Multiprocessor Implementation of Transitive Closure

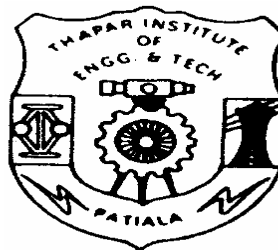
A Thesis submitted for the partial fulfillment of the requirement for the award of the Degree of

Master of Engineering
In
Software Engineering

By

Sunidhi Bhalla
8023119

Under the guidance of
Mrs. Seema Bawa
Asstt. Professor & H.O.D (Computer Sc. & Engg. Deptt.)



Computer Science and Engineering Department
Thapar Institute of Engineering and Technology
(Deemed University), Patiala-147004

May, 2004

Declaration

I hereby declare that the work which is being presented in the thesis entitled, "Multiprocessor Implementation of Transitive Closure", in the partial fulfillment of the requirements for the award of the degree of Master of Engineering in Software Engineering at Computer Science & Engineering Department, Thapar Institute of Engineering & Technology (Deemed University), Patiala is an authentic record of my own work carried out under the supervision of Mrs. Seema Bawa, Asstt. Professor. & Head, Computer Science & Engg. Department.

The matter embodied herein has not been submitted to any other University for the award of any other degree.

(Sunidhi Bhalla)
M.E. (Software Engg.)
8023119

This is to certify that above statement made by the candidate is correct and true to best of my knowledge.

(Seema Bawa) **(Dr. D.S.Bawa)**

Supervisor Dean

(Academic Affairs) Assistant Professor

& Head TIET,

Patiala-147004

Computer Science & Engineering Department

TIET, Patiala-147004

Acknowledgement

To discover, analyze and to present something new is to venture on an untrodden path towards an unexplored destination is an arduous adventure unless one gets a true torchbearer to show the way. This enlightening guidance, I found in my revered guide Mrs Seema Bawa, Asstt. Professor & H.O.D., Computer Science & Engineering Department, Thapar Institute of Engineering & Technology (Deemed University), Patiala, without whose patronization it was never possible to give final shape to this thesis. I express my heartfelt gratitude towards her for her valuable guidance, encouragement, constant involvement, inspiration and the enthusiasm with which she solved my difficulties.

I shall be failing in my duties if I do not express my deep sense of gratitude towards Mr. Maninder Singh, Asstt. Professor, Computer Science & Engineering Department, Thapar Institute of Engineering & Technology (Deemed University), Patiala, for his valuable advices and suggestions.

I would like to thank all the faculty and staff members of Computer Science & Engineering Department for providing me all the facilities required for the completion of this work.

Last but not the least, I express my heartfelt thanks to my parents, my friends for co-operation, which they were always ready to extend.

(Sunidhi Bhalla)
M.E. (Software Engg.)
8023119

Abstract

Graph theoretic algorithms are found quite effective for solving complex real life problems. Computing the transitive closure in directed graphs is a fundamental graph problem. Transitive closure can be thought of as establishing a data structure that makes it possible to solve reachability questions (can I get to *x* from *y*?) efficiently. After the preprocessing of constructing the transitive closure, all reachability queries can be answered in constant time by simply reporting a matrix entry. Transitive closure is fundamental in propagating the consequences of modified attributes of a graph *G*.

For efficient system utilization and fast response to the user, it is necessary to use parallel algorithms for solving a single problem. Transitive closure is a highly parallelizable problem; it belongs to the class NC of problems that can be solved in polylogarithmic time (i.e. $O(\log^c n)$ for some constant c) with polynomial number of processors.

This thesis proposes an efficient scalable multiprocessor algorithm for transitive closure computation in a distributed computing environment. It focuses on the role of transitive closure in the Graph Theory, Very Large Databases, Relational Database Management Systems, and VLSI Test Generation.

Serious efforts have been made in investing new parallel and /or distributed algorithm for the transitive closure so that real life problems can be solved in reasonable time. The proposed algorithm divides the transitive closure computation into *n* tasks using data parallelism. The performance metrics like

speedup, efficiency, and scalability are taken into account for the algorithm development.

The algorithm, so developed, has been implemented in a heterogeneous distributed computing environment, using PVM. Experimental results given in the thesis show the effectiveness of the proposed algorithm.

Table of Contents

Contents **Page No**

Certificate i
Acknowledgement ii
Abstract iii
Table of Contents v
List of Figures viii
List of Tables ix

1. INTRODUCTION **1-4**

1.1 Motivation

2 1.2 Objective of the Study

2

1.3 Problem Definition

3

1.4 Organization of Thesis

3

2. LITERATURE SURVEY **5-45**

2.1 Principles of Parallel Algorithm Design 6

2.1.1 Decomposition into Tasks 7

2.1.2 Dependency Graphs 8

2.1.3 Granularity and Concurrency 11

2.1.4 Mapping the Processes 13

2.1.5 Characteristics of Tasks 15

2.2 Performance Metrics for Parallel Systems 16

2.2.1 Run Time 17
2.2.2 Total Parallel Overhead 17
2.2.3 Speedup 17
2.2.4 Efficienc y 18
2.2.5 Scalabilit y 18

2.3 Programming Environment for Distributed Computing 20
2.3.1 Parallel Virtual Machine (PVM) 20

2.4 Graph Theoretic Preliminaries 23

2.5 Applications of Transitive Closure 30

2.5.1 Transitive Closure in Graph Theoretic Problems 30

2.5.1.1 Transitive Closure in Shortest Path Problem 30

2.5.1.2 Transitive Closure in Transitive Reduction 31

2.5.1.3 Transitive Closure in Limited Space Algorithms 32

2.5.2 Transitive Closure in Very Large Databases 33

2.5.3 Transitive Closure in RDBMS
34
2.5.4 Transitive Closure in VLSI
Test Generation 35
2.6 Transitive Closure Algorithms
39
2.6.1 Serial Warshall Algorithm
39
2.6.1.1 Illustration of Warshall
Algorithm 39
2.6.2 Serial Warren Algorithm
41
2.6.3 Parallel Algorithm for
Transitive Closure using 42
Double Hash-Based Clustering

3. PROPOSED TRANSITIVE CLOSURE ALG. 46-51

3.1 Parallel Algorithm Design 47

3.1.1 Illustration of Proposed Parallel Algorithm 49

3.2 Complexity Calculations 50

3.2.1 Best Case 50

3.2.2 Worst Case 51

4. IMPLEMENTATION DETAILS AND EXPERIMENTAL RESULTS 52-57

4.1 Implementation Details 52

4.1.1 Computation at Master 53
4.1.2 Computation at Slave 54
4.2 Experimental Results 54
4.2.1 Computational Time vs. Number of Processors 55
4.2.2 Speedup vs. Number of Processors 56

5. CONCLUSIONS & FUTURE SCOPE OF WORK 58-59
5.1 Conclusion 58
5.2 Future Scope of Work 59

REFERENCES **60-63**

List of Figures

Figure No.	Caption	Page No.
2.1	Decomposition of matrix multiplication into four tasks	7
2.2	Decomposition of matrix multiplication into eight tasks	8
2.3	Task Dependency Graph	8
2.4	Tables and their dependencies in a query processing operation	11
2.5	Decomposition of dense matrix multiplication into n-tasks and four tasks	12
2.6	PVM Computational Model	21
2.7	Interprocess Communication	22
2.8	Example Graph G and its representation	24
2.9	Transitive Closure of Graph G	26
2.10	Spanning Tree for Graph G of Fig 2.8	27
2.11	Strong Components of Graph G	29
2.12	Condensation Graph	29
2.13	Directed Graph	40
3.1	Logical Design of Master-Slave Communication	47
3.2	Adjacency Matrix	49
3.3	Transitive closure of above adjacency matrix	49
4.1	Average execution time vs. No of Processors	56
4.2	Speedup vs. No of Processors	57

List of tables

Table No.	Caption	Page No.
2.1	Database string information about used vehicles	9
5.1	Efficiency as a function of n & p for adding n numbers on p processors	19
4.1	Computational Time vs. No of processors for proposed algorithm	51

Chapter 1

Introduction

Graph theoretic algorithms are found quite effective for solving complex real life problems and a vast amount of research has already been done in order to exploit the computational power available with the machines. Standard graph algorithms have been used to solve the computationally intensive problems like traveling sales-man problem, max-cut/min-cut problem, network analysis, transportation problem, VLSI-CAD problems like automatic test generation, high-level synthesis, layout problem etc. In many of these applications, one often needs to determine connectivity between any two vertices in a graph and it is done by computing the transitive closure of a graph. Computing the transitive closure in directed graphs is a fundamental graph problem.

Efficient computation of the transitive closure of a directed graph is required in many applications, for instance, in the reach ability analysis of transition networks representing distributed and parallel systems [4] and in the construction of parsing automata in compiler construction [3,23]. Recently, efficient transitive closure computation is required in many database queries [30].

Several transitive closure algorithms have been presented during the last thirty years. Despite the increased efficiency of computers, the need for more efficient transitive closure algorithms and representations remains. This is for two

reasons. First, the size of the inputs seems to grow in proportion to the growth of the memory capacity. Since the CPU speed has grown at the same rate as the memory capacity, only linear algorithms have retained their execution times on typical inputs. Second, typical inputs and outputs in modern applications, e.g., in the area of databases, do not fit into the main memory.

Transitive closure can be thought of as establishing a data structure that makes it possible to solve reachability questions (can I get to *x* from *y*?) efficiently. After the preprocessing of constructing the transitive closure, all reachability queries can be answered in constant time by simply reporting a matrix entry.Transitive closure is fundamental in propagating the consequences of modified attributes of a graph *G*.

1.1 Motivation

For efficient system utilization and fast response to the user, it is necessary to use parallel algorithms for solving a single problem. Transitive closure is a highly parallelizable problem; it belongs to the class NC of problems that can be solved in polylogarithmic time (i.e. $O(\log^c n)$ for some constant c) with polynomial number of processors. This has been extended significantly to more general classes of recursive queries: linear and piecewise linear queries, programs with the polynomial fringe property, and the polynomial stack property. Although the basic problem of transitive closure is in NC, in many respects we still lack a parallel algorithm.

1.2 Objective of the Study

The objective of research reported here is to design and implement an efficient multiprocessor algorithm for computing the transitive closure of a directed graph in the real heterogeneous distributed network environment comprised of P-II and P-III machines of different speeds and architecture. In the study, we have used the notion of message passing based on PVM Master/Slave model as the criterion for implementing parallel algorithm.

1.3 Problem Definition

In the transitive closure problem, we are given a directed graph G = (*V,E*), where *V*={x_1,….x_n) is a finite set of vertices and *E* is a finite set of edges; An edge *E* is an ordered pair (x_i,x_j), where x_i,x_j € V and an edge(x_i,x_j) means that vertices x_i and x_j are connected.
The transitive closure of G is defined as the Graph G * = (*V,E*), where *E** ={(x_i,x_j)| there is a path from x_i to x_j in G}.

The transitive closure of a graph is computed by computing the connectivity matrix A*.The connectivity matrix of G is a matrix A* = (a^*_{ij}) such that a^*_{ij} = 1 if there is a path from v_i to v_j or i = j, and a^*_{ij} = 0 otherwise.

$a^*_{i,j}$ = 1 if i = j or(v_i,v_j) € *E*

$a^*_{i,j}$ = 0, otherwise

The input of the transitive closure problem is in the form of adjacency matrix. Note, that we require no data structure that contains the vertices adjacent to a vertex v, only the vertices adjacent from v. The output of the transitive closure is given as a successor lists that represent the successor sets of the vertices.

1.3 Organization of Thesis

Chapter 1 introduces the problem and describes the objective of the work, along with the organization of the thesis.

Chapter 2 presents the parallel algorithm design principles, performance metrics for parallel systems, graph theoretic preliminaries, programming environment for distributed computing, applications of transitive closure, transitive closure algorithms.

Chapter 3 deals with the design of the proposed parallel transitive closure algorithm and its complexity issues.

Chapter 4 gives the implementation details and experimental results of the proposed parallel transitive closure algorithm. Chapter 5 finally gives the conclusions and the scope for future work. After this chapter, a list of important references used to comprehend the theoretical concepts related to the thesis work, are included.

Chapter 2 Literature Survey

Parallel Processing refers to the concept of speeding-up the execution of a program by dividing the program into multiple fragments that can execute simultaneously, each on its own processor. A program being executed across n processors might execute n times faster than it would be using a single processor.

The complexity of today's applications coupled with the widespread use of parallel processing has made the design and analysis of parallel algorithms a necessity. Many problems can be solved by massive parallelism. The emergence of high-performance, massively parallel computers demand the development of parallel algorithms to take advantage of this technology.

Parallel Processing is information processing that emphasizes the concurrent manipulation of data elements belonging to one or more processes. It involves multiple processes, which are active simultaneously, and solving a given problem, generally on multiple processors. The critical aspect here is "solving a given problem". The processes all must be concerned with the solution of one single problem [27]. In other words, there must be interactions among the units.

Problems are parallelizable to different degrees. For some problems, assigning partitions to other processors might be more time consuming than performing the processing locally. Other problems may be completely serial. For example, consider the task of digging a hole. Although one person can dig a hole in a certain amount of time, employing more people does not reduce this time. Because, it is impossible to partition this task, it is poorly suited to parallel processing. Therefore, a problem may have different parallel formulations, which result in varying benefits, and all problems are not equally amenable to parallel processing.

2.1 Principles of Parallel Algorithm Design

Algorithm development is critical to using computers to solve problems. A sequential algorithm is essentially a recipe or a sequence of basic steps for solving a given problem using a serial computer.

Similarly, a parallel algorithm is a recipe that tells us how to solve a given problem using multiple processors [27]. A parallel algorithm has the added dimension of concurrency and the algorithm designer must specify sets of steps that can be executed in parallel. This is essential for obtaining any speed benefit from the use of a parallel computer. In practice, specifying a parallel program may include the following:

- *Identifying portions of the work that can be performed concurrently.*
- Mapping the concurrent pieces of work onto multiple processes running in parallel.
- Distributing the input, output, and intermediate data associated with the program.
- Managing accesses to data shared by multiple processors.
- Synchronizing the processors at various stages of the parallel program execution.

Typically, there are several choices for each of the above steps, but usually, relatively few combinations of choices lead to a parallel algorithm that yields

performance commensurate with the computational and storage resources employed to solve the problem. Often, different choices yield the best performance on different parallel architectures or under different parallel programming paradigms. The following section presents the various principles of parallel algorithm design [27]:

2.1.1 Decomposition into Tasks

The processes of dividing a computation into smaller parts, some or all of which may potentially be executed in parallel, is called decomposition. Tasks are programmer-defined units of computation into which the main computation is subdivided by means of decomposition. Simultaneous execution of multiple tasks is the key to reducing the time required to solve the entire problem. Tasks can be of arbitrary size, but once defined, they are the smallest units of concurrency.

Example 2.1 Matrix multiplication

Consider the problem of multiplying two n × n matrices A and B to yield a matrix C. Figure 2.1 show a decomposition of this problem into four tasks.

Each matrix is considered to be composed of four blocks or sub matrices defined by splitting each dimension of the matrix into half. The four sub matrices of C, roughly of size n/2 × n/2 each, are then independently computed by four tasks as the sums of the appropriate products of sub matrices of A and B: -

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

(a)

Task 1: $C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$
Task 2: $C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$
Task 3: $C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$
Task 4: $C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

(b)

Figure 2.1 Decomposition of Matrix Multiplication into four tasks

Task 1: $C_{1,1} = A_{1,1}B_{1,1}$	Task 1: $C_{1,1} = A_{1,1}B_{1,1}$
Task 2: $C_{1,1} = C_{1,1} + A_{1,2}B_{2,1}$	Task 2: $C_{1,1} = C_{1,1} + A_{1,2}B_{2,1}$
Task 3: $C_{1,2} = A_{1,1}B_{1,2}$	Task 3: $C_{1,2} = A_{1,2}B_{2,2}$
Task 4: $C_{1,2} = C_{1,2} + A_{1,2}B_{2,2}$	Task 4: $C_{1,2} = C_{1,2} + A_{1,1}B_{1,2}$
Task 5: $C_{2,1} = A_{2,1}B_{1,1}$	Task 5: $C_{2,1} = A_{2,2}B_{2,1}$
Task 6: $C_{2,1} = C_{2,1} + A_{2,2}B_{2,1}$	Task 6: $C_{2,1} = C_{2,1} + A_{2,1}B_{1,1}$
Task 7: $C_{2,2} = A_{2,1}B_{1,2}$	Task 7: $C_{2,2} = A_{2,1}B_{1,2}$
Task 8: $C_{2,2} = C_{2,2} + A_{2,2}B_{2,2}$	Task 8: $C_{2,2} = C_{2,2} + A_{2,2}B_{2,2}$

Figure 2.2 Decomposition of Matrix Multiplication into eight tasks.

Note that all four tasks in Figure 2.1 are independent and can be performed all together or in any sequence. However, that is not the case with the decompositions shown in Figure 2.2. Each of the even numbered tasks in both decompositions of Figure 2.2 can be performed only after the immediately preceding odd numbered task is complete.

2.1.2 Dependency Graphs

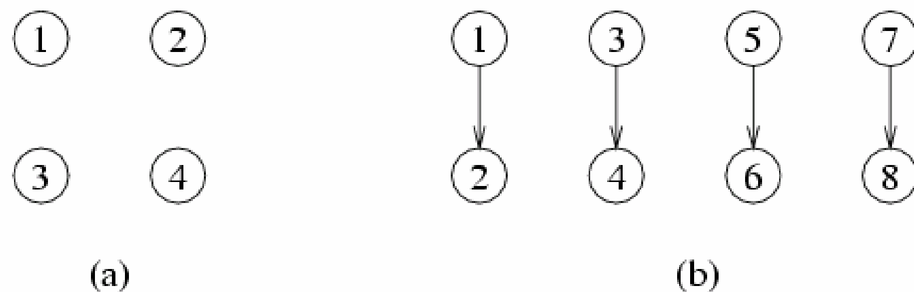


Figure 2.3 Task Dependency Graph

A task-dependency graph is a directed acyclic graph in which the nodes represent tasks and the directed edges indicate the dependencies amongst them. The task corresponding to a node can be executed when all tasks connected to this node by incoming edges have a property of a task-dependency graph that places a lower bound on the average degree of concurrency for a given granularity is its critical path. In a task-dependency graph, the nodes with no incoming edges are referred as start nodes and the nodes with no outgoing edges are referred as finish nodes. The longest directed path between any pair of start and finish nodes is known as the critical

path and the number of nodes along this path is known program. The average degree of concurrency usually increases as the granularity becomes smaller (finer). The task-dependency graphs for the matrix multiplication problem are shown in Figure 2.3. To show a task-dependency graph, consider the following database query-processing example.

Example 2.2 Database query processing Consider the computations performed on the vehicle relational database shown in Table 2.1 in processing the following query:
MODEL=”Accord” AND YEAR=”1999” AND (COLOR=”Green” OR COLOR=”Black”)

ID#	Model	Year	Color	Dealer	Price
4523	Accord	2000	Blue	MN	$19,000
3476	Taurus	2000	Green	IL	$18,000
7623	Camry	1999	Black	NY	$21,000
9834	Civic	1997	Black	MN	$10,000
6734	Accord	1999	Green	AR	$18,000
5342	Altima	1999	Black	FL	$18,000
3845	Maxima	1999	Blue	TX	$22,000
8354	Malibu	2000	Black	ID	$16,000
4395	Accord	1999	Red	CA	$20,000
7352	Accord	2000	Red	SD	$17,000

Table 2.1 A Database storing information about used vehicles

This query looks for all 1999 Accords whose color is either Green or Black. On a relational database, creating a number of intermediate tables processes this query. One possible way is to first create the following four tables: A table containing all Accords, a table containing all 1999-model cars, a table containing all green-colored cars, and a table containing all black-colored cars. Next, the computation will proceed by combining these tables via what is known as a database-join operation. In particular, it will compute the intersection of the Accord-table with the 1999-model year table, to construct a table of all 1999-modelAccords. Similarly, it will compute the union of the green- and black-colored tables to compute a table storing all cars whose color is either green or black. Finally, it will compute the intersection of the table containing all the 1999 Accords with the table containing all the green or black vehicles, and return the desired list.

The various computations involved in Example 2.2 can be visualized by the task-dependency graph shown in Figure 2.4. Each node in this figure is a task that corresponds to an intermediate table that needs to be computed and the arrows between nodes indicate dependencies between the tasks. For example, before we can compute the table that corresponds to the 1999 Accords, we must first compute the table of all the Accords and a table of all the 1999-model cars.

Note that often there are multiple ways of expressing certain computations, especially those involving associative operators such as addition, multiplication, and logical AND *and* OR*. Different ways of arranging computations can lead to different task-dependency graphs with different characteristics. For instance, first computing a table of all green or black cars, then performing a join with a*

table of all 1999 model cars, and finally joining the resulting table with that of all Accords can solve the database query in the sequence of computation results in a task-dependency graph shown below-

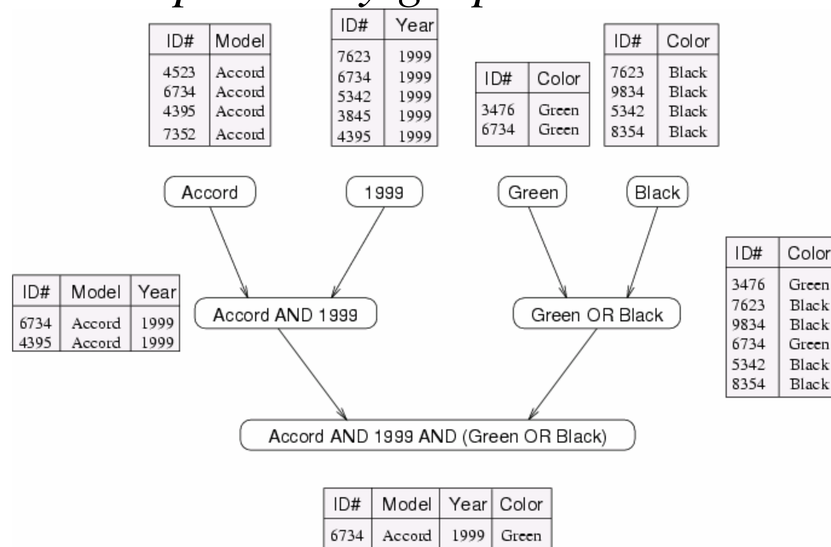


Figure 2.4 The tables and their dependencies in a query processing operation

2.1.3 Granularity and Concurrency

The number and size of tasks into which a problem is decomposed determines the granularity of the decomposition. Decomposition into a large number of small tasks is called fine-grained and decomposition into a small number of large tasks is called coarse-grained. A concept related to granularity is that of degree of concurrency. The maximum number of tasks that can be executed simultaneously in a parallel program at any given time is known as its maximum degree of concurrency. A more useful indicator of a parallel program's performance, the average degree of concurrency is the average number of tasks that can run concurrently over the entire duration of execution

of the For example, the decomposition of matrix-vector multiplication shown in Figure 2.5 has a fairly small granularity and a large degree of concurrency.

The decomposition for the same problem shown in Figure 2.5(b) has a larger granularity and a smaller degree of concurrency. In most cases, the average degree of concurrency is less than the total number of tasks due to dependencies among the tasks. For example, there are four tasks in the decomposition of matrix multiplication shown in Figure 2.1 and the average and maximum degrees of concurrency are also four because all four tasks are independent and can be performed simultaneously. However, the decompositions shown in Figure 2.2 have a degree of concurrency of four too, despite a smaller granularity.

This is because task-dependency graph shown in Figure 2.3(b) permits at most four tasks to run simultaneously. The degree of concurrency also depends on the shape of the task-dependency graph and the same granularity, in general, does not guarantee the same degree of concurrency. For example, the maximum degree of concurrency in the database query decompositions shown in Figures 2.4 is four respectively.

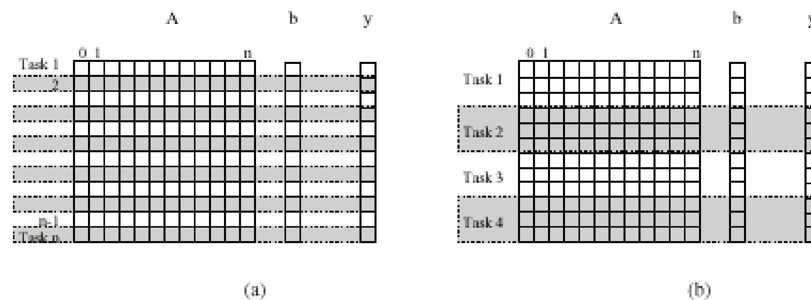


Figure 2.5 Decomposition of dense matrix-vector multiplication into (a) n tasks and (b) four tasks.

Other than granularity and degree of concurrency, there is another important practical factor that limits our ability to obtain unbounded speedup from parallelization. This factor is the interaction among tasks running on different physical processors. The tasks that a problem is decomposed into often share input, output, or intermediate data. For example, in matrix-vector and matrix-matrix multiplication, multiple tasks share the input data. In the database query example, tasks share intermediate data; the table generated by one task is often used by another task as input. The resulting interaction among tasks can be time consuming. While the total amount of useful work required to solve a problem remains constant, increasing the number of tasks usually increases the total amount of interaction. Therefore, increasing the number of tasks beyond a certain point may not reduce the parallel solution time of a problem if the interaction overhead

exceeds that time gained due to parallel execution of tasks.

2.1.4 Mapping the Processes

The tasks, into which a problem is decomposed, run on physical processors. Process is an abstract entity that uses the code and data corresponding to a task to produce the output of that task within a finite amount of time after the task is activated by the parallel program. During this time, besides performing computations, a process may synchronize or communicate with other processes, if needed. In order to obtain any speedup over a sequential implementation, a parallel program must have several processes active simultaneously, working on different tasks. The mechanism by which tasks are assigned to processes for execution is called mapping. For example, four processes could be assigned the task of computing one sub matrix of C each in the matrix-multiplication computation of Example 2.1. The task-dependency and task-interaction graphs that result from a choice of decomposition play an important role in the selection of a good mapping for a parallel program.

Mapping should seek to maximize the use of concurrency by mapping independent tasks onto different processes, it should seek to minimize the total completion time by ensuring that processes are available to execute the tasks on the critical path as soon as such tasks become executable, and it should seek to minimize interaction among processes by mapping tasks with a high degree of mutual interaction onto the same process. In most non-trivial parallel programs, these tend to be conflicting goals. For instance, the most efficient decomposition-mapping combination is a single task mapped onto a single process. It wastes no time in idling or interacting, but achieves no speedup either. Finding a balance that optimizes the overall parallel performance is the key to a successful parallel program.

A good mapping of tasks onto processes strives to achieve the twin objectives of

- *Reducing the total amount of time when some processes are idle while the others are engaged in performing some tasks.*
- *Reducing the amount of time processes spend in interacting with each other due to interaction among tasks mapped onto different processes.*

These two objectives often conflict with each other. For example, the objective of minimizing the interactions can be easily achieved by assigning sets of tasks that need to interact with each other onto the same process. In most cases, such a mapping will result in a highly imbalanced workload among the processes. In fact,

following this strategy to the limit will often map all tasks onto a single process. As a result, the processes with lighter load will be idle when those with a heavier load are trying to finish their tasks. Due to the conflicts between these objectives, finding a good mapping is a non-trivial problem.

Therefore, mapping of tasks onto processes plays an important role in determining how efficient the resulting parallel program is. Even though the degree of concurrency is determined by the decomposition, it is mapping that determines how much of that concurrency is actually utilized, and how efficiently. For example, an efficient mapping for the decompositions of Figure 2.2 onto four processes may assign Tasks 1 and 2 to process P_0, Tasks 3 and 4 to process P_1, Tasks 5 and 6 to process P_2, and Tasks 7 and 8 to process P_3. Note that, in this case, a maximum of four processes can be employed usefully, although the total

number of tasks is eight. This is because the maximum degree of concurrency in the task-dependency graph shown in Figure 2.3(b) is four. Note that the set of the first four tasks and the set of the last four tasks can be mapped arbitrarily among four processes each to satisfy the constraints of the task-dependency graph.

2.1.5 Characteristics of Tasks

The following four characteristics of the tasks have a large influence on the suitability of a mapping scheme:

a) Task Generation

The tasks that constitute a parallel algorithm may either be generated statically or dynamically. Static task generation refers to the scenario where all the tasks are known before the algorithm starts execution. Data decomposition usually leads to static task generation. Examples of data decomposition leading to a static task generation include matrix-multiplication. Certain decompositions lead to a dynamic task generation during

the execution of the algorithm. In such decompositions, the actual tasks and the task-dependency graph is not explicitly available a-priori, although the high level rules or guidelines governing task generation are known as a part of the algorithm. Recursive decomposition can lead to dynamic task generation. For example, consider the recursive decomposition in quick sort the tasks are generated dynamically, and the size and shape of the task tree is determined by the values in the input array to be sorted. An array of the same size can lead to task-dependency graphs of different shapes and with a different total number of tasks.

b) Task Sizes

The size of a task is the relative amount of time required to complete it. The complexity of mapping schemes often depends on whether or not the tasks are uniform; i.e., whether or not they require roughly the same amount of time. If the amount of time required by the tasks

varies significantly, then they are said to be non-uniform. For example, the tasks in the decompositions for matrix multiplication would be considered uniform. On the other hand, the tasks in quick sort are non-uniform.

c) Knowledge of Task Sizes

The third characteristic that influences the choice of the mapping scheme is the knowledge of the task size. If the size of all the tasks is known, then this information can often be used in mapping of tasks to processes. For example, in the various decompositions for matrix multiplication discussed so far, the computation time for each task is known before the parallel program starts.

d) Size of Data Associated with Tasks

Another important characteristic of a task is the size of data associated with it. The reason this is an important consideration for mapping is that the data associated with a task must be available to the process performing that task, and the size

and the location of this data may determine the process that can perform the task without incurring excessive data-movement overheads.

Different types of data associated with a task may have different sizes. For instance, the input data may be small but the output may be large, or vice-versa. In the problem of computing the minimum of a sequence, the size of the input is proportional to the amount of computation, but the output is just one number. In the parallel formulation of the quick sort, the size of both the input and the output data is of the same order as the sequential time needed to solve the task.

2.2 Performance Metrics for Parallel Systems

A sequential algorithm is usually expressed in terms of its execution time, expressed as a function of the size of its input. The execution time of a parallel algorithm depends not only on the input size but also on the architecture of parallel computer and the number of processors.

Various performance metrics for parallel systems are described in the following section [27]:

2.2.1 Run Time

The serial run-time of a program is the time elapsed between the beginning and the end of its execution on a sequential computer. The parallel run time is the time that elapses from the moment that a parallel computation starts to the moment that the last processor finishes execution. We denote the serial run time by T_s and the parallel run time by T_P. We also formalize the size of the problem W *as the serial runtime of the program to solve the problem instance. This is in contrast to other measures of problem size such as the input size or output size.*

In an ideal world, if computation can be carried out in *p* equal parts, the total execution time will be nearly *1/p* of the time required by a single proc;

$$T_p = T_s / p.$$

2.2.2 Total Parallel Overhead

The overheads incurred by a parallel program are encapsulated into a single expression referred to as the overhead function [27]. We define total overhead or overhead function of a parallel system as

the total time collectively spent by all the processors over and above that required by the fastest known sequential algorithm for solving the same problem on a single processor. We denote the overhead function of a parallel system by the symbol T_o*. To is a function of* W *and* p*, and we often write it as* T_o(W, p).

The total time spent in solving a problem summed over all processors, is pT_P. *W* units of this time are spent performing useful work, and the remainder is overhead. Therefore, the overhead function (T_o) is given by

$$T_o = p \times T_p - W$$

2.2.3 Speedup

When evaluating a parallel system, we are often interested in knowing how much performance gain is achieved by parallelizing a given application over a sequential implementation. Speedup is a measure that captures the relative benefit of solving a problem in parallel. It is defined as the ratio of the time taken to solve a problem on a single processor to the time required to solve the same problem on a parallel computer with p identical processors [27]. Speedup is denoted by the symbol S.

$$S = T_s / T_p$$

2.2.4 Efficiency

Only an ideal parallel system containing p *processors can deliver a speedup equal to* p*. In practice, ideal behavior is not*

achieved because while executing a parallel algorithm, the processors cannot devote 100 percent of their time to the computations of the algorithm. Efficiency is a measure of the fraction of time for which a processor is usefully employed; it is defined as the ratio of speedup to the number of processors [27]. In an ideal parallel system, speedup is equal to p *and efficiency is equal to one. In practice, speedup is less than* p *and efficiency is between zero and one, depending on the effectiveness with which the processors are utilized. Efficiency is denoted by the symbol E. Mathematically, it is given by*

$$E = S / P$$

2.2.5 Scalability

It is a well-known fact that given a parallel architecture and problem size of fixed size, the speed up of a parallel algorithm does not continue to increase with increasing number of processors. The speedup tends to saturate at a certain value.

The effect of increasing the problem size keeping the number of processors constant can degrade the performance [26]. We know that the total overhead function T_o is a function of both problem size W *and the number of processors p. In many cases, T_o grows sub-linearly with respect to* W. *In such cases, we can see that efficiency increases if the problem size is increased keeping the number of processors constant. For such algorithms, it should be possible to keep the efficiency fixed by increasing both the size of the problem and the number of processors simultaneously. For instance, in following Table the efficiency of adding 64 numbers using four processors is 0.80. If the number of processors is increased to 8 and the size of the problem is scaled up to add 192 numbers, the efficiency remains 0.80. Increasing p to 16 and n to 512 results in the same efficiency. This ability to maintain efficiency at a fixed value by simultaneously increasing the number of*

processors and the size of the problem is exhibited by many parallel systems. We call such systems scalable parallel systems. The scalability of a parallel system is a measure of its capacity to increase speedup in proportion to the number of processors. It reflects a parallel system's ability to utilize increasing processing resources effectively. The scalability and cost-optimality of parallel systems are related.

N	*P=1*	*P=4*	*P=8*	*P=16*	*P=32*
64	*1.0*	*0.80*	*0.57*	*0.33*	*0.17*
192	*1.0*	*0.92*	*0.80*	*0.60*	*0.38*
320	*1.0*	*0.95*	*0.87*	*0.71*	*0.50*
512	*1.0*	*0.97*	*0.91*	*0.80*	*0.62*

Table 5.1 Efficiency as a function of n *and* p *for adding* n *numbers on* p *processors.*

Example 2.3 Scalability of adding n numbers for the cost-optimal addition of n numbers on p processors n= p log p. As shown in above Table, the efficiency is 0.80 for n= 64 and p= 4. At this point, the relation between n and p is n = 8p log p. *If the number of processors is increased to eight, then* 8p log p= *192. Table shows that the efficiency is indeed 0.80 with* n= *192 for eight processors. Similarly, for* p= *16, the efficiency is 0.80 for* n = 8p log p *= 512. Thus, this parallel system remains cost-optimal at an efficiency of 0.80 if n is increased as* 8p log p.

2.3 Programming Environment for Distributed Computing

In the initial days of computing, all the data & computation were centralized on a mainframe computer. Users had to connect to this central computer through dumb terminals. Later, with the proliferation of personal computers, more & more data were being stored and processed at the user's desktop. Although this process moved data closer to where it was being used, it made data-sharing a difficult task. This was one of the motivations for developing the distributed computing paradigm.

Distributed computing can be defined as computing that involves the cooperation of two or more machines communicating over a network

To make distributed computing feasible, we need environments, which support certain basic services to provide the advantages. These are: -

- ***Parallel Virtual Machine***
- XPVM
- Harness

2.3.1 Parallel Virtual Machine

Stand-alone workstations delivering several tens of millions of operations per second are commonplace, and continuing increases in power are predicted. When these computer systems are interconnected by an appropriate high-speed network, their combined computational power can be applied to solve a variety of computationally intensive applications. Further, under the right circumstances, the network-based approach can be effective in coupling several similar multiprocessors, resulting in a configuration that might be economically and technically difficult to achieve with supercomputer hardware.

The advances in high-speed networking promise high throughput with low latency and make it possible to utilize distributed computing for years to come. Consequently, increasing numbers of universities, government and industrial laboratories, and financial firms are turning to distributed computing to solve their computational problems. The objective of PVM is to enable these institutions to use distributed computing efficiently.

The PVM software provides a unified framework within which parallel programs can be developed in an efficient and straightforward manner using existing hardware [11]. PVM enables a collection of heterogeneous computer systems to be viewed as a single parallel virtual machine. PVM transparently handles all message routing, data conversion, and task scheduling across a network of incompatible computer architectures.

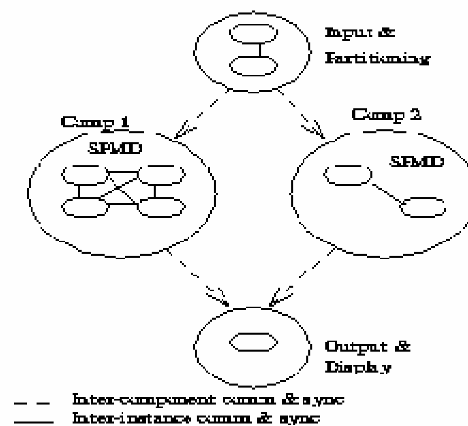


Figure 2.6 PVM Computation Model

In PVM the user writes his application as a collection of cooperating tasks. Tasks access PVM resources through a library of standard interface routines. These routines allow the initiation and termination of tasks across the network as well as communication and synchronization between tasks. The PVM message-passing primitives are oriented towards heterogeneous operation, involving strongly typed constructs for buffering and transmission. Communication constructs include those for sending and receiving data structures as well as high-level primitives such as broadcast, barrier synchronization, and global sum [11].

PVM tasks may possess arbitrary control and dependency structures. In other words, at any point in the execution of a concurrent application, any task in existence may start or stop other tasks or add or delete computers from the virtual machine. Any process may communicate and/or synchronize with any other. Any specific control and dependency structure may be implemented under the PVM system by appropriate use of PVM constructs and host language control-flow statements.

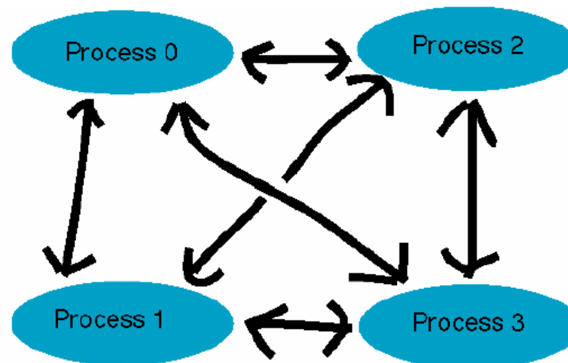


Figure 2.7 Interprocess Communication

With PVM, multiprocessor systems can be included in the same configuration with workstations. Shared-memory computers with a small number of processors can be linked to deliver supercomputer performance. The virtual machine hides the configuration details from the programmer. The physical processors can be a network of workstations, or they can be the nodes of a multicomputer. The programmer doesn't have to know how the tasks are created or where they are running; it is the responsibility of PVM to schedule user's tasks onto individual processors. The user can, however, tune the program for a specific configuration to achieve maximum performance [11].

2.4 Graph Theoretic Preliminaries

Since the definitions of graph theoretical concepts differ somewhat in the literature, we define here the basic concepts. The definitions are adapted from references [1, 2, 9, 13].

Definition: A directed graph G is a pair $(V; E)$ where V is a set of elements called vertices and $E \subseteq V \times V$ is a set of ordered pairs called edges. The cardinality of V is denoted by n and the cardinality of E by e. The size of graph G is $n + e$. Given an edge (v; w), v is the tail and w the head of the edge. A sub graph of a graph $G = (V; E)$ is a graph $S = (V^1; E^1)$ where $V^1 \subseteq V$ and $E^1 \subseteq E$.

Definition: If (u; v) is an edge of G, we say that u is adjacent to v, and v is adjacent from u. The number of vertices adjacent to v is the in-degree of v, denoted Indeg(v), and the number of vertices adjacent from v is the out-degree of v, denoted Outdeg(v).The in-degrees and out-degrees are connected by the following equation:

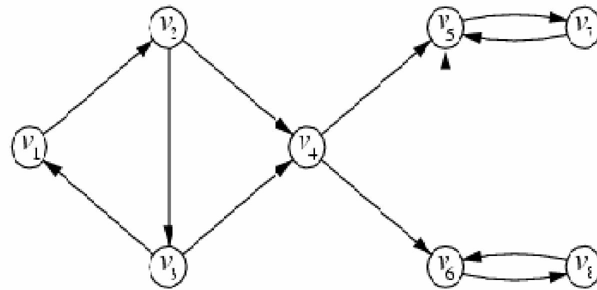
$$\sum_{v \in V} \text{Indeg}(v) = \sum_{v \in V} \text{Outdeg}(v) = e$$

To process graphs, we have to select a representation for them. One common representation is the adjacency matrix.

Definition: An adjacency matrix of a graph $G = (V, E)$ is an $n \times n$ Boolean matrix A such that A[i, j] is true iff (if and only if) G has an edge (v_i, v_j).

Definition: An adjacency list AdjFrom(v) of vertex v is a list that contains the vertices adjacent from v. The adjacency list representation of a graph consists of the adjacency lists of its vertices.

Example 2.4 In Figure 2.8, we present an example graph G = *(V,E).* The vertices are shown as circles and the edges as arrows going from the tail of the edge to the head of the edge. Below G we present its adjacency matrix and adjacency list representations.



	1	2	3	4	5	6	7	8
1	0	1	0	0	0	0	0	0
2	0	0	1	1	0	0	0	0
3	1	0	0	1	0	0	0	0
4	0	0	0	0	1	1	0	0
5	0	0	0	0	0	0	1	0
6	0	0	0	0	1	0	0	1
7	0	0	0	0	1	0	0	0
8	0	0	0	0	0	1	0	0

$$
\begin{aligned}
AdjFrom(v_1) &= \{v_2\} \\
AdjFrom(v_2) &= \{v_3, v_4\} \\
AdjFrom(v_3) &= \{v_1, v_4\} \\
AdjFrom(v_4) &= \{v_5, v_6\} \\
AdjFrom(v_5) &= \{v_7\} \\
AdjFrom(v_6) &= \{v_5, v_8\} \\
AdjFrom(v_7) &= \{v_5\} \\
AdjFrom(v_8) &= \{v_6\}
\end{aligned}
$$

Figure 2.8 An example graph $G = (V; E)$ and its representations.

Definition: A path from vertex v_0 to v_k vertex in G, denoted $v_0 \rightarrow^* v_k$, is a sequence of edges of the form (v_0, v_1); (v_1, v_2);. ;(v_{k-1}, v_k), where each edge is in *E*. The length of a path is the number of edges in it. A path from v to u is non-null, denoted $v \rightarrow^+ u$, if its length is positive. A path is simple if all edges and vertices on the path, except possibly the first and the last vertex, are distinct. A cycle is a nonnull simple path that begins and ends at the same vertex. A graph that contains no cycles is acyclic. If we do not explicitly say that a graph is acyclic, we assume that it may contain cycles, i.e., it may be cyclic.

Example 2.5 In Figure 2.8, ((v_1 ; v_2); (v_2 ; v_3); (v_3 ; v_4)) is a simple path and path ((v_1 ; v_2); (v_2 ; v_3); (v_3 ; v_1)) is a cycle. Path ((v_4 ; v_6); (v_6 ; v_8); (v_8 ; v_6); (v_6 ; v_5)) is not simple. Graph *G* of Figure 2.8 is cyclic.

Definition: The transitive closure of graph $G = (V, E)$ is a graph $G^+ = (V, E^+)$ such that E^+ contains an edge (v; w) iff *G* contains a nonnull path $v \rightarrow^+ w$. The size of the transitive closure is denoted by e^+. The successor set of a vertex v is the set Succ(v) = {w (v; w) € E^+ }, i.e., the set of all vertices that can be reached from vertex v via non-null paths. The predecessor set of a vertex v is the set Pred(v) = {u j (u,v) € E^+}, i.e., the set of all vertices that v is reachable from via non-null paths. The vertices adjacent from vertex v are the immediate successors of v and the vertices adjacent to v are the immediate predecessors of v.

Example 2.6 The transitive closure of graph *G* of Figure 2.8 is presented in Figure 2.9. As we see, the successor set of vertices v_1, v_2, and v_3 is *V* ; the successor set of vertices v_4 , v_6 , and v_8 is {v_5, v_6 ; v_7, v_8}; and the successor set of v_5 and v_7 is {v_5,v_7}. Similarly, several vertices have a common predecessor set.

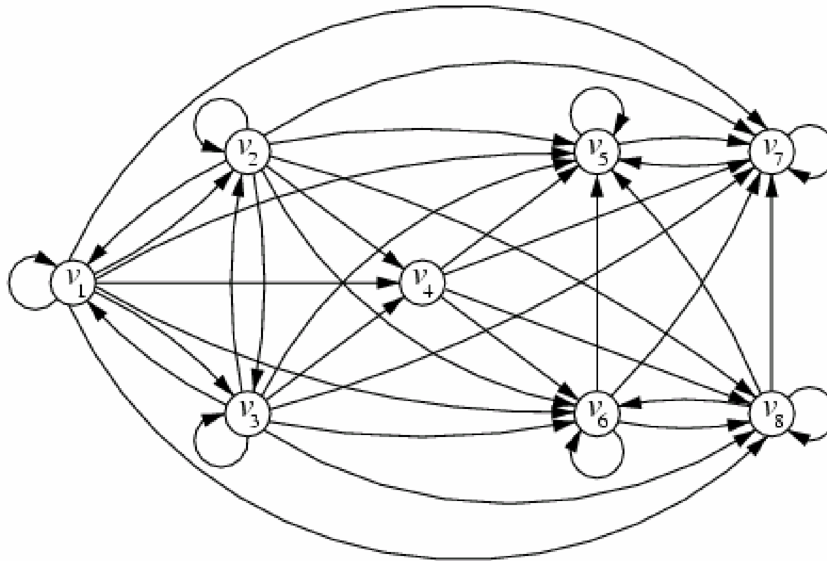


Figure 2.9 The transitive closure of graph G of Figure 2.8.

Definition: The reflexive transitive closure of graph $G = (V,E)$ is a graph $G^* = (V; E^*)$ such that E^* contains an edge (v; w) iff G contains a (possibly null) path $v \rightarrow^* w$. The size of the reflexive transitive closure is denoted by e^*.

Example 2.7 The transitive closure of graph G, presented in Figure 2.9, can be changed to the reflexive transitive closure of graph G by inserting the edge (v_4, v_4), since $E^* = E^+ \cup I$ where $I = \{(v,v) \mid v \in V\}$, the identity relation.

Definition: The transitive reduction $G_r = (V, E_r)$ of a graph $G = (V, E)$ is a graph that has the same transitive closure as G, but has as few edges as possible. The size of the transitive reduction is denoted by e_r. The transitive reduction is not necessarily unique.

Definition: A (directed rooted) tree T is an acyclic graph satisfying the following properties:

1. T has exactly one vertex r (called the root) that is the head of no edge.

2. Each vertex except the root is the head of exactly one edge.

3. From the root r to each vertex v exists a unique path $r \rightarrow^{*} v$.

If v is a vertex in a tree T, a sub tree T_v is the maximal sub graph of T that has $\{v\} \cup Succ(v)$ as its vertex set. A graph consisting of a collection of trees is a forest. A tree T (a forest F) is a spanning tree (a spanning forest) of a graph G if T (F) is a subgraph of G and T (F) contains all the vertices of G.

Example 2.8 Figure 2.10 shows one spanning tree of graph G of Figure 2.8.

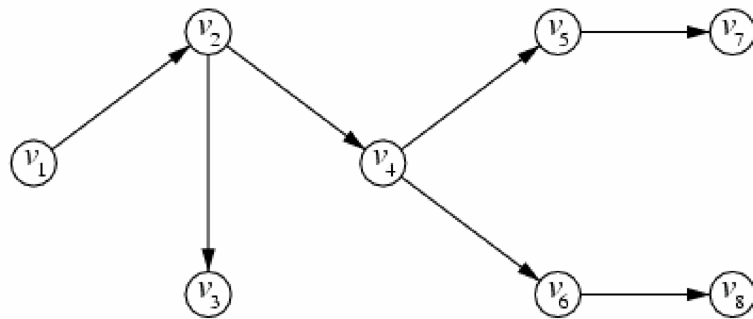


Figure 2.10 A spanning tree of graph G of Figure 2.8.

The set of edges $E \subseteq V \times V$ is a binary relation. Conversely, every binary relation R in a domain V can be seen as a directed graph with vertex set V and edge set R. We can formulate the transitive closure also in the terms of binary relations.

Definition: The transitive closure of a binary relation $E \subseteq V \times V$ is a relation $E^+ \subseteq V \times V$: $E^+ = \cup_{n>0} En$ where $E^0 = I$ and $E^{n+1} = E^n \circ E = E \circ E^n$. Here I is the identity relation and $\circ$ is the composition operator.

Example 2.9 Consider the example graph G = (V, E) of Figure 2.8. We can compute the transitive closure of any graph by using only its simple paths.

Generally, the longest simple path in a graph has at most n edges, but in our example graph the longest simple path has five edges. Thus, $E^+ = E \cup E^2 \cup E^3 \cup E^4 \cup E^5$.

Definition: Two vertices v and w in G are path equivalent iff G contains a path $v \to^* w$ and a path $w \to^* v$. Path equivalence divides V into maximal disjoint sets of path equivalent vertices. These sets are called the strong components of G. The strong component containing vertex v is denoted Comp(v). A strong component that contains only one vertex is called a trivial component. The condensation graph $\overline{\boldsymbol{G}} = (\overline{\boldsymbol{V}}, \overline{\boldsymbol{E}})$ induced by the strong components of graph G is a graph such that $\overline{\boldsymbol{V}}$ is the set of the strong components of G and $\overline{\boldsymbol{E}}$ contains an edge (X,Y) iff E contains an edge (u; v) such that Comp(u) = X and Comp(v) = Y .

Each vertex of a strong component has the same successor set; this can be used in designing efficient transitive closure algorithms.

Example 2.10 The strong components of graph *G* are encircled in Figure 2.11. The condensation graph $\overline{\boldsymbol{G}}$ induced by the strong components of *G* is shown in Figure 2.12. Note that the condensation graph is always acyclic if we remove the self loop edges.

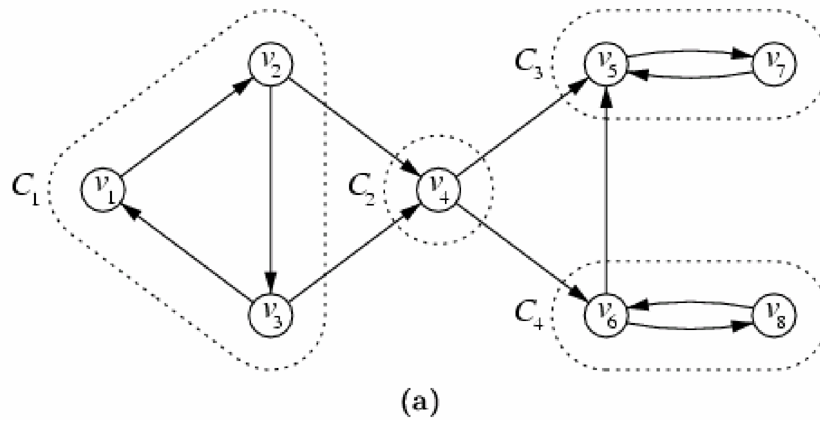


Figure 2.11 The Strong Components of figure 2.8

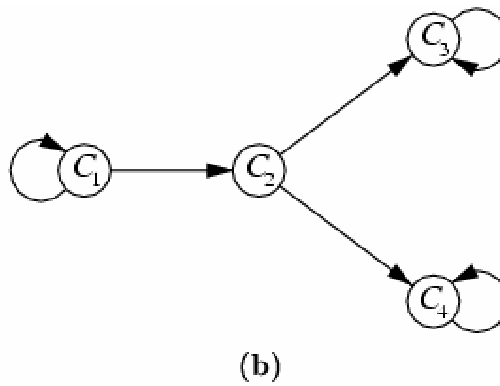


Figure 2.12 The condensation graph

2.5 Applications of Transitive Closure

Transitive closure computation is required in many applications. Most important of these are listed below:

2.5.1 Transitive Closure in Graph Theoretic Problems

Graphs provide a powerful tool to model objects and relationships among objects. The study of graphs dates back to Euler days in the 18th century, when he defined the Koinesberg bridge problem. Specifically graphs can be used to model problems in many areas such as transport scheduling, network robotics, VLSI, compilers and software engineering. The most fundamental graph problem is to compute the reachability information. In the all pairs transitive closure problem, we are given a graph *G* and want to compute the set of all pairs of nodes (*u,v*) such that u can reach *v*, i.e. there is a path in G from node *u* to node *v*.

2.5.1.1 Transitive Closure in Shortest Path Problem

Two straightforward applications of the transitive closure problem are the Boolean closure of the matrix (where + and × are the boolean *OR* and boolean *AND* operations respectively) and all-pairs shortest path computations in graphs (where + and ×) are the *MIN* and + operations). Less clear cut is the computation of minimum spanning trees : if we consider + and × as *MIN* and *MAX* then the minimum spanning tree consists of all edges (i,j) such that $A(i,j) = A^*(i,j)$, , assuming that all initial weights $A(i,j)$ are distinct [15].

The Transitive Closure and Shortest Path Problem can be defined using an abstract notion of semiring with operations + and × [15]. In both problems there are two operations + and ×, we sum the values of the paths from a given vertex i to a given vertex j using the generalized summation + and the value of each path is the multiplication of the values of its edges in terms of the generalized multiplication operation ×. In the case of TC we sum only simple

paths – simple means that no vertex appears twice, except possibly the first and last one.

Transitive Closure

Instance : A matrix *A* with elements from a semiring *S*.

Output : The matrix A^*, $A^*(i,j)$ is the sum of all simple paths from i to j.

Shortest Path Problem:

Instance: A matrix *A* with elements from a closed semiring *S*

Output: The matrix whose (i,j)th entry is the sum of all paths from i to j.

The classical algorithm for computing the transitive closure is the Warshall algorithm which takes $O(n^3)$ time steps and works on the elements of matrix A as follows ..

```
for k:=1 to n do
    for i:=1 to n do
        for j:= 1 to n do
        A(i,j) = A(i,j) + A(i,k) × A(k,j)
```

The shortest path can be computed similarly; by replacing $A(i,k) \times A(k,j)$ *with* $A(i,k) \times A(k,k)^* \times A(k,j)$, the algorithm is transformed into a version of Kleene's algorithm[15].

2.5.1.2 Transitive Closure in Transitive Reduction

Transitive reduction (also known as *minimum equivalent digraph*) is essentially the inverse operation of transitive closure[12], namely reducing the number of edges while maintaining identical reachability properties. The transitive closure of *G* is identical to the transitive closure of the transitive reduction of *G*. The primary application of transitive reduction is space minimization, by eliminating redundant edges from *G* that do not effect reachability. Transitive reduction also arises in graph drawing, where it is important to eliminate as many unnecessary edges as possible in order to reduce the visual clutter [12] .

2.5.1.3 Transitive Closure in Limited Space Algorithms

Transitive closure is an important problem in studying space complexity classes. The graph reachability problem is in NSPACE(*logn*), i.e., it can be solved by a nondeterministic algorithm using logarithmic extra space (besides the input); the algorithm just “guesses” a path from the source to the sink node by node [16]. Conversely, a nondeterministic computation that uses a given amount of extra space *S* can be modeled by a reachability probhem in a graph of exponentially larger sizes, whose nodes correspond to the configurations of the machine and the arcs correspond to the transitions. Graph reachability is a complete problem for the class NSPACE(*logn*) [16] . Immerman and Szelepcsenyi showed recently the surprising result that nondeterministic space complexity classes are closed under complementation, by providing a *logn*-space nondeterministic algorithm for verifying that a source node cannot reach a sink node**.**

*The relation of nondeterministic to deterministic space complexity is an old open problem. By Savitch’s result, NSPACE(*logn*) is contained in DSPACE(*log2n*). The deterministic log2n-space algorithm for transitive closure that is implied by this simulation rediscovers the same path facts over and over again, and as a consequence, its time complexity is not even polynomial, but n’g”. Very little has been achieved in designing*

algorithms that are simultaneously space- and time efficient.

At present, we do not know of any algorithm that works in polynomial time with less than linear space. Tompa proves that no implementation of this algorithm can achieve both properties: if an implementation uses sublinear space, then it must take superpolynomial time. He also shows that any implementation of Warshall's algorithm requires linear space.

2.5.2 Transitive Closure in Very Large Databases (VLDB)

As in many areas of computer science and other disciplines, graph theoretic tools play an important role; this is true for databases also in databases. Many concepts are best captured in terms of graphs or hyper graphs. Problems can then be formulated and solved using graph theoretic algorithms. There are a great number of such examples from schema design, dependency theory, transaction processing, query optimization, data distribution, and a host of other areas [16]. With the increasing 'non-traditional' uses of databases, several extensions have been have been proposed to the relational

query languages in order to efficiently support these applications [28]. A common operator that appears in many of these proposals is the transitive closure operation, it has been shown that every recursive query can be expressed as a transitive closure possible preceded and followed by operations already available in relational algebra, once again emphasizing the importance of transitive closure as a primitive database operation [10].

Database application domains require the computation of queries, such as transitive closure, that cannot be expressed in first-order languages like relational algebra and calculus. Solutions proposed to increase the capabilities of relational systems include the addition of a transitive closure operator as a primitive, stepping up to a more powerful logic-based query language that permits recursion, or the use of a graph-oriented language.

In the last few years the support of deductive capability in database systems (DBS) has been extensively required by the increasing number of so-called "non-standard" applications. One of the most promising approaches to support this requirement is to extend DBS to include the whole functionality underlying the theoretical foundation of first order logic. In so doing, a user of such a Deductive DBS (DDBS) can descriptively express his queries as well as his integrity constraints in first order logic. A traditional DBS user, on the other hand, has to implement such deductions by means of application programs embedding DB-calls.

In general, DDBS consist of a set of base relations (Extensional DB: EDB) with explicitly stored tuples and a set of rules defining virtual relations (Intensional DB: IDB). The tuples of a virtual relation are derived from the clauses of the IDB and

from the tuples contained in the EDB. Such a derivation can be achieved by the dynamic view mechanism of relational DBS, however, only to a certain extent: this mechanism cannot handle recursively defined virtual relations [10].

In the last few years this problem has received growing interest in the DB research area, leading to several proposals of strategies to efficiently evaluate recursion in logical queries. Most of the existing strategies have been developed to deal with all types of recursion. However, due to the existence of different types of recursion, it is almost impossible to develop a universal strategy, which handles all types of recursions and at same time works efficiently. For this reason, researchers have concentrated on a special type of recursion - transitive closure problem, since most of the recursion problems occurring in real world applications are of this type.

Furthermore, some large classes of recursive queries, e.g. linear and nonlinear recursive queries, can be translated into a transitive closure problem preceded and followed by relational algebra operations.

Transitive closure operation has been widely recognized as a necessary extension to relational query languages.

2.5.3 Transitive Closure in Relational Databases

Relational Database Management Systems (RDBMS) are a theoretically sound and industrially proven technology for managing large amounts of shared data, but RDBMS have not been used for managing knowledge bases. This is because most RDBMS do not support transitive closure queries, which are an indispensable tool for managing concept such networks.

Transitive closure computation is extensively used in RDBMS queries [9]. Relational databases can be viewed as labeled "directed" hyper graphs, where the nodes are the domain elements, the labels are the database relation schemes, and the labeled edges are the tuples of the relations. If the relations are binary, then the database is simply a directed labeled graph (with multipledges allowed).

The generalization relationship ('ISA') is probably the most commonly used relationship in relational databases. One common use case for querying ISA links is to test whether a given concept matches an expected concept. This requires querying the ISA relationship between the two concepts. The ISA relationship however is transitive. With the SQL standard, implemented by most commercial RDBMS today, one cannot query the complete transitive

closure. Instead one would either write complex queries that allow a transitive chain up to a certain length, or one would need to write a program that would walk the links to form the transitive closure. Programmatically following links is not only cumber-some (requiring programming instead of just ad-hoc queries,) it is also slow since many queries and their responses need to be communicated between the database client and server.

2.5.4 Transitive Closure in VLSI Test Generation

Transitive Closure application of deriving of deriving all logical conclusions implied by a set of binary relations is found extremely useful in VLSI test generation.

VLSI test generation is simply a process of generating test patterns in order to detect failures in a circuit and is known to be an NP-complete problem [17, 18]. Conventionally, it is characterized as a search of *N*-dimensional 0-1 state spaces, where *N* is number of primary inputs in a circuit [19]. Although, several algorithms [25] were developed in the past, but there is an enormous computational requirement as most of them are serial algorithms. However, today huge amount of computational power is available with the affordable parallel machines and distributed network of idle workstations in most the VLSI-CAD environments which has opened a new front for the development of efficient parallel and/or distributed algorithms to harness the available raw computational power. Considering these facts, some optimization methods [7, 5, 6] have also been proposed to solve the test generation problem. These approaches have two significant advantages: First, several operations research techniques like linear and non-linear programming and graph-theoretic algorithms can be applied to the test generation problem. Second, the non-causal form of the model makes parallel processing possible. As a result, one of the most significant approaches is the transitive closure algorithm for solving VLSI test generation problem, as this approach demands a very

efficient and effective parallel or distributed solution of the problem in reasonable and affordable time.

The application of transitive closure algorithm for test generation is basically the Boolean Satisfiability approach in which we construct a formula expressing the Boolean difference between fault-free and its corresponding faulty circuit. This formula one may derive either in the form of a Boolean false function [6] or in Boolean truth function proposed by Larrabee [24] and for test generation, one needs to either minimize the false function to zero or satisfy the Boolean truth function. However, we consider the test generation problem as minimization of a Boolean false function, which consists of binary (pairwise) and ternary relations of the signals of a test generation network. For example, consider a 2-input NAND gate with a, b as inputs and c as an output of the gate. Since $c = (\overline{\boldsymbol{ab}})$, it is easy to see that the equation:

$$F_{NAND} = c \oplus (\overline{\boldsymbol{ab}}) = 0 \qquad (1)$$

is satisfied only by those values of a, b and c that satisfy the NAND gate function.

F_{NAND} is referred as the Boolean false function for the NAND gate and can be written as

$$F_{NAND} = \overline{\boldsymbol{a}}\boldsymbol{c} + \overline{\boldsymbol{b}}\boldsymbol{c} + abc \qquad (2)$$

where + denotes the logical *OR* operation. The false function for a digital circuit is the logical *OR* of the false functions for all gates in the circuit. It consists of a set of binary and ternary relations. However, the problem of determining a signal assignment that satisfies the false function is equivalent to minimizing the energy function of the digital circuit [5], where a,b, c . . . can be treated as arithmetic variables. For test generation, determining signal values that satisfy the Boolean equation derived for the test generation constraint network in which fault-free circuit is first augmented with the faulty copy of the circuit in order to incorporate the necessary conditions for fault activation and path sensitization. Then, the Boolean equation is being derived

for the test generation constraint network, which consists of binary and ternary relations. An implication graph is being constructed for the binary relations in terms of a directed graph.

For example, the term $\overline{\boldsymbol{ac}}$ represents the binary relations $\bar{\boldsymbol{a}} \rightarrow c$ and $\bar{\boldsymbol{c}} \rightarrow a$ and the term $\overline{\boldsymbol{bc}}$ represent the binary relations $\bar{\boldsymbol{b}} \rightarrow c$ and $\bar{\boldsymbol{c}} \rightarrow b$, where $\rightarrow$ denotes logical implication.

The transitive closure based algorithm is a sequence of two main steps that are repeatedly executed [22]: transitive closure computation and decision-making. Since the implication graph only includes local pair wise (or binary) relations, it is a partial representation of the netlist. Higher-order signal relationships are represented as additional ternary relations. The transitive closure of the implication graph determines global pair wise logical relationships among all signal pairs. These relationships either force values on some signals or indicate contradictory requirements caused by a redundant fault. A test is found if signals thus determined satisfy the Boolean equation. Otherwise, this algorithm enters into the decision-making phase in which an unassigned signal is fixed and the transitive closure is updated to determine logical consequences of this decision.

Transitive closure determines four basic conclusions: contradiction – an impossible signal relation, fixation – a 0 or 1 value for a signal, identification – two signals must assume identical value, and exclusion – two signals must not assume certain states. A key feature of the algorithm [22] is that dependencies derived from the transitive closure are used to reduce ternary relations to binary relations that in turn dynamically update the transitive closure. The signals are either determined from the transitive closure or are enumerated until the Boolean equation is satisfied. After successful attempt of

satisfying the Boolean equation, the signal values of primary inputs form the test vector.

Keeping in view of the gate complexity of today's VLSI circuits which is too high, i.e. more than 100,000 gates are common so the number of signals in such circuits would be multiple of the gate complexity that is n to be quite large.

Hence, a practical approach for transitive closure based test generation is essentially required in which parallel or distributed algorithm must consider the available number of processors and the computational power available with the processors and/or machines. Serious efforts are being made in the consideration of the important factors for the application of an adequate parallel and/or distributed algorithm for transitive closure computation so that real-life problems like test generation could be solved in reasonable time. The proposed practical solution is described in the next chapter.

2.6 Transitive Closure Algorithms

Over the years, researchers have proposed number of algorithms for transitive closure computation. These algorithms usually fall into two categories, serial as well as parallel and a lot of theoretical study has been done on this topic. This section describes the most significant contributions made so far.

2.6.1 Serial Warshall Algorithm

Warshall's algorithm is probably the best-known sequential transitive closure algorithm. The algorithm transforms an adjacency matrix representing the input graph into an adjacency matrix representing the transitive closure by scanning the potential paths of the input graph in three nested loops between machines [8].

Given an $n \times n$ matrix of elements a_{ij} over a n-node graph with a_{ij} being 1, if there is an arc from node i to node j and 0 otherwise, the warshall algorithm requires that every element of this matrix be 'processed' column by column from left to right, and from top to bottom with a column. 'Processing' of an element a_{ij} involves examining if a_{ij} is 1, and if it is, them making every successor of j a successor of i [8].

2.6.1.1 Illustration of Warshall Algorithm

Consider a matrix A, of $n \times n$, whose elements represent edges of corresponding graph. Each edge in the graph is represented in matrix by placing the value of corresponding element as 1 otherwise we place the value as 0 of the element. The sequential version of the algorithm can be written as:

```
for k: = 1 to n do
        for i:= 1 to n do
                if i != k and A[i,k] then
                        for j:= 1 to n do
                        A[i,j] := A[i,j] or A[k,j];
```

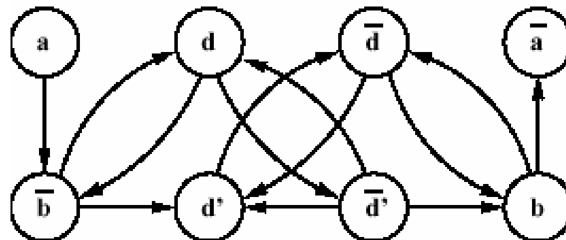


Figure 2.13 An example directed graph.

To illustrate this serial algorithm, we may consider a directed graph as shown in figure 2.13. It is assumed that all the nodes contain self loops. The transitive closure of the graph is computed by complete execution of the serial algorithm, which implies how each node is connected to other nodes in the

graph. This sequential algorithm for computing transitive closure consists of three loops and each of which is to be executed n times. So, the time complexity will be O (n^3).

This implies that Warshall's algorithm is not economical for a sparse input graph. Warshall's algorithm examines the matrix position A[i; k] column by column, but the matrix positions A[i,j] and A[k; j] row by row [8].

The drawback of Warshall's algorithm is that it scans by columns (so that it may "or" by rows). This has very bad consequences in a paging environment if the entire matrix will not fit in real main storage: the whole matrix must be paged in at least n times, where the matrix is n × n (it must be paged in once for each column scan; we assume an LRU page replacement algorithm, and assume that a row of the matrix will fit in one page). If we could scan by rows (and also "or" by rows), the algorithm would have better paging characteristics, and furthermore various coding tricks would emerge for speeding up the row scan [8].

Unfortunately, if Warshall's algorithm is modified to scan (and "or") by rows, it doesn't work, at least not in one pass. As a simple example, consider the digraph:

Using Warshall's algorithm modified to scan by rows, the arcs (1,3) and (1,1) are added during the scan of row 1. In scanning row 2, the arc (2, 1) is added. Scanning row 3 adds (3, 2) and (3, 3), but it takes a second pass to add the final arc (2, 2).

2.6.2 Serial Warren's Algorithm

Warren noticed that we could change the Warshall algorithm to examine also the position *A[i,k]* row by row if we split the algorithm into two passes [8]. The first pass tests the positions below the main diagonal of the matrix and the second tests the positions above the main diagonal. Both algorithms examine and set the same positions of the input matrix, but in a different order.

```
for i:= 2 to n do
   for k:= 1 to i-1 do
       if A[i,k] then
            for j:= 1 to n do
            A[i,j] := A[i,j] or A[k,j]
             for i:= 1 to n-1 do
            for k:= i+1 to n do
                        if A[i,k] then
                  for j:= 1 to n do A[i,j] := A[i,j] or A[k,j];
```

In the number of page faults generated, Warren's algorithm is better. Warren showed that if the adjacency matrix of a sparse graph does not fit into the main memory and if the LRU (least recently used) page replacement policy [117] is used, then his algorithm causes fewer page faults than Warshall's by a factor that is between one and n/2.The algorithm takes O (n^3) time steps.

2.6.3 Parallel Algorithm for Transitive Closure using Double Hash-Based Clustering

To reduce the overall complexity of the sequential algorithm of transitive closure, we need a parallel approach. The efficient implementation of a transitive closure operator today appears to be one of the keys to the evaluation of recursive queries in a deductive DBMS. The idea of decreasing execution times of relational operations by the intensive use of parallelism is very prevalent. Its effectiveness has been proven in numerous propositions and parallel implementations. The more recent data base machine architectures base their execution strategies around parallel algorithms. The multiple backend approach is very dominant; besides the performance improvements, it provides other advantages such as reliability and possibilities for operating in a degraded mode. This approach lends itself to situations in which high-volume sequential processing is performed which is the case with transitive closures. One of the bases for this type of architecture is the division among the processors of the data to be processed. A simple, generally adopted way to perform this division is to form, using an appropriate function, as many hash buckets as there are processors available.

The Double Hash-Based Transitive Closure algorithm is directly usable in such an environment [14]. Thanks to the use of hashing, it lends itself naturally to parallelization. With a multiple backend configuration, as in, the join loop processing time can be divided by p, where p is the number of processors available.

The advantages of parallelization rest on the following principle: the initial relation is divided horizontally into several fragments, each of which is then processed by a separate processor. In an environment where memory is not shared ("shared nothing"), each processor processes the data from its own disk

in its own memory. Each processor- memory-disk set can be viewed as a node in a network. In a centralized multi-processor architecture, the network consists simply of a bus, whereas with a divided configuration it consists of a veritable communication network. The main problem for parallel execution of a relational operation in general, and of a transitive closure in particular, is to give a maximum amount of local tasks to each processor while limiting data and message transfers at the same time. Unlike joins, the execution of a transitive closure cannot be totally "localized" by simple horizontal partitioning of the initial relation. In fact, tuples newly-produced in a particular node at a particular moment in the processing can be required by another node in order to continue the task. Transfers are thus absolutely necessary during execution of the transitive closure. However, inter-processor transfers can be limited to the smallest set necessary for each step.

In this algorithm the idea of a division of tuples among several nodes is maintained, thanks to a partitioning by hashing on the attribute that will be the join attribute [14]. The second hashing permits the composition of the buckets to be transferred in preparation for the next iteration. The sets of tuples exchanged between processors are minimized and formed directly. The bucket indices (i.e. the hash values) represent, for each bucket, the receiving node. Without extra processing or special messages, transfers are thus performed in a minimum of time. No extra synchronization or master node is necessary.

In a multiple backend configuration, each processor performs the same action. The data are divided into as many subsets as there are processors available for the operation. If each processor has its own disk (we shall assume, for simplicity's sake, such a configuration), the data are divided among these disks by hashing. At the outset, the relation R is distributed among as many sub-relations as there are processors. This division is calculated by hashing on the X attribute. Each sub-relation is rehashed locally according to the Y

values. Overall, *n** sub-buckets are to be dealt with (hashing in *n* buckets on the X attribute followed by rehashing of each bucket in n buckets on the Y attribute). If p processors are available, $n = p$ is chosen.

Let p be the processor index; the buckets of index i_p from R, *i* between 0 and *n-l*, will be clustered on the disk assigned to processor *p* (it is supposed that there is one disk per processor). The buckets of index *p* from the different ΔR's, produced during each step, are likewise distributed. In this way, for one iteration, the join of buckets R_{ip} and a ΔR_p remains local to each processor. No transfers whatsoever are necessary per iteration for the processing of each step of an algorithm. The processing time during this phase of the algorithm is thus divided by the number of processors *p*. Actually, a non-uniformity in the hashing can considerably reduce this factor because the total time used is that of the slowest processor, i.e. the processor processing the largest volume of data. This problem, shared by all hashing methods, emphasizes the importance of a careful choice of the partitioning function.

The results from each join R_{ip} IXI ΔR_p must next be processed by processor *i*. Indeed, these tuples have the *i* hash value according to X; they form the Δ R_i sub-relation for the next iteration, which will be processed by processor *i.* A transfer is thus necessary. However, no rehashing has to be performed. The tuples destined for .processor *i* are already stored in a specific sub-relation, thanks to the values from the second hashing. An initial version of the parallel algorithm can be given:

```
for each processor p do
    R*_p := R_p
    ΔR_p := R_p ;
    while ΔR_p contains new tuples do
        begin
            for i:=0 to p-1 do
                begin
                    Z_ip := ΔR_p ° R_ip ;
                    send Zip to node i
                end;
            receive (p-1) Z_pk ;        k=0..p-1,  k≠p
            ΔR_p := ∪_k Z_pk ;          k= 0..p-1
            R_p* := R_p* ∪ ΔR_p
        end ;
```

The final result (R*) will be formed simply by the union of the different R^*_p's. It must be pointed out that this union does not require a suppression of duplicate tuples. Indeed, the sets to be combined are hash buckets and are thus necessarily disjoint. In this algorithm [14], each processor awaits the arrival of all of the new tuples produced during one iteration in order to begin the next iteration (each of the *p* processors receives *p-l* sub-relations coming from *p-l* other nodes). A procedure that performs the "receive" function can easily take this role and verify that all the sub-relations have arrived.

The next chapter deals with the proposed parallel algorithm for transitive closure computation.

Chapter 3

Proposed Parallel Transitive Closure Algorithm

The complexity of sequential transitive closure algorithms could be drastically reduced if the transitive closure could be computed in parallel for which efficient parallel algorithms is required. Although number of parallel transitive closure algorithms have been developed in the past but in order to see the reduction in the overall time complexity a parallel algorithm is described here.

We present a new efficient parallel algorithm for transitive closure. In our approach, efforts have been made to develop a true efficient parallel / distributed algorithm by considering resources normally available in the distributed computing environment.

A very important factor for an efficient parallel algorithm development is the dividing mechanism of the complete problem into sub problems so that each workstation has sufficient workload and takes almost the equal amount of time to process the same. Another important factor which leads to an efficient algorithm is the proper load distribution among different machines so that minimum communication overhead will be required between machines. Both these factors have been considered for an efficient algorithm development in our approach.

3.1 Parallel Algorithm Design

We propose here a multi-processor implementation of a transitive closure computation. The algorithm is based on divide and conquer strategy: the task is divided into number of smaller tasks that

can be assigned to several processors. A very large transitive closure thus amounts to a collection of smaller operations. The proposed algorithm divides the transitive closure computation into n *tasks using data parallelism*

The proposed parallel transitive closure algorithm computes the transitive closure of a cyclic directed graph using Master-Slave approach and the PVM Message-Passing architecture. The algorithm employs the data parallelism

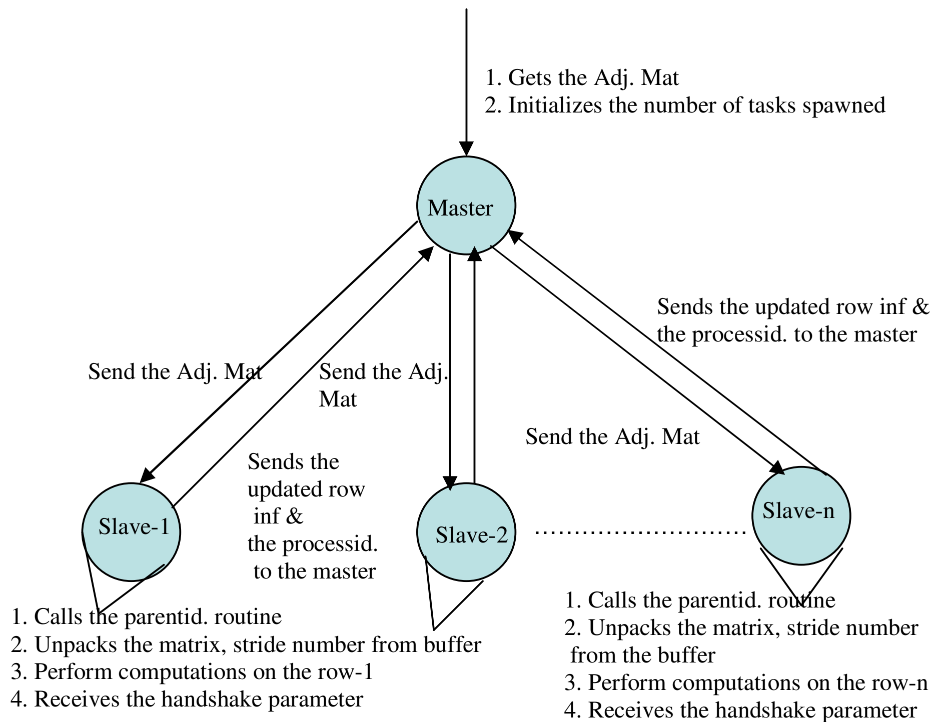


Figure 3.1 Logical Design of the Master-Slave Communication

The parallel algorithm for transitive closure computation of a graph G consists of n vertices which means the adjacency matrix of the graph is of size $n \times n$. Firstly, the master process gets the adjacency matrix from the user whose transitive closure is to be computed. The master process divides the computational tasks into number of sub-tasks depending on the available slave processors each of which is assigned to an independent slave processor for execution. The distribution of subtasks in the transitive closure computation is based on the rowwise distribution of the matrix. In a parallelized algorithm

with rowwise distribution, all the *n* sub tasks execute concurrently as each slave does computation for one row.

In this algorithm, the master process first creates *n* slaves and the entire adjacency matrix is first transferred to each slave. Then, each slave will work on to its assigned row, which means that i^{th} processor will work on i^{th} row. The ith processor will scan the entire row, and wherever it find the value of element 1, it will scan the corresponding row, change the value of element to 1, if it finds the corresponding element of second row as 1. The whole process is illustrated in the figure 3.1.

3.1.1 Illustration of Proposed Parallel Algorithm

	a	$\bar{a}$	b	$\bar{b}$	d	$\bar{d}$	d'	$\bar{d}'$
a	1	0	0	1	0	0	0	0
$\bar{a}$	0	1	0	0	0	0	0	0
b	0	1	1	0	0	1	0	0
$\bar{b}$	0	0	0	1	1	0	1	0
d	0	0	0	1	1	0	0	1
$\bar{d}$	0	0	1	0	0	1	1	0
d'	0	0	0	0	0	1	1	0
$\bar{d}'$	0	0	1	0	1	0	1	1

Figure 3.2 Adjacency Matrix

	a	$\bar{a}$	b	$\bar{b}$	d	$\bar{d}$	d'	$\bar{d}'$
a	1	1	1	1	1	1	1	1
$\bar{a}$	0	1	0	0	0	0	0	0
b	0	1	1	0	0	1	1	0
$\bar{b}$	0	1	1	1	1	1	1	1
d	0	1	1	1	1	1	1	1
$\bar{d}$	0	1	1	0	0	1	1	0
d'	0	1	1	0	0	1	1	0
$\bar{d}'$	0	1	1	1	1	1	1	1

Figure 3.3 Transitive Closure of above Adjacency Matrix

Consider the adjacency matrix shown in Figure 3.2, to compute the transitive closure using the parallel algorithm, we need eight iterations of the variable j and in each iteration, one row is assigned to one slave, e.g., in the first iteration of j, the 1st row, is being processed by 1st slave; the 2nd row, is being processed by 2nd slave; the 3rd row, is being processed by 3rd slave and so on.

Finally, after the last iteration i.e. 8th iteration, the transitive closure will be obtained.

Thus, in the worst case, scanning of each row will be required and it will take n^2 time to scan each row. So, for the worst-case time complexity of this algorithm is O (n^2). However, in the best case, scanning of only 1 row for each slave will be required, thus, for the best case; the time complexity comes out to be O (n) only.

It is reported in the literature that the transitive closure can be computed in polylog time using O ($n^{2.376}$) processors [6] or in constant time for related graph problems on processor arrays and reconfigurable bus system. Although the parallel algorithm takes a polylog time or constant time but employing large number of processors is not feasible for solving real-life problems where n is fairly large. One such real-life problem is the test generation problem for VLSI circuits where the transitive closure based approach has been proposed as the future parallel solution of this complex problem [6].

3.2 Complexity Calculations

The following section discusses the complexity issues of the proposed algorithm.

3.2.1 Best Case

When there are available n *processors for computation of the Transitive Closure -*

Serial Run Time (T_s) = n^3

Parallel Run Time (T_p) = n

Number of Processors = n

Speedup = T_s/T_p = $n^3/n = n^2$

Efficiency = Speedup / Number of Processors = $n^2/n = n$ *As Serial Complexity is $O(n^3)$ and Parallel Complexity is O(n) which means, Serial Cost is proportional to Parallel Cost therefore the algorithm is optimal.*

As Serial Complexity is $O(n^3)$ and Parallel Complexity is O(n) thus using the proposed algorithm, Time Complexity for the computation of Transitive Closure can be reduced from $O(n^3)$ to O(n) using n *number of processors.*

3.2.2 Worst Case

When we have only one processor available for computation of the Transitive Closure -

Serial Run Time = n^3

Parallel Run Time = n^3

Number of Processors = 1

Speedup = $T_s/T_p = n^3/n^3 = 1$

Efficiency = Speedup / Number of Processors = 1/1 = 1

As serial complexity is O(n^3) and parallel complexity is O(n^3) which means, serial cost is proportional to parallel cost therefore algorithm is Optimal. As serial complexity is O(n^3) and parallel complexity is O (n^3) which is the same as the old parallel algorithm hence the cost is same. In essence what is done is that the computation is distributed evenly to all the available processors to achieve the result in as short time interval as possible.

The implementation details of this proposed parallel algorithm are given in the next chapter.

Chapter 4

Implementation Details and Experimental Results

4.1 Implementation Details

The proposed algorithm has been implemented in the real heterogeneous distributed network environment comprised of P-II and P-III Linux machines of different speeds and architecture networked via LAN using Parallel Virtual Machine (PVM) version 3.0. The PVM system currently supports C, C++, and Fortran languages. We have used C language to develop the underlying application. The algorithm has been implemented using the Master/Slave Model. We have successfully tested the prototype of the proposed solution for different problem size, i.e. n for a given n × n matrix of 0's and 1's.

The Master program is responsible for initializing the processors, distributing the entire matrix and the information about the allotted rows to the available

slave processes, sending/receiving rows from slaves and assembling the resultant matrix. Each slave receives from the Master program gets the block it needs for computation, computes and sends back the results.

The implementation has been done using a master-slave configuration in which the master processor spawns n tasks where each task does the computation on a single row. Each task is a member of a group with a unique identifier. All tasks are identified by an integer task identifier (TID). Messages are sent to and received from tids. Since tids must be unique across the entire virtual machine, they are supplied by the local pvmd and are not user chosen.

4.1.1 Computation at Master

The computation at master side consists of the following steps:

(i) It gets the adjacency matrix from the file and initializes the number of tasks to be spawned. The tasks spawned are equal to the number of rows in the matrix.

(ii) It sends this adjacency matrix to the slaves (i.e. the tasks spawned).Enrolls the calling tasks in the group that is defined in the parameter.

(iii) It multicasts the message stored in the active send buffer to the number of tasks that are spawned specified by their respective tids. The contents of the message can be distinguished by the message identifier.

(iv) It blocks the calling processes until all the processes in a group have called it.

(v) It receives the information about the tasks spawned from the slaves and un-packs the process id and stores in the array iteratively.

(vi) It unpacks the rows of the resultant matrix from the tasks spawned iteratively.

(vii) It displays the resultant transitive closure of the adjacency matrix.

(viii) It performs the handshaking to check whether the slaves have correctly received the data, if the data is not received it sends the data again and receives the resultant matrix by following the above procedure again.

(ix) It kills the slaves and writes the final output to a file. This is the resulting transitive closure of the adjacency matrix.

(x) The total execution time *t* is calculated, where t = t2 – t1; t2 = final time and t1 = initial time.

4.1.2 Computation at Slave

The computation at master side consists of the following steps:

(i) It calls the parent id routine to get the parentid and receives the groupid.

(ii) It unpacks the matrix, size of the matrix and the stride number from the buffer.

(iii) For each row the slave processor scans it for entry 1, if it finds a 1 at element *(row,i).* Now it checks *(i,j)* and *(row,j)* for 1 . If the element is found then it changes the value of *(row,j)* to 1.Otherwise

checks for *(row,i+1)* and so on till the last element of the row is checked. Insert the coordinates of *j,* in case *j* is less than *i*, into a stack of maximum size of *n* where *n* is the number of elements in the row. Here computation is done based on data parallelism concept.

(iv) It sends this information to the master by specifying its process id

(v) It receives the handshake parameter from the master and if the data has not been received then it resends the data.

The above process is to be repeated until a single process is left. The proposed parallel transitive algorithm has been successfully tested for different problem size, i.e. n for a given $n \times n$ matrix of 0's and 1's.

4.2 Experimental Results

This section depicts the practical behavior of the parallel transitive closure algorithm for matrix size ranging from 75 × 75 to 130 × 130.

All experiments were performed on P-II and P-III Linux workstations networked via LAN using Parallel Virtual Machine 3.0.

4.2.1 Computational Time vs. Number of Processors

The following table shows the average execution time (in seconds) for the proposed algorithm for different sized matrices and for varying number of processors.

No of Processors	75× 75	100 ×100	130 ×130
1	0.05	0.11	0.36
2	0.03	0.06	0.18
4	0.02	0.05	0.14

Table 4.1 Computational Time vs. Number of Processors for proposed Algorithm

The results in table 4.1 show that there is decrease in execution time as the matrix dimension increases. It implies that the proposed parallel transitive closure algorithm is suited for larger matrix dimensions. This is due to larger communication overhead in smaller matrix dimensions.

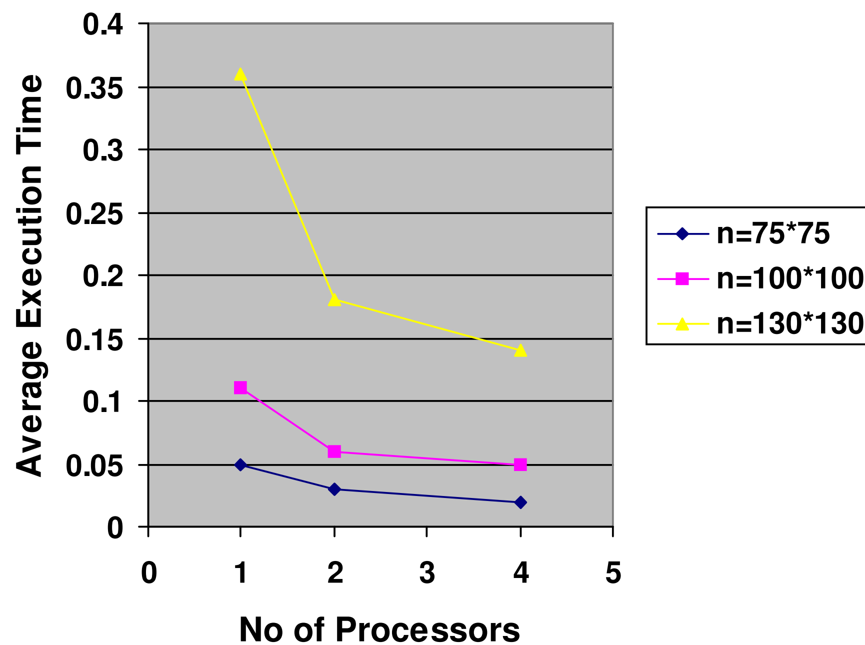


Figure 4.1 Average Execution Time (in seconds) vs. No of Processors for Transitive Closure Computation

4.2.2 Speedup vs. Number of Processors

The speedup of the average execution times of the proposed parallel transitive closure algorithm is calculated by the formula:

$$\text{Speedup} = T_s/T_p$$

where T_s is the serial time & T_p is the parallel time of the proposed parallel algorithm. The following graph shows the speedup vs. number of processors

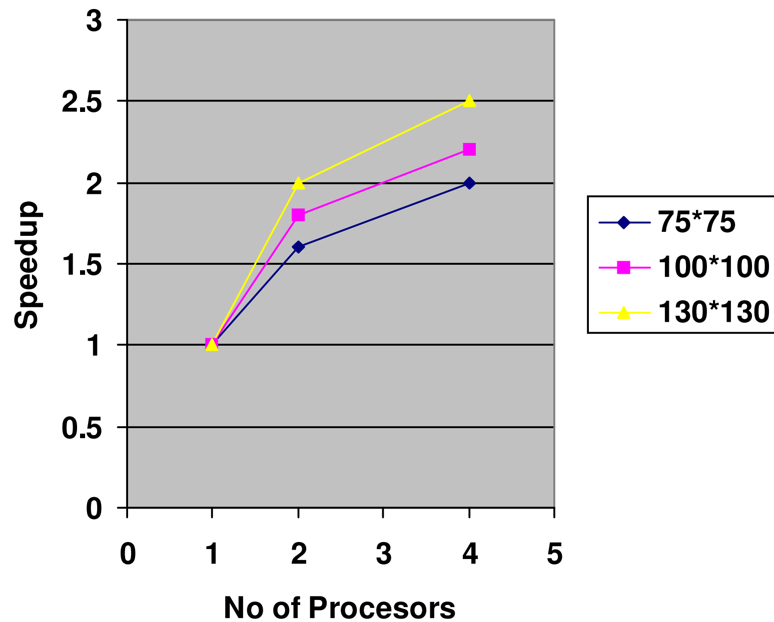


Figure 4.2 Speedup vs. No of Processors

From the above graph we can conclude that the proposed parallel algorithm for transitive closure computation shows the speedup with increase in the number of processors. It has a linear speedup. It has been observed that the speedup improves with increase in the size of the matrix. i.e. the bigger the matrix size, the more is the speedup. Therefore, parallelization is ideally required for larger matrix dimensions.

The algorithm is highly efficient because the speed up increases considerably with the increase in the number of processors. This is obvious from the above graphs. As the efficiency E is given by

$$E = S / p$$

where S is the speedup and *p* is the number of processors.

The more the speedup with increase in the number of processors, the more is the efficiency.

Chapter 5

Conclusion and Future Scope of Work

In this thesis, we have proposed the multiprocessor implementation of transitive closure computation. This chapter summarizes the main conclusions and suggests the future scope of work

5.1 Conclusion

This study focuses on the algorithmic issues related to parallel and distributed computing and suggested that input, output and concurrency should be regarded as primitives of parallel programming. The platform for parallel and distributed computing is discussed.

Our experimental study leads us to conclude that in today's era of network computing it could be quite cost-effective to solve compute-intensive problems on distributed computing concept and Parallel Virtual Machine

(PVM) has become very popular in implementation of parallel algorithms in heterogeneous computing platforms.

This thesis proposes an efficient and practical multiprocessor solution for the transitive closure computation. Parallel algorithm for transitive closure computation, divides the problem into several tasks in order to run on different processors. The proposed solution is based on the distributed transitive closure computation in which the idle time of the processors is found very low as the computational work is almost evenly distributed among the available number of machines on the network. The parallel algorithm, so developed, has been implemented by using Parallel Virtual Machine (PVM) in a distributed computing environment mostly available for VLSI-CAD related activity. The algorithm has shown good results. The experimental results reported in the thesis show the effectiveness of the proposed algorithm.

5.2 Future Scope of Work

There is a limitation in the proposed approach that for small example circuits, if we keep on adding the number of processors involved in the computation, there is an increase in the speedup but the speedup starts decreasing beyond a certain limit for small problem size due to the large communication overhead. Thus the proposed approach recommends that number of processors to be employed should be based on the size of the problem assigned to each processor. Hence, the future scope of the work will be to carry out the study for performance metrics and test the proposed algorithm for generating test patterns for large practical circuits using large number of processors in a more general-purpose parallel and/or distributed heterogeneous computing paradigm.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, "The Design and Analysis of Computer Algorithms", Addison-Wesley, Reading, Mass., 1974.

[2] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, "Data Structures and Algorithms", Addison- Wesley, Reading, Mass., 1983.

[3] A.V. Aho, R. Sethi, and J.D. Ullman, "Compilers. Principles, Techniques, and Tools", Addison-Wesley, Reading, Mass., 1986.

[4] Bolognesi and Smolka, "Fundamental results for the verification of observational equivalence: A survey", proc. of the IFIP Int. Conference on Protocol Specification, Testing and Verification VII. North-Holland, 1988.

[5] Chakradhar, V. D. Agrawal, M.L. Bushnell, "Automatic test generation using quadratic 0-1 programming", 27th ACM/IEEE Design Automation Conference, 1990, pp. 654-659.

[6] Chakradhar, Agrawal and Rothweiler, “A transitive closure algorithm for test generation”, IEEE Transactions on Computer-Aided Design, 1993, pp. 1015-1028.

[7] E.H.L. Arts and J.H. Korst, “Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimizations and Neural Computing”, Wiley, New York, 1989.

[8] E. Nuttila, “Efficient Transitive Closure Computation in Large Digraphs”, Acta Polytechnica Scandivinica, Mathemetaics and Computing in Enhh. Series No 74, Helsinki 1995, pp. 125 -130

[9] Gunther Schadow, Michael R. Barnes, Clement J. Mcdonald, “Representing and Querying Conceptual Graphs with Relational Database Management Systems is Possible”, Registries Institute and Indiana University School of Medicine, Indianapolis, pp. 1-4.

[10] H. Jagdish, R. Aggarwal, L.Ness, “A Study of Transitive Closure as a Recursion Mechanism”, proc of ACM SIGMOD 1987 Annual Conference, San Francisco, May 1987, pp. 331-344.

[11] http://www.csm.ornl.gov/pvm/pvm_home.html.

[12]http://www.toki.or.id/book/AlgDesignmanual/Book/Book4/Node163.html

[13] J.A. McHugh., “Algorithmic Graph Theory”, Prentice Hall, Englewood Cliffs, N.J., 1990.

[14] Jean Pierre, Cheiney, Christophe De, “ A Parallel Strategy for Transiitve Closure Computation using Double Hash Based Clustering”,VLDB , 1990, pp. 347-358.

[15] Ken Chan, Alan Gibbons, Marcelo Pias, Wojciech Ryterr, “On the PVM Computations of Transitive Closure and Alegbraic Path Problems”, EuroPVM-MPI97, Lecture Notes in Comp. Science, Springer Verlag, 1997, pp. 1-8.

[16] Mihalis Yannakakis, “Graph-Theoretic Methods in Database Theory”, proc of ACM, 1990, pp. 1-13.

[17] M.R. Garey and D.S. Johnson, Computers and Intractability, “A Guide to the Theory of NP-Completeness”, W.H. Freeman & Company, San Francisco, 1979.

[18] O.H. Ibarra and S.K. Sahni, “Polynomially Complete Fault Detection Problems”, IEEE Trans. On Computers, Vol. C-24, No 3, pp. 242-249, march, 1975.

[19] P.Goel, “An Implicit Enumeration Algorithm to Generate Tests for Combinatational Circuits”, IEEE Trans. On Computers, Vol. C-30, No 3, pp. 215-222, March 1981.

[20] R. Aggarwal, H. Jagdish, "Direct Algorithms for Computing the Transitive Closure of Database Relations", proc of the 13th VLDB Conference, Brighton 1987, pp. 255-256.

[21] R.E. Tarjan, "Depth first search and linear graph algorithms", SIAM Journal of Computing, pp.146-160, June 1972.

[22] Seema Bawa, G.K. Sharma, "A Parallel Transitive Closure Algorithm for VLSI Test Generation", Applied Parallel computing, proceedings PARA 2002, Springer-Verlag, Berlin Heidelberg 2002, pp. 243-252.

[23] S. Sippu and E. Soisalon-Soininen, "Parsing Theory, volume I, Languages and Parsing", Springer-Verlag, Berlin, 1988.

[24] T. Larabee, "Test pattern generation using Boolean Satisfiability", IEEE Transactions on Computer –Aided Design, Vol 7, Jan. 1992, 4-15.

[25] V.D.Aggarwal and S.C. Seth, "Test Generation for VLSI chips", IEEE Computer Society Press, Los Alamitos, CA, 1988.

[26] Vipin Kumar and Anshul Gupta, Department of Computer Science University of Minnesota, "Analyzing Scalability of Parallel Algorithms and

Architectures", journal of parallel and distributed computing,1994.

[27] Vipin Kumar, Anath Grama, Anshul Gupta and George Karypis, "Introduction to Parallel Computing - Design & Analysis of Algorithms", Benjamin / Cummings Publications.

[28] W.Yan and N.Mottos, "Transitive Closure and the LOGA- Strategy for its Efficient Evaluation", proc. 2nd Symposium on Mathematical Foundations of Database Theory, Visegrad, Hungary, pp. 1- 4.

[29] Yannis E. Ionnadis, Raghu Ramakrishnan, "Effecient Transitive Closure Algorithms", proc. of the 14th VLDB Conference, Los Angels, California 1988, pp.382-383.

[30] Y. Ioannidis, "On the Computation of thee Transitive Closure of Relational Operators", proc. of the 12th VLDB Conference, Kyoto, 1986, pp. 403-411.