharing will be exacerbated. Allowing level relaxation will also demand a more complicated algorithm.

A further variation would result from using the Logarithmic method *locally* to exhaust a machine's drivers. We have not investigated either of these intermediate Wavefront algorithms further. I do think that the option deserves further study.

The Local Wavefront algorithm achieves a high degree of level relaxation at the cost of *increased local processing*. While the local results from the Global method can simply be concatenated, the Local method's local results must be merged. This matter is discussed in greater detail in chapter 4, "Analysis", under "Combined Results" (subsection 4.3).

The Direct Algorithm

Like any dynamic programming algorithm, Warshall's algorithm computes the globally optimal solution, the TC, by successively combining locally optimal subsolutions. This approach rests on an orderly decomposition of the problem into subproblems. The vertices of the input graph are numbered, say 1 through n. What particular labelling scheme is used is immaterial, so long as it constitutes a well-ordering of the vertices.[vii] This labelling imposes what amounts to a sequential traversal of the vertices (the outer loop). The optimal subsolution after iteration k consists of all paths running through no vertex numbered higher than k. The globally optimal solution is the input graph's TC.[viii]

Iteration k's solution is obtained by combining optimal subsolutions produced by iteration k-1. Only if *those particular* subsolutions are combined can iteration k's results be guaranteed optimal. In other words, if iteration k is not fed all paths discovered previous to it, it will in turn leave those paths out of *its* results. Ultimately, the algorithm may produce incomplete results. The sequential order of traversal is not, then, some incidental property of the algorithm that can be removed by tinkering!

[vii] This follows from the applicability of the Principle of Optimality. Note that this allows us to "label" the vertices by the order in which they appear in the primary chains of the hash table. There is no need to sort the input data.

[viii] "Globally" here means "belonging to the whole graph", rather than "inclusive of the states of all participating machines".

In the relational adaptation, each iteration k will involve merging k's successor list into the successor list of each of k's currently-known predecessors. Each predecessor p of k is resident at some machine. That machine will do the merging. It is essential that it use an up-to-date copy of k's successor list! This leaves us with two choices. Either we can ensure that all of k's currently-known predecessors (and their successor lists) are on the same machine as k, or we can send k's current list to each machine that has such a predecessor. The first alternative demands that the *current global state* be replicated on k's machine! This degenerates to the serial algorithm. We adopt the second alternative, which involves broadcasting k's current successor list at the beginning of each iteration k.

Warren's algorithm is one of a number of adaptations of Warshall's which are motivated by the objective of minimizing disk access. The techniques used are like level relaxation, in that they exploit clustering. The graph vertices are divided into partitions of close-to-adjacent vertices. By keeping processing within a partition, the number of page faults is minimized. Within a partition, the vertices are traversed sequentially. The partitions *themselves* are swapped in and processed *one after the other*. Indeed, these conditions *must* be enforced, because no computation can be allowed to overlook possibly optimal previous results! A partitioning strategy that does not allow the partitions to be processed concurrently is not what exactly what we distributors seek. In fact, Warren's algorithm is even *less* amenable to distribution than is Warshall's. Since it uses two loops in series, we would have to broadcast twice as often.

What *can* be done concurrently is the processing *within* an iteration. Each vertex's successor list is kept on a different machine. This ensures that each list is updated only by its host. When that machine receives k's successor list in iteration k, it therefore has all the state required to merge the two together. Predecessor lists are maintained for each vertex either resident or received. This allows the machine to find any locally resident predecessors efficiently.[ix]

2.2.3. Data Distribution

Since the machines are loosely coupled, it makes sense to send fewer long messages rather than more short ones. The level of granularity is the *horizontal fragment* of a relation. This is finer than the relation, and larger than the individual tuple.

For each relation in each algorithm, two independent choices are made. The first is between redistributing

[ix] In essence, we must do the scanning sequentially, but can OR in parallel. This is analogous to the serial bit-ORing technique.

the relation during each iteration (distribute) and not redistributing (serial). Not redistributing does not preclude the relation's *initially* being distributed between the various machines - ie. the serial option does not preclude load sharing. The second choice is between horizontally fragmenting the relation (partition) and not (replicate). Thus for each relation there are four possibilities:

```
sr:     (serial,replicated)
sp:     (serial,partitioned)
dr:     (distributed,replicated) and
dp:     (distributed,partitioned).
```

If option sr is chosen, then each machine simply starts with a copy of the original relation. They proceed in parallel, and communicate only at the end. Under option dr, the machines *gossip* the relation being replicated, and each unions the incoming copies. The unit of distribution is the *tuple*.

The following discussion pertains only to the iterative algorithms. The Direct algorithm is treated separately.

Correctness

A distributed algorithm would be invalid if it generated any tuples not generated by the serial algorithm. New tuples are only generated by the join operation. Thus they can result only from a match between two existing tuples. Thus tuples can only be generated if there is a path between their left and right members, in the underlying graph. Fragmenting the relations does not introduce new tuples. Validity, therefore, is not problematical.

The algorithm will be *incomplete* if it fails to generate all tuples in TC. This could happen if, in the *global* relations, there exist matching tuples (possibly originating on different machines) which are not brought together on some *particular* machine and there joined.

There are two ways to make sure that matching tuples find each other. One is to replicate *the current global state* of one of the relations. Any given tuple can then find its match (if any), since:

i) *all* of the relation to be matched into is present locally (at each machine), and

ii) every tuple in the global relation to be matched must reside in *some* machine.

I will call a relation "static" if its contents do not change over the course of the algorithm, and "dynamic" otherwise. (In the Delta-Wavefront Algorithm, R is static. All of the relations in the Logarithmic Algorithm *except for R* are dynamic.) Option sr, serial and replicated, will be sufficient to guarantee completeness only if the relation is static. Otherwise, its state has to be updated at each iteration by gossiping (dr).

The distributed join can also be guaranteed complete if we ensure that for each partition (machine), any tuple that can be driven from that partition is available locally, and similarly that all drivers for a given "passenger" are available. In other words, if both relations are fragmented, the fragments must be properly matched (I will say that the tuples have to be "synchronized".).

For completeness' sake, we must also ensure that no tuples generated, other than duplicates, are thrown away by the unions or differences. As long as all *local* results are kept, the global result can be computed by unioning them at the end of the algorithm.

2.2.3.2. Eliminating Uninteresting Versions

If an iterative TC algorithm uses n relations, specifying a type (sr, sp, dr or dp) for each allows us to enumerate 4^n different versions of it. The advantage of this approach is its exhaustiveness - we can be assured that all data-distribution alternatives at the relation-fragment level of granularity have been considered. Fortunately, almost all of these alternatives can be discarded a priori. Many are simply incorrect. Others are either inefficient, or are not really "distributed" versions.

The work done by a machine is a function of the size of the input to its joins. Alternative load sharing strategies thus depend on how the joins' input relations are distributed between machines. The other relations are basically just storage bins, used to accumulate results. There is no point in communicating these relations during processing; gathering them together at the end is sufficient. Suppose that there are n relations in the serial algorithm, but only r of them are join inputs. The number of *genuine* alternatives can now be pared down from 4^n to 4^r.[x]

The key to distribution is, as mentioned, the manner in which the join processing is shared. Suppose that *both* input relations to a join are replicated at some machine. Then *no* speedup will be achieved for that machine. Any advantage gained by data distribution hinges on presenting each machine with only a *fragment* of the global state. This leads us to make the following assumption:

Locality assumption:
For any pair of join input relations, at most one of them is replicated.

The locality assumption allows us to eliminate from consideration a large number of "pseudo-distributed"

---

[x]A join input relation may be composed by unioning other relations. If all of component relations are available locally, then that input relation can be composed, and the join will be complete. It may be cheaper to distribute one or more of the *components*, rather than the join input relation itself. In the Logarithmic Algorithm, for instance, D_R is distributed, rather than T.

versions, along with the serial one. (In the serial case, both relations joined are of type sr. An example of a pseudo-distributed version would be one where those relations were both of type dr. Data *is* distributed. But this leads only to duplication of effort, rather than to load sharing.)

Some other versions will, depending on correctness criteria, involve redundant communication. For instance, if (dp,sp) is a correct option for a given join operation, (dr,sp) is also correct. But the latter constitutes wasteful overkill. If a relation is static, then any version which calls for distributing it (after initial set-up) is patently inefficient.

2.2.3.3. Data Distribution for the Delta-Wavefront Algorithm

I will test two versions of the Delta-Wavefront Algorithm.

The Delta-Wavefront Algorithm has only a single join. Thus we can immediately narrow our attention down to the $2^4 = 16$ versions that result from choosing for R_Δ and R. I will specify versions by ordered pairs, where the first element represents the choice for R_Δ, and the second the choice for R. For example, the serial version is (sr,sr).

Versions (sr,sp), (sr,dp), (sp,sp) and (sp,dp) can be discarded as incorrect. (The dynamic relation, R_Δ, can be serial *only if* the static one, R, *is replicated.* New tuples generated at a given machine may not be in that machine's partition. To find their passengers, either those drivers must themselves be redistributed, or all potential passengers must be available locally.) Four more options, (sr,sr), (sr,dr), (dr,sr) and (dr,dr) can be rejected by the Locality Assumption. All remaining versions which involve (re)distributing R can be eliminated as involving redundant communication - four more versions bite the dust. We are left with (dr,sp), (dp,sp), (dp,sr) and (sp,sr). Since (dp,sp) is correct, we can eliminate (dr,sp) and (dp,sr) as involving redundant communication. We are left with only (dp,sp) and (sp,sr). The first yields what I call the "Global Wavefront Algorithm", and the second the "Local Wavefront".

Figures 2-2 and 2-3 illustrate how, in the Global Wavefront algorithm, the join operation fragments the driver relation. Figure 2-2 depicts the situation before the join. Each machine has a separate horizontal fragment of both join input relations. The driver, R_Δ, is partitioned on the 2nd column, the passenger, R on the 1st. This allows all matching tuples in each partition to be found by the machine dedicated to that partition.

Figure 2-3 shows the Global Wavefront's fragmentation pattern after the join. Since the passenger relation

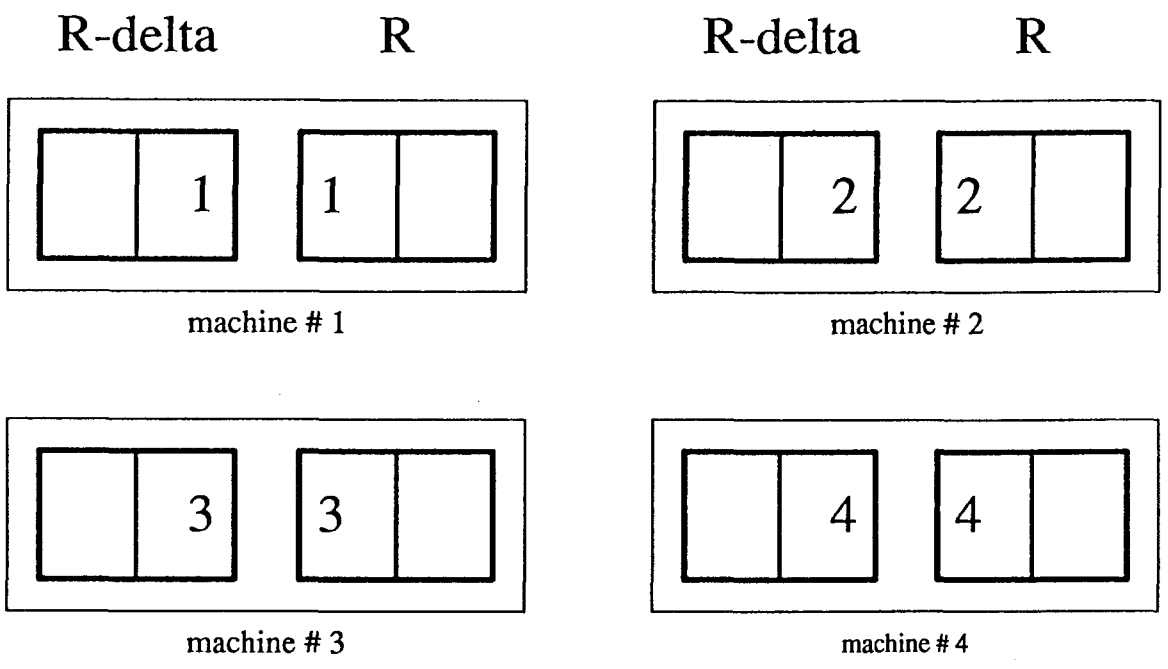

Figure 2-2: Global Wavefront - before join

(R) is static (ie. it does not change throughout the course of the algorithm), it is still correctly partitioned. The join output forms the driver, R_Δ, for the next iteration. Note that the original join column(s) has been projected away. Thus each machine can, and probably will, have new tuples belonging to other machines' partitions. These must be sent to the appropriate hosts, so that all matching tuples are brought together for the next join.

In the Local Wavefront algorithm, we avoid exchanging by replicating all of the source (passenger) relation on each machine. Figure 2-4 shows the situation before the join in the first iteration.

Figure 2-5 shows the situation after the first join. As with the Global algorithm, the passenger relation is static. The join will again produce tuples which fall outside their host machines' partitions. But now, those tuples can find any passengers which match them, since the entire *global* passenger relation is present on *each* machine. It is possible to maintain that condition without communication, since the passenger relation is static.

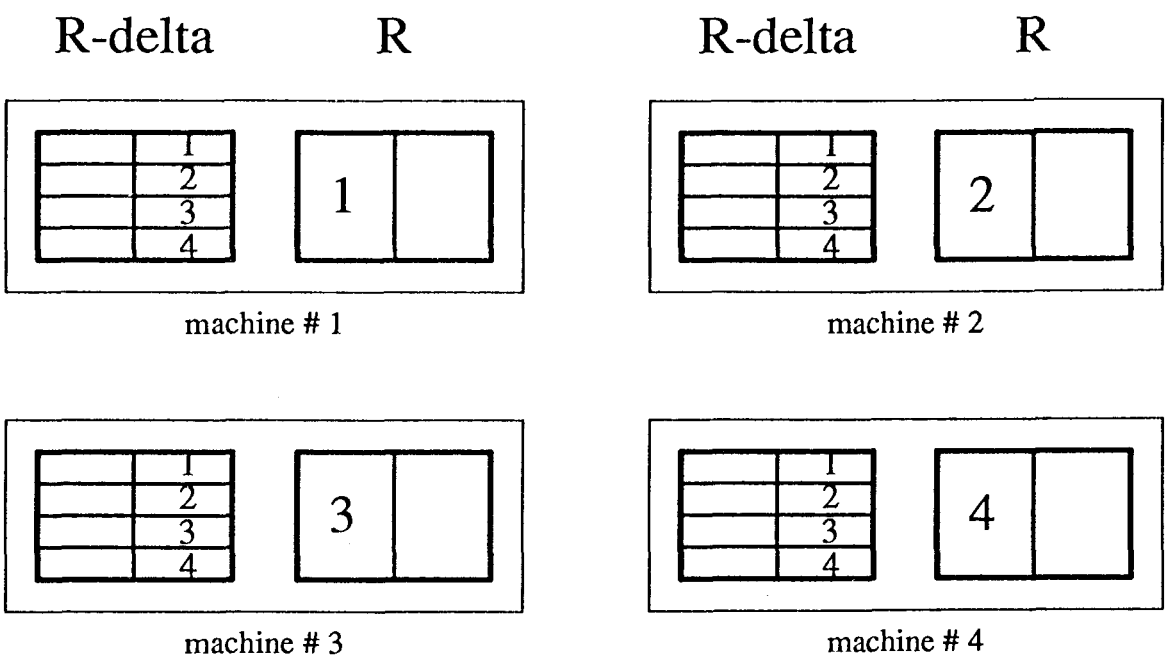

Figure 2-3: Global Wavefront - after join

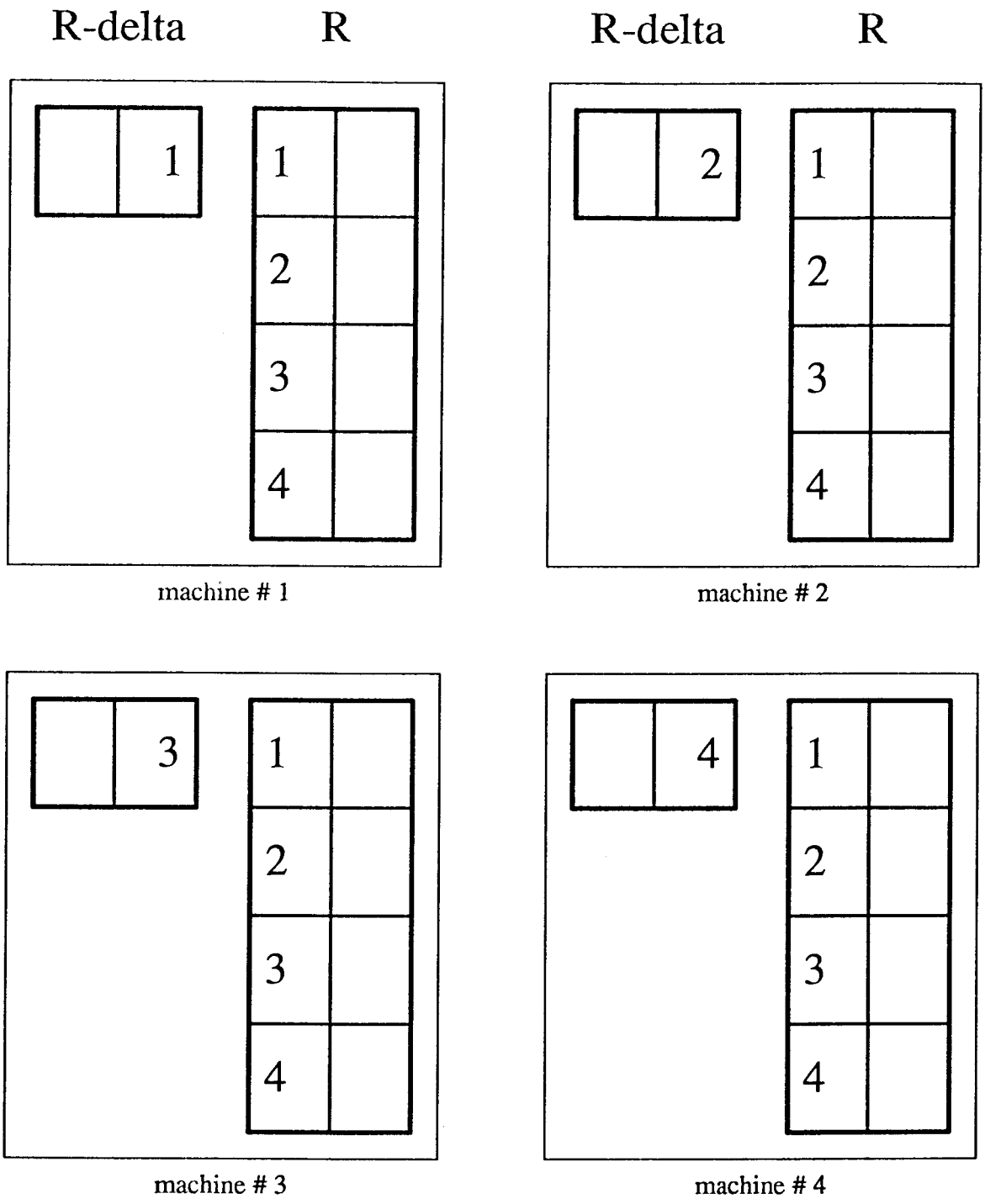

Figure 2-4: Local Wavefront - initial distribution of fragments

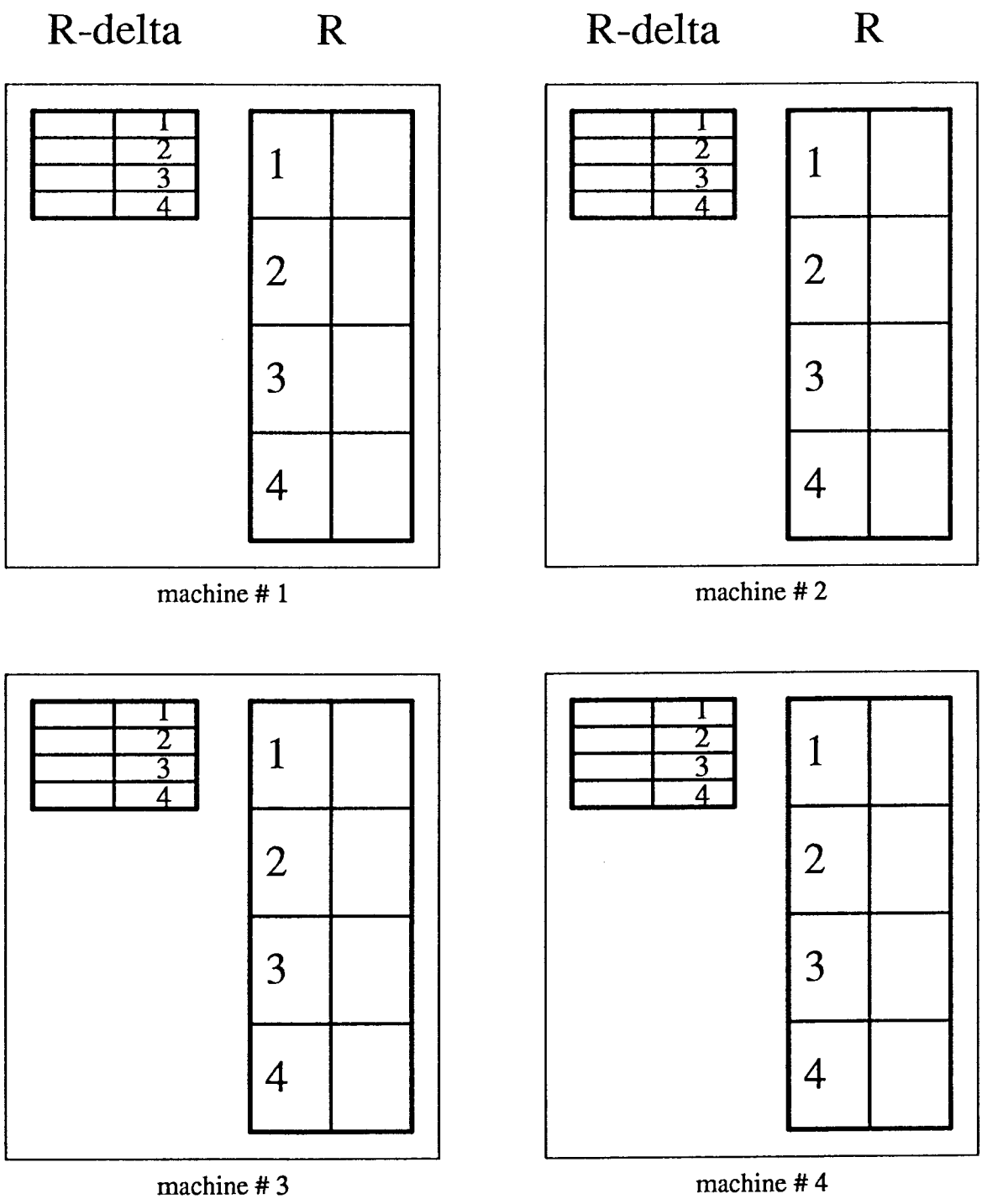

Figure 2-5: Local Wavefront - after the initial join

2.2.3.4. The Logarithmic Algorithm

The Logarithmic Algorithm has three relations that are amenable to distribution, D_R and two copies of R_Δ. (R is only used for purposes of initialization, and T is just a storage bin.) In more explicit notation, line 4 would read:

$$R_\Delta := \pi_{1,4}\, R_\Delta.2 \text{ join } R_\Delta.1$$

Those relations which are fragmented are partitioned by the hash value of their join attribute. A given tuple in R_Δ may hash into different partitions, depending on whe ther it is hashed on attribute 2 (left side) or 1 (right side). Consequently, when R_Δ is horizontally fragmented (sp or dp), it must be implemented as two separate relations. While these will be globally identical, their local partitions will not.

Of all 64 permutations, I suggest that the following are the only versions of the distributed Logarithmic Algorithm that are not clearly incorrect or inefficient. The option chosen for a given relation is specified in the column under its name.

	D_R	R_Δ_Left	R_Δ_Right	Comments
1.	sp	sp	dr	Since R_Δ is replicated, no synchronization is necessary. It is easy to see that all tuples that should be generated by the first iteration are. A simple inductive proof demonstrates that all subsequent iterations are completely calculated (with a lot of redundancy).
2.	sp	sr	sr	This is just the serial algorithm.
3.	sp	dp	dr	R_Δ_Right must be replicated, since D_R is serial, and thus cannot be synchronized.
4.	dp	dp	dp	This is the only version that I will actually implement.
5.	dr	dr	sp	Since R_Δ_Right is serial, the others must

be replicated.

Version 2, the serial algorithm, is fine for a single machine. 1, 3 and 5 all involve gossiping. I expect that to be a prohibitively expensive operation. Thus I will only implement version 4. I hope that the foregoing theoretical analysis justifies my choice.

The fragmentation of the join input relations is traced in figures 2-6 through 2-9. Each iteration has two joins. Figure 2-6 depicts the situation before the first (each iteration).

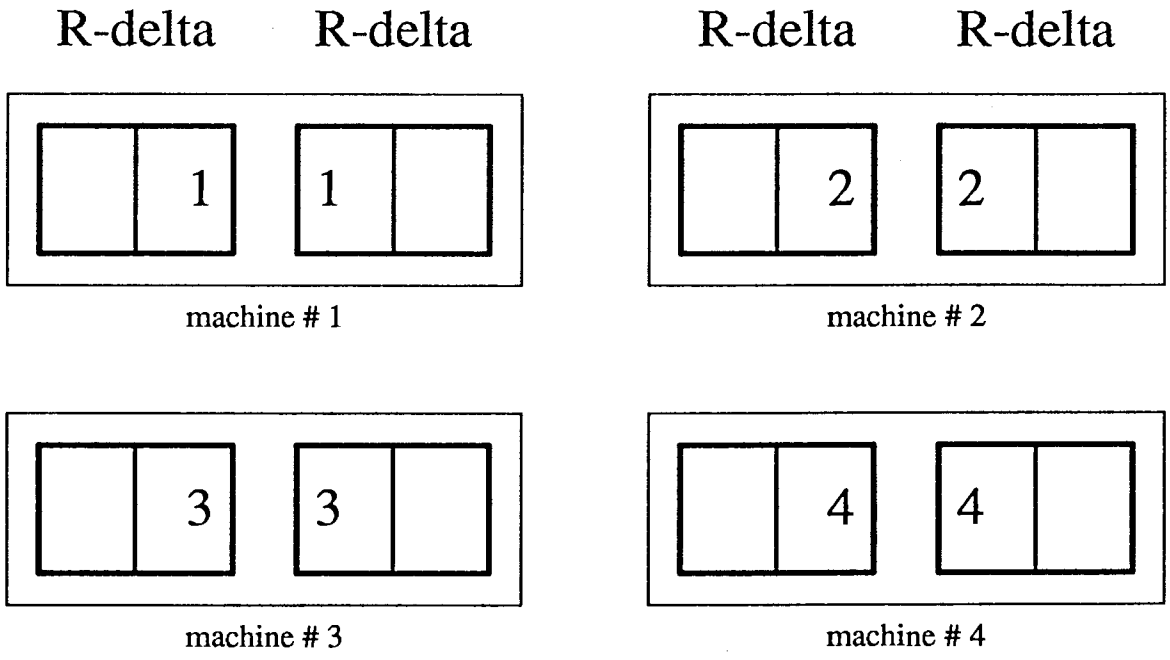

Figure 2-6: Logarithmic - before 1st join

Note that R_Δ is being joined with itself. Two copies of that relation are fragmented between the machines, one on each column. Thus the fragments resident on a given machine will in general *not* contain the same tuples as each other.

Figure 2-7 shows the situation after the first join. Neither join input is static. Thus both sides (ie. two copies of R_Δ) have to be exchanged between machines. Once this has been done, the right-hand input for the *second* join is now properly partitioned. The situation prior to the second join is shown in figure 2-8.

The output from the second join is D_R, which consists of *new* tuples to be added to T. R_Δ has already been exchanged. So now, we need only exchange D_R.

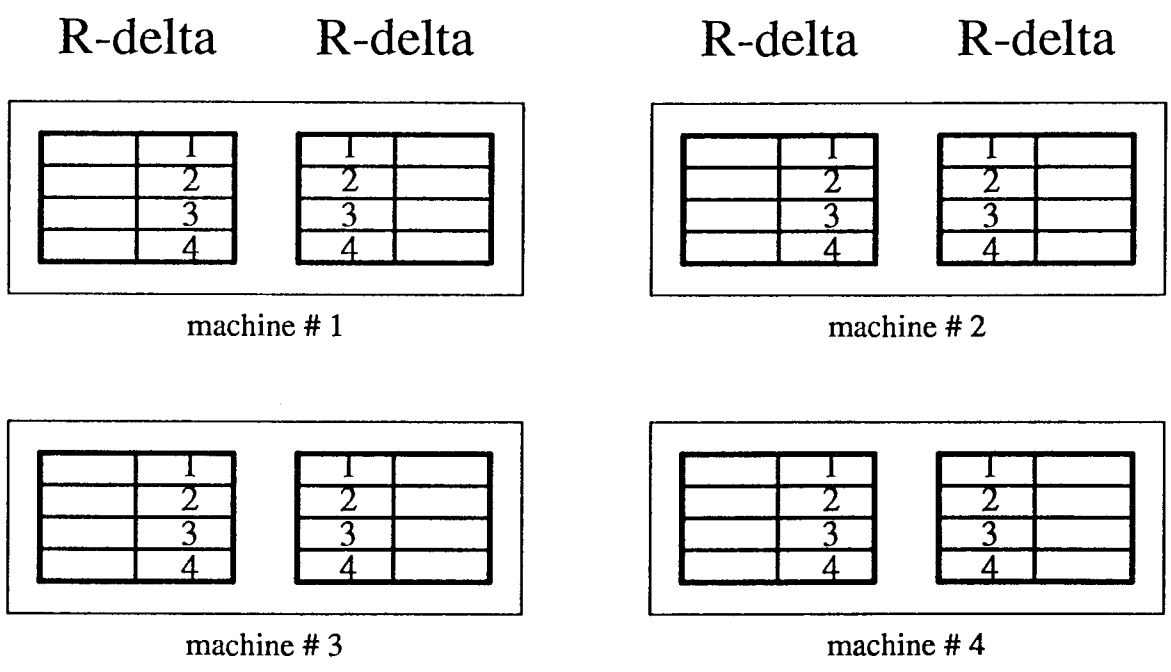

Figure 2-7: Logarithmic - after 1st join

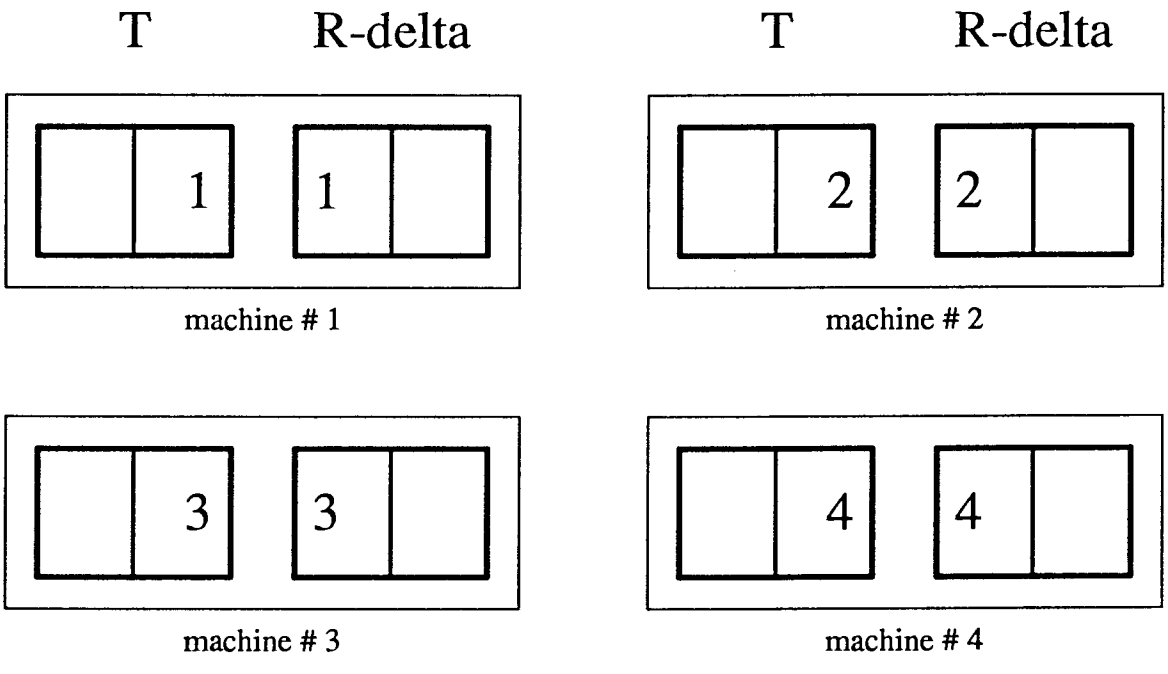

Figure 2-8: Logarithmic - before 2nd join

Since all join input relations are dynamic, and none are replicated, all of them have to be exchanged once per iteration.

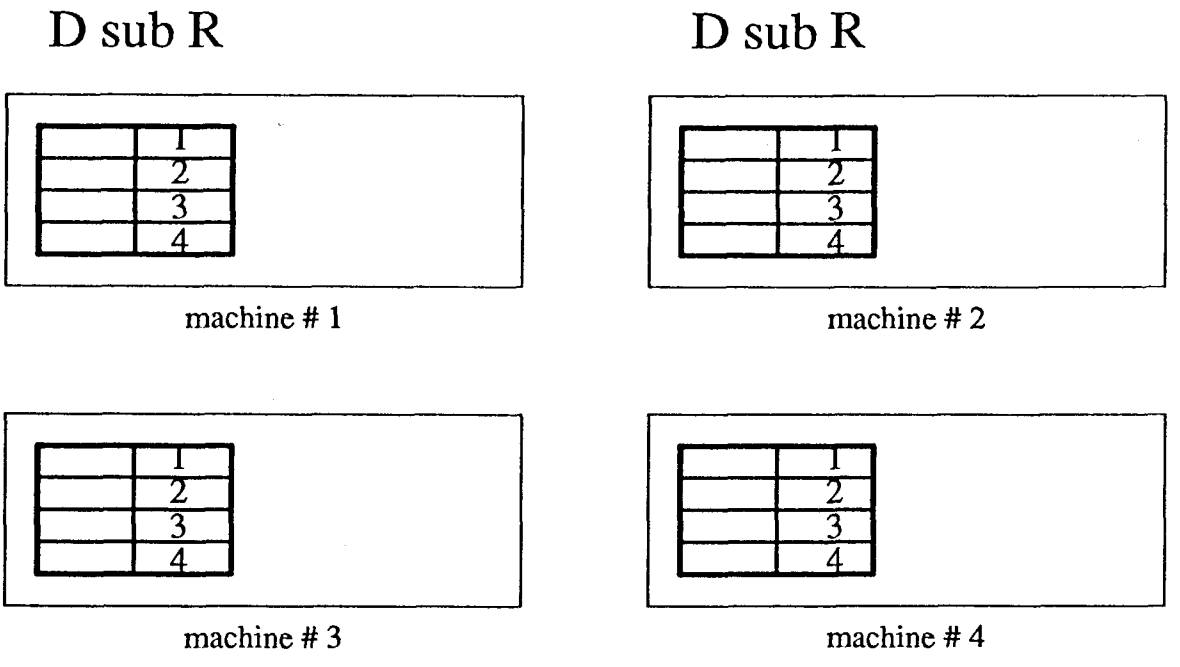

Figure 2-9: Logarithmic - after 2nd join

2.2.3.5. The Direct Algorithm

The Direct algorithm really uses only a single relation, the set of successor lists. (The predecessor list structure is a copy of the same relation, though it will be fragmented differently between the machines.) Thus there are only four alternative coarse-grained data distribution alternatives, depending on how the successor lists are distributed. The sr alternative is just the serial algorithm - dr would exacerbate communication problems by sending predecessor lists *as well as* the current successor list. The sp alternative is incorrect - successor lists *must* be broadcast. This leaves only the dp option, whereby the input relation is partitioned between machines, and part of it (namely the current successor list) exchanged during processing.

Figure 2-10 shows how the data fragmented between the machines.[xi] Note that the two fragments on each are *identical*. The only difference is in the method of storage - one copy is stored as successor lists, the other as predecessor lists. Since a machine will only generate new tuples that fall within its own partition, there is no need for the exchange of intermediate results. Of course, the current successor list must still be broadcast at the start of each iteration.

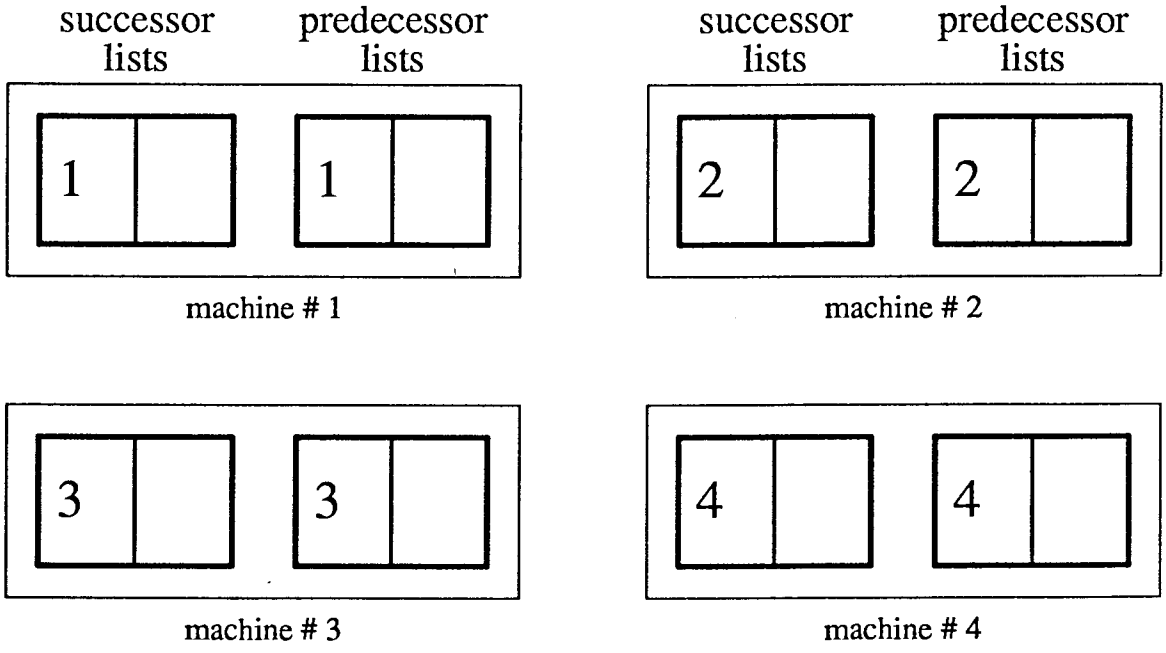

Figure 2-10: Direct - partitioning of successor and predecessor lists

[xi] This is best thought of as a partitioning of the input graph's vertices - actually just those with out-edges - between the participating machines.

Chapter 3
Implementation Issues

Each of the algorithms was simulated on a single machine. From these simulations, we were able to determine the following important quantities:

- the sizes of intermediate and result relations for the input relation and key domain sizes tested
- the number of duplicate tuples generated by each algorithm, and of these, the number that could be detected locally (by the machine generating them)
- the number of iterations each algorithm takes to complete

These quantities depend on statistical characteristics of the input data - the values obtained are "expected" values taken by averaging the results of ten runs for each sample set analyzed.

3.1. Communication Issues

We assume that communication is by means of the V-System's synchronous send and synchronous broadcast operations. A sender is blocked until it gets an acknowledgement from the receiver. Thus each data exchange step constitutes a synchronization point for all machines. With an asynchronous send, the sender can proceed with its other business as soon as it has transmitted. V supports asynchronous communication, through its datagram service. With the datagram service, however, the receiver can receive the message only if it is receive-blocked on it. [2] So if the message arrives while the receiver is busy processing, the message is lost. For all of our algorithms, that is intolerable. (We would need to implement an application-level protocol for retransmission, and a *large* number of retransmissions would be required.)

The V-System's blocking send can be used to implement a non-blocking one. Message sending is delegated to a server process. Although the server process is blocked, its client (the main process) can continue.

To use this method, we have to modify the Exchange step of each algorithm, and the Distribution step (3a) of the Direct algorithm. After a machine sends or broadcasts, it is allowed to continue processing before receiving acknowledgements. We must, of course, be careful not to overwrite the locations containing the

message before all acknowledgements of it *are* received. We must also enforce a retransmission policy - the sender retransmits if it fails to receive an acknowledgement after a timeout period.

In the analysis of chapter 4, we assume that the following method is for the exchange step:

> For p participating machines, $\lceil \log_2 p \rceil - 1$ iterations are used. In each iteration, each machine is paired off with a different partner. Thus once all iterations are finished, each machine has been paired with each other. An iteration consists of two steps. In the first step, one partner sends to the other. In the second step, their roles are reversed. Of course, the blocking send is used.

An asynchronous alternative method would require each machine to create a sender-process for each other machine. The main process would fire up each of its sending servers, then wait for incoming messages. Unlike the scheduled method above, a machine would not have to wait for a response from a *particular* partner to respond. By not imposing an order on the point-to-point transmissions, the total time required for the exchange will probably be decreased. This gain has to be weighed against the process creation and switching costs entailed.

While the asynchronous method is worth investigating, the total time saving is unlikely to make much difference to the *overall* performance of the algorithm. In deriving cost estimates for the exchange step, we have assumed that a sender *never* has to wait. Any performance difference between the two exchange methods will not, therefore, show up in our estimates.

The Local Wavefront algorithm communicates only at the beginning and end of the algorithm. At result collection, unblocking senders saves no time, since those machines are finished processing. Thus using the asynchronous send only allows the controller to start processing a bit sooner. The extra overhead required to spawn sender processes and switch control to them is hard to justify.

The Global Wavefront algorithm exchanges data once per iteration (step 2dii). Since the helper processes needed for the asynchronous send need only be created once, the relative advantage of that technique is better than in the Local algorithm (creation cost is amortized over more exchanges). A more significant gain might be had by allowing a machine to continue with *local processing* before it has received data from all the others. This amounts to another way to achieve level relaxation. So long as a machine has *any* drivers, it processes them. Late-arriving messages from iteration i can safely be processed during iteration i+1. I think that this method is well worth investigating. It cannot be used in the context of the Logarithmic algorithm, for the same reasons that level relaxation cannot (see chapter 2).

In the Direct algorithm, the broadcast originator never requires data from the other machines, at least until the next iteration. Thus allowing it to proceed with local processing before receiving acknowledgement is safe. Concommitant to this, though, is the fact that little advantage is gained from asynchronicity - only the sender benefits. If it gets very far ahead of the other machines, it will ultimately have to wait for them to catch up - it will eventually need data from them.

The V-System's broadcast unblocks the originator as soon as a *single* acknowledgement is received. To retrieve the other acknowledgements, the "GetReply()" system call must be used. This is expensive. The following scheme allows us to avoid that system call:

> The originator timestamps the message with the current iteration number. If a machine receives a message with an out-of-order timestamp, it knows that an intervening message has been lost. It broadcasts an alert, with the timestamp of the missing message. Broadcasting the alert ensures that the originator gets it. The originator can retrieve the appropriate successor list, and send it to the site which lost it. Since elements are never *deleted* from successor lists, the receiving a copy that is newer than the original will not lead to incorrect results. On first and last iterations of the algorithm, the broadcast originator should retrieve all acknowledgements.

Note that the broadcast *primitive* used is synchronous![i]This primitive is more expensive than is a point-to-point send, because the *operating system* keeps track of *all* acknowledgements, even if the application does not retrieve them with GetReply(). This matter is discussed in more detail in chapter 4, section 4.2.1.

3.2. Local Processing Issues

For the iterative algorithms, the most important implementation decision regarding local processing was the choice of join method. We settled on the hash-based join, for reasons discussed in subsection 1. Hash tables are also used for accumulating results and filtering out duplicates. In the Direct algorithm, both predecessor and successor lists are implemented as hash tables. The design of the hash tables is thus an important implementation concern. The other major data structure used is basically a stream-oriented implementation of a relation.

I will first discuss the join method, then the implementation of hash tables and of the stream-structured relation. With the amount of global state replicated locally, the Local Wavefront algorithm could actually detect more duplicates than our implementation of it does. Our reasons for not making use of all the state available are discussed at the end of this chapter.

---

[i]The datagram service is inappropriate, since it requires that the receivers by receive-blocked.

3.2.1. Join Method

Wang [17] [16] investigated sort-merge, hash-based and tree-based join algorithms. In the distributed iterative algorithms (Wavefront and Logarithmic), the join operation is not *itself* distributed. Rather, each participating machine does a serial join on its local input fragments. Though Wang studied *distributed* joins, his results are also applicable to the *serial* versions. The Direct algorithm's merging process can be interpreted as a distributed join. That interpretation is only useful for comparison with the iterative algorithms. In this section, implementation issues germane to the Direct algorithm are described and discussed as *merges*.

I now turn to the various join methods considered.

3.2.1.1. Sort-Merge

Wang's experimental results indicate that this method is the most efficient, if the input relations are already sorted. If not, the hash-based method is superior. [17] When a join is done during a TC computation, the join attributes are *both* projected away - ie. they are removed from the result relation. The keys on which the input relations were sorted are thus not even *in* the result. Thus the sort order is not preserved from one iteration of the TC algorithm to the next. If the sort-merge algorithm were used, the wavefront relation would have to be resorted at the beginning of each iteration, and the final result relation would still be unsorted. I conclude that this method is unsuitable for use in TC algorithms.

3.2.1.2. Tree-Based

If tree indices already exist for both input relations, the tree-based method is the most efficient. There are actually two such methods. One uses the indices to generate an inorder traversal of each relation then merge. The other is more like the hash-based method - loop through one relation, and look up matches in the other. [16] The first approach suffers from the same deficiencies as does the sort- merge method. The second (nested loop) approach might well be attractive in the serial case, since we would not have to build the hash table for the source relation. (Note that in the nested loop method, it is not essential that the driver relation - the wavefront - be traversed in sorted order.) As Wang points out, however, this advantage does not withstand distribution, since the indices for partitions of the source would have to be rebuilt on the load sharing machines. Consequently, this method turns out to be very similar to the hash-based one. Both methods will involve building the same number of structures, and performing the same number of lookups. The performance difference will be a function of the relative costs of building and using one data structure over another. Since I am not interested in investigating data structures for their own sake, I will not implement the tree-based method. I would in any case be surprised if it outperformed hashing.

3.2.1.3. Hash-Based

All of my algorithms use hash-based joins. The join algorithm is serial - ie. each machine does its joins locally. One advantage of hashing is that the hash function constitutes a good value on which to partition the data relatively evenly between machines.

3.2.2. Major Data Structures

The simulations used the same data structures that would be used in the actual algorithms.

A relation is stored in one of four structures. Three of these are hash table varieties. The other is what I will refer to as the "stream-structured relation". I will discuss the latter first, then the hash tables.

3.2.2.1. The Stream-Structured Relation

The relation is stored as an unsorted contiguous array of "tuples". A tuple is implemented as an array of two "tuple values". We used long integers (4 bytes) for tuple values. The array of tuples is allocated just *once* - it does not grow dynamically. This facilitates fast insertion, since no storage allocation is required. A header contains read and write pointers into the array of tuples, as well as a pointer to that array. The only operations supported on this structure are creation, insertion and deletion.

Even a main memory model should acknowledge *some* practical constraints on the amount of RAM available. I strove for a realistic implementation, ie. one that would be usable. The disadvantage to static allocation is that the size of the result must be estimated a priori. Since this structure is quite compact (it does not contain a lot of pointers), allocating enough storage for the *theoretical* limit on result size does not impose unrealistic demands on memory.

3.2.3. Hash Tables

All of our algorithms make use of hash tables. They are used in implementing the hash-based join and the union, and for filtering duplicates out of relations. In the Direct algorithm, they are used to store successor and predecessor lists. Each use involves particular operations on the table - the structure of the table is predicated on optimizing those operations.

Storage for hash table entries is allocated dynamically, using the UNIX "malloc()" function. This adds greatly to the cost of insertions. It is very difficult to estimate a priori the sizes of hash table chains. Static

storage allocation for them is therefore difficult to implement without making unrealistic demands on RAM. (It hardly seems reasonable to allocate enough for the worst for each chain!) Maintaining a heap in user space *would* be a realistic alternative, and would considerably improve performance. I doubt, though, that this would greatly change the *relative* performances of the algorithms.

3.2.3.1. Hash Table Operations

The operations of concern in this study are list retrieval, insertion, tuple lookup and insertion with duplicate detection.

List Retrieval

By list retrieval I mean, "Given the value of vertex v_i, retrieve all known successors (or predecessors) of v_i." This interpretation is most pertinent to the Direct algorithm, whose rationale is graph theoretic. In relational terms, we could say, "Given a value i for the join column of the stored relation, retrieve all tuples with a join column value of i ." This latter interpretation is perhaps more pertinent to the Wavefront and Logarithmic algorithms.

To expedite discussion, I introduce the terms "primary value" and "secondary value". The primary value is the value of the join column, the secondary value the value of the other column.

List retrieval is central to the hash-based join operation. For each driver tuple, we want to find *all* tuples in the table which match it. (The driver tuple's join column value is the primary value. It is through the primary value that we access the join and hybrid hash tables - types A and C. When we find its link, that link will contain a pointer to a structure holding the associated secondary values.) If the table is used to implement the join, then it clearly should support efficient list retrieval.

In the Direct algorithm, the two major data structures are both adjacency list representations of the graph (transitive closure) being generated. One represents that graph as a list of predecessor lists, the other as a list of successor lists. Finding predecessor lists and merging successor lists are frequent operations. (In this case, the vertex number is the primary value, and the lists are comprised of secondary values.) Both major structures must therefore support efficient list retrieval.

List Creation

A matter that the Direct algorithm has to address is, "How are successor and predecessor lists initialized?"

We can find the successor lists by sorting the source relation on its first column; the predecessor list can be grouped by sorting on the second column. This is expensive! The hash table implementation that we adopt removes the need for sorting. It does not matter in which order tuples are inserted into the A and C type tables. Successor (or predecessor) lists will be stored together. (We had to ensure this so that we could retrieve them.) That it removes the need for sorting strongly recommends the hash table implementation of this algorithm's merge operation.

Insertion

I will call a structure "static" if, once initialized, it need not be modified for the duration of the algorithm. In the Wavefront algorithms, the hash tables used for the join are static. All other tables used are "dynamic". They must either be entirely rebuilt each iteration, or they grow as new closure tuples are found. Obviously, efficient insertion is more important for dynamic tables than for static ones. As it turns out, the static tables used support very efficient insertion, since they do not have to support tuple lookup.

Tuple Lookup

By "tuple lookup", I mean the operation of either finding a tuple (i,j), given the values of i and j, or reporting that it is not in the table. This I contrast with just "lookup", which is what I call the operation of finding the secondary structure associated with a specified primary value.[ii] The latter operation is essential to list retrieval. But note that retrieving *a set of* tuples $(b,j_1),...(b,j_n)$ not tell us whether a *particular* tuple, say (b,c) is present.

The difficulty arises because the entities stored are *binary, not unary.* Random access on a *unary* value is easy to arrange. But there may be a huge number of tuples with the same join column value. Accessing them *as a group* may be easy - picking out one from among that group may be difficult.

The most efficient solution is to "fold" the primary and secondary values together. This provides a unary index for each binary entity, thereby making random access possible. (If the hash table chains get too long, we just decrease the loading factor by allocating more buckets.)[iii] Unfortunately, folding is not compatible

---

[ii] Given primary value b, "lookup" would return a pointer to the set of values j_k, $1 \le k \le n$ such that all tuples (b,j_k) are in the table.

[iii] When evaluating the cost of operations on the type B table, in chapter 4, I use the *lowest* cost, regardless of the expected number of entries in the table. The assumption behind this is that if performance deteriorates, this table type can always be "tuned" (ie. buckets can be added or removed) to bring the performance back to near optimum.

with efficient list retrieval, since the relationship between the two element values of a tuple is lost. We can no longer ensure that all elements with a common primary value get stored together.

If we want *both* list retrieval and tuple lookup, we need a "two-tiered" structure. We first look up the primary value. This gives us access to the group of associated secondary values. We then employ some search strategy to find the particular secondary value sought. The specific structure used is discussed under "Type C - Hybrid Table" below.

When is tuple lookup required? The iterative algorithms all use a set difference operation (though it is, in our implementations, done "on the fly", as part of the union). Set difference clearly requires tuple lookup on the subtracted relation - in this case the result relation. Tuple lookup is also a necessary ingredient in insertion with duplicate detection.

Insertion With Duplicate Detection

If we insert tuples into a table without first checking that they are not already stored there, the table can come to contain duplicates. Of course, if we can guarantee a priori that every tuple the algorithm inserts is unique, then we do not have to check for duplicates. In the Wavefront algorithms, the join tables are static. These are initialized with the tuples in the source relation (the input to the transitive closure operation). So as long as the source is duplicate-free, we need not worry about duplicates getting into the join tables.

Each algorithm uses a table to store its result relation(s). The source relation and all join outputs are unioned into this result table. A given tuple in the source relation may also be generated by some join; a tuple may be generated more than once. As a consequence, a substantial number of duplicates will be presented for insertion into the result table.

We filter out duplicate tuples as follows:

```
For each tuple (i,j) to be inserted
        - look up (i,j) in table
        - if not found
                insert (i,j)
```

There are at least two good reasons for filtering out duplicates. The first is that their presence increases the size of the table - this in turn leads to higher processing costs (eg. when communicating the relation).[iv]The

[iv]The extra processing cost from carrying duplicates would be especially (intolerably) high in the Direct algorithm, since that algorithm generates so many of them.

second, more important reason is that to be a relational operation, transitive closure *must output a relation.* Since duplicates must *eventually* be removed anyway, we might as well filter them out at insertion time.

Note that the extra cost of duplicate detection comes from the tuple lookup. In the iterative algorithms, duplicate detection is, as mentioned above, needed for the set difference operation. So filtered insertion comes at a low marginal cost.

3.2.3.2. The Tables' Structures

The type (A, B or C) of table implemented for a particular relation depends on which operations will commonly be done on that relation. Result tables should support insertion with duplicated detection, and join tables must support list retrieval. A table used for *both* the join and result collection must support *all* the operations described in the previous subsection.

Each Wavefront algorithm uses one static join table and one result table. For static join tables, neither tuple lookup nor filtered insertion is required. Thus only list retrieval is optimized for. Here we use the type A table. Neither result table is involved in a join, so we can optimize for tuple lookup and insertion with duplicate detection. Here we use the type B table.

The Logarithmic algorithm uses two joins per iteration. Each needs a join table. Since all join input relations are dynamic, both join tables must also be dynamic. This means that insertions should be efficient. The first join (step 2dii) takes as input a *single* relation R_{i-1}, and joins it with itself. Its output R_i in iteration i constitutes its input in iteration i+1. Two copies of that output are required for input to the next iteration's execution of the join (step 2dii). The join output may contain duplicates. Thus once the two copies of it (P_i *and* Q_i) are assembled at the machine in whose partition they fall, they must *both* be filtered. One copy has to be made into a join table. To accomodate both duplicate filtering and list retrieval, the type C table is used. This table is a compromise - all operations are supported with "tolerable" performance, none of them with "optimal" efficiency. The other copy, the driver, is filtered by insertion into a type B table, copying unique tuples to a new stream-structured relation. The type B table is used here because operations on it are cheaper. This table is used *only* for filtering.

For the sake of correctness alone, it is not necessary to filter the inputs to the first join. These are only intermediate relations - the result is accumulated in the table used for the *second* join. As long as *that* table is kept duplicate-free, the result will be correct. The decision to filter the first join's inputs came after implementing the simultation. *Not* filtering led to a huge proliferation of duplicates, severely impairing performance.

The Logarithmic algorithm's second join takes as input the result relation so far, and the output R_i from the first join. We must decide which to loop through, and which to build the table for. The time complexity of the hash-based join is a linear function of the size of the relation looped through. This suggests that we loop through the smaller relation, which is almost certainly R_i. If we build the join table on the result relation, though, we will need a type C table, since those operations germane to result collection must still be supported.

Alternatively, we could build the join table on R_i. (This would not be an "extra" table - the table used for the second join in iteration i could be reused for the first join in iteration i+1.) Which choice is better will depend on the nature of the data. In anticipation that building the join table on the result relation and looping through R_i will usually be more efficient, I have implemented the second join table and the result table as one, using type C.

In the Direct algorithm, a type C table used for the successor lists, since that structure must support all operations. Any duplicates generated are detected while merging into the successor lists. The corresponding predecessor is inserted only if it is not a duplicate. Thus the simpler type A table is used for predecessor lists. (The type B table is inappropriate, since list retrieval must be supported.)

Type A - Join Table

Tables of type A are useful *only* for implementing the join - they are not suitable for collecting results in. The structure is depicted in figure 3-1.

Hashing the specified join column takes us to a bucket, which is the header for a list of "primary links". Each of these contains a unique primary value, a pointer to the next primary link, and a pointer to a "secondary chain". Each "secondary link" contains a value which is unique *within its chain*, and a pointer to the next secondary link. Neither primary nor secondary chains are kept sorted. A tuple is inserted as follows:

- hash join column value into the correct bucket
- search primary chain for join column value of tuple to be inserted
- prepend secondary value of that tuple to head of its primary value's secondary chain

The type A table is designed to optimize the list retrieval operation. Insertion is inexpensive, since no attempt is made to keep the chains sorted. This table is the most space-efficient of the three, since the chains are only singly-linked. Tuple lookup is expensive, demanding a sequential search of a potentially very long secondary chain. But none of the algorithms entails searching in this structure.

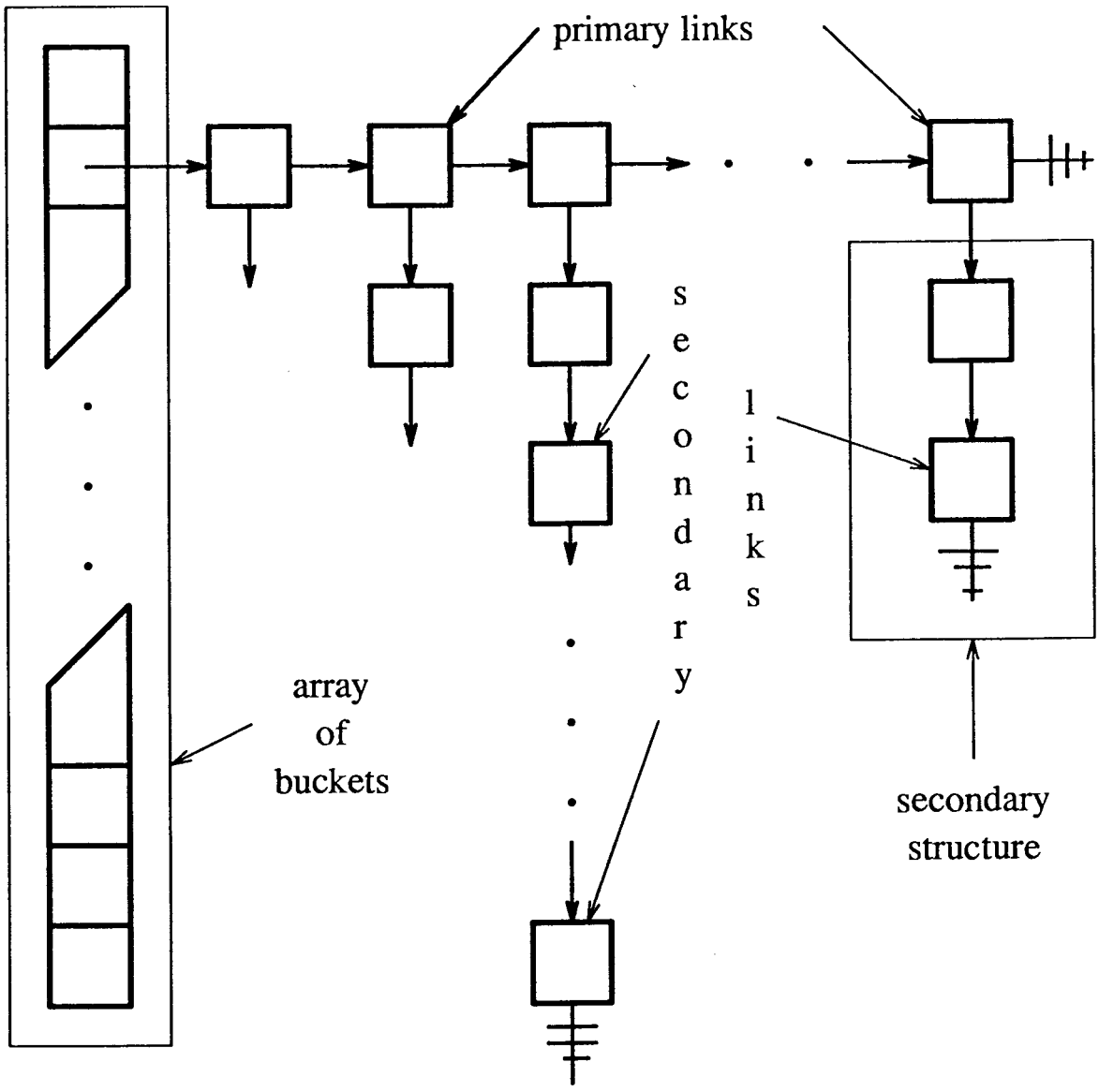

Figure 3-1: Join Hash Table - Type A

Type B Tables

Tables of type B are useful *only* for result collection - they are not suitable for implementing the join. The structure is depicted in figure 3-2.

Folding the tuple's column 1 and column 2 values together takes us to a bucket, which is the header for a

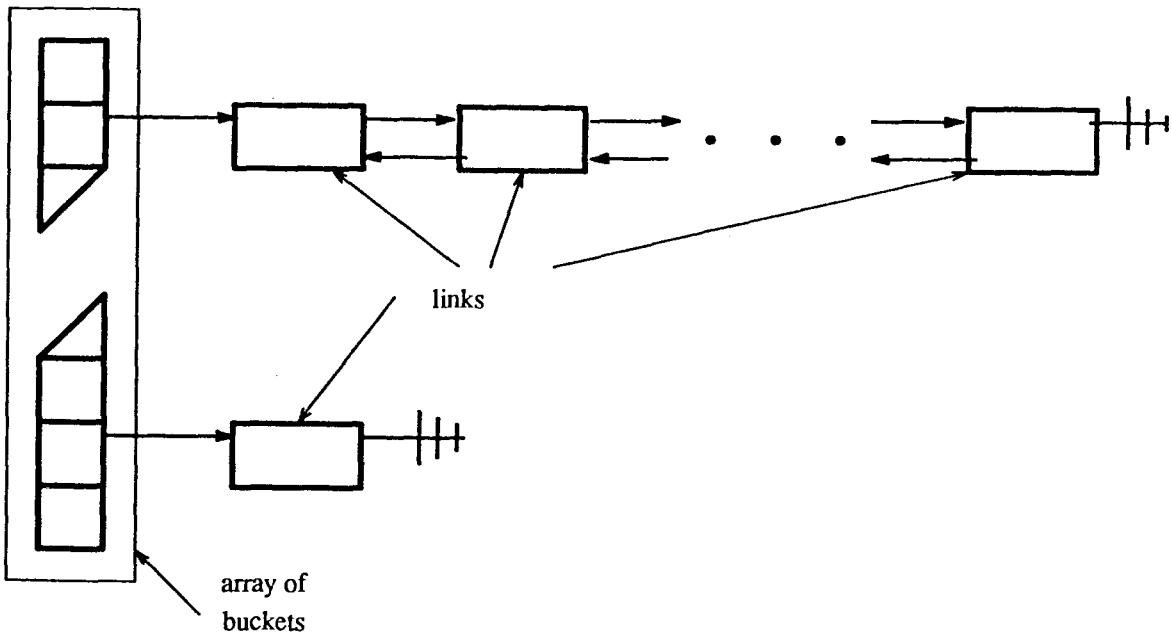

Figure 3-2: Result Hash Table - Type B

list of *tuple* links. Each link in this chain contains *both* column values for a tuple, a pointer to the next link, and a pointer to the previous link. Note that all that tuples in a given chain have in common is that their *folded* value is identical. The chains are doubly-linked, and kept in sorted order. There are no secondary structures hanging off of these tuple links.

Type B tables are designed to optimize tuple lookup and insertion with duplicate detection. Folding allows us to dispense with secondary chains. We can make tuple chains arbitrarily short by increasing the size of the hash (fold function) domain. List retrieval (finding all of a vertex's successors or predecessors) demands a sequential search of *the entire table*. None of the algorithms attempts a list retrieval on this table type.

Type C Tables

Tables of type C can be used for both the join and result collection, though they are less than optimal for either. The structure is depicted in figure 3-3.

The main table (array of buckets) and primary chains are identical to those for the type A table. But now

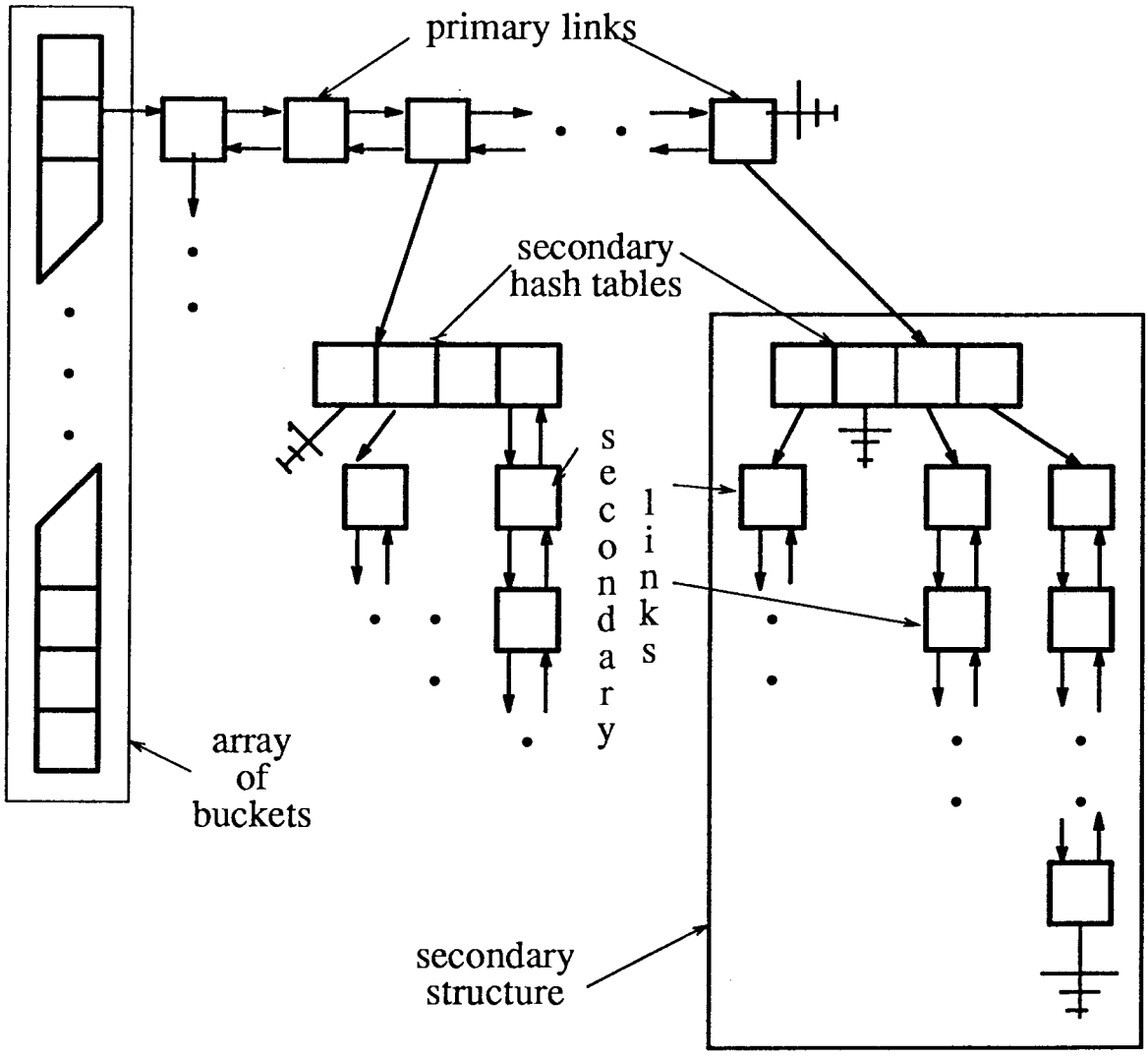

Figure 3-3: Hybrid Hash Table - Type C

the secondary values for each primary value are themselves stored in a hash table. There is one such "secondary hash table" for each distinct primary value in the table. This involves considerable space overhead.

For close-to-random access, we want a hash table to have about as many buckets as there are values stored in it. We could achieve a good loading factor for the *main* table by allocating, say, one bucket per two tuples expected to be in the relation. How big, then, should the *secondary* tables' hash domains be? If the relation's graph consists of a connected component, then some vertices' successor lists will include every other vertex! (Of course the average successor list will surely be smaller than the number of vertices in the graph.) To achieve rapid access into such list, we would want as many buckets as the main table has.

Large secondary tables cause some problems, however. They consume an amount of space that, for large relations, may be intolerable even for a "main memory" model.[v]The list retrieval operation will involve retrieving the *entire* contents of the secondary table. If we use a lot of buckets, many of them will be empty. Traversing empty buckets will be expensive. We might chain non-empty ones together, but this will make the table even more complicated, and will require yet more space.

Even a very small secondary table (say 10 or less buckets) will markedly improve tuple lookup. With 10 buckets, we cut the average chain length to $\frac{1}{10}$ its length in the type A table. We chose to use even smaller secondary tables - each has just 4 buckets. By keeping secondary chains (of which there is one per bucket in each secondary hash table) sorted, we can again cut average search length in half. Both primary and secondary chains doubly linked and sorted.

The type C table supports tuple lookup and insertion with duplicate detection tolerably well. List retrieval involves traversing a small number of short secondary chains instead of one long one - the cost will be very slightly larger.

3.3. Duplicate Detection

In the Global Wavefront and Direct algorithms, all duplicates generated can by detected and removed locally. For these algorithms, the implementation of duplicate detection is straightforward.

Because the machines do not exchange intermediate results, the Local Wavefront algorithm cannot detect all duplicates locally. This duplicate problem can be alleviated, if we take advantage of the fact that the entire source relation is replicated on each machine. Duplicates are filtered out by looking them up in the result table. If we install all of the source into each machine's result table, any tuple generated that is also in the source can be detected locally.

Note, though, that this "optimization" is not exactly free. Firstly, it increases table setup costs. Secondly, it increases the cost of result collection. For p machines, p-1 copies of the source end up being sent, in sequence, to the controller. Then all of those copies are run through the merge operation, which removes them.

---

[v]If we want m buckets for the main table, we'd need m^2 more for the secondary tables. m could easily grow into the thousands.

The alternative is to initially insert only the *local fragment* of the source relation into the result table. With this method, the question is, "How many duplicates that *could* be detected will go undetected?" The answer is, "Not many." We simulated the algorithm using both alternatives. It turned out for all sample sets tested, it was cheaper to suffer the presence of a few more duplicates, and include only the local fragment of the source relation in the local result. In Appendix B, "Cardinalities", I tabulate the global result relation size, as well as the number of duplicates detected. The values shown assume that each machine initializes its local result relation just to its own fragment of the source relation.

In the Logarithmic algorithm, it is not necessary for correctness' sake to remove internal duplicates from the join driver relation, A^{2^i}. Initially, our implementation did not do so. As it transpired, however, this resulted in so many duplicates' being stored that the capacity of our test machine was exceeded. Each duplicate *not* removed from A^{2^i} can cause the generation of *more* duplicates when input to the join in iteration $i+1$! Especially when cycles are present, this causes an explosion in their number, with attendant severe degradation of performance. Accordingly, the algorithm was implemented so as to filter all relations. Each intermediate relation fragment exchanged is filtered *after* it has been collected at its host.

Chapter 4
Analysis

In this chapter, the running times of the four algorithms are estimated, and their cost characteristics compared. The first section introduces some terminology and offers motivation for the approach taken. The second section includes the actual step- by-step analysis of each of the algorithms. Variables (eg. unit costs and relation names) used in the analysis are defined in that section. Together, the first two sections constitute the framework for the empirical study. In the third section, I explain how experimental results are applied to the analysis of section 2, to obtain total cost estimates for each of the algorithms. I discuss the empirical methods by which values for unit costs and relation sizes were obtained. The results are presented in the fourth section. Local processing, communication and combined costs are presented for each algorithm. The combined results are compared with the cost of the serial Delta-Wavefront algorithm. These results are interpreted, and reasons for performance differences between the algorithms are discussed. Speedup over the serial implementations, scaling to more machines, and variance in algorithms "cardinalities" are also discussed.

4.1. Terminology and Motivation

The basic unit of data is the *tuple*, which is assumed to be an ordered pair of key values. (Our test data consist of 8-byte tuples.) The molecule of data is the *relation*. This is the level of granularity at which the algorithms are defined (see appendix) and at which the analysis is done. Unit costs are per tuple, allowing us to take into account the size of the relation being operated on.

The analysis refers to "source", "intermediate" and "result" relations and subrelations. The source and result are the input and output, respectively, of the transitive closure (TC) operation. Intermediate relations are either partial results, or temporary relations used in generating the TC.

Clearly, complexity will be dominated by the sizes of the intermediate and result relations. An important methodological concern is to describe the sizes of those relations in terms of the input (source) size. Asymptotically, the size of the transitive closure of a relation R is $\in O(|R|^2)$. That bound is achieved when the TC of R is a complete graph.

The engine of all the algorithms is the join - it is this operation which generates new tuples. (The Direct algorithm is actually implemented as a series of list merge operations, but these can be interpreted as a series of joins.) The semantics of the join are probably best captured by a pair of loops, one nested within the other. If relations A and B are joined, then, the output relation size will be i ($|A|\,|B|$), $i \leq 1$. The scalar i captures the effects of statistical properties of the input relations. We decompose i into α, σ and γ, which stand for "join selectivity", "difference selectivity" and "internal difference selectivity", respectively. No doubt a statistical or combinatorial investigation could predict expected values for α, σ and γ, with only the source relation size, key domain size and data distribution characteristics as givens. The probabilistic or combinatorial research required to make those predictions is, however, decidedly non-trivial. We have not taken the a priori approach - our estimates of expected intermediate and result relation sizes are taken from empirical tests. Join and difference selectivity are introduced as descriptive, not predictive, notational devices. In fact, they are never actually evaluated. The use of those terms does, however, shed some light on the growth dynamics of the intermediate relations.

4.1.1. Join Selectivity

Join selectivity has been defined as follows:

$$\alpha = \frac{|A \textit{ join } B|}{|A|\,|B|}\,|B| \text{) [13]}$$

Thus, given input relations A and B, we estimate the size of their join to be $\alpha|A|\,|B|$. In the Wavefront algorithms, relation B, the source, remains the same throughout. Only A, the wavefront relation, changes in size. Thus it might seem more enlightening to define join selectivity simply as:

$$\alpha = \frac{|A \textit{ join } B|}{|A|}.$$

However, this would make any comparison with the Logarithmic and Direct algorithms more obscure. In the Log algorithm, there are two joins per iteration. For *each* of these, the sizes of *both* input relations vary over time. If the Log method's intermediate relations grow faster, this is not so much due to a different join selectivity, as to the different sizes of its joins' input relations. In the Direct algorithm, each iteration joins a fragment of the "current" closure (the graph generated so far) with the entire current closure. Again, the join input relations are both dynamic.

let p_i = the probability that a given tuple a from relation A matches *exactly* i tuples $b_1, b_2, \ldots, b_i$ in relation B.

If all tuple values are equally likely to occur in A (ie. the data are uniformly distributed), then we would expect that:

$$|A \textit{ join } B| = |A| \sum_{i=1}^{|B|} p_i$$

$$\text{so } \alpha = \frac{\sum_{i=1}^{|B|} i p_i}{|B|}$$

Unfortunately, estimating values for the p_i's is not as easy as it might seem. It *does*, however, seem safe to say that p_i depends on:

- the sizes of A and B
- the domain size of the join column
- the distribution of values within the join column

As the size of the input relations approaches the join domain size, the probability of matches increases, and thus so does α. In all four algorithms, the join input sizes will vary from iteration to iteration, while the domain size remains constant. Thus we would expect join selectivity to vary over time. Accordingly, α is subscripted, so that a different α exists for every iteration. Similarly, the typical input size at a given iteration will differ from one algorithm to another. To reflect this, α is superscripted, so that a different α exists for each algorithm.

Another difference between algorithms has to do with the distribution of join column values. If two *different* relations are being joined, using a uniform random number generator will guarantee uniform tuple distribution. But consider what happens when we join a relation with itself, as in the Log algorithm. If each tuple is unique within its relation, then the join column of the left copy will not be independent of the join column of the right copy. For instance, if $|A| \geq 2$, then $2 \leq |A \textit{ join } A| \leq 2(|A|/2)^2$, regardless of what particular tuples occur in A. But if A and B are generated independently, $0 \leq |A \textit{ join } B| \leq |A|\,|B|$. This consideration again militates for the use of separate variables for the different algorithms.

4.1.2. Duplicates

The "duplicate problem", and how it is addressed, is the most important factor affecting the algorithms' relative efficiency. All four algorithms are assumed to take the same input and, of course, produce the same output. Their expected complexities vary a great deal, mainly because some generate more data than others. The "extra" data either consist of duplicates, or are kept in order to facilitate their detection and removal.

I distinguish between two kinds of duplicates, "internal" and "external". If a tuple occurs more than once *within* a single relation, all copies but one are internal duplicates. By a single relation, I mean a *logical* relation, which may be horizontally fragmented between machines. This I call the "global relation"; each *local* fragment of it is a "local relation". Similarly, a single *logical* operation (eg. join) may in fact be implemented as a number of concurrent operations, each on different fragments of the input relations. Our

implementations ensure that, taken as *separate* relations, the local join input fragments do not contain internal duplicates. We also assume that the source relation, whose transitive closure we are trying to compute, is duplicate-free.

Any relational operator should produce as output a relation. By definition, a relation contains no internal duplicates. A join cannot be *prevented* from generating internal duplicates. Relational purity thus demands their removal. In practice, it is sometimes more efficient to delay duplicate removal until a later relational operation (such as union) is applied to the join output.

While the "no internal duplicates" constraint can always be enforced on *local* relations (or fragments), it is sometimes impossible to maintain with respect to *global* ones. The reason is that it is possible that no machine has enough global state information to allow it to detect internal duplicates resident on *other* machines.

An "external duplicate" is any tuple in the driver relation which is already in the result relation (closure). This category is only of interest in the context of the Wavefront algorithms. Our implementations of the Wavefront algorithms removes internal duplicates from the join output during the same operation (Union) that removes external ones. In the Logarithmic algorithm, removal of external duplicates can lead to incomplete results. In the Direct algorithm, all duplicates are internal to the current result relation - these are filtered out by the merge procedure.

4.1.3. Difference Selectivity

Difference selectivity, represented by σ, relates the size of a relation before duplicates (both internal and external) are removed to its size after. Consider the Wavefront algorithms, and suppose that at iteration i, the join produces a new wavefront relation W_i. Let C_i be the union of all W_j's, $0 \leq j < i$, and W_i' be W_i with external duplicates removed. Then $W_i' = W_i - C_i$ (where the "-" operator represents relational difference). Now we define:

$$\sigma = \frac{|W_i'|}{|W_i|}$$

The higher the percentage of duplicates found, the smaller W_i' will be. We therefore expect σ to *decrease* with time, since $|C_i|$ is monotonically non-decreasing in i. This *could* be reflected by including $|C_i|$ in the definition of σ. In the interest of removing notational clutter, I have decided instead to individuate the difference selectivity for each iteration by subscripting σ (as was done with α).

As mentioned in the preceding subsection, "Duplicates", the concept of "external" duplicates is germane only to the Wavefront algorithms. "Difference selectivity" (σ), since it applies to external (as well as internal) duplicates, is thus not applicable to the Logarithmic or Direct algorithms. The Logarithmic algorithm cannot be guaranteed complete if external duplicates are removed. In the Direct algorithm, the merge operation in effect joins the current result relation with itself. All duplicates generated are *internal.* (These are filtered out as part of the merging process.) For the analysis of the Logarithmic and Direct algorithms, I introduce "internal difference selectivity", γ. This is defined in the same way as σ, except that only *internal* duplicates are removed.

In the Wavefront algorithm, at the start of each iteration, all tuples in a given fragment are gathered together on the same machine - this machine can detect *all* duplicates (both kinds) in the *logical* relation. Thus, in the context of the Global algorithm, "removing detected duplicates" means "removing *all* duplicates". In the Direct algorithm as well, all duplicates can be detected. The (current) result relation is in the form of a list of successor lists. Each vertex $v(i)$ in the graph of the source relation is resident on *exactly one* machine.[i] At the start of iteration i, that machine broadcasts v_i's successor list to all other machines. Any predecessor v_j of $v(i)$ resides on *exactly one* machine (all tuples with j as their column 1 value are on a single machine). Thus all duplicates produced by merging v_i's successor list into any particular $v(j)$'s will be generated on the same machine, and can be detected and removed on that machine during the merge.

In the Local Wavefront algorithm, machines do *not* communicate with each other on each iteration. The effect is a distributed equivalent of level relaxation - as the algorithm progresses, a given machine's local data can come to include a lot of tuples that are not in its own fragment. Meanwhile, other machines may be generating tuples that *are*. If a tuple occurs twice *locally*, it will be detected. But if two or more different machines produce the same tuple, none of them will know about it. In the context of the Local algorithm, the set of detected duplicates is very likely to be a *proper subset* of the set of duplicates. Thus we expect its σ values to be larger than those for the Global algorithm.

Like α, we individuate σ and γ by algorithm with a superscript.

---

[i] When I say that a vertex is resident on some machine, I mean that its *successor list* is there resident. A vertex's number is the *header value* for its associated successor list.

4.1.4. Broadcast Scalar

On a LAN, a broadcast originator only needs to send a single message. Receivers can take that message off the Ethernet in parallel. In this study, we assume that both the send and the broadcast are blocking. [ii] Although it unblocks the originator after the first acknowledgement is received, the V-System's synchronous broadcast collects acknowledgements from *all* receivers involved. (The user process can extract the extra ones with the "GetReply()" system call.) Because of the extra acknowledgements involved, the synchronous broadcast is inevitably more expensive than a point-to-point send. *How much* longer is a function of the number of receivers broadcast to. In the analysis, broadcast costs are related to send costs (both in tuples) by multiplying the former by a *broadcast scalar*. This I denote by "b_p", where p is the number of machines, and b a system-dependent constant. Actually, the *exact* relationship between point-to-point and broadcast costs cannot be captured simply by multiplying by a scalar. In the step-by-step analysis, however, I wanted to avoid to fine-grained an approach. In compiling cost results, a more accurate method of estimation is used (see section 3, "Experimental Methodology").

4.1.5. Parallelism

We assume that if one machine takes longer to complete a phase than does another, this is due to one's doing more work than the other.[iii] Since senders block, each send or broadcast constitutes a synchronization point. I will refer to the time between two synchronization points as a "phase". No machine can go to phase i+1 until all others have finished phase i. In other words, they proceed in lockstep (although the Local Wavefront algorithm's single step is a very long one).

In the step-by-step analysis (section 2), I have divided *total* cost of concurrent operations by the number of machines participating. This is really only justified if the load is distributed absolutely uniformly. Even though we assume uniform input data, we can hardly expect there to be *no* variance between loads from one machine to another. The more unevenly the load is distributed between machines, the longer a phase will take. I have made no attempt to quantify the effects of uneven load distribution. Further observations concerning those effects are offered in section 4.

[ii] For a discussion of asynchronous options, see chapter 3, "Implementation Issues."

[iii] For analytical purposes, we want to rule out possible causes which are independent of the algorithms under study. We do not, for example, allow for the possibility that the machines have different cpu's, or that one is being used by five other users while the rest are dedicated solely to this computation.

Local processing can obviously proceed in parallel *within* a phase. It is only when a synchronization point is reached that someone must wait. So the fewer synchronization points, the more parallelism. In order of increasing number of synchronization points, the algorithms can be listed as follows: Local Wavefront, Logarithmic, Global Wavefront, Direct. Thus the Local Wavefront algorithm allows the greatest parallelism, while the Direct algorithm allows the least.

In accounting for communication costs, I have also assumed linear speedup when it is theoretically possible (eg. during exchange steps). In fact, most "communication" time is really spent doing the local processing germane to transmission, such as copying buffers contents. This surely *can* proceed in parallel. But of course ethernet collisions *will* occur, necessitating retransmission of messages. I assume that collisions are not frequent enough to contribute significantly to running time. I also assume that machines have sufficient buffer space to prevent message over-run. This assumption is appropriate for a main memory model, and is realistic for Sun-3 workstations. memory model.

4.2. The step-by-step analysis

I begin this section with a discussion of what constitute the units of communication and local processing. I then introduce that notation which is general to all of the algorithms. The specific algorithm analyses follow. Notation particular to an algorithm is introduced at the head of the analysis of that algorithm.

4.2.1. Communication Costs

Communication is analyzed in terms both of tuples transmitted and of the number of messages sent. Communication cost per tuple decreases markedly as the message size grows. It is important that the number of messages sent be quantified, so that message sizes can be estimated. Broadcast costs are multiplied by a scalar (b_p) greater than 1, to reflect the cost of processing extra acknowledgements.

4.2.2. Basic Local Processing Operations

A relation is stored either in stream form, or in one of three varieties (A, B or C) of hash table. Operations on tuples, then, involve searching for, inserting or retrieving them from those structures. The analysis of local processing is defined in terms of the following basic operations:

PART — Partition. A tuple is hashed on the specified column value, and copied to a subrelation holding the appropriate horizontal fragment of the relation being partitioned (the input relation). Note that PART includes the cost of copying the tuple, as well as computing the hash value. It may also include the cost of retrieving the tuple from the input relation.

COPY	Insert a tuple into a stream-form relation. The tuple may be copied *from* any of the aforementioned structures, in which case COPY includes the cost of retrieval. Alternatively, the tuple may be already available. (It may have been retrieved as part of a previous operation.)
JTL	Join table lookup. Look up a key value in a hash table of type A or C. Includes the cost of retrieving the tuple from a stream-form relation.
JTI	Join table insertion. Insert a tuple into a hash table of type A. Includes the cost of retrieving the tuple from a stream-form relation.
RTI	Result table insertion. Insert a tuple into a hash table of type B. Includes the cost of retrieving the tuple from a stream-form relation.
JTID	Join table insertion with duplicate detection. Insert a tuple into a hash table of type C. The JTID function informs its invoker whether or not the tuple to be inserted is already in the table. If so, the duplicate is not inserted. Includes the cost of retrieving the tuple from a stream-form relation.

4.2.3. Notation common to all algorithms

A	source relation
C	result relation (closure)
C_i	intermediate result accumulated after iteration *i*
p	number of machines used
r	number of iterations to complete (Wavefront algorithms only) = length of longest path in source relation

4.2.4. The Global Wavefront Algorithm

terms specific to the Global Wavefront algorithm:

α_i^A : join selectivity at iteration i

σ_i^A : difference selectivity at iteration i

G_i : output relation of join at iteration i

G_i' : G_i with duplicates removed

$G_i' = G_i - C_i$

$|G_i| = \alpha_i^A |G_{i-1}'| |A|$

$G_0 = G_0' = A$

$|G_i'| = \sigma_i^A |G_i|$

$|C| = \sum_{i=0}^{r} |G_i'| = (\sum_{i=1}^{r} (\prod_{j=1}^{i} \sigma_j^A \alpha_j^A) |A|^{i+1}) + |A|$

Timing:

For each major step in the algorithm (see Section ?), the total cost of that step (ie. cost of one execution of it times the number of times it gets executed) is described. Steps 4, 5, 6 and 7 are executed once per iteration, while the others are executed exactly once.

1. **Distribution :**

$(p-1)\frac{|A|}{p}$ tuples,

$p-1$ messages.

Since only the coordinator has the source relation, and each machine gets a different fragment of it, the work of distributing it cannot be shared.

2a. **Partition :**

$2\frac{|A|}{p}$ partitioning

The factor of 2 is present because each machine takes as input *one* fragment of the source, and

produces *two* partitioned copies of it. When a cost is divided by p, this is because each of p machines works concurrently on $1/p^{th}$ of a global relation.

2b. **Exchange :**

$$4(p-1)\frac{|A|}{p^2} \text{ tuples.}$$

$$4(p-1) \text{ messages.}$$

We assume only a simplex connection. Thus for two machines to exchange messages with each other involves two calls, with each machine sending once and receiving once, consecutively. Each machine exchanges with $p-1$ others. Thus at least $2(p-1)\frac{|A|}{p}$ communication cost is incurred to exchange a single relation. Here, each machine is sending two copies - hence the 4. We could get away with only $2(p-1)$ longer messages, by sending both fragments destined for a given machine together. This, however, would involve expensive fragmentation and reassembly.

Since the source relation (A) contains no internal duplicates, the unioning can be done just by concatenation - this contributes negligible cost.

2c. **Build Local Hash Tables :**

$$\frac{|A|}{p} \text{ JTI}$$

$$\frac{|A|}{p} \text{ RTI}$$

2di. **Join and Partition :**

$$\frac{\sum_{i=0}^{r-1} |G_i'|}{p} = \frac{|C|}{p} \text{ JTL}$$

The preceding formula describes the size of the driver input relation to each join done by the algorithm. Note that $|G_r'| = 0$.

$$+\frac{\sum_{i=1}^{r} |G_i|}{p} \text{ partitioning}$$

This latter formula sums up the join *output* sizes, before external duplicates have been removed. Duplicates are not removed until the Union (step iii). That is the first time at which *all* of them can be detected.

2dii. **Exchange :**

$$\frac{2(p-1)}{p^2}\sum_{i=1}^{r} |G_i| \text{ tuples}$$

During each iteration, the join output (partitioning from step i) is redistributed. The expected size of the fragment one machine will send to another is $\frac{1}{p^2}|G_j|$.

$2r\,(p-1)$ messages.

Unioning the Q_{ij}'s does not add any cost. At this stage, they are simply concatenated. Then internal duplicates are removed along with external ones, as Q_i is unioned into C_i (step iii).

2diii. **Union :**

$$\frac{\sum_{i=1}^{r} |G_i|}{p} \text{ RTI}$$

$$+\frac{\sum_{i=1}^{r} |G_i'|}{p} = \frac{1}{p}(|C| - |A|) \text{ COPY}$$

Partitioning of new drivers is also done "on the fly" in the course of Unioning. If a join output tuple is *not* found in the result (closure) table, it is written into G_i'.

3. **Result Collection :**

$$\frac{|C|}{p} \text{ COPY}$$

At this point, the local fragment of C is in the form of a hash table. For the sake of efficient transmission, its contents are first copied into contiguous storage.

$$\frac{p-1}{p} \sum_{i=0}^{r} |G_i'|$$

$$= \frac{p-1}{p} |C| \text{ tuples}$$

The coordinator receives (consecutively) $p-1$ fragments of C, each of expected size $\frac{|C|}{p}$. Since the fragments are disjoint, result merging is unnecessary; concatenation suffices.

$p-1$ messages.

4.2.5. The Local Wavefront Algorithm

terms specific to the Local Wavefront algorithm:

α_i^B : join selectivity at iteration i

σ_i^B : difference selectivity at iteration i

C^B : result relation - closure $\cup$ undetected duplicates

L_i' : L_i with duplicates removed

$|L_i| = \alpha_i^B |L_{i-1}'|$

$$L_0 = L_0' = A$$

$$|L_i'| = \sigma_i^B |L_i|$$

$$|C^B| = \sum_{i=0}^{r} |L_i'| \geq |C|$$

Note that the last expression is an *inequality*. Since not all duplicates can be detected, the sum of the sizes of the local result relations will typically be greater than the size of the logical, *global* result relation.

Timing:

Step 5 is repeated r times; the formulae reflect this. Each other step is performed just once.

1. **Distribution :**

   $|A|$ tuples

   1 message

   Since *every* machine gets a copy of the *same* relation, it can be broadcast. For large messages and few machines, the cost due to the extra acknowledgements is insignificant compared with the cost of transmitting the message itself. That extra cost is even less significant when we consider that just one message is involved. I have therefore not applied the broadcast scalar.

2a. **Partition :**

   $\frac{|A|}{p}$ partitioning

2b. **Exchange :**

   $2(p-1)\frac{|A|}{p^2}$ tuples

   $2(p-1)$ messages

   Only a single copy of the relation is exchanged.

   Since the source relation (A) contains no internal duplicates, the $P_{i,j}$'s can just be concatenated, at negligible cost.

2c. **Build Local Hash Tables :**

   $|A|$ JTI

   $\frac{|A|}{p}$ RTI

   A join table for the entire source relation must be built on each machine. Otherwise, the algorithm cannot be guaranteed complete.

Only the local fragment of A is inserted into the result table, C_i. If *all* of A were inserted into C_i, the algorithm could detect and hence remove more duplicates. But this would also entail adding $\frac{p-1}{p}$ tuples to each C_i at the outset. Experiment indicates that this added cost outweighs the benefit of the enhanced duplicate detection.

2d. **Local Closure :**

$$\frac{\sum_{i=0}^{r-1} |L_i'|}{p} = \frac{|C^B|}{p} \text{ JTL}$$

This is the sum of the costs of looping through the join driver relation for each iteration, matching each of its tuples into the join table. Note that $|L_r'| = 0$.

$$+ \frac{\sum_{i=1}^{r} |L_i|}{p} \text{ RTI}$$

$$+ \frac{\sum_{i=1}^{r} |L_i'|}{p} = \frac{1}{p}(|C^B| - |A|) \text{ copying}$$

New tuples output by the join at iteration i are copied into the wavefront relation δA_i for iteration i+1. As in the Global algorithm, internal duplicates are filtered out along with the external ones, during the Union operation.

3. **Result Collection :**

Each machine has part of the global closure, but these fragments cannot be assumed disjoint. Therefore the local results cannot be just concatenated - the controller machine has to be merge them. This is done by inserting them into a type B hash table. Once merging is complete, the table contents must be copied into a stream-form relation structure. (The TC algorithm takes its input in that form - the output form should conform.)

local processing:

$$\frac{|C^B|}{p}$$

$$+ \frac{p-1}{p}|C| \text{ COPY}$$

First, each machine prepares its local results for transmission, by copying them into stream form. The second term gives the cost of converting the merged results to stream form.

$$\frac{p-1}{p}|C^B| \text{ RTI}$$

communication:

$$\frac{p-1}{p}|C^B| \text{ tuples}$$

p -- 1 messages

The controller receives the incoming local results in sequence. Hence no parallelism is possible.

4.2.6. The Logarithmic Algorithm

terms specific to the Logarithmic algorithm:

α_i^{C1} : join selectivity of the first join at iteration i

γ_i^{C1} : internal difference selectivity of the first join at iteration i

α_i^{C2} : join selectivity of the second join at iteration i

γ_i^{C2} : internal difference selectivity of the second join at iteration i

$\overline{A^{2^i}}$: the result of recursively joining the source relation A, with all *internal* duplicates removed

A^{2^i} : the result of the i^{th} recursive join of the source relation without all internal duplicates removed

D_i : those tuples generated during iteration i whose end-points are not 2^i apart from each other. Internal duplicates generated during iteration i not yet removed.

$\overline{D_i}$: D_i with internal duplicates removed

C_i^C : the result relation after the ith iteration

$A^{2^0} = \overline{A^{2^0}} = A$

$A^{2^i} = \overline{A^{2^{i-1}}}\ join\ \overline{A^{2^{i-1}}}$

$\overline{A^{2^i}} = \gamma_i^{C1} A^{2^i}$

$|A^{2^i}| = \alpha_i^{C1} |\overline{A^{2^{i-1}}}|^2$

$$|A^2| = \alpha_1^{C1}|A|^2$$

$$|\overline{A}^2| = \gamma_1^{C1}\alpha_1^{C1}|A|^2$$

$$|\overline{A}^4| = \gamma_2^{C1}\alpha_2^{C1}|\overline{A}^2|^2$$

$$= \gamma_2^{C1}\alpha_2^{C1}(\gamma_1^{C1}\alpha_1^{C1}|A|^2)^2$$

$$= \gamma_2^{C1}\alpha_2^{C1}(\gamma_1^{C1}\alpha_1^{C1})^2|A|^4$$

$$|\overline{A}^8| = \gamma_3^{C1}\alpha_3^{C1}|\overline{A}^4|^2$$

$$= \gamma_3^{C1}\alpha_3^{C1}(\gamma_2^{C1}\alpha_2^{C1})^2(\gamma_1^{C1}\alpha_1^{C1})^4|A|^8$$

$$|\overline{A}^{2^i}| = \prod_{j=0}^{i-1}(\gamma_{i-j}^{C1}\alpha_{i-j}^{C1})^{2^j}|A|^{2^i} \quad , \; i>0$$

If we superscript the Global algorithm's closure with A, and the Log method's with C, we get:

$$C_i^C = C_{2^i}^A$$

$$|C_i^C| = (\sum_{j=1}^{2^i}\prod_{k=1}^{j}\sigma_k^A\alpha_k^A|A|) + |A|$$

$$D_i = \overline{A}^{2^i} \; join \; C_{i-1}$$

$$|D_i| = \alpha_i^{C2}|A^{2^i}||C_{i-1}^C|$$

$$\overline{D}_i = \gamma_i^{C2}|D_i|$$

$$= \gamma_i^{C2}\alpha_i^{C2}(\prod_{j=0}^{i-1}(\alpha_{i-j}^{C1})^{2^j}|A|^{2^i})((\sum_{j=1}^{2^{i-1}}\prod_{k=1}^{j}\sigma_k^A\alpha_k^A|A|) + |A|)$$

Timing:

Steps 4 through 9 each get executed $\lfloor log_2 r \rfloor$ times, each other step exactly once.

1. **Distribution :**

$$(p-1)\frac{|A|}{p} \text{ tuples}$$

$p-1$ messages

2a. **Partition :**

$2\frac{|A|}{p}$ partitioning

As in the Global Wavefront algorithm, this step produces *two* output relations. Hence the costs of steps 2 and 3 are doubled.

2b. **Exchange :**

$4(p-1)\frac{|A|}{p^2}$ tuples

$4(p-1)$ messages

For explanation, see analysis of Global Wavefront Algorithm.

2c. **Initialize Hash Tables :**

$\frac{|A|}{p}$ JTID

This is the cost for installing the source relation into the join table, P_i. Setting up the table for C_i has negligible cost, since data insertion is deferred until step di.

2di. **Union :**

$\frac{1}{p}\sum_{i=0}^{\lfloor log_2 r \rfloor} |\overline{A}^{2^i}|$ JTID

Cost of inserting output tuples from the first join, as well as initial source relation, into closure table. A "hybrid" table (type C) is used. It supports both join and union operations fairly efficiently. Because the relation stored (C_i) is involved in a join operation (step iv), the simpler type B table is not appropriate.

2dii. **Join and Partition :**

$\frac{1}{p}\sum_{i=0}^{\lfloor log_2 r \rfloor - 1} |\overline{A}^{2^i}|$ JTL

$+\frac{2}{p}\sum_{i=1}^{\lfloor log_2 r \rfloor} |A^{2^i}|$ partitioning

Two copies (P_i and Q_i) of the join output must be made. These are the inputs for the same join at the next iteration.

2diii. **Exchange :**

$\frac{4(p-1)}{p^2}\sum_{i=1}^{\lfloor log_2 r \rfloor} |A^{2^i}|$ tuples

$4\,(p-1)\,\lfloor log_2 r \rfloor$ messages

The $4(p-1)$ stems from there being two relations to be exchanged, and each machine having to both send to and receive from each other. (See analysis of Global Wavefront algorithm for further explanation.)

2div. **Build Join Table and Filter**

$$\frac{1}{p}\sum_{i=1}^{\lfloor log_2 r \rfloor} |A^{2^i}| \text{ RTI}$$

$$+\frac{1}{p}\sum_{i=1}^{\lfloor log_2 r \rfloor} |\overline{A}^{2^i}| \text{ copying}$$

The join output from step ii may include a large number of internal duplicates, which should be removed before the next iteration (see chapter 2, "Implementation Issues"). Q_i is filtered by inserting it into a type B table (RTI), and copying only non-duplicates to the output (filtered) relation.

$$+\frac{1}{p}\sum_{i=0}^{\lfloor log_2 r \rfloor} |A^{2^i}| \text{ JTID}$$

The join table for the next iteration of step ii is built on P_i. If a type C table is used, then the JTID operation removes any internal duplicates as the table is being built. Building the next iteration's join table "on the fly", as the current join's output is produced, allows us to avoid copying P_i twice (once to filter it, then again to install it in the table).

2dv. **Join and Partition :**

$$\frac{1}{p}\sum_{i=1}^{\lfloor log_2 r \rfloor} |\overline{A}^{2^i}| \text{ JTL}$$

$$+\frac{1}{p}\sum_{i=1}^{\lfloor log_2 r \rfloor} |D_i| \text{ partitioning}$$

Only one copy of this join output is required, since D_i is not joined with itself.

2dvi. **Exchange :**

$$\frac{2(p-1)}{p^2}\sum_{i=1}^{\lfloor log_2 r \rfloor} |D_i| \text{ tuples}$$

$$2\,(p-1)\,\lfloor log_2 r \rfloor \text{ messages}$$

Only a *single* relation is exchanged. The receiving machine just concatenates the incoming fragments of D_i. Duplicates are removed by the Union in step vii.

2dvii. **Union :**

$$\frac{1}{p}\sum_{i=1}^{\lfloor log_2 r \rfloor} |D_i| \text{ JTID}$$

3. **Result Collection :**

$$\frac{|C|}{p} \text{ copying}$$

$$\frac{p-1}{p}|C| \text{ tuples}$$

$p-1$ messages

Same as for Global Wavefront Algorithm.

4.2.7. The Direct Algorithm

terms specific to the Direct algorithm:

α_i^D : join selectivity at iteration i

σ_i^D : difference selectivity at iteration i

m_i : number of distinct column 1 values that match the head of the successor list broadcast at iteration i

dups : number of duplicate tuples generated by the algorithm

S_i : output relation of join at iteration i

S_i' : S_i with duplicates removed

C_i^D : result (closure) relation at the end of iteration i

w : number of distinct column 1 values in the source relation, A

v_i : the vertex (1st col value) whose successor list is processed in iteration i

$\{succ(v_i)\}$: v_i's successor list, at the beginning of iteration i

Note that i refers to both the label of the vertex *and* to the current iteration. They coincide because each iteration broadcasts and processes a different vertex's successor list. The vertices are processed in the order of their labelling.

b_p : broadcast scalar.

This scalar is applied to broadcast communication, to make its cost commensurate with that of point-to-point transmission. Since the extra cost attendant to broadcasting is a function of the number of receivers, b is subscripted by p, the number of machines. See subsection 1.3 of this chapter.

$b_p > 1$

$C_0^D = A$

$|\{succ(v_i)\}| \approx \frac{|C_{i-1}^D|}{w}$

$|S_i| \approx \alpha_i^D \frac{|C_{i-1}^D|^2}{w}$

$|S_i'| = \sigma_i^D |S_i| \approx \sigma_i^D \alpha_i^D \frac{|C_{i-1}|^2}{w}$

$C_i^D = C_{i-1}^D \cup S_i = C_{i-1}^D + S_i'$

$|C_i^D| \approx |C_{i-1}^D| + \sigma_i^D \alpha_i^D \frac{|C_{i-1}^D|^2}{w}$

$m_i \approx \frac{\textit{size of join output}}{\textit{average size of each successor list in Table A before the join}} = \frac{|S_i|}{(\frac{1}{w}|C_{i-1}^D|)^2}$

$\Sigma_{i=0}^{w} |S_i'| = |C|$

$\Sigma_{i=0}^{w} |S_i| = |C| + \text{dups}$

The reference to a "join" should be clarified. Consider the middle loop in step 3b, "FOR EACH $v_i \in \{pred(v_i)\}$ resident on machine h". This loop takes a single vertex v_i's successor list, and merges it into the successor list of each of v_i's predecessors. Let R be the relation consisting of all tuples whose first element value is v_i. The loop's effect is that of joining R with the current transitive closure, and unioning the join result into that closure. Step 3b can, then, be considered a distributed join/union operation.

For analytical purposes, it is useful to interpret step 3b as a join/union - this interpretation furthers comparison with the other algorithms. The lower-level definition, in terms of lists and merges, better reveals the structure of and motivation behind the algorithm. Thus I chose that notation for the algorithm itself.

Timing:

1. **Distribution :**

$\frac{p-1}{p}|A|$ tuples

$p-1$ messages

2a. **Partition :**

$\frac{|A|}{p}$ partitioning

This is the same as in the Global Wavefront algorithm, except that only one partition output relation is produced.

2b. **Exchange :**

$2(p-1)\frac{|A|}{p^2}$ tuples.

$2(p-1)$ messages.

Again, this is like the Global Wavefront algorithm, except that only one copy of the relation is exchanged.

2c. **Build Local Hash Tables :**

$\frac{|A|}{p}$ JTI

$\frac{|A|}{p}$ JTID

Table 2, the successor list structure, is used to filter duplicates. Accordingly, a Type C table is used. JTID, the insertion function, returns TRUE if a tuple to be inserted is a duplicate. (The duplicate is not inserted.) Each tuple generated by the merge is first JTID'ed into Table 2 (step 3bi). Only if JTID returns FALSE is it then JTI'ed into Table 1 (step 3bii). This procedure, along with the condition that the source relation is duplicate-free, allows us to use the non-filtering Type A table for Table 1.

3a. **Distribution :**

$$\sum_{i=0}^{w-1} |\{succ(v_i)\}| \approx \sum_{i=0}^{w-1} \frac{|C_i^D|}{w}$$

$$\approx (|A| + \frac{|C| - |A|}{2}) \text{ tuples}$$

In the Global Wavefront and Logarithmic algorithms, *all* machines both send and receive each iteration. Here, only *one* machine is sending - all others are receiving. Thus we can broadcasting. There are always exactly w iterations.

The successor list representation of tuples allows us to transmit them in a more compact form. Since all tuples in a given list share the same column 1 value, that value need not be repeated for each tuple. The actual amount of data transmitted is approximated by the following formula:

$$\frac{1}{2}(|A| + \frac{|C| - |A|}{2}) + 2w$$

The "2w" term accounts for the header that must be sent with each list, plus a timestamp used to detect message loss.[iv] It is assumed that the average successor list has $|A| + \frac{|C| - |A|}{2}$ elements in it.

Each successor list is broadcast exactly once:

$$b_p \cdot w \text{ messages}$$

For small messages, a broadcast is significantly more expensive than is a point-to-point send. Because of this, and the large number of broadcasts done by this step, the number of messages is multiplied by the "broadcast scalar" b_p. The scalar is applied to the number of *messages*, since the extra cost it reflects is, for small messages, independent of message size.

Before a successor list can be transmitted, it has to be copied out of the hash table where it resides, into contiguous storage:

$$\sum_{i=0}^{w-1} |\{succ(v_i)\}| \text{ COPY}$$

3b. **Join and Union :**

$$w \text{ JTL}$$

There are w iterations of step 3. Each iteration, each machine, in parallel, first looks up v_i in Table A, in order to find its predecessors. (middle loop of step 3b)

$$+\frac{1}{p}\sum_{i=1}^{w} m_i \text{ JTL}$$

Each predecessor of v_i (the header of the current successor list) must now be looked up in Table B.

$$+\frac{1}{p}\sum_{i=1}^{w} m_i \frac{|S'_{i-1}|}{w} = \frac{1}{p}\sum_{i=1}^{w} |S_i|$$

$$= \frac{1}{p}(|C| + dups - |A|) \text{ (JTID - JTL)}$$

The preceding formula describes the cost of step 3bi, merging the successor lists.

$$+\frac{1}{p}|S'_i| = \frac{1}{p}(|C| - |A|) \text{ JTI}$$

Cost of step 3bii, updating predecessor lists.

4. **Result Collection :**

[iv]Timestamping is discussed in chapter 3, "Implementation Issues".

$\frac{|C|}{p}$ copying

$+\frac{p-1}{p}|C|$ tuples

$+p-1$ messages

As in the Global Wavefront and Logarithmic algorithms, the local result relations are disjoint, and can simply be concatenated to form the global result.

4.3. Experimental Methodology

The empirical study is guided by the step-by-step analysis of the preceding section. Variables used in that analysis can be separated into two categories, which I will call "cardinalities" and "unit costs". The former denote sizes, and include such entities as |C|, $\Sigma_{i=0}^{\lceil log_2 r \rceil} |A^{2^i}|$, and number of iterations. Examples of unit costs are "JTID" (for local processing) and "tuples" (for communication). Values for cardinalities and local processing unit costs were taken from test results. Communication costs were calculated on the basis of experimental results presented in [12]. All values used in the calculation of results are tabulated in the appendix. Cost estimates are based on *averages* of the cardinalities - see table B-1 in the appendix.

4.3.1. Cardinalities

In section 1, "Terminology and Motivation", I alluded to the difficulty of estimating a priori the sizes of intermediate and result relations. I therefore resorted to experiment - the values for these variables are taken from that empirical study.

For each of the algorithms (serial as well as distributed), cost estimates were based on average cardinalities. By "cardinalities" I mean "number of tuples" in one of the following categories:

- sums of intermediate relations
- relations as of iteration 0 or the last iteration
- sizes of lists (eg. successor lists) for the Direct algorithm
- duplicates
- number of iterations the algorithm takes to complete

Cardinalities were estimated by simulating the algorithms, counting the appropriate tuples. The values used are averaged over ten runs. Maximum and minimum cardinalities are shown in tables B-2 through B-5 in the appendix.

The notation used allows us to denote a given relation *during a particular iteration*. In general, however, that fine a granularity is neither needed nor illuminating. The quantities which emerge as salient are of three kinds:

- Relation sizes at iteration 0, ie. the *source* relation.
- Relation sizes during the *last* iteration.
- Summations from iteration 0 or 1 to the last or second last.

Summations are in general treated like unanalyzed variables. In evaluating $\sum_{i=0}^{r-1} |L_i|$, for example, we did not record empirical results for each $|L_i|$, but rather a *single* average for the entire sum. I retained the summation notation because it reflects the variable's meaning.

Since summations referred to may or may not include the first and last interations, up to four different summation ranges may be of interest for a given relation. The analysis of the Logarithmic algorithm, for instance, refers to both $\sum_{i=0}^{\lfloor log_2 r \rfloor - 1} |\overline{A}^{2^i}|$ and $\sum_{i=1}^{\lfloor log_2 r \rfloor} |\overline{A}^{2^i}|$. It is only necessary to measure *one* of these sums. If we know the sizes of the relation involved during the first (0^{th}) and last iterations, we can adjust the measured sum to any of the other three, by adding or subtracting the size for the first (0^{th}) or last iteration.

Values for average (mean) relation size sums are listed in table B-1, in the appendix. (The summation notation is not used in the table. A key beneath it relates the table variables to the notation used in the analysis.) All values used are for long integer (4 byte) keys. A tuple therefore consists of 8 bytes.

In the analysis, all sums of G_i and L_i that are mentioned run through iteration r. These algorithms terminate when the wavefront relation is empty. Thus $|G_r| = |L_r| = 0$. Accordingly, size values for the last iteration were not required. Those values are required, and are therefore supplied, for the Logarithmic and Direct algorithms.

During iteration 0, all relations are either coextensional with the source relation or are empty. Sums can be adjusted for the presence or absence of the first iteration simply by adding or subtracting the size of the source relation.

For each relation name, there is a primed (or overlined) and an unprimed version. The primed version results from the removal of those duplicates which can be detected locally by the algorithm concerned. (The precise definitions are given in the preceding section, "The Step-by-Step Analysis", where the notation is introduced.) The experiments measured the sizes of the *primed* relation size sums and the number of duplicates directly. The unprimed relation sums were arrived at by adding the number of duplicates to the sums for their unprimed counterparts. The values for last iterations were measured directly.

4.3.2. Local Processing Unit Costs

The "basic" operations used in the analysis are rather abstract - they do not reflect implementation details. Though a finer-grained analysis would be more accurate, it would also be more obscure. Rather than adopt a lower-level model, I have adjusted the values given for the cost parameters, to reflect *important* contextual influences.

4.3.2.1. Hash Table Operations

JTI, RTI, JTID and JTL are operations on hash tables. As the loading factor of the table increases, these unit costs will also increase, sometimes significantly.

To reflect this, we tested these operations under various loading conditions. Two sets of results were obtained, one with a larger table size (number of buckets) than the other. The results are shown in tables C-1 and C-2 in the appendix. In assigning a value for one of these parameters, we first estimated the number of entries in the table at the time of the relevant operation. The closest "table load" in the table was then cross-indexed with the operation name, yielding the cost for that operation.

Of course, judicious tuning of the hash tables can improve performance. (For a discussion of hash table tuning, see chapter 3.) In table C-1, hash table types A and C have 1000 buckets, and type B has 2000. In C-2, all table types have 5000 buckets. This allows us to assume that at least a modicum of table tuning is done. Each cost is looked up in *both* tables - the lower cost is used. (The only stipulation is that whatever size table is assumed, that assumption must be adhered to for the course of the algorithm. The number of buckets does not magically change to suit the needs of the moment.)

The JTID and RTI operations filter out duplicates - if the input tuple is already in the table, it is not inserted. Actually inserting a tuple involves considerable overhead over merely looking it up. Storage must be allocated for it, and it must be linked into the table. Since this cost is not incurred for duplicates, a JTID or RTI on one will be considerably cheaper. We therefore broke JTID and RTI costs into separate costs for duplicates and non-duplicates. Accordingly, the tables include columns headed "RTI dups" and "JTID dups". The values under "RTI" and "JTID" are for non-duplicates.

The number of duplicates handled by each algorithm is listed in table B-1. For an iterpretation of these values, see the key beneath that table.

The RTI operation is something of a special case. This operation uses table type B. Unlike the other tables,

type B is single-tiered. This makes it especially amenable to tuning. (For a full explanation, see chapter 3.) Except for very small relations (very few tuples in the table), we assume that the type B table has a favourable number of buckets. This allows us to use *constants*, .072 and .042, for RTI and RTI dups, respectively, regardless of relation size.

4.3.2.2. COPY

The COPY operation may involve retrieving the copied relation from a hash table or from a stream-structure relation. Again, two tables (C-3 and C-4) are provided, offering a choice of hash table size. Values for the latter case are listed under "from relation". In some cases, the tuples to be copied are already available - the cost is then determined by subtracting the cost of retrieving from a stream- stuctured relation from the "from relation" COPY cost. These results are listed under "no retr".

Looking at tables C-3 and C-4, you will see that the values under "from relation" and "no retr" are virtually constant. For these cases, we use values of .014 and .009 throughout. These are listed in table C-5. (For tuple retrieval, we always charge .005 ms..)

4.3.2.3. PART

Test results that suggest that partitioning cost is constant, regardless of the size of the relation being partitioned. PART always includes the cost of copying the tuple to the appropriate partition.

Ref(CONST) gives two values for PART. The higher value (.024) includes the cost of retrieving the tuple from a stream-structure relation (not from a hash table). The lower value (.019) does not include retrieval cost. Consider step 2a of the Global Wavefront algorithm. In this step, a single input relation is partitioned twice. We do not want to double-charge for tuple retrieval. Hence the cost for this step is calculated as $\frac{1}{p}\frac{(.024+.019)}{1000}$ seconds.

4.3.2.4. How Values Were Obtained

To arrive at the local processing costs listed in the appendix, each operation was performed on 100,000 tuples, and the average cost taken. Consider for example the value (.091 ms.) for JTI, with 5,000 tuples in the table. The following test was iterated 20 times:

> A new relation with 5,000 tuples in it was generated. The contents of this relation were then inserted into an initially empty type A table.

The costs for each iteration were summed, then divided by 20. For a table with 100,000 entries, only one test was run. Values for RTI, RTI dups, JTID, JTID dups and JTL were obtained in the same way.

As mentioned above, COPY (from a relation), COPY (no retrieval), PART and "retrieve tuple" are treated as constants. This decision followed from empirical results - varying the input relation size for these operations did not appear to affect their values. Each of these constant values was obtained by averaging the costs of at least 100,000 operations.

The tests from which the table values were derived were run in the dead of night. I reran some of them at different times, so as to detect any variance resulting from different load conditions on the facilities used. Any differences between test runs were insignificant.

4.3.3. Communication Costs

Table D-1 lists the costs of point-to-point sends for a variety of message sizes. The costs per tuple listed in table D-2 are derived from those message costs. Key values are assumed to be long integers - each tuple therefore consists of 8 bytes. Note that the cost per tuple decreases markedly as message size increases.

Let

m = the number of messages sent within a given step of the algorithm

t = the average size, in tuples, of the messages sent during that step

c = the total communication cost for that step, in seconds

For message sizes under 128 tuples (1K bytes), the following cost formula was used:

$c = m\ (2 + (t-4)(.033)) / 1000$

For such small messages, the communication cost is dominated by *the number of messages*, rather than by the amount of data sent. The first 4 tuples (32 bytes) in a message are "free" - this is the minimum-size message. The minimum-sized message costs 2 ms.. Sending $\frac{1}{2}$ K bytes costs an additional 2 ms.. The scalar .033 ms./tuple is just 2/((512/8)-4). For a message of 128 tuples, the formula yields a cost of 6.1 ms., compared with the measured value of 6 ms. shown in D-1.

For larger messages (128 tuples and greater), the cost is dominated by the amount of data transmitted. Costs for these are obtained by interpolating linearly between the costs listed in D-2.

Average message sizes are estimated simply by dividing the total amount of data transmitted within a given algorithm step by the number of messages sent during that step.

Let

m = the number of iterations of the step

n = the total number of tuples transmitted therein

q = the number of messages sent per iteration

The average message size is then estimated by:

n / qm

The assumption here is that the size of the relation transmitted grows *linearly* over the course of the algorithm (from one iteration to the next). Since this assumption is almost certainly unsound, the estimates of communication costs are bound to be inaccurate. I trust, however, that they are reasonable approximations of the actual costs.

Only the broadcast at the beginning of the Local Wavefront algorithm (step 1) will, in the expected case, involve broadcasting large messages. Since that step only executes once, the cost of retrieving acknowledgements from more than one receiver contributes very little to the total running-time of the algorithm. Accordingly, I cost that broadcast as if it were a point-to-point send.

The broadcast in step 3a of the Direct algorithm will almost invariably transmit *very* small messages - usually the minimum size. Experiments done by Garnik Haftevani at Simon Fraser University indicate that to broadcast the minimum-sized message to 3 remote processes takes 3.9 ms, using the V-System broadcast system call. That result is averaged over 10,000 messages, using Sun-3 workstations. The broadcast cost for small messages, with four participating machines, is then obtained from the following formula:

c = m (3.9 + (t-4)(.033)) / 1000

Note that for any but minimum-sized messages, the broadcast cost is not *exactly* obtained by scaling the point-to-point cost. I use a simple scalar in the step-by-step analysis in order to avoid notational clutter.

The cost to broadcast the minimum-sized message to *seven* machines was not available at the time of writing. If we used a logarithmic broadcast, with a point-to-point send costing 2 ms., we would incur a cost of 6 ms.. This figure is no doubt too high for a LAN-type multicast. (The V multicast operation scales very well.) The cost *will* rise as more receivers are added, since more acknowledgements must be handled. I used a value of 5.0 ms. for the broadcast with 8 participating machines. The formula for that case is thus:

c = m (5.0 + (t-4)(.033)) / 1000

An important assumption with respect to broadcast cost is that the *machines have sufficient system buffer*

capacity to prevent the occurrence of overrun. For a main memory model like ours, this assumption is surely justified. For Sun-3 workstations, it is realistic. To *prevent* overrunning the originator with replies (acknowledgements), the V broadcast has each receiver wait for a random time period before acknowledging. With ample buffer capacity, this is no longer necessary. *Our broadcast cost (3.9 ms. to broadcast 32 bytes to 3 machines) is based on the assumption that those delays have been removed!* Since the Direct algorithm has the highest communication costs, I wished to estimate its costs as optimistically as was reasonable.

4.3.4. The Input Data Used

The relative efficiencies of the algorithms will depend on the computation size resulting from the particular input data used. By "computation size", I mean the total amount of data processed. This includes input, intermediate and result relations. Computation size depends on :

- the size of the source relation
- the size of the key domain
- the distribution of data values within the source relation
- the mechanics of the algorithm used (eg. how many duplicates are generated, and how soon they can be detected and removed from further processing)

Note that computation size will depend on which algorithm is used.

All key values were generated by means of the UNIX "random()" function, with uniform distribution. Given that distribution, we generated data sets with varying computation sizes by varying the key domain and source relation sizes. I refer to particular test runs by the ordered pair (source relation size, key domain size). For example, a data set consisting of 750 tuples (source relation size) chosen from a key domain size of 1000 is denoted by (750,1000).

4.3.4.1. The Sample Sets Used

We wished the computation sizes tested to fall within certain limits. Firstly, we wanted those sizes to be large enough to be interesting. Below a certain size, distributing the computation is counterproductive - a serial implementation will complete sooner. Practical considerations imposed an upper bound on the problem sizes tested. For a given domain size d, the size of the TC for a relation on that domain is bounded above by d^2. Test runs indicated that with input relation sizes as low as 1.5 x d, the result relation size could closely approach that limit. The consequent computational explosion overwhelmed the computing resources on which the tests were conducted. In fact, this occured even when testing the Logarithmic algorithm on input data (1250,1000). (Accordingly, values for that test are listed as "n/a" in table B-1.)

The sizes of the intermediate relations depend to a large extent on the connectivity of the problem graph. As connectivity increases, the probability of asynchronous and cyclic data occurring will also increase. Such data will lead to the generation of duplicates. If there is one factor which above all accounts for the different performances of the algorithms, it is their method of handling duplicates. The choice of test data was thus strongly motivated by a desire to test the algorithms on input graphs of varying degrees of density.

We used domain sizes of 1000 and 5000. With domain size 1000, tests were run for source relation sizes of 750, 1000 and 1250 tuples. With domain size 5000, tests were run for source relation sizes of 3750 and 5000. The particular source sizes used were chosen only after some experimentation. Source sizes less than (750,1000) and (3750,5000) produced very sparse graphs. Input graphs denser than (1250,1000) and (5000,5000) resulted in computational explosions.

4.3.4.2. Variance in Cardinalities

Given the source and domain sizes, the closure size can vary a great deal, depending on the *particular* tuples generated for the source relation. The reason for this is best explained in graph theoretic terms. Interpreting each tuple as an edge, and each element of a tuple as a vertex, we can refer to the "connectivity" of the relation's graph. Minute "random" effects can hugely change the connectivity of a graph, and thus the density of its TC. Adding a single edge, for instance, can transform a graph from one consisting of two strongly connected components to a single component. If the size of the input relation is n, that incidental change could increase the size of its TC by a term on the order of n^2! For any given (source, domain) choice, then, we expect a large variance in computation sizes. Accordingly, for each such choice, ten tests were run, each with a different seed passed to the UNIX "random()" function. The values listed in table B-1 are averages over ten test runs.

Tables B-2 through B-5 in the appendix show, for each distributed algorithm, maximum and minimum test results for the relevant cardinalities. The table for the Local Wavefront algorithm assumes that four machines are used - the results for eight machines would be very similar. For the other algorithms, the cardinalities are not dependent on the number of participating machines. I did not calculate standard deviation or variance. In this discussion, I will refer to the ratio of maximum to minimum values as the "ratio".

The values for the Global and Local Wavefront algorithms show the smallest ratio. Neither of these algorithms generates a great many duplicates. What the ranges show is the range in *output size* for a given input. It seems that as the graph gets denser, the effects of random variations in the input graph become more

pronounced. As expected, the ration of maximum to minimum is lowest for sparse graphs - (750,1000) and (3750/5000). When the ration of relation to domain size is low, so is the probability of, for instance, return edges. Since the probability of matches rises with the ratio of relation to domain size, we would expect the sort of structure that results in large output size - asynchronous data, cycles, cliques, etc. - to occur more commonly in denser graphs. For fairly dense graphs ((1000,1000) and (5000,5000)), the ratio of maximum to minimum is about $5\frac{1}{2}$: 1. In fact, (1000,1000) showed the highest variance of all, for the Wavefront algorithms. I am not sure if this is what we should expect, or if it is really more probable that (1250,1000) should show the highest ratio. It is hard to make much of the ratio for number of iterations, except that as the output approaches being a complete graph, that ratio should fall.

The ratios are highest for the Logarithmic algorithm. This is due to the presence of large numbers of duplicates. The effects of structures like cliques are compounded in this algorithm, since not all external duplicates can be removed. Of course, the ratio for "iterations" is low, since that quantity is a logarithmic function of maximum path length. What the high ratios show for this algorithm is its very high susceptibility to minute changes in the input graph.

As graph density rises, the Direct algorithm's ratio for duplicates rises quickly. As with the Logarithmic algorithm, minute changes in the input graph can result in the presence of those structures which lead to duplicate explosion. The ratio for number of iterations is very low. "Iterations" is in this case just the number of vertices with out-edges in the input graph. Since the relation sizes are fairly large, and tuple values are generated with a random number generator, this low ratio is to be expected.

The Logarithmic and Direct algorithms have the highest *average* cardinalities (except for iterations in the Logarithmic), and also the highest variation in them. Thus in the *worst case*, these algorithms will significantly underperform the Wavefront methods.

Even for the Wavefront algorithms, we can expect a high variance in relation sizes, from run to run, with a concomitant high variance in run-time. This, I suggest, is a product of the *problem*. Both join and transitive closure are potentially quadratic in input size. The join operation seldom reaches that limit. But the presence of graph structures like cliques exposes that underlying quadratic potential.

4.4. The Results

In the first subsection, I present comparative cost results for each of the distributed algorithms, and for the serial Delta-Wavefront algorithm. In the second, I discuss how much speedup the distributed algorithms achieve over their serial progenitors, and how well they scale, ie. how much improvement in performance results from using more machines.

4.4.1. Cost Comparisons

In this subsection I present, for each algorithm, total local processing, communication and combined costs. Two sets of graphs are presented. One set shows results when four machines participate in the computation. The other shows the results for eight participating machines. The results for each quantity are averages over ten test runs. As mentioned above, they can vary greatly from run to run. (This matter is discussed in the following subsection.)

Performance differences between algorithms are explained in a fairly abstract way. A detailed comparison of the Global Wavefront algorithm with each of the other distributed algorithms is presented in appendix F.

Cost results are presented graphically in figures 4-1 through 4-12.[v] In the graphs, data points are connected with straight lines. This was done only to make them more readable. We do *not* suggest that complexity in fact grows linearly between the sample sizes tested! I will first discuss local processing costs, then communication costs. Within each of those categories, I consider first results with domain size 1000, then those with domain size 5000. Finally, combined costs are presented. In addition to being compared with each other, the distributed algorithms' performances (combined costs only) are compared with that of the serial Delta-Wavefront algorithm, indicated by "Serial" in the graphs.

4.4.1.1. Local Processing

A separate curve is drawn for each algorithm. Relation sizes are plotted on the x-axis. y-values indicate total average local processing cost, in seconds.

Note first that for all algorithms, costs rise sharply once the source relation size exceeds the domain size. As the density of the input graph rises, the TC tends rapidly towards a complete graph. As the result size grows, so too will the sizes of intermediate relations. Once a complete graph *is* obtained, any increase in source size will not, of course, increase the size of the TC.

[v]The actual values for the data points are tabulated in E-1 through E-4 in the appendix.

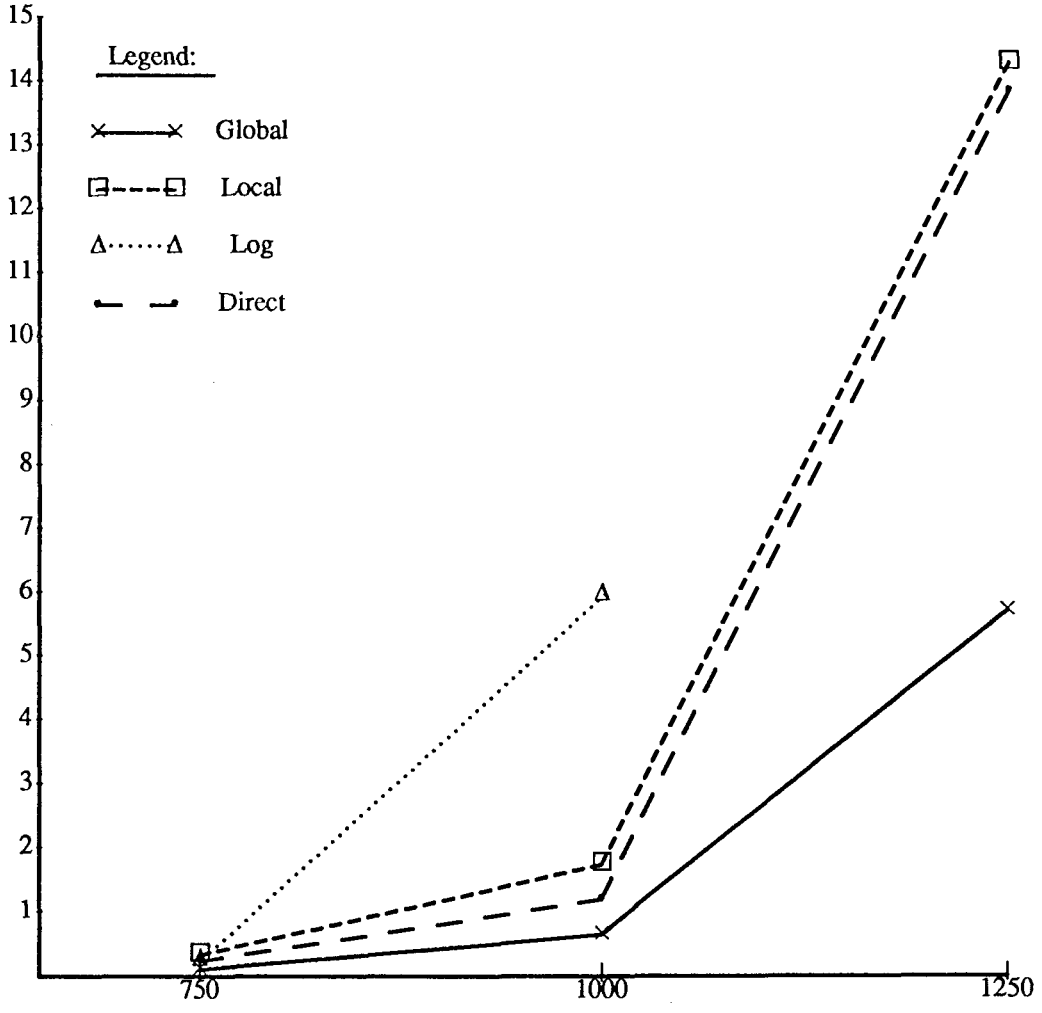

Figure 4-1: Local Processing Costs - domain size 1000 - 4 machines

Each of the algorithms is based on a serial progenitor. Certain of the performance differences between them would also arise between their serial counterparts, while others are peculiar to the distributed versions. Costs for the serial algorithms are given in table E-4 in the appendix. I will first discuss those factors which stem from the structure of the underlying serial models, and subsequently those which are artifacts of distribution.

Examination of table B-1 in the appendix will reveal the main reason for the differing performances of the algorithms. In general, the higher the cost for the algorithm, the more duplicates it handles!

The Logarithmic algorithm is especially bedeviled by the duplicate problem. To understand why duplicates

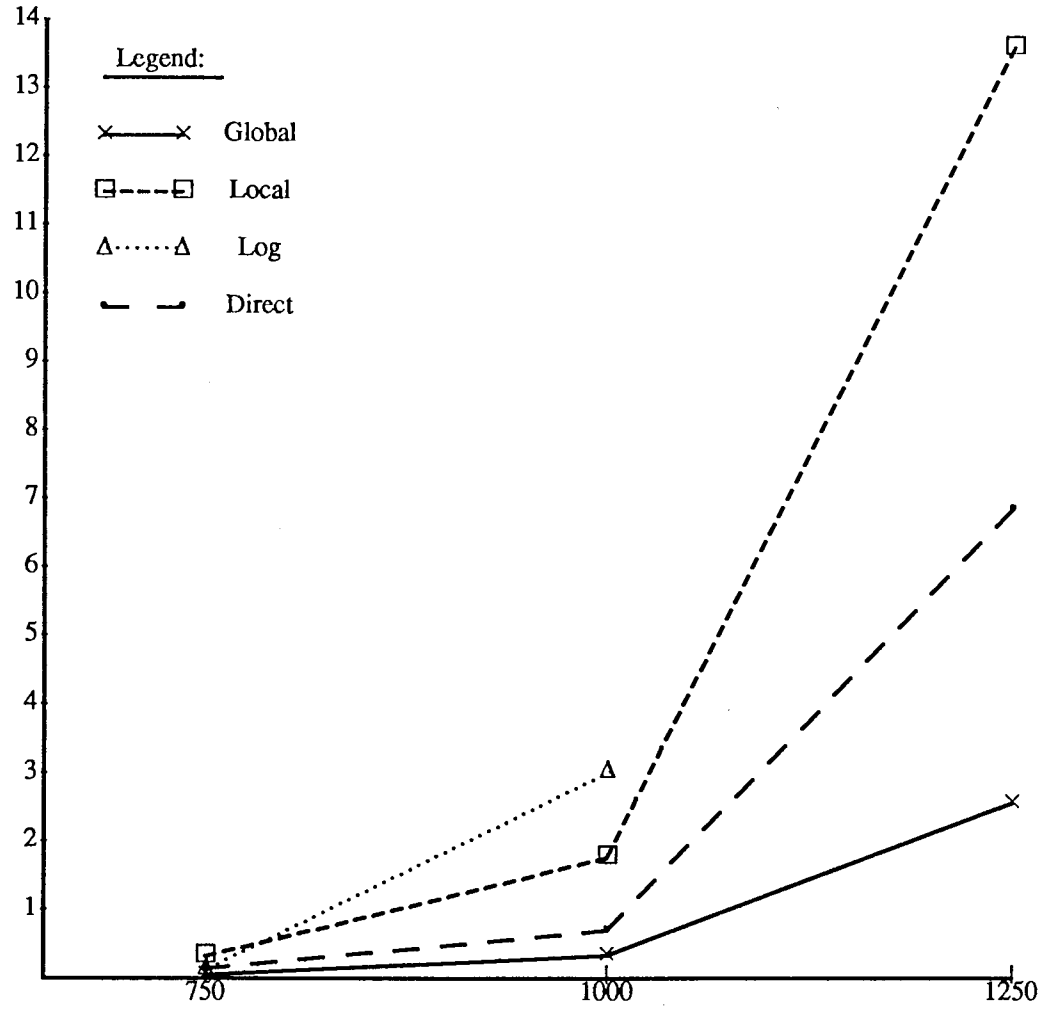

Figure 4-2: Local Processing Costs - domain size 1000 - 8 machines

proliferate in that algorithm, consider first what happens when the relation A^{2^i} contains a clique of size l. When that relation is joined with itself, in step 2dii, each edge in that clique will match each other. The join output attributable to the clique will contain l^2 tuples! Of these, $l^2 - l$ are duplicates. While the duplicates are removed from $A^{2^{i+1}}$, the original clique cannot be, since it *is* part of $A^{2^{i+1}}$. Thus those duplicates will *again* be generated during iteration $i+1$. The proliferation of duplicates feeds on itself. (In the Wavefront algorithms, any cliques are removed by the set difference operation, and will thus not contribute further to processing costs.)

The second join generates even *more* duplicates. These duplicates can be traced to various causes. One is that this join may be fed external duplicates from the first join. When joined with the current result, these

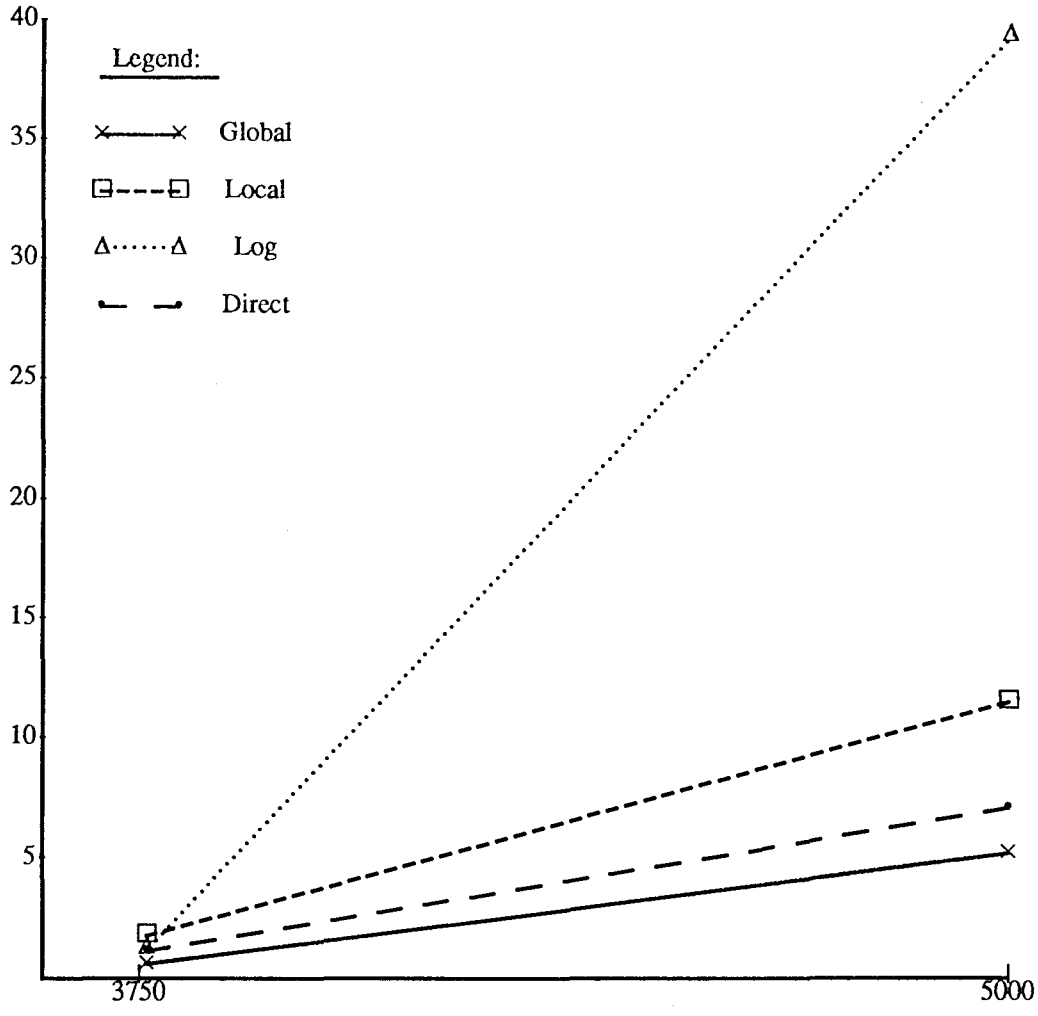

Figure 4-3: Local Processing Costs - domain size 5000 - 4 machines

will produce yet more duplicates. (Since C_i will generally be larger than A^{2^i}, joining with it will generally produce more duplicates.) Another cause has to do with the termination point of this algorithm. In the last iteration, it generates all relations from $A^{2^{\lceil \log_2 r-1 \rceil}+1}$ to $A^{2^{\lceil \log_2 r \rceil}}$ inclusive. What if r is just 1 greater than a power of 2, ie. $2^{\lceil \log_2 r \rceil} = 2r-2$? Then, even after internal duplicates are removed from the join output, almost the entire last iteration can consist of duplicates!

Note that for sparse inputs ((750,1000) and (3750,5000)), the Logarithmic algorithm compares quite favourably with the others. If we can guarantee that the data are synchronous, this algorithm is reasonably attractive.

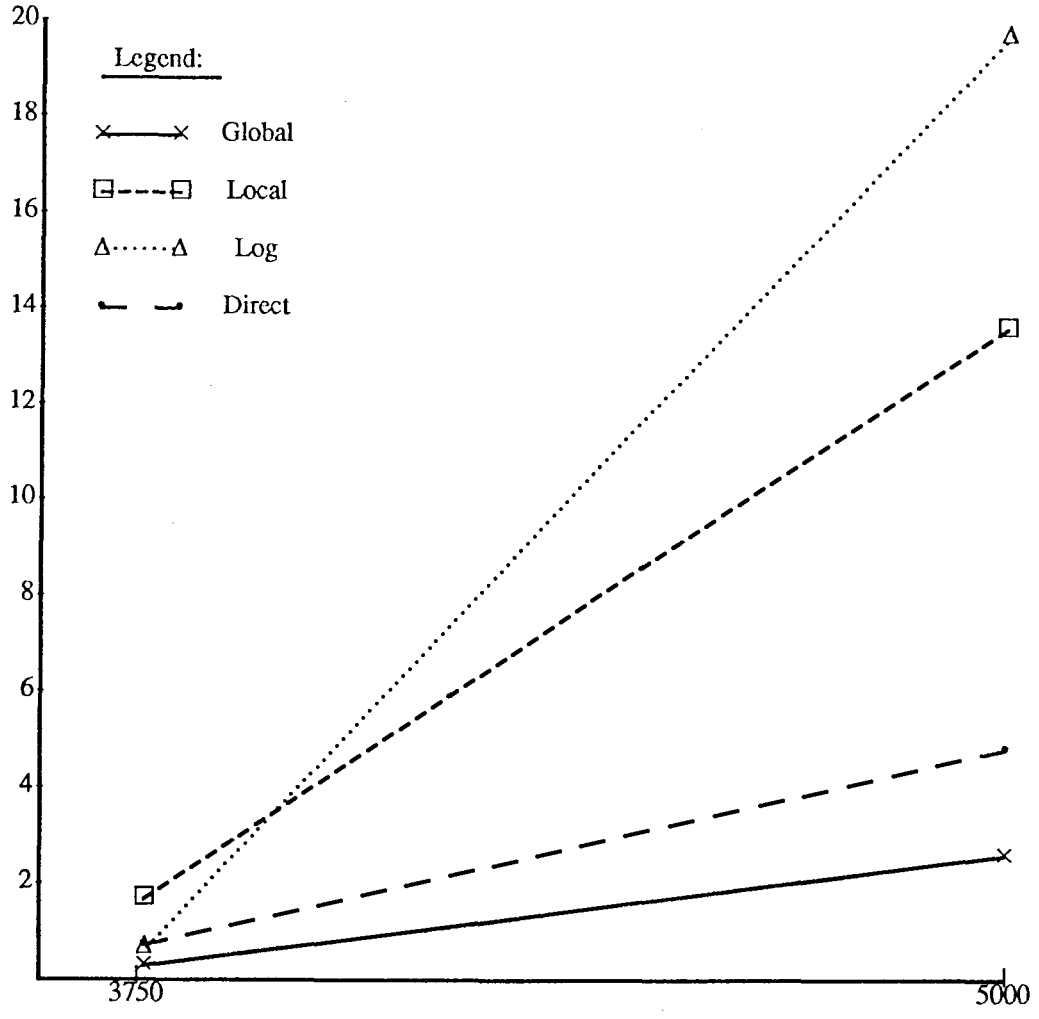

Figure 4-4: Local Processing Costs - domain size 5000 - 8 machines

The Direct algorithm also generates a plethora of duplicates (though considerably fewer than does the Logarithmic algorithm). Especially during its later iterations, we would expect the successor list being merged to contain numerous vertices already present in the lists being merged into. Note, though, that a given successor list is processed (merged into the others) *exactly once*. A duplicate generated by merging list 1 may be regenerated by merging list 5. But no copy of it is cycled back into the computation - all duplicates are removed as soon as they are detected. In the worst case, a clique of size l will produce $l^2 - l$ duplicates over the course of the algorithm, rather than that many *each iteration*, as in the Logarithmic algorithm.

The Direct algorithm never actually inserts a duplicate into a hash table. It is simply searched for - when found, it is discarded. Compared with table insertion, looking up a duplicate is an inexpensive operation. In

short, the Direct algorithm incurs very little cost per duplicate generated. This accounts for its surprisingly good performance, in light of the number of duplicates it handles.

Since it generates and carries the fewest duplicates, it is not surprising that the Global Wavefront algorithm consistently incurs less local processing cost than the other algorithms. The duplicate problem will plague *serial* implementations of the Logarithmic and Direct algorithms as well. (Looking at table E-4 in the appendix, you will see that the Delta-Wavefront algorithm outperforms either of the others in the serial case.)

Since the Local and Global Wavefront algorithms both derive from the same serial algorithm, any performance difference between them results from distribution-related matters.

Both the Logarithmic and Direct algorithms make use of the rather complicated type C hash table. This implementation decision springs from the need to detect duplicates. Of all the table types, the type C variety is the most expensive to build. This will adversely affect the performance of these algorithms. (The Wavefront algorithms can make do with the simpler type A and B tables.)

The performance of the Logarithmic algorithm is impaired by the need to rebuild the join hash table for A^{2^i} during each iteration. (Note as well that $|A^{2^i}|$ can be considerably larger than $|A|$.) This exacerbates the duplicate problem. Not only are cliques (and other external duplicates) retained, but they must be reinserted in that table in each iteration! As mentioned, table insertion is an expensive proposition. In fact, it is the dominant local processing cost in all the algorithms.

I now discuss performance factors due to distribution.

What the duplicate problem *cannot* entirely account for is the rather poor performance of the Local Wavefront Algorithm. For data set (1000,1250), for instance, the Direct algorithm outperforms the Local Wavefront. Yet for that data set, the Direct algorithm generates substantially more duplicates. The problem with the Local Wavefront algorithm is that the local results that it generates cannot be guaranteed disjoint. This complicates result collection. In order to eliminate duplicates from the result, the local closures have to be *merged*. This merging process contributes almost half the local processing cost.

The reason that the merge is so expensive is that it is done by the controller *sequentially*. Merging *can* be done in parallel - on the model of the multi-way merge. This method, however, results in *higher* costs in this context. An efficient parallel merge can be implemented for a *tightly coupled* system, with a very large number of processors. I did not attempt to adapt that parallel method to our coarse-grained environment.

The other three algorithms produce disjoint local closures. Their local results can thus simply be concatenated, at negligible cost.

Of course, maintaining sharp partition boundaries throughout the algorithm is not without cost. Most of that cost falls under the heading of communication. There may also be extra local processing costs. The Global Wavefront algorithm must partition its driver (wavefront) relation once each iteration. The Logarithmic algorithm pays more dearly. *Two* copies of A^{2^i}, *plus* D_i, must be partioned during each iteration. In addition, a new hash table for A^{2^i} has to be built for each iteration. (This exacerbates the duplicate problem, since duplicates must also be inserted into that table.) Mercifully, the Logarithmic algorithm uses the fewest iterations. What emerges is a cost tradeoff between allowing the participating machines to proceed independently, then paying the price at result collection time, and maintaining sharp partition boundaries throughout the algorithm. This tradeoff will be discussed at greater length in the context of the *combined* results.

Note that the Direct algorithm "automatically" keeps the local partitions disjoint. In that algorithm, a machine will only generate tuples within its own partition. No "extra" processing is required.

4.4.1.2. Communication

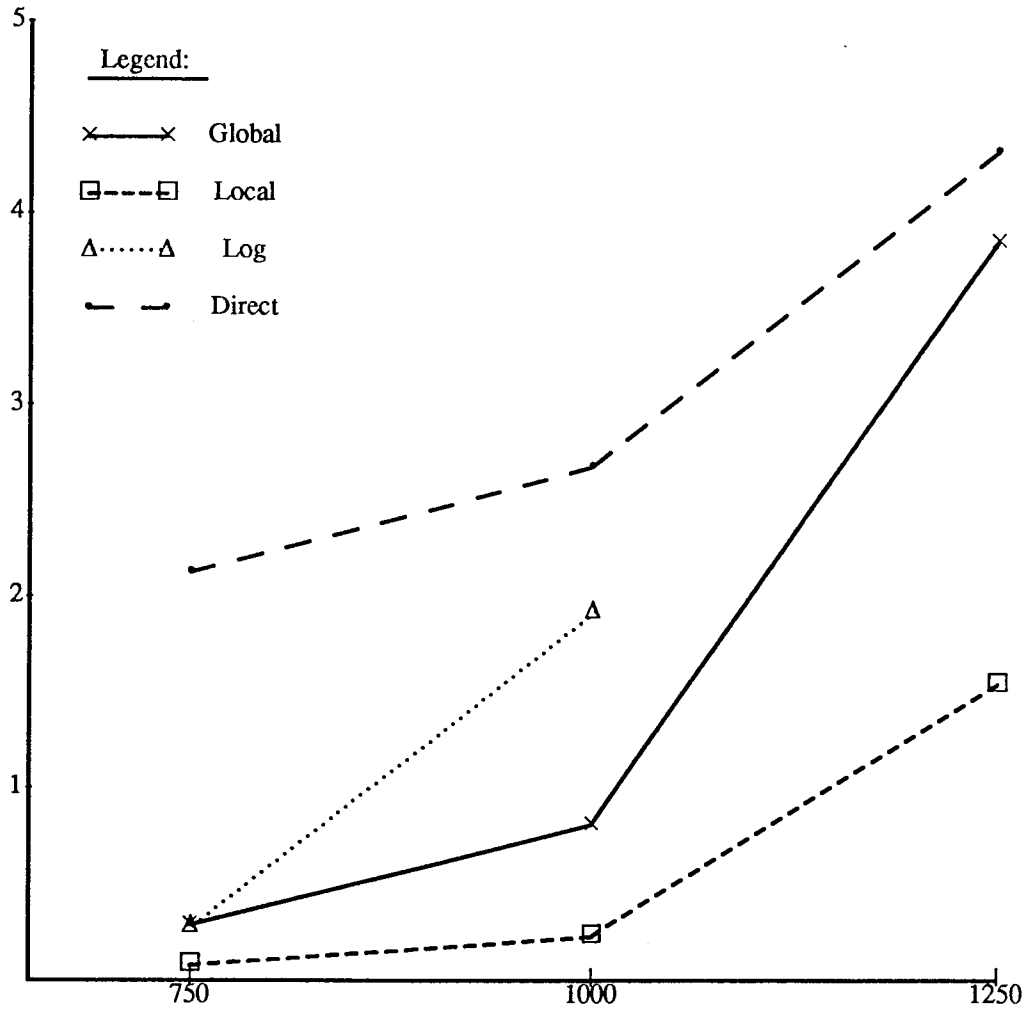

Figure 4-5: Communication Costs - domain size 1000 - 4 machines

Not surprisingly, commuication costs are consistently lowest for the Local Wavefront algorithm. In that algorithm, only the source and result relations are communicated.

Distribution of the source and collection of results are identical for this algorithm and the Global Wavefront method. The difference lies in the amount of data exchanged during generation. For sparse relations, the Logarithmic algorithm incurred lower communication costs than did the Global Wavefront. This no doubt results from its using fewer iterations. For dense graphs, the Logarithmic algorithm again shows poorly. This can be attributed to the transmission of duplicates.

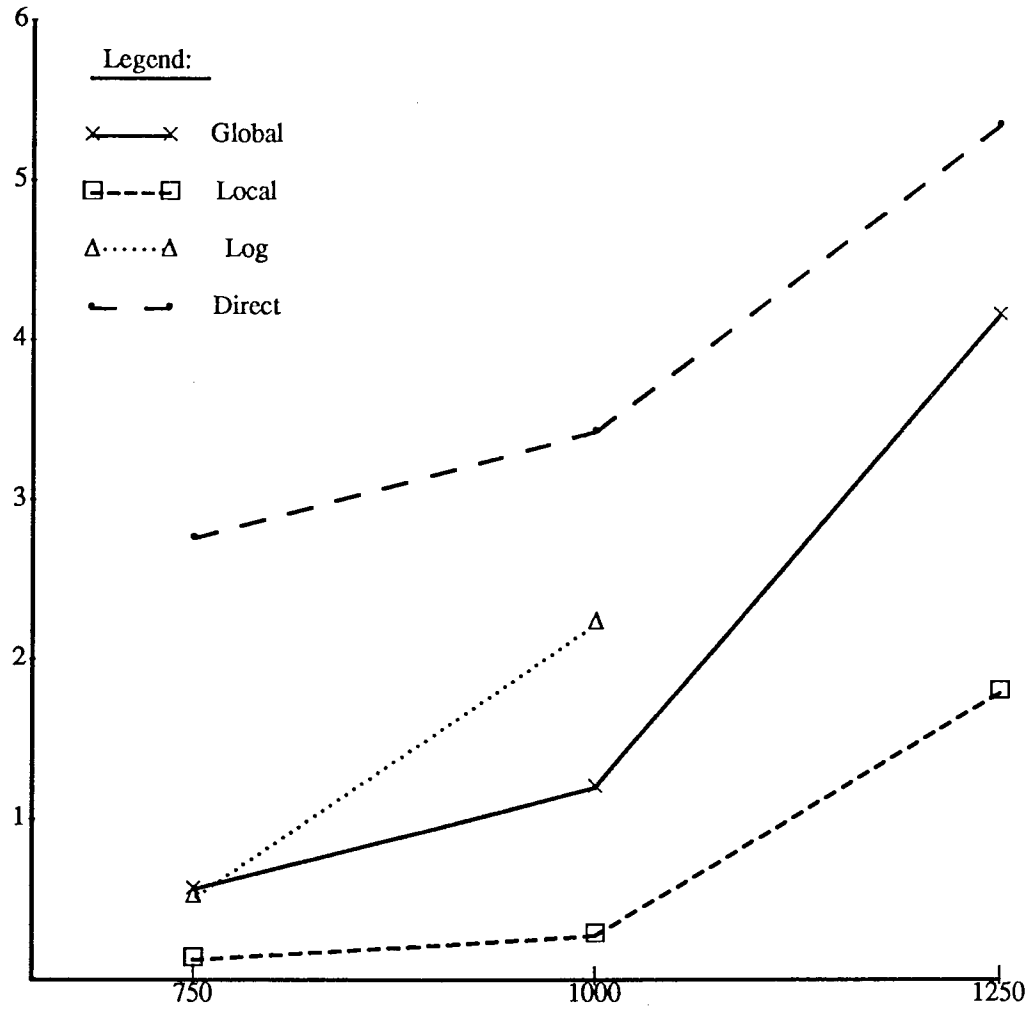

Figure 4-6: Communication Costs - domain size 1000 - 8 machines

For all data sets, the Direct algorithm's communication costs were highest. The reason is that that algorithm takes by far the most iterations to complete - data are transmitted during each iteration. Actually, only the Local Wavefront algorithm sends fewer *tuples*. The high cost results from *the number of messages sent*. It is vastly cheaper to send a few large messages than miriad small ones![vi] The Direct algorithm requires an iteration for each vertex in the input graph. The number of iterations thus depends more on the *source* size than on the size of the result. (TC adds only edges, not vertices.) This explains the very high costs for sample sets (3750,5000) and (5000,5000). Corollary to this is the observation that its communication costs are relatively independent of graph density.

[vi] Of course, a broadcast is also more expensive than is a point-to-point send. However, each iteration of the Direct algorithm uses a *single* broadcast, while an exchange of intermediate results in the iterative algorithm must use p-1 sends.

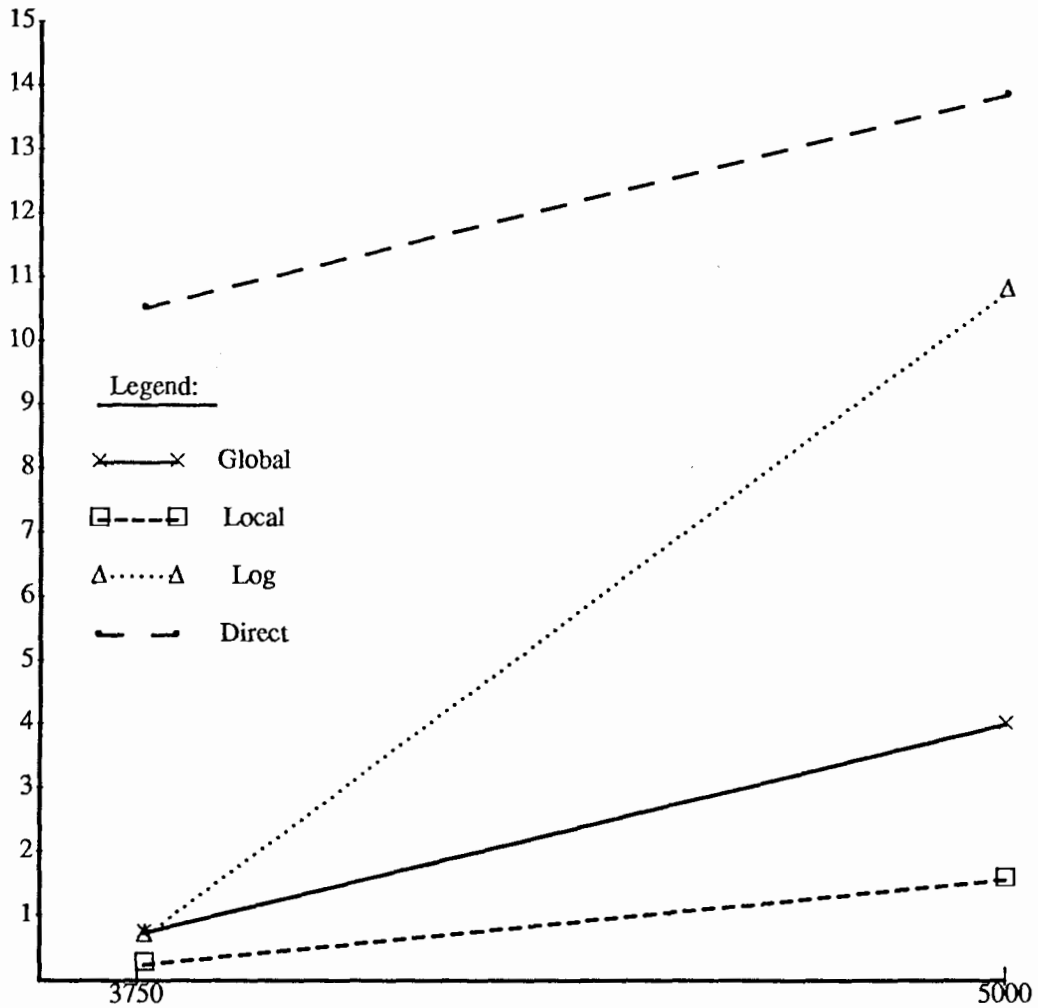

Figure 4-7: Communication Costs - domain size 5000 - 4 machines

You will notice that the y-axes of the different cost graphs are not uniformly scaled. The graphs for communication are, in effect, skewed on the y-axis. This obscures the fact that communication costs in general rise more slowly with problem size than do local processing costs. The reason for this is that communication costs per tuple decrease markedly as message size increases. For the Wavefront and Logarithmic algorithms, larger closures do tend to require more iterations to compute.[vii] However, the most pronounced result of increased computation size is larger messages.

[vii] Note that this is not a hard-and-fast rule. If the input relation is a complete graph, those algorithms will terminate after only two iterations.

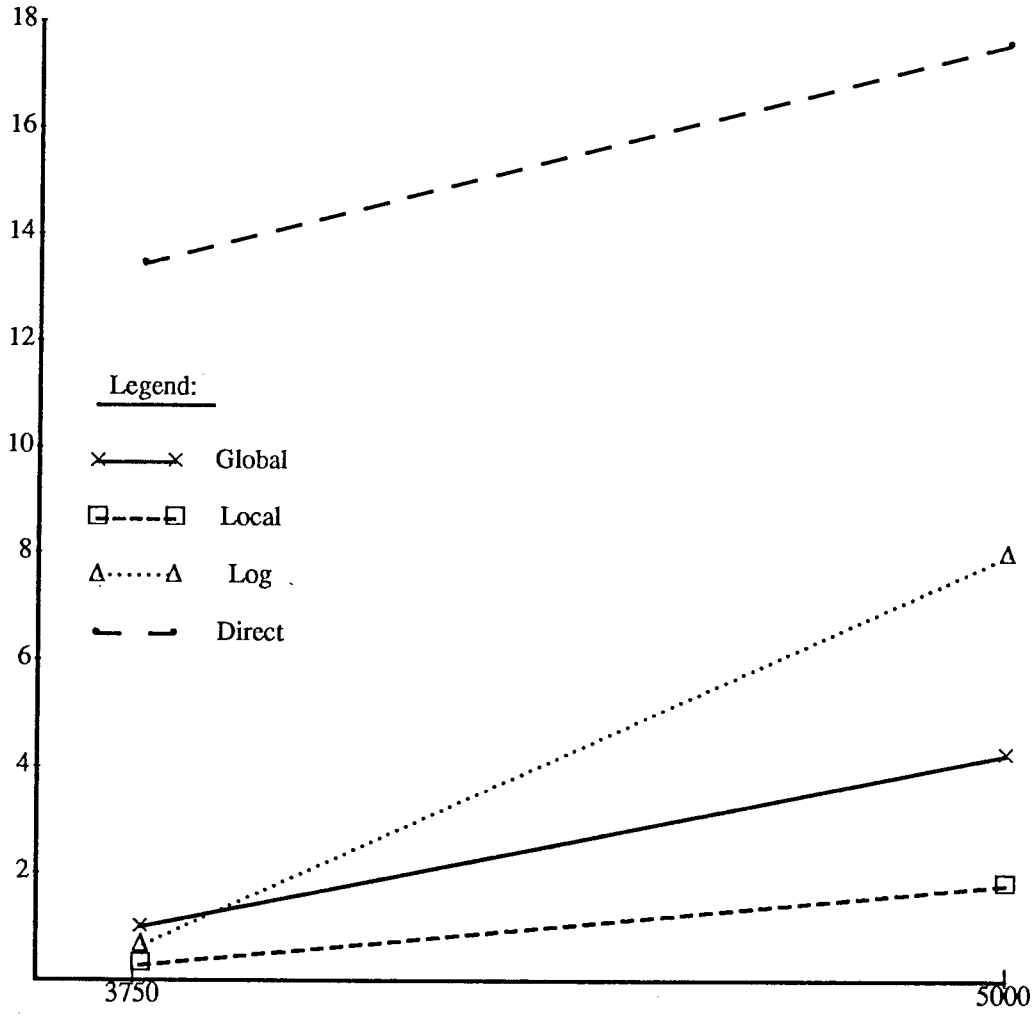

Figure 4-8: Communication Costs - domain size 5000 - 8 machines

4.4.1.3. Combined Results

Figures 4-9 through 4-12 show costs not only for the distributed algorithms, but for a Serial one as well. The serial algorithm is the Delta-Wavefront, which for all data sets outperformed either of the other serial algorithms. This discussion is centred around the *distributed* algorithms. The serial Delta-Wavefront costs are shown on the graphs merely to put these results in context. As you can see, *none* of the distributed algorithms achieves spectacular speedup over the best serial one.

The combined costs depicted in the following figures should be read under the following qualification: they do *not* reflect the effects of uneven load distribution between participating machines. They are based on the assumption of absolutely even load sharing. In practice, this condition is hardly to be expected. Since the algorithms are synchronous (they proceed independently only *within* iterations), no machine may move to the next iteration until the slowest has completed the present one. Running time will thus be dictated not by the average load on each machine, but by the heaviest one. The less uniformly the data are distributed, the slower the algorithms will run. All of the results shown are too optimistic (except for the Serial algorithm). How over-optimistic they are depends on the algorithm, and on the distribution of "the real" data.

Each data exchange or distribution step constitutes a "sychronization point". The more synchronization points an algorithm has, the more it will suffer from uneven load distribution. Each algorithm has a synchronization point at the start, when the source relation is parcelled out, and one at the end, when results are collected. In addition, the Global Wavefront and Direct algorithms have to synchronize once per iteration, and the Logarithmic alogithm twice per iteration. Therefore the Local Wavefront algorithm can be expected to be the most resilient to uneven load distribution, followed by the Logarithmic, Global Wavefront and Direct algorithms, in that order. The Direct algorithm, with its very large number of iterations, will be seriously affected by this problem. In interpreting the following figures, this matter should be kept in mind. (It should also be kept in mind that the Serial algorithm - Delta-Wavefront - suffers *not at all* from load imbalance. So the speedup actually achieved by the distributed algorithms will be even less than that suggested by the graphs.)

Although the Logarithmic algorithm performs well on sparse input graphs, its cost for dense graphs is intolerably high. As mentioned, this results mainly from that algorithm's problems with duplicates. This problem results not from distibuting the algorithm (though the presence of duplicates adds to communication costs), but from the nature of the underlying *serial* algorithm.

For the sample sets tested, the Direct algorithm consistently underperforms the Wavefront algorithms. The

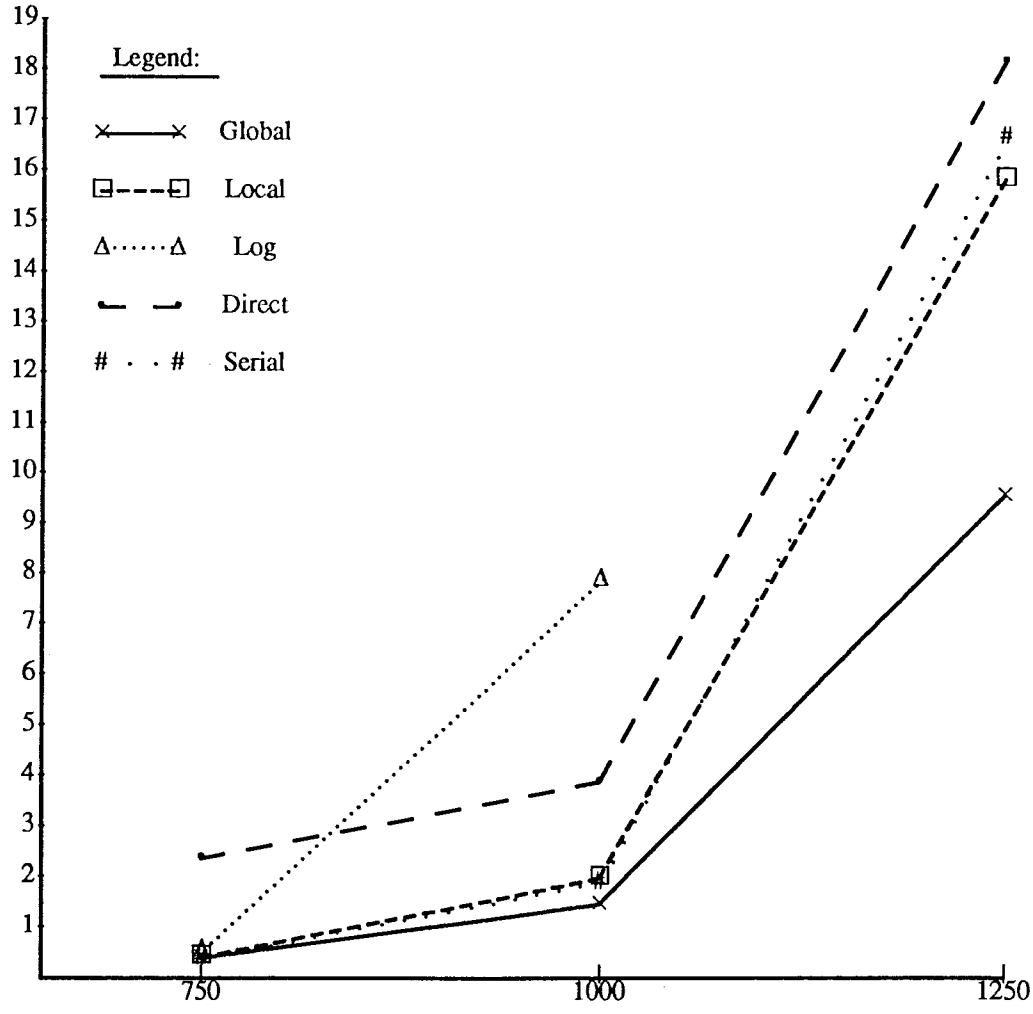

Figure 4-9: Combined Costs - domain size 1000 - 4 machines

primary cause for this poor performance is the large number of messages that the Direct algorithm sends. The state of each successor list $\{succ(v_i)\}$ must be known to *all* machines during iteration i. This entails broadcasting the current list at the start of each iteration. With this algorithm, an inherently sequential relationship between subsolutions from one iteration to the next inhibits distribution. This leads me to conclude that this algorithm is poorly suited to distribution. Only with very small domain sizes will the Direct algorithm compare favourably.

Even in the serial case, the Delta-Wavefront algorithm consistently outperforms the serial Direct one. For dense graphs, the latter must deal with far more duplicates than must the former. But even for sparse graphs, the Delta-Wavefront is much faster. The Delta-Wavefront inserts one copy of the source relation and one

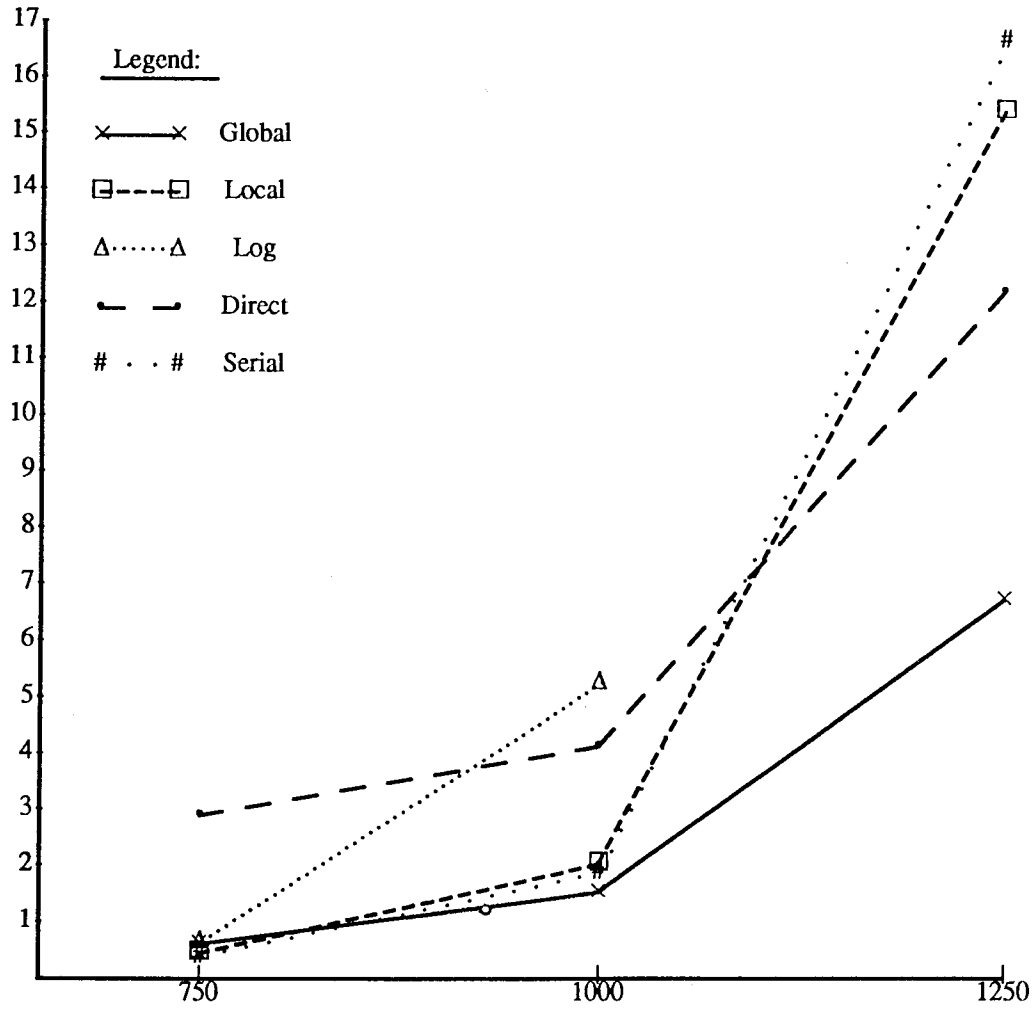

Figure 4-10: Combined Costs - domain size 1000 - 8 machines

copy of the result relation into hash tables. Both of these tables are quite simple, and thus support efficient insertion. The Direct algorithm inserts *two copies* of the result relation into hash tables, one for successor lists and one for predecessor lists. One of the tables is of the complicated and hence expensive type C. In both serial algorithms, total costs are dominated by hash table insertion.

The Direct algorithm has lately been touted as up to an order of magnitude faster than the Delta-Wavefront. But those costs estimates are based on *disk access costs*. Since we have adopted the main memory model, we ignore disk access.

Most interesting is the relative performance of the Global and Local Wavefront algorithms. It came as something of a surprise that for *all* sample sets, the Global Wavefront algorithm outperformed the Local. (In

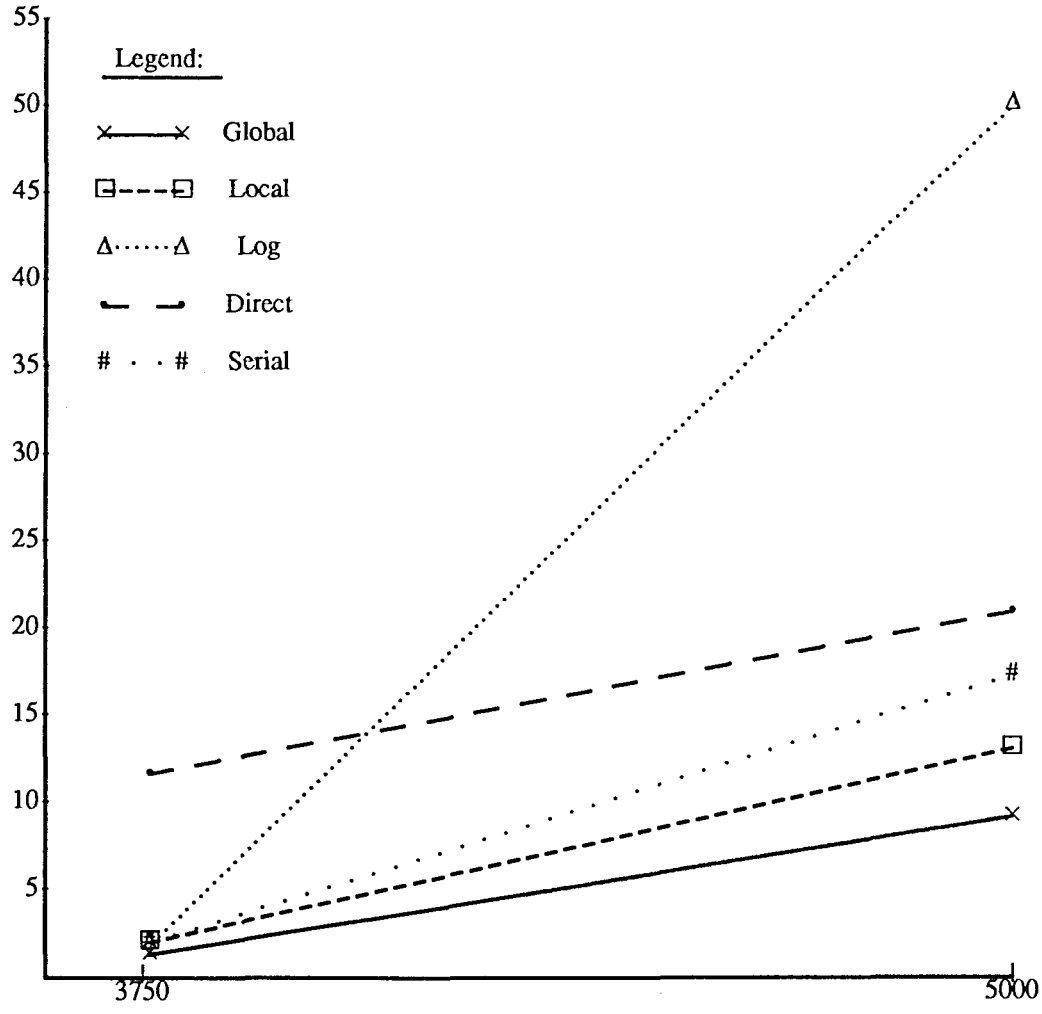

Figure 4-11: Combined Costs - domain size 5000 - 4 machines

all cases except one, the Global Wavefront had the lowest total cost. The exception is the sample set (750,1000), for which the *serial* Delta-Wavefront algorithm outperformed all others.) The key factor here, alluded to above, is the tradeoff between local processing and communication. By exchanging intermediate results, the Global method can maintain sharp partition boundaries between the participating machines. The Local method avoids that data exchange cost, but cannot ensure that local results are disjoint. This complicates the result collection process, necessitating a *very expensive* merge. Though there are some local processing costs attendant to synchronizing the data (eg. partitioning intermediate relations), the bulk of that cost is directly attributable to communication. The effect of the tradeoff thus depends on the relative costs of processing and communication. Faster connection technology would favour the Global method, while faster processers would tend to improve the relative performance of the Local algorithm.

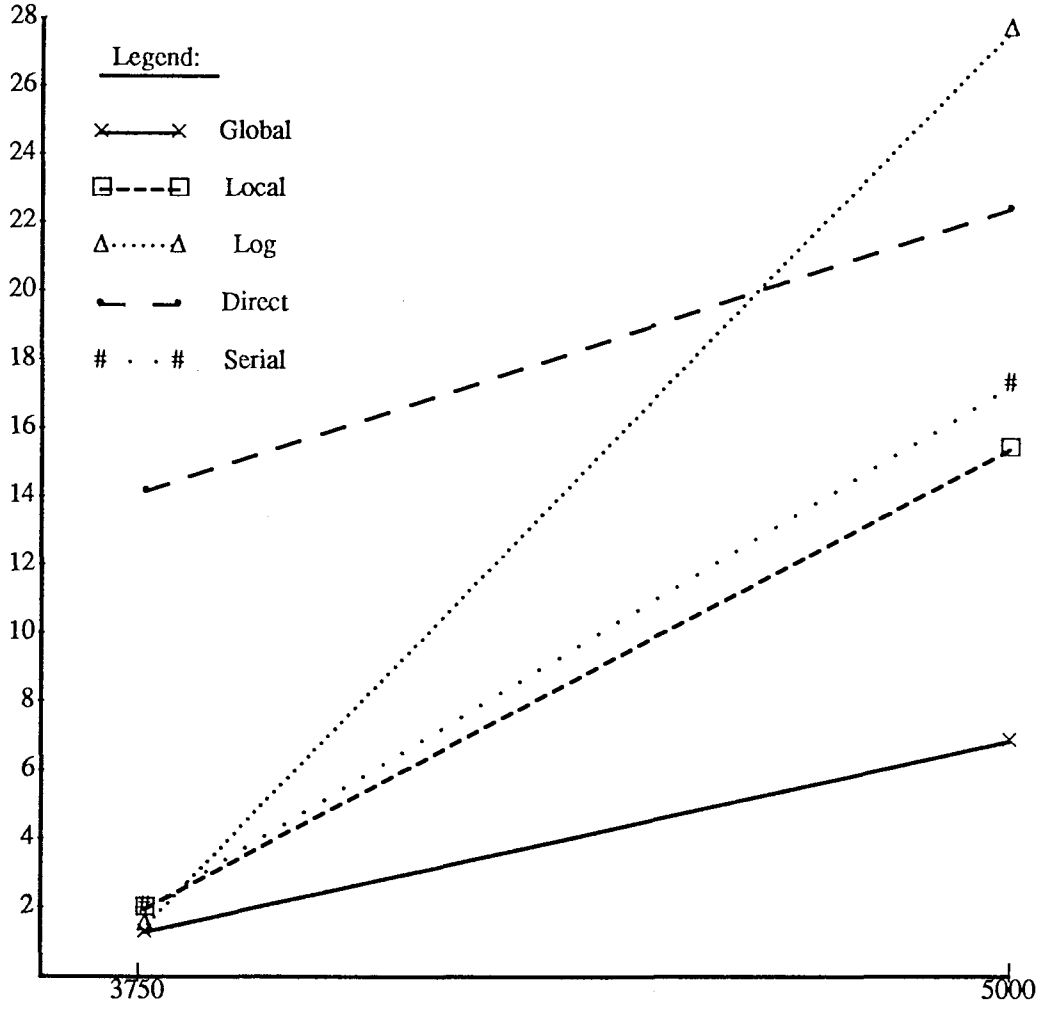

Figure 4-12: Combined Costs - domain size 5000 - 8 machines

In designing coarse-grained distributed algorithms, partitioning is a widely-applicable technique. *Strict* partitioning certainly has its advantages. In the particular cases of the Global and Local Wavefront algorithms, tested on the particular equipment that we used, the evidence favours strict partitioning. For the Wavefront algorithm, that partitioning is quite easy to enforce. Consider, though, the case of the Logarithmic algorithm. Here, keeping the data sychronized (ie. keeping local data within *globally* defined limits) is both more complicated and more costly. *Two* copies of each A^{2^i} must be exchanged, along with the very large relation D_i. (Note that this sychronization is unavoidable - if each iteration's results are not exchanged, the algorithm may not produce complete results.) The lesson here is that one should strive for a model which lends itself to strict data partitioning.

What the partitioning issue is symptomatic of is the need, in any distributed solution, to maintain a certain amount of global state information. Maintaining strict partitioning amounts to enforcing a *global* invariant condition. In the Global Wavefront algorithm, the support cost for maintaining that invariant consists of exchanging *just one* relation. In the Logarithmic algorithm, *three* relations must be exchanged. The Logarithmic algorithm exploits more complex relationships between data entities. This complexity works against it when that algorithm is distributed.[viii] A general heuristic can be gleaned from this case. A design objective in *any* distributed context should, I propose, be to keep the amount of global state information required to a minimum. If the basic structure (eg. a serial progenitor of the algorithm) involves complicated relationships between entities, it is likely that considerable global state will have to be maintained, in order to keep track of those entities. This consideration will *tend* to favour simple, "naive" models over complicated, "sophisticated" ones.

4.4.2. Speedup and Scaling

The appropriateness of an algorithm for distribution is best indicated by its speedup. The higher the speedup, the better that algorithm distributes. I define the "speedup" of a distributed algorithm over its serial version as follows:

$$\frac{\textit{cost for serial algorithm}}{\textit{cost for distributed algorithm}}$$

Table E-5 in the appendix shows the speedup for each of the distributed algorithms. The number in parentheses after the algorithm name indicates the number of machines participating in the computation.

The distributed version can take considerably *longer* than the serial one. (This is indicated in the table by a value less than 1.) This is especially true of the Direct algorithm. Except for very dense graphs, the communication cost incurred in distribution is not regained by the benefits of load sharing. The Local Wavefront algorithm also shows poor speedup. This results not from *communication* costs, but from the cost of the *merge*, which does not have to be done in the serial case.

The Global Wavefront algorithm's speedup increases as the result relation size increases (also as graph density rises). For small outputs, the total cost of this algorithm is dominated by communication. The benefits of load sharing are outweighed by this communication cost.

For all sample sets, the Logarithmic algorithm achieves rather good speedup. From this, one might surmise

---

[viii]Note, though, that the distributed Logarithmic algorithm achieves greater speedup over its serial counterpart than is the case for either the Delta-Wavefront or Direct algorithms. The reasons for this will be discussed in the following subsection.

that it is especially amenable to distribution. The main reason for the speedup, however, is that this algorithm's costs are dominated by local processing - distribution allows the local processing to be done concurrently. Local processing costs dominate because the intermediate relations are so large - they contain a lot of duplicates. Of course, those duplicates are also *communicated.* But recall that communication cost per tuple falls rapidly (up to about 8 Mbytes) as the message size increases. Thus as intermediate relations get larger, the relative cost of communication drops.

This last observation is applicable to *all* of the distributed algorithms. As the computation size grows, communication costs per tuple fall quickly, while local processing costs remain virtually constant. Thus the benefits of load sharing come to outweigh the cost of communication.

Many of the observations regarding speedup apply to scaling as well. The main penalty paid when using more machines is that the the data are sent in the form of more, smaller messages, rather than fewer, larger ones. Communication costs per tuple fall very rapidly up to about 8K bytes, then level off. Against this penalty, however, there is an advantage to exchanging data between more machines. This is that more communication concurrency can be achieved. The cost saving due to communication concurrency rises quadratically with the number of machines participating in the data exchange.

Let:

m = global size of intermediate relation to be exchanged

p = the number of machines

s = expected cost of exchange step

t = cost/tuple for message of size $\frac{m}{p^2}$

then

$$s = 2(p-1)t\ \frac{m}{p^2}$$

As $\frac{m}{p^2}$ gets larger, the cost per tuple rises more slowly. At some point, intermediate relations will be large enough that dividing them into more messages adds less to cost than is subtracted by the effects of more concurrency.

When the intermediate relation is large enough, subdividing it between more machines does not, then, result in higher communication costs per tuple. When the exchange step(s) sufficiently dominate the other,

more sequential communication steps (initial distribution and result collection), the *total* communication costs can *drop* as more machines are used. This occurs with the Logarithmic algorithm, for sample sets (3750,5000) and (5000,5000).

The Global Wavefront algorithm scales well for dense graphs. In such a case, local processing costs dominate total processing, the exchange step dominates communication, and the intermediate relations exchanged are large. The Local Wavefront method does not scale well for *any* of our sample sets. The reason is that the merge is more expensive for more machines. Note also that the exchange step is only executed once for this algorithm.

Since the V-System's multicast operation scales well, we might also expect the Direct algorithm to scale well. The basic cost for broadcasting a message (ie. the cost for the minimum-sized message) will grow quite slowly as the number of receivers is increased. The messages will, of course, remain the same size - the current successor list's size is independent of the number of machines. This algorithm generally scales poorly because its cost is always, for the sample sets tested, dominated by communication. Having more machines never *decreases* communication cost, since the exchange step in only executed once, at the start of the algorithm. For very dense graphs (eg. (1250,1000), the algorithm both distributes and scales well.

Chapter 5
Conclusions, Other Results and Research Directions

In the first section, I summarize our results in terms of a three-way interplay between distributing global state, communication and local processing. I also suggest some general heuristics for coarse-grained distributed computation. In the second section, I compare our results with results from the literature which seem to contradict them. In the third, I suggest directions for future research.

5.1. Conclusions

This research follows work done by Wang at Simon Fraser University. [16]Wang's research focused on the processing requirements of the join methods, as well as on the trade-off between local processing and communication. While that trade-off is important to this research as well, we also gave considerable attention to the design of data structures, and to the distribution of *global state*.

When joining and sorting are distributed, the exchange of *intermediate* results is relatively straightforward. The same cannot be said of transitive closure. A machine may be unable to complete processing of its initial data partition without receiving tuples generated by other machines - matching join inputs must be brought together.[i]Various solutions to this "matching" problem may be achieved, depending on how much *global state* is made available locally. The need to distribute global state is compounded by the fact that the computation of transitive closure can, unlike joining or sorting, generate huge numbers of duplicates.

What emerges is a *three-way* interplay between *local processing*, *communication*, and *availablility of global state information*. If local partitions are kept up-to-date by frequent exchange between the machines, then more duplicates can be detected, and redundant processing thereby avoided. But communicating that state adds to communication costs. Having more global state available will generally decrease local processing, since less duplicates will have to be carried. It might seem that global state availablility is just an

[i]Compare this with a distributed merge-sort. In that algorithm, a machine can at least sort whatever tuples are locally resident without consulting the others. Communication is required only to merge local results together.

aspect of the processing/communication tradeoff. The following considerations, however, count against that characterization:

- Some algorithms, by the nature of their underlying structures, require more replication of global state than do others. The Delta-Wavefront is well-suited to distribution largely *because* it requires a minimum of state exchange. In fact, that algorithm can be distributed in such a way that *no* intermediate results need be exchanged - this is exemplified by the Local Wavefront algorithm. The Direct algorithm suffers because it *must* communicate global state that only be derived sequentially.
- Even when global state *is* locally available, the cost of retrieving and using it may outweigh the benefit. In the Local Wavefront algorithm, for instance, the entire source relation is replicated on each machine. It is thus *possible* to detect any locally-generated tuples that are already in the *global* source relation. To support this, however, entails extra hash-table insertions. It turned out, for all the data sets that we tested, that the number of extra duplicates removed did not "pay for" the cost required to detect them. Thus *using* global state can actually *add to* net local processing costs.
- While exchanging state between machines obviously requires communication, having global state available locally can substantially *decrease* communication costs. Because the Local Wavefront algorithm replicates the source relation on each machine, intermediate results need not be exchanged.

Our most stiking discovery was the huge effect that the presence of duplicate tuples can have on performance. This stands out clearly in the abysmal performance of the Logarithmic algorithm. As the input graph's density increases, the number of duplicates that this algorithm must carry explodes - in one case to the point of overwhelming our computing resources. Both local processing and communication costs are, in this algorithm, dominated by the presence of duplicates.

The Direct algorithm also suffers, though much less, from generating a large number of duplicates. In light of this, its local processing costs are surprisingly low. Because it can detect and remove each duplicate very cheaply, the duplicate problem does not prove extreme.

In the Delta-Wavefront algorithm, one of the join input relations is unchanged throughout the course of the algorithm. Because of this, any tuple which has already driven the join during a previous iteration can be removed from subsequent join driver relations. This removal vastly reduces the number of duplicates that are carried by the computation. Because it exchanges *current* (as opposed to just initial) global state, the Global Wavefront algorithm can eliminate more duplicates than can the Local Wavefront method. While the Local Wavefront is not optimal with respect to duplicate removal, however, it is not terribly plagued by duplicates. The main reason for its poor performance, relative to the Global Wavefront's, is the need to merge local results.

The performance effects of duplicates cannot be captured merely by counting them. Their presence can

considerably complicate the contruction of hash tables used to implement the join, union and difference operations. Because they need to support duplicate detection, the tables used in the Logarithmic and Direct algorithms are considerably more complex than are those used in the Wavefront algorithms. This complexity adds to the overhead of hash table operations. In short, the duplicate problem is compounded by side-effects that appear at the implementation level of abstraction.

The poor performance of the Direct algorithm is attributable mainly to its high communication costs. The fundamental structure of the algorithm imposes severe constraints on the amount of concurrency that can be achieved. An iteration is required for each vertex in the input graph - at the start of each iteration, a successor list must be broadcast. Except for very small relations on small domains, this results in the sending of a very large number of small messages. The running time of the algorithm comes to be dominated by this message sending. Although this algorithm may be quite suitable for the serial case, especially if the data will not fit in main memory, it is it is not very amenable to coarse-grained distribution.

Of the four algorithms tested, the Global Wavefront method emerges almost unequivocally as the winner. In both local processing and combined cost, it is the most efficient for all data sets tested. Its serial progenitor, the Delta-Wavefront algorithms, consistently outperforms the serial Logarithmic and Direct algorithms. I conclude that the Global Wavefront algorithm's good performance relative to the others stems mainly from the relative efficiency of the underlying serial algorithm on which it is based, rather than any special amenability to distribution. That the Global outperforms the Local Wavefront algorithm is, of course, due to its distributing more efficiently.

Not surprisingly, the Local Wavefront algorithm always sustains the lowest communication cost. The shortcoming of this algorithm is that it produces local results which overlap with each other. Result collection thus involves merging. The merge cost accounts for about half of total local processing costs.

From this case study emerge some heuristics which seem applicable to coarse-grained distributed processing in general.

In the serial case, structural characteristics of the data can often be exploited to advantage. But reliance on such structure can make it harder to distribute the algorithm. For example, the Logarithmic algorithm relies on the fact that its join drivers are at some particular level (a power of 2). To ensure that all such tuples which can drive each other do so, *three copies* of the driving relation have to be exchanged each iteration. The Delta-Wavefront algorithm, which does not rely on such specific structural properties, can be distributed

without the need for exchanging *any* intermediate results. (This option is instantiated in the Local Wavefront algorithm.)

A related condition which proves problematic is reliance on complicated relationships between a multiplicity of data objects. We simply have more objects to keep track of. On a serial model, this is relatively cheap. On a distributed one, communication between objects usually means communication *between machines.*

These matters point to a more general concern. The more global state each machine requires, the more overhead is incurred. Complicated algorithms will generally require the maintainance of complicated state. In distributing them, efficiency will tend to be impaired by the need to communicate that state.

Partitioning on hash values suggests itself as an attractive method of parcelling work out between machines. (This is an example of a serial technique, used to optimize disk access, which *does* translate well to the distributed environment.) Maintaining strict partition boundaries as an invariant condition yields an important benefit - it vastly simplifies the collection of results. It is primarily for this reason that the Global Wavefront algorithm outperforms the Local Wavefront. There is, of course, a cost to maintaining this invariant. Intermediate results must be exchanged. In other words, more global state must be made available to the individual participants. Comparing the Local and Global Wavefront algorithms, we see a tradeoff between maintaining global state and local processing. This manifests itself as a tradeoff between communication and local processing.

A rather obvious general dictum for coarse-grained distribution is that the granularity of data distribution should indeed be coarse. Small messages are much more costly per byte than are large ones. In addition, more messages require more synchronization points, resulting in less concurrency.

Clearly, the heuristics I suggest are not always mutually compatible. The Global Wavefront algorithm must communicate more global state than must the Local version. The Local Wavefront's communication consists of fewer messages, and no small ones. Yet the Global method outperforms the Local. In cases of *tradeoff* between global state replication, communication and local processing, it may be difficult to estimate a priori which choices will be best. In such cases, one resorts to experiment.

5.2. Other Results in the Literature

For the *serial* case, results from Lu [11] indicate that the Delta-Wavefront algorithm[ii] performs considerably *worse* than the Logarithmic. This result is confirmed by Ioannidis [9]. Again for the serial case, Agrawal and Jagadish [1] give results indicating that the Direct method (a version of Warren's algorithm) *substantially* outperforms the Logarithmic. It would seem then that a forteriori the Direct outperforms the Delta-Wavefront.

In view of these results, ours must seem surprising indeed! According to our results for both the serial and the distributed cases, versions of the Delta-Wavefront algorithm turn out to be the *most* efficient. The Global Wavefront had the lowest costs for all samples tested, except for the (750,1000) sample set, where the serial Delta-Wavefront algorithm triumphed. The second-best overall performance was achieved by the Local Wavefront algorithm. For small computation sizes, however, the serial Delta-Wavefront algorithm actually achieved a shorter running time. Actually, the serial Delta-Wavefront compares even better than the figures in tables E-3 and E-2 indicate. The costs for the distributed algorithms are based on the unrealistic assumption of perfect load balancing - load balancing is not an issue for a serial algorithm.

The main reason for the seemingly contradictory results has to do with how costs are accounted. The other studies account for complexity either mainly or exclusively in terms of *disk access costs.* In terms of disk access, it is not surprising that the Logarithmic algorithm outperforms the Delta-Wavefront method. Lu says, "Intuitively, the source relations are only read from the disks once in one iteration. The more tuples generated in one iteration, the fewer number of iteratios needed to complete the computation." [11] The Logarithmic algorithm certainly uses fewer iterations.

Lu's method of accounting does not reveal the deleterious effects of duplicates in the Logarithmic algorithm. "Furthermore," he writes, "we assume that duplication elimination costs are the same for all three algorithms, and they are not taken into account." [11] If we only charge for disk access, this assumption is perfectly sound - duplicates would *not* be written to disk. The cost of duplicates shows up in *cpu time*, which Lu does not charge for at all. Considering that disk access costs unit costs are several orders of magnitude higher than cpu costs, failing to account for cpu costs is justifiable, *if we assume that main memory is scarce.* In this study, we have assumed a *main memory* model - we do not charge for disk access at all! This makes it hard to compare our results with Lu's (or with Ioannidis'). What we *can* conclude is that cheaper memory will favour the serial Delta-Wavefront algorithm over the Logarithmic method.

[ii]Lu calls it the Semi-Naive algorithm

Lu estimates the problem size by counting join *inputs*. The running time (cpu cost) of the Logarithmic algorithm is, for dense graphs, dominated by join *output* sizes. It is because those outputs contain so many duplicates that the algorithm fairs so poorly. (The Direct algorithm also carries suffers from this duplicate problem, though less so.) Our simulations count *both* inputs and outputs.

The serial Direct algorithms suggested by Agrawal and Jagadish are very efficient in minimizing disk access. They too estimate cost in terms of disk access - their consideration of cpu cost is very abstract. According to our estimates, the Direct algorithm is slower than the Delta-Wavefront even in the *serial* case. There are two reasons for this. The first is that the Direct method generates more duplicates. The second has to do with implementation details. In the Direct algorithm, two copies of the output relation are inserted into hash tables. In the Delta-Wavefront, one copy of the source and *just one* copy of the result are inserted into tables. Hash table insertion is expensive. The cost for the Direct method is compounded by the fact that it uses more complicated table structures.

It might seem that considerations of disk access for the serial case would carry over to communication in the distributed case. For a variety of reasons discussed in this thesis, the cases differ. But there is a most salient difference between disk traffic and communication that I have *not* mentioned. And this is the most important difference of them all. Sending tuples across a LAN is much cheaper than writing them to (or reading them from) a disk! Communication costs are on the same order as cpu costs. In fact, it is often cheaper to send a tuple to another site for processing than it is to process it locally. If this were not so, the tradeoff between local processing and communication would hardly be worth investigating. If the data will not fit into main memory, a lot of processing can be justified to avoid disk access. In the main memory distributed case, extra processing to avoid communication is much less attractive.

Lu suggests a "hybrid" of the Delta-Wavefront, employing level relaxation and source relation reduction. His results indicate that this hybrid algorithm outperforms the Logarithmic or the unmodified Delta-Wavefront (semi-naive), at least in terms of disk access. As I mentioned in Chapter 2, level relaxation seems well worth investigating for the distributed main memory context as well. Source relation reduction involves quite a lot of processing overhead. If efficiency is estimated in terms of *cpu* time, this extra overhead will seriously impair performance. The only *processing* advantage gained is that the loading factor of the join hash table will be reduced, resulting in slightly less costly list lookups. Our tests indicate that those lookups do not contribute a great deal to total running time - table *insertions* are much more costly. Accordingly, I doubt that source relation reduction will be worth the extra processing cost. (If the join relations have to be written to disk, reducing their size undoubtedly *will* significantly improve performance. Again, we note that disk access is, relative to processing, very expensive.)

Ioannidis and Ramakrishnan have proposed a method of computing transitive closure based on a depth first search. [10] Their study is for the *serial* case. The method is *very* good with respect to detecting and removing duplicates. This leads me to think that for the serial case, the approach has much promise, even for a main memory model. They say, "A preliminary study of the potential for parallel execution and performance improvements through good blocking and paging heuristics shows much promise. However, as we progress from *Basic_TC* to the more sophisticated algorithms, it appears that there is a tradeoff between these factors and the benefits of a strictly depth-first search order." [10] Any new serial algorithm for TC should at least be considered as a candidate for distribution. However, depth-first search does not easily lend itself to concurrent processing. In addition, reliance on topological features of the input graph can be expected to complicate the task of distributing the algorithm - there is more global state to keep track of. I suggest that this depth-first approach may be more appropriate for a *tightly coupled* environment, where such techniques as parallel prefix computation can be efficiently implemented.

5.3. Suggestions for Future Research

It would seem that expected join result sizes could, given input relation size, domain size, and a uniform distribution, be easily predicted by a combinatoric or probabilistic study. The case proved resistant to my efforts. I think that the problem is interesting enough to warrant more research.

The Global Wavefront algorithm's performance might be improved if it allowed some level relaxation. A higher degree of level relaxation could be obtained if the exchange step employed asynchronous communication.

While we are confident in the soundness of our methodology, it would be reassuring to do some experiments to confirm them. We might also reanalyze the studies on joining and sorting, using this methodology, and then to compare the analytical with the empirical results. This more general application of the methodology is in itself an interesting research topic.

Appendix A
The Distributed Total Closure Algorithms

In this section I present the distributed algorithms which we ultimately simulated. The pseudo-code used to describe them is perhaps a bit cryptic. Specifics of implementation are discussed in the chapter 3, "Implementation Issues". Comprehension can be abetted by reading that chapter and the motivation provided in chapter 2, section 2 ("The Distributed Context").

The "Exchange" step, which occurs at least once in each algorithm, includes a union operation. As it receives fragments from other machines, each host unions them together. Sometimes simple concatenation suffices. In other cases, two or more different fragments may include copies of the same tuple. In this case, a true union is required. These cases are distinguished, for each exchange step of each algorithm, in chapter 4, "Analysis". How the other steps are implemented can be gleaned from in chapter 3, "Implementation Issues".

In each algorithm, the source (input) relation is denoted by "A", and local result relations by C_i. The other variables denote intermediate relations or relation fragments. Their meaning should be clear from context.

A.0.1. Algorithm 1: Global Wavefront

1. **Distribution:**

   Divide A into p equal-sized subrelations, and distribute A_i to machine i, for $1 \leq i \leq p$.

2. FOR EACH MACHINE i, DO THE FOLLOWING:

   a. **Partition:**

      Partition A_i into p parts, i.e., $P_{i,j}$, $1 \leq j \leq p$, by hashing on the 1st column of A_i.

      Partition A_i into p parts, i.e., $Q_{i,j}$, $1 \leq j \leq p$, by hashing on the 2nd column of A_i.

   b. **Exchange:**

      Each $P_{i,j}$ and $Q_{i,j}$ is sent from machine i to machine j, for $1 \leq i,j \leq p$.

      $A_i = \cup P_{j,i}$, $1 \leq j \leq p$

      $\delta A_i = \cup Q_{j,i}$, $1 \leq j \leq p$

   c. **Build Local Hash Tables:**

      Using δA_i from step 2a as input, build a hash table for the result relation, C_i.

      Also build a hash table for the local fragment of the source relation, A_i.

   d. WHILE (δA_i not all null) DO THE FOLLOWING :

      i. **Join and Partition:**

         $\Delta A_i = \delta A_i \, join \, A_i$

         Implemented by looping through δA_i, hashing each member into A_i. ΔA_i is partitioned into p parts, $Q_{i,j}$, $1 \leq j \leq p$, by hashing ΔA_i on its 2nd column, as in step 2a. Partitioning is done on the fly, as each join output tuple is generated.

      ii. **Exchange:**

         Each $Q_{i,j}$ is sent from machine i to machine j, as in step 2a.

         $\Delta A_i = \cup Q_{j,i}$, $1 \leq j \leq p$

      iii. **Union and Set Difference:**

         $\delta A_i = \Delta A_i - C_i$

         $C_i = C_i \cup \delta A_i$

3. **Result Collection:**

   Each machine i, $2 \leq i \leq p$ sends C_i to machine 1, the Controller, which concatenates the C_i's together.

A.0.2. Algorithm 2: Local Wavefront

1. **Distribution:** Send the entire source relation A to each machine.

2. FOR EACH MACHINE i, DO THE FOLLOWING:

   a. **Partition:**

   Choose a segment A_i of A, $1 \leq i \leq p$, where p is the number of machines. Each machine gets a disjoint segment, and all segments are within one tuple of being equal in size. Using a uniform hash function, partition segment A_i into p fragments,$P_{i,j}$, $1 \leq j \leq p$.

   b. **Exchange:**

   Each $P_{i,j}$ is sent from machine i to machine j.

   $\delta A_i = \cup P_{j,i}$, $1 \leq j, i \leq p$.

   c. **Build Local Hash Tables:**

   $C_i = \delta A_i$

   Set up a join hash table (A) and a local result collection hash table (C_i). All of the source relation (A) is installed into A. Only the *local fragment* of A (δA_i) is installed into C_i.

   d. **Local Closure:**

   WHILE (δA_i not null), do the following:

   i. *Local Join*:

   $\Delta A_i = \delta A_i \, join \, A$

   ii. *Local Union and Set Difference*:

   $\delta A_i = \delta A_i - \Delta A_i$

   $C_i = C \cup \delta A_i$

3. **Result Collection:**

   Each machine i, $2 \leq i \leq p$ sends C_i to machine 1, the Controller. For each incoming fragment, sequentially, the Controller merges that fragment into the combined result C, by inserting it into a hash table, filtering out duplicates. Once merging is complete, the contents of that table are copied into the result relation.

A.0.3. Algorithm 3: Logarithmic Method

1. **Distribution:**

   Divide A into p equal-sized subrelations, and distribute A_i to machine i, for $1 \leq i \leq p$.

2. FOR EACH MACHINE i, DO THE FOLLOWING:

   a. **Partition:**

   Partition A_i into p parts, i.e., $P_{i,j}$, $1 \leq j \leq p$, by hashing on the 1st column of A_i.

   Partition A_i into p parts, i.e., $Q_{i,j}$, $1 \leq j \leq p$, by hashing on the 2nd column of A_i.

   b. **Exchange:**

   Each $P_{i,j}$ and $Q_{i,j}$ is sent from machine i to machine j, for $1 \leq i,j \leq p$.

   $P_i = \cup P_{j,i}$, $1 \leq j \leq p$

   $Q_i = \cup Q_{j,i}$, $1 \leq j \leq p$

   c. **Initialize Hash Tables:**

   Set up a hash table for P_i and for the closure relation, C_i.

   FOR EACH tuple $t \in P_i$ DO THE FOLLOWING:

   - insert t into table for P_i

   d. REPEAT THE FOLLOWING :

   i. **Union:**

   $C_i = C_i \cup P_i$

   FOR EACH tuple $t \in P_i$ DO THE FOLLOWING :

   - insert t into table for C_i

   ii. **Join and Partition:**

   $R_i = P_i \, join \, Q_i$

   Implemented by looping through Q_i, hashing each member into into P_i. R_i is partitioned into two separate output relations, $P_{i,j}$ and $Q_{i,j}$, $1 \leq i,j \leq p$. $P_{i,j}$ is formed by hashing R_i on its 2nd column, and $Q_{i,j}$ by hashing it on the 1st. Partitioning and copying are done on the fly, as each join output tuple is generated.

   iii. **Exchange:**

   Each $P_{i,j}$ and $Q_{i,j}$ is sent from machine i to machine j, as in step 2b.

   iv. **Build Join Table and Filter:**

   $P_i = \cup P_{j,i}$, $1 \leq j \leq p$

   $Q_i = \cup Q_{j,i}$, $1 \leq j \leq p$

FOR EACH tuple $t \in P_i$ DO THE FOLLOWING:

- insert t into table for P_i

FOR EACH tuple $u \in Q_i$ DO THE FOLLOWING:

- insert u into table for Q_i
- if u not a duplicate, copy to output version of Q_i

v. **Join and Partition:**

$\delta D_i = C_i join\ Q_i$

Implemented by looping through Q_i, hashing each member into into C_i. δD_i is partitioned into p parts, $\delta D_{i,j}$, $1 \le j \le p$, by hashing δD_i on its 2nd column.

vi. **Exchange:**

Each $\delta D_{i,j}$ is sent from machine i to machine j.

$D_i = \cup \delta D_{j,i}$, $1 \le j \le p$

vii. **Union:**

$C_i = C_i \cup D_i$

Implemented by looping through D_i, inserting each tuple into the table for C_i. Unioning removes duplicates.

UNTIL (D_i is null at all machines)

3. **Result Collection:**

Each machine i, $2 \le i \le p$ sends C_i to machine *1*, the Controller, which concatenates the C_i's together.

A.0.4. Algorithm 4: Direct Method

1. **Distribution:**

   Divide A into p equal-sized subrelations, and distribute A_i to machine i, for $1 \leq i \leq p$.

2. FOR EACH MACHINE i, DO THE FOLLOWING:

   a. **Partition:**

      Partition A_i into p parts, i.e., $P_{i,j}$, $1 \leq j \leq p$, by hashing on the 1st column of A_i.

   b. **Exchange:**

      Each $P_{i,j}$ is sent from machine i to machine j, for $1 \leq i,j \leq p$.

      $A_i = \cup P_{j,i}$, $1 \leq j \leq p$

   c. **Build Hash Tables:**

      Build Table 1, predecessor lists, from A_i, by hashing on the 2nd column.

      Build Table 2, successor lists, from A_i, by hashing on the 1st column.

3. FOR EACH unique column 1 value v_i, $1 \leq i \leq v$:

   a. **Distribution:**

      v_i's host machine broadcasts v_i and its successor list, $\{\text{succ}(v_i)\}$.

   b. FOR EACH MACHINE h, DO IN PARALLEL:

      FOR EACH $v_j \in \{pred(v_i)\}$ resident on machine h:

         FOR EACH $v_k \in \{succ(v_i)\}$:

            i. $\{\text{succ}(v_j)\} = \{\text{succ}(v_j)\} \cup \{v_k\}$

            ii. $\{pred(v_k)\} = \{pred(v_k)\} \cup \{v_j\}$

4. **Result Collection:**

   Each machine i, $2 \leq i \leq p$ sends C_i to machine 1, the Controller, which concatenates the C_i's together.

Appendix B
Cardinalities

By "cardinalities", I mean "numbers of units". The specific units denoted are detailed in the key beneath table B-1. The key also cross-indexes the names used in the table with the corresponding ones used in the analysis. Cardinalities referred to in the analysis which are *not* listed can be derived from those which are. Those derivations can be gleaned from the equations preceding each algorithm in the step-by-step analysis (Chapter 4). All values in this table were obtained by simulating the algorithm concerned. For each algorithm and relation size, ten tests were run, each with different randomly-generated (uniform) data. The values shown are averages over the ten runs.

source relation size (tuples)		750	1000	1250	3750	5000
ALGO-RITHM	VARI-ABLE	key domain size 1000			key domain size 5000	
GLOBAL WAVE	Gi-sum	2777	19,911	173,117	15,566	158,390
	<Gi>-sum	2759	18,810	138,647	15,537	153,227
	dups	18	1101	34,470	29	5163
	iterations	14	29	50	22	51
LOCAL WAVE	Li-sum	2785	20,845	203,197	15,575	161,938
	<Li>-sum	2770	19,739	163,007	15,551	156,799
	dups	26	2035	64,550	38	8711
	iterations	14	29	50	22	51
LOGARITHMIC	A2i-sum	1021	32,959	n/a	5163	90,768
	<A2i>-sum	1021	13,434	n/a	5163	60,550
	Di-sum	1296	221,052	n/a	7199	1,770,597
	<Di>-sum	1245	39,745	n/a	7087	293,271
	ast-<A2i>	27	5511	n/a	28	26,855
	dups	51	200,832	n/a	112	1,507,544
	iterations	4	5	n/a	5	6
DIRECT	succ-sum	1364	3460	17,027	7063	20,769
	mi-sum	700	2373	11,978	3707	12,961
	last-succ	4	8	248	12	87
	dups	29	8036	977,633	58	71,061
	iterations	525	631	715	2635	3161

Table B-1: Cardinalities

Key

Gi-sum ::= $\sum_{i=0}^{r} |G_i|$

<Gi>-sum ::= $\sum_{i=0}^{r} |G_i'| = |C|$

Li-sum ::= $\sum_{i=0}^{r} |L_i|$

<Li>-sum ::= $\sum_{i=0}^{r} |L_i'|$

A2i-sum ::= $\sum_{i=1}^{\lfloor log_2 r \rfloor} |A^{2^i}|$

<A2i>-sum ::= $\sum_{i=1}^{\lfloor log_2 r \rfloor} |\overline{A^{2^i}}|$

Di-sum ::= $\sum_{i=1}^{\lfloor log_2 r \rfloor} |D_i|$

<Di>-sum ::= $\sum_{i=1}^{\lfloor log_2 r \rfloor} |D_i'|$

last-<A2i> ::= $|\overline{A^{2^{\lfloor log_2 r \rfloor}}}|$

Values for each of the variables just listed indicate number of *tuples*.

succ-sum ::= $\sum_{i=1}^{w} |\{succ(v_i)\}|$

A different successor list is broadcast at the start of each iteration. The formula above sums them.

last-succ ::= $|\{succ(v_{w-1})\}|$

Values for succ-sum and last-succ indicate the number of *elements* (ie. vertices) in successor lists.

mi-sum ::= $\sum_{i=1}^{w} |m_i|$

Values for mi-sum indicate the number of *matches* found.

iterations ::= the number of iterations that the algorithm takes to complete

dups ::= the number of duplicate tuples that the algorithm processes.

For the Global Wavefront algorithm, this is just Gi-sum - <Gi>-sum - all duplicates can be detected locally.

For the Local Wavefront algorithm, it is Li-sum - <Gi>-sum; not all duplicates generated can be detected locally. The number of duplicates that *can* be detected locally is found by Li-sum - <Li>-sum.

For the Logarithmic algorithm, dups = (A2i-sum - <A2i>-sum) + (Di-sum - <Di>-sum). This just includes *internal* duplicates - even the *serial* version cannot remove external duplicates. Note that <A2i>-sum + <Di>-sum > |C|. The difference is attributable to external duplicates.

In the Direct algorithm, all duplicates generated *are* detected locally.

	key domain size 1000			key domain size 5000	
relation size (tuples)	750	1000	1250	3750	5000
Gi-sum (min)	2343	7329	93,611	13,251	65,381
Gi-sum (max)	3584	38,099	242,785	21,621	335,831
<Gi>-sum (min)	2330	7205	80,326	13,243	64,768
<Gi>-sum (max)	3573	36,121	190,471	21,581	314,185
iterations (min)	10	17	41	13	37
iterations (max)	18	47	63	41	72

Table B-2: Cardinality Ranges - Global Wavefront

	key domain size 1000			key domain size 5000	
relation size (tuples)	750	1000	1250	3750	5000
Li-sum (min)	2343	7414	104,615	13,256	65,723
Li-sum (max)	3586	40,372	295,128	21,624	347,308
<Li>-sum (min)	2331	7307	90,016	13,253	65,168
<Li>-sum (max)	3577	38,396	231,891	21,586	325,222
iterations (min)	10	17	41	13	37
iterations (max)	18	47	63	41	72

figures are for 4 participating machines

Table B-3: Cardinality Ranges - Local Wavefront

	key domain size 1000			key domain size 5000	
relation size (tuples)	750	1000	1250	3750	5000
A2i-sum (min)	865	2399	n/a	4609	16,447
A2i-sum (max)	1353	191,938	n/a	5922	219,230
<A2i>-sum (min)	864	2390	n/a	4609	16,440
<A2i>-sum (max)	1353	44,226	n/a	5919	157,634
Di-sum (min)	813	4579	n/a	4948	48,143
Di-sum (max)	2871	1,243,564	n/a	11,925	8,193,062
<Di>-sum (min)	801	4431	n/a	4947	46,400
<Di>-sum (max)	2664	96,126	n/a	11,922	783,683
iterations (min)	4	5	n/a	4	6
iterations (max)	5	6	n/a	6	7

Table B-4: Cardinality Ranges - Logarithmic

	key domain size 1000			key domain size 5000	
relation size (tuples)	750	1000	1250	3750	5000
succ-sum (min)	1268	2398	10,039	6506	15,341
succ-sum (max)	1527	5453	29,131	7642	25,441
mi-sum (min)	601	1424	4738	3408	9093
mi-sum (max)	869	4344	19,731	4060	17,291
dups (min)	5	202	91,256	2	1104
dups (max)	64	35,165	2,891,676	288	293,655
iterations (min)	514	614	693	2614	3131
iterations (max)	544	645	731	2660	3184

Table B-5: Cardinality Ranges - Direct

Appendix C
Local Processing Unit Costs

table load	basic local processing operation					
	JTI	RTI	RTI dups	JTID	JTID dups	JTL
1,000	.098	.075	.042	.149	.045	.018
5,000	.091	.071	.042	.131	.052	.024
10,000	.087	.078	.046	.118	.054	.026
25,000	.079	.135	n/a	.110	.060	.027
50,000	.077	.200	n/a	.108	.065	.027
100,000	.076	.335	n/a	.107	.072	.027

1000 buckets in A and C type hash tables
2000 buckets in B type hash table

Table C-1: Local Processing Unit Costs - Small Hash Tables

Tables C-1 and L3 show unit costs, in milliseconds/tuple, for operations on hash tables. Note that those costs can vary, depending on the number of buckets in the table. "table load" refers to the maximum number of tuples in the table. For each table load, each operation was executed 100,000 times, and the average cost per operation taken.

JTI operates only on type A tables, RTI only on type B tables, and JTID only on type C. JTL is applied to both A and C type tables. The columns headed by "RTI dups" and "JTID dups" give the costs for detecting duplicates in the type B and C tables, respectively. These are lower than the values for RTI and JTID, since no storage allocation isrequired - duplicates detected by RTI and JTID operations are simply discarded.

table load	basic local processing operation					
	JTI	RTI	RTI dups	JTID	JTID dups	JTL
1,000	.107	.072	.037	.161	.047	.017
5,000	.091	.072	.042	.131	.047	.019
10,000	.082	.076	.047	.113	.047	.019
25,000	.075	.085	.063	.097	.049	.019
50,000	.074	.099	.088	.093	.052	.020
100,000	.070	.126	.140	.095	.058	.020

5000 buckets in each hash table

Table C-2: Local Processing Unit Costs - Large Hash Tables

table load	from table A	from table B	from table C	from relation	no retr
1,000	.000021	.000017	.000029	.000014	.000006
5,000	.000018	.000013	.000027	.000015	.000009
10,000	.000016	.000013	.000022	.000015	.000010
25,000	.000015	.000013	.000019	.000015	.000009
50,000	.000015	.000012	.000017	.000016	.000009
100,000	.000016	.000012	.000015	.000015	.000009

1000 buckets in tables A and C, 2000 in table B

Table C-3: Unit Costs - COPY

Tables C-3 and C-4 give COPY costs in milliseconds/tuple. Again, each value listed was obtained by averaging 100,000 tuple copies. Since tuples are sometimes copied from hash tables, two tabulations are

table load	from table A	from table B	from table C	from relation	no retr
1,000	.000032	.000025	.000042	.000014	.000008
5,000	.000020	.000015	.000029	.000015	.000009
10,000	.000018	.000014	.000023	.000015	.000008
25,000	.000016	.000013	.000019	.000015	.000009
50,000	.000015	.000012	.000017	.000015	.000009
100,000	.000016	.000012	.000016	.000016	.000010

5000 buckets in each table

Table C-4: Unit Costs - COPY

presented. (The cost to retrieve from a hash table depends on its loading factor, which in turn depends on the number of buckets.) The "from relation" column lists costs for copying from a relation stored in stream form. The "no retr" column gives costs for copying when the tuples do not have to be retrieved - ie. they are available as separate local variables.

For the "from relation" and "no retrieval" columns, "table load" should be read as "length of (number of tuples in) stream". Not surprisingly, unit costs under these columns do not vary significantly with input size. In all cases, the PART operation either retrieves its input from a stream-structured relation, or does not require retrieval. Likewise, the "retrieve from relation" cost is constant. Costs which are constant (do not vary with the size of the input to the operation) are tabulated in C-5. Values are in ms./tuple.

OPERATION	COST
retrieve from relation	.005
COPY (no retrieval)	.009
COPY (with retrieval)	.014
PART (no retrieval)	.019
PART (with retrieval)	.024

Table C-5: Constant Local Processing Costs

Appendix D
Communication Unit Costs

message size (bytes)	32	512	1K	2K	4K	8K	16K
cost (ms.)	2	4	6	9	11	16	25

Table D-1: Sun-3 V-System IPC Timing Information

message size (tuples)	4	64	128	256	512	1024	2048
cost (ms. / tuple)	.5000	.0625	.0469	.0352	.0215	.0156	.0122

Table D-2: Sun-3 V-System Communication Costs per Tuple

Table D-1 shows, for various message sizes, the total cost of sending a message of that size. This table is taken from [12]. From those figures, I have derived the values shown in D-2. In computing communication costs, small messages (message size less than 1K bytes, or 128 tuples) are treated differently from larger ones. The *message* cost of a small message is taken from the following formula:

cost = 2 + (message-size in tuples - 4)(.033) milliseconds.

Larger messages are charged for on a per-tuple basis. The cost per tuple is found by interpolating linearly between the values in table D-2.

For broadcasting large messages, we use the same method as for point-to-point transmission (ie. we charge the same as for a point-to-point send). For small messages, however, the cost processing more

acknowledgements adds significantly to the cost per tuple. For the minimum-size message to three receivers, we charge 3.9 ms., and to seven receivers 5.0 ms.. Additionaly data up to 1K are charged at the normal rate (.033 ms./tuple). The formulas for three and seven receivers respectively are:

cost = 3.9 + (message-size in tuples - 4)(.033) milliseconds.

cost = 5.0 + (message-size in tuples - 4)(.033) milliseconds.

Appendix E
Results

The tables in this section show *total* costs for each algorithm, for each sample set tested. All costs are in seconds. The values shown are averages over ten simulation runs.

	key domain size 1000			key domain size 5000	
relation size (tuples)	750	1000	1250	3750	5000
ALGORITHM					
Global Wave (4)	0.10	0.65	5.71	0.57	5.19
Global Wave (8)	0.05	0.33	2.56	0.28	2.59
Local Wave (4)	0.33	1.74	14.30	1.74	11.50
Local Wave (8)	0.32	1.76	13.62	1.67	13.60
Logarithmic (4)	0.25	5.96	n/a	1.22	39.43
Logarithmic (8)	0.13	3.02	n/a	0.64	19.72
Direct (4)	0.23	1.20	13.86	1.11	7.08
Direct (8)	0.14	0.70	6.85	0.71	4.80

Table E-1: Local Processing Costs

	key domain size 1000			key domain size 5000	
relation size (tuples)	750	1000	1250	3750	5000
ALGORITHM					
Global Wave (4)	0.29	0.81	3.86	0.73	4.00
Global Wave (8)	0.56	1.20	4.16	0.99	4.25
Local Wave (4)	0.08	0.23	1.54	0.24	1.56
Local Wave (8)	0.12	0.27	1.79	0.27	1.80
Logarithmic (4)	0.28	1.92	n/a	0.68	10.81
Logarithmic (8)	0.51	2.23	n/a	0.64	7.99
Direct (4)	2.13	2.68	4.33	10.52	13.88
Direct (8)	2.76	3.43	5.35	13.45	17.62

Table E-2: Communication Costs

	key domain size 1000			key domain size 5000	
relation size (tuples)	750	1000	1250	3750	5000
ALGORITHM					
Global Wave (4)	0.39	1.46	9.56	1.30	9.19
Global Wave (8)	0.61	1.53	6.72	1.27	6.84
Local Wave (4)	0.40	1.97	15.84	1.98	13.06
Local Wave (8)	0.44	2.03	15.41	1.94	15.40
Logarithmic (4)	0.53	7.88	n/a	1.90	50.23
Logarithmic (8)	0.64	5.25	n/a	1.49	27.71
Direct (4)	2.36	3.88	18.18	11.63	20.95
Direct (8)	2.90	4.13	12.20	14.16	22.42

Table E-3: Combined Local Processing and Communication Costs

	key domain size 1000			key domain size 5000	
relation size (tuples)	750	1000	1250	3750	5000
ALGORITHM					
Delta-Wave	0.37	1.87	16.66	1.97	17.28
Logarithmic	0.76	15.86	n/a	3.37	103.71
Direct	0.73	3.34	51.35	2.71	26.34

Table E-4: Costs for Serial Algorithms

	key domain size 1000			key domain size 5000	
relation size (tuples)	750	1000	1250	3750	5000
ALGORITHM					
Global Wave (4)	0.95	1.28	1.74	1.52	1.88
Global Wave (8)	0.61	1.22	2.48	1.55	2.53
Local Wave (4)	0.93	0.95	1.05	0.99	1.32
Local Wave (8)	0.84	0.92	1.08	1.02	1.12
Logarithmic (4)	1.43	2.01	n/a	1.77	2.06
Logarithmic (8)	1.19	3.02	n/a	2.26	3.74
Direct (4)	0.31	0.86	2.82	0.23	1.26
Direct (8)	0.25	0.81	4.21	0.19	1.17

Table E-5: Speedup

Appendix F
Inter-algorithm Cost Comparisons

Each table details the cost differences between the Global Wavefront alogorithm and one of the others. For each basic cost parameter, the *difference* between the number of units charged to the Global algorithm, and the number charged to the algorithm being compared with it, is listed. Each term is accounted for in the appropriately-numbered comment beneath its table.

operation	Global Wavefront	LocalWavefront
tuples	$(p-1)\frac{\|A\|}{p}$ 1. + $\frac{2(p-1)}{p^2}\|A\|$ 2.	$\|A\|$ 1.
	+ $\frac{2(p-1)}{p^2}\sum_{i=1}^{r}\|G_i\|$ 3.	+ $\frac{p-1}{p}\|C^B\|$ 4.
	+ $\frac{p-1}{p}\|C\|$ 4.	
messages	$(p-1)$ 1. + $2(p-1)$ 2.	b_p 1.
	+ $2r(p-1)$ 3.	
PART	$\frac{\|A\|}{p}$ 5. + $\frac{1}{p}\sum_{i=1}^{r}\|G_i\|$ 6.	
COPY	$\frac{1}{p}(\|C\|-\|A\|)$ 7. + $\frac{\|C\|}{p}$ 8.	$\frac{1}{p}(\|C^B\|-\|A\|)$ 7. + $\frac{\|C^B\|}{p}$ 8.
		+ $\frac{(p-1)\|C\|}{p}$ 9.
JTI	$\frac{\|A\|}{p}$ 10.	$\|A\|$ 10.
RTI	$\frac{1}{p}\sum_{i=1}^{r}\|G_i\|$ 11.	$\frac{1}{p}\sum_{i=1}^{r}\|L_i\|$ 11.
		+ $\frac{p-1}{p}\|C^B\|$ 12.
JTL	$\frac{\|C\|}{p}$ 13.	$\frac{\|C^B\|}{p}$ 13.

Table F-1: Global Wavefront vs. Local Wavefront

1. Initial distribution of source relation. The Global algorithm sends a *different* fragment to each other machine, while the Local one broadcasts the *entire* source relation. b_p (Local alg.) is applied to the number of messages broadcast (in this case 1), to reflect added cost of broadcasting over point-to-point send.
2. The Global algorithm exchanges two copies of each partition, the Local just one.
3. The total of intermediate data that the Global algorithm exchanges. The Local does not exchange intermediate results.
4. $|C^B|$ includes duplicates that cannot be detected before result collection.
5. See point 2. above. The extra copy exchanged by the Global method is partitioned differently, so the relation exchanged has to be partitioned twice. The Local algorithm exchanges only one copy.
6. This is the cost of partitioning intermediate results, preparatory to exchanging them. The total amount of data partitioned is divided by p, since the machines can partition their local relations concurrently.
7. Copying required to make new driver relations during processing.
8. Preparation of local results for transmission to controller - machines can work in parallel.
9. The Local algorithm merges by insertion into a hash table at the controller. We assume that each algorithm takes as input a relation in stream form - the output should be in the same form. Nonduplicates must be copied to the stream implementation of the result (TC) relation.
10. In the Global algorithm, each machine builds a join hash table for just its own partition. In the Local one, that table must include the *entire source* relation, since data are not exchanged during processint.
11. L_i will generally include more duplicates that G_i.
12. The controller merges the local results from each other machine into its own. This is done by filtering them through a type B hash table.
13. In the join operation, each tuple in the driver (wavefront) relation is looked up in a table containing the source relation. Each member of the local closure is included in the wavefront relation exactly once during the entire running time of the algorithm.

operation	Global Wavefront	Logarithmic
tuples	$\frac{2(p-1)}{p^2}\sum_{i=1}^{r}\lvert G_i\rvert$ 1.	$\frac{4(p-1)}{p^2}\sum_{i=1}^{\lvert log_2 r\rvert}\lvert A^{2^i}\rvert$ + $\frac{2(p-1)}{p^2}\sum_{i=1}^{\lvert log_2 r\rvert}\lvert D_i\rvert$ 1.
messages	$2r(p-1)$ 1.	$6(p-1)\lvert log_2 r\rvert$ 1.
PART	$\frac{1}{p}\sum_{i=1}^{r}\lvert G_i\rvert$ 2.	$\frac{2}{p}\sum_{i=1}^{\lvert log_2 r\rvert}\lvert A^{2^i}\rvert$ + $\frac{1}{p}\sum_{i=1}^{\lvert log_2 r\rvert}\lvert D_i\rvert$ 2.
COPY	$\frac{1}{p}(\lvert C\rvert - \lvert A\rvert)$ 3.	$\frac{1}{p}\sum_{i=1}^{\lvert log_2 r\rvert}\lvert\overline{A}^{2^i}\rvert$ 3.
JTI	$\frac{\lvert A\rvert}{p}$ 4.	
RTI	$\frac{\lvert A\rvert}{p}$ 5. + $\frac{1}{p}\sum_{i=1}^{r}\lvert G_i\rvert$ 6.	$\frac{1}{p}\sum_{i=1}^{\lvert log_2 r\rvert}\lvert A^{2^i}\rvert$ 7.
JTID	$\frac{1}{p}[\sum_{i=1}^{\lvert log_2 r\rvert}(\lvert A^{2^i}\rvert + \lvert\overline{A}^{2^i}\rvert + \lvert D_i\rvert) + \lvert A\rvert]$ 8	
JTL	$\frac{\lvert C\rvert}{p}$ 9.	$\frac{1}{p}[\sum_{i=0}^{\lvert log_2 r\rvert-1}\lvert\overline{A}^{2^i}\rvert + \sum_{i=1}^{\lvert log_2 r\rvert}\lvert\overline{A}^{2^i}\rvert]$

Table F-2: Global Wavefront vs. Logarithmic

1. The initial setup and result collection costs are identical for these algorithms. Each exchanges intermediate results. The Global algorithm has only a single join per iteration, while the Log one has two. In the Log algorithm, two copies of the first join's output have to be exchanged, since that relation is joined with itself. While $|A^{2^i}|$ and $|D_i|$ tend to be much larger than G_i, the Log method uses only $\lceil log_2 r \rceil$ iterations.
2. The relations to be exchanged have to be partitioned. The total amount of data to be partitioned is divided by p, since the machines can work in parallel.
3. For each iteration > 0, each algorithm builds its driver for the next iteration.
4. The JTI operation is for insertion into the type A hash table, which the Logarithmic algorithm does not use.
5. This term describes the cost of initializing the result (type B) table for the Global Wavefront algorithm. The Log algorithm collects results in a type C table.
6. This term covers the cost of inserting generated tuples into the result. G_i includes duplicates. These are filtered out by the RTI operation.
7. A^{2^i} may contain a lot of internal duplicates. They are removed by filtering the relation through a type B hash table.
8. Both joins use a type C table. The join table for the second join is also the result table. This expression covers the cost of building both tables.
9. Costs of looking up drivers in join tables. The Global algorithm has one join, the Logarithmic two. The JTL operation works on both type A and type C hash tables.

operation	Global Wavefront	LocalWavefront
tuples	$\frac{2(p-1)}{p^2}\lvert A\rvert$ 1.	$b_p \sum_{i=0}^{w-1} \lvert\{succ(v_i)\}\rvert$
	$+ \frac{2(p-1)}{p^2}\sum_{i=1}^{r}\lvert G_i\rvert$ 2.	$\approx \frac{b_p}{2}[(\lvert A\rvert + \frac{\lvert C\rvert - \lvert A\rvert}{2}) + 2w]$ 2.
messages	$2(p-1)$ 1. $+ 2r(p-1)$ 2.	$b_p w$ 2.
PART	$\frac{\lvert A\rvert}{p}$ 3. $+ \frac{1}{p}\sum_{i=1}^{r}\lvert G_i\rvert$ 4.	
COPY	$\frac{1}{p}(\lvert C\rvert - \lvert A\rvert)$ 5.	$\sum_{i=0}^{w-1} \lvert\{succ(v_i)\}\rvert$
JTI	$\frac{1}{p}(\lvert C\rvert - \lvert A\rvert)$ 6.	
RTI	$\frac{1}{p}\sum_{i=0}^{r}\lvert G_i\rvert$ 7.	
JTID	$\frac{1}{p}\sum_{i=0}^{w}\lvert S_i\rvert$ 8.	
JTL	$\frac{\lvert C\rvert}{p}$ 9.	w 10. $+ \frac{1}{p}\sum_{i=1}^{w} m_i$ 11.
		$- \frac{1}{p}\sum_{i=1}^{w}\lvert S_i\rvert$ 12.

Table F-3: Global Wavefront vs. Direct

1. In the initial setup, the Global algorithm exchanges two copies of the the source relation, the Direct algorithm just one. The result collection procedure is identical for both algorithms.

2. In the Global algorithm, machines *exchange* intermediate results. In the Direct, each iteration involves a single machine *broadcasting* a successor list. b_p is applied to the number of messages (w) broadcast by the Direct algorithm, to reflect the added cost of a broadcast over a point-to-point send.

3. JTI is the insertion operator for the type A hash table. The Global algorithm installs only the initial source fragment - this is used in the join operation. The Direct algorithm stores the predecessor lists in a type A table. Between the participating machines, the *entire closure* ends up being stored in this structure.

4. The Global algorithm accumulates its results in a type B table.

5. Successor lists are stored in a type C table. This expression includes the cost of initializing each machines table to include its fragment of the source relation, and of merging each successor list broadcast into that table (one list per iteration). Note that S_i includes duplicates.

6. The lookup costs for driving the join.

7. Cost of finding predecessors of each vertex (once per iteration).

8. Each iteration, the successor list of each predecessor of the "current" vertex has to be looked up in the successor list structure (hash table type C).

9. Merging two successor lists together involves inserting one of them into the secondary structure which contains the other. The JTID operation *includes* the cost of finding that secondary structure. But when this list merging is done, the algorithm already has a pointer to it. Thus the lookup cost has to be subtracted.

References

1. Agrawal, Rakesh and Jagadish, H.V. Direct Algorithms for Computing the Transitive Closure of Database Relations. Proceedings of the 13th VLDB Coonference, 1987, pp. 255-266.

2. Cheriton, D.R. and Lantz, K.E., Principal Investigators. *V-System 6.0 Reference Manual.* Distributed Systems Group, Stanford University, 1986.

3. DeWitt, D.J., Katz, R.H., Olken, F., Shapiro, L.D., Stonebraker, M.R. and Wood, D. Implementation Techniques for Main Memory Database Systems. Proc. of ACM-SIGMOD Conference on Management of Data, 1984, pp. 1-8.

4. Han, J. and Henschen, L.J. Handling Redundancy in the Processing of Recursive Database Queries. Proceedings of ACM-SIGMOD Conference on Very Large Data Bases, 1987, pp. 73-81.

5. Han, J., Qadah, G. and Chaou, C. The Processing and Evaluation of Transitive Closure Queries. Han - School of Computing Science, Simon Fraser University, Qadah and Chaou - Dept. of Electrical Engineering and Computer Science, Northwestern University, 1987.

6. Han, J. and Luk, W.S. What Kinds of Recursions Can Be Processed by Transitive Closure Strategies? To appear in Proc. of 3rd International Symposium on Methodologies for Intelligent Systems, 1988.

7. Henschen, L.J. and Naqvi, S. "On Compiling Queries in Recursive First-Order Databases". *JACM 31*, 1 (1984).

8. Horowitz, E. and Sahni, S.. *Fundamentals of Computer Algorithms.* Computer Science Press, 1984.

9. Ioannidis, Y.E. On the Computation of the Transitive Closure of Relational Operators. Proceedings of the 12th International VLDB Coonference, 1986, pp. 403-411.

10. Ioannidis, Y.E. and Ramakrishnan, R. Efficient Transitive Closure Algorithms. Computer Sciences Department, University of Wisconsin, 1988.

11. Lu, H. New Strategies for Computing the Transitive Closure of a Database Relation. Proc. of the 13th International VLDB Conference, 1987, pp. 267-274.

12. Luk, W. S., Wang, Xiao and Ling, Franky. On the Communication Cost of Distributed Database Processing. Proc. of the 8th International Conference on Distributed Computing Systems, 1988.

13. Lu, H., Mikkilineni, K. and Richardson, J.P. Design and Evaluation of Algorithms to Compute the Transitive Closure of a Database Relation. Proc. of the 3rd International Data Engineering Conference, 1987, pp. 112-119.

14. Raschid, L. and Su, S.Y.W. A Parallel Processing Strategy for Evaluating Recursive Queries. Proceedings of the 12th International Conference on VLDB, 1986.

15. Valduriez, P. and Boral, H. Evaluation of Recursive Queries Using Join Indices. Proc. of the 1st International Expert Database Systems Conference, 1986, pp. 197-208.

16. Wang, X. and Luk, W.S. Parallel Join Algorithms for a Network of Workstations. to appear in Proc. of International Symposium on Databases in Parallel and Distributed Systems, 1988.

17. Wang, X. Query Processing for Distributed Main Memory Database Systems. School of Computing Science, Simon Fraser University, 1987.

18. Warren, Henry S. Jr. "A Modification of Warshall's Algorithm for the Transitive Closure of Binary Relations". *Communications of the ACM 18*, 4 (1975), 218-220.

19. Warshall, S. "A Theorem on Boolean Matrices". *JACM 9*, 1 (1962), 11-12.