PARALLELIZATION OF THE FAST ALGORITHM FOR COMPUTATION OF DOMINATORS IN A FLOWGRAPH

By

SHARMILA SHANKAR

Bachelor of Science
University of Poona
Poona, India
1985

Master of Science
Indian Institute of Technology
Bombay, India
1987

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the degree of
MASTER OF SCIENCE
July, 1992

PARALLELIZATION OF THE FAST

ALGORITHM FOR COMPUTATION

OF DOMINATORS IN A

FLOWGRAPH

Thesis Approved:

Blayne E. Mayfield
Thesis Adviser

J. Chandler

William David Miller

Thomas C. Collins
Dean of the Graduate College

ACKNOWLEDGEMENTS

I wish to express my sincere appreciation, thanks and deep sense of gratitude to Dr. Blayne Mayfield for his constant encouragement and advice throughout my thesis research. I wish to thank Dr. David Miller for his helpful suggestions throughout the study and for serving on my graduate committee. I would also like to place on record my thanks to Dr. John P. Chandler for serving on my graduate committee.

I would also like to thank my sister Suchorita, and most of all my parents, Ashish and Kamala Mookerjee for their unflinching, unquestioned, and constant support in everything that I have done so far. I wish to thank my husband Shankar for his love, understanding, moral support and his strong belief in my abilities, which helped me to give my thesis the present orientation and form.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

It is imperative that software quality be a primary concern in any software development effort, the prime objective being the efficiency of computer programs. Every computer program can be visualized as a flowgraph (See definition in Section 2.1) of edges and vertices [ROBI80] as shown in Figure 1 (page 2) with its branch (or decision points) represented by vertices, and the program codes between branch points represented by edges. The dominators (See definition in Section 2.1) problem arises in the study of global data flow analysis and object code optimization [LENG79].

The compilation process converts programs from a form which is flexible to a form which is efficient in a given computing environment. Compiler writers are challenged on the one hand by increasingly complex hardware and on the other hand by the fact that much of the complexity and rigidity of large, costly programs results from conscious efforts to build in efficiency. Methods of analyzing the control flow and data flow of programs during compilation are applied to transforming the program to improve object time efficiency. Dominance relationships, indicating which

statements are necessarily executed before others, are used to do global common expression elimination and loop identification [LOWR69].

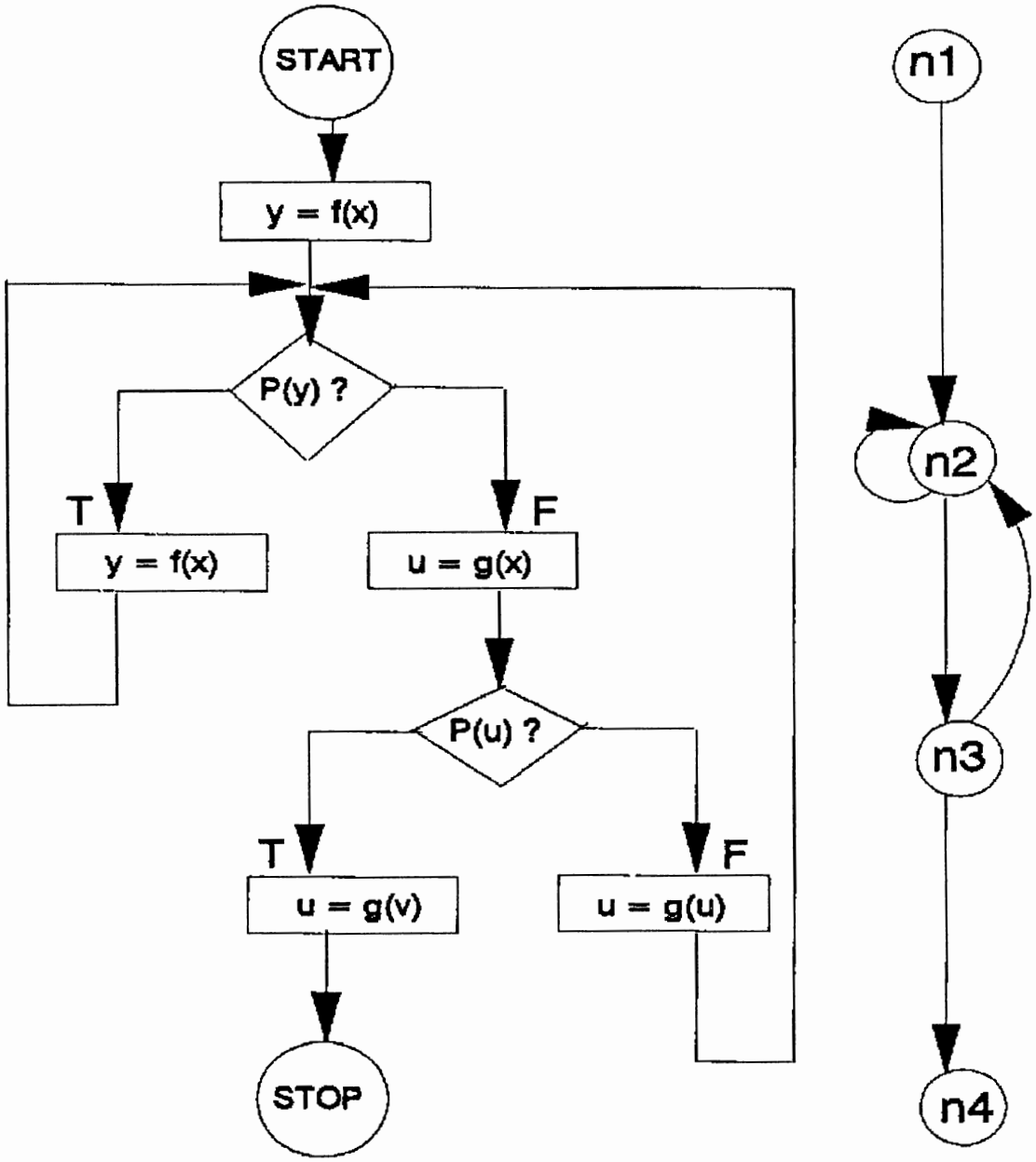

Figure 1. Computer Program modeled by a graph

The class of problems arising while analyzing computer programs for code improvement known as the "global data flow analysis problems" [HECH75], involve the local collection of information distributed throughout the program. Some examples of global data flow analysis problems are "available expressions" (expressions such as A+B are available at point p in a flow graph if every sequence of branches which the program may take to p causes A+B to have been computed after the last computation of A or B), "live variables" (variables are live in a flow graph if their current value might be used before they are redefined), and "very busy variables" (variables are busy at a point in the program if at that point they contain data that will be subsequently fetched).

In the arithmetic translator the program is broken into computational blocks whose relationship is represented by a directed graph (See definition in Section 2.1) that illustrates the flow of control through the program, with each block consisting of a sequence of statements, only the first of which may be branched to, and only the last of which contains a branch as shown in Figure 2 (page 4).

The idea of dominance relations between the blocks of a program is suggested by Lowry and Medlock. A block I "predominates" a block J if every path along a sequence of successors from a program entry block to J always passes through I as shown in Figure 3 (page 5). The relation is

transitive : If I predominates J and J predominates K, then I predominates K.

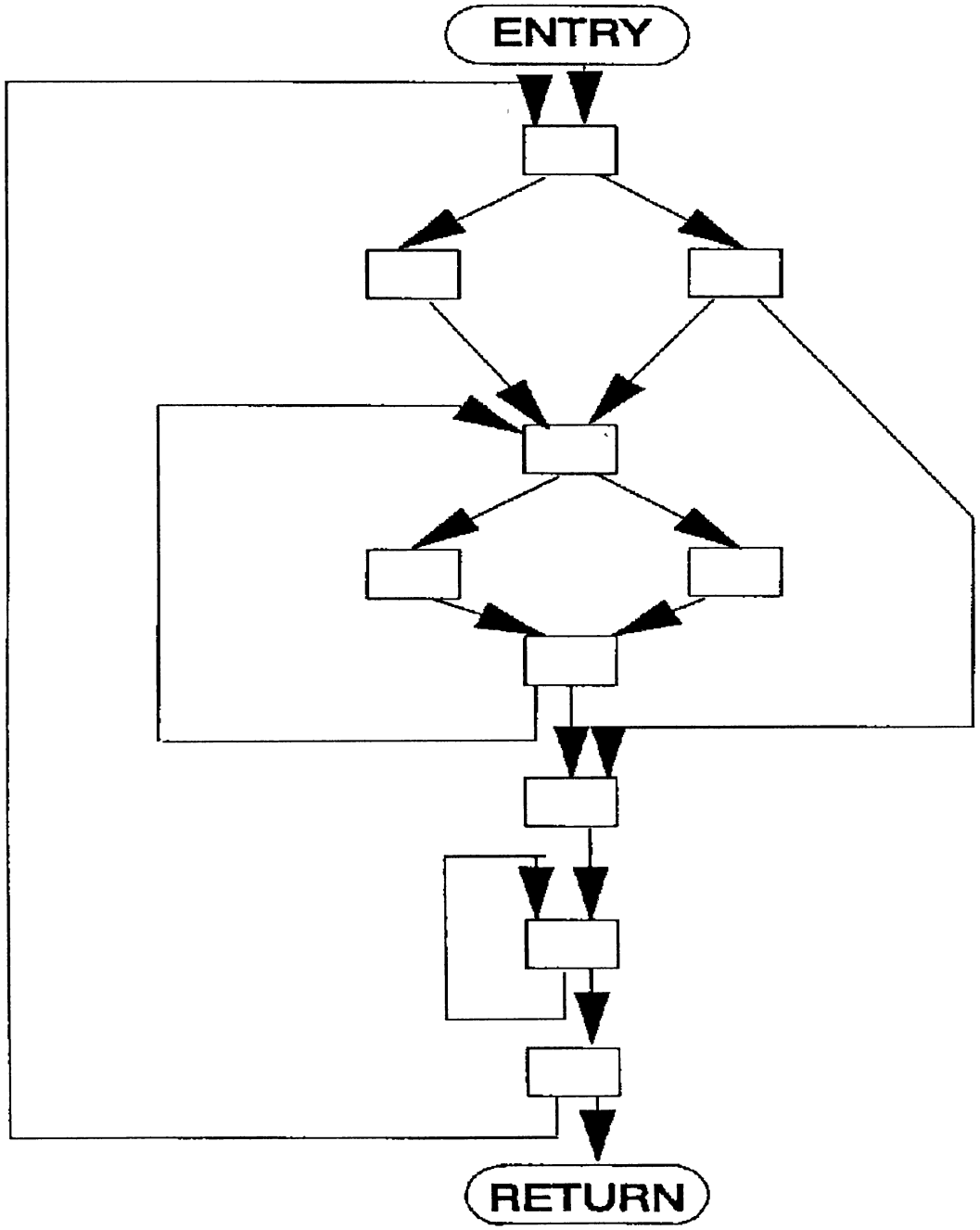

Figure 2. Block structure

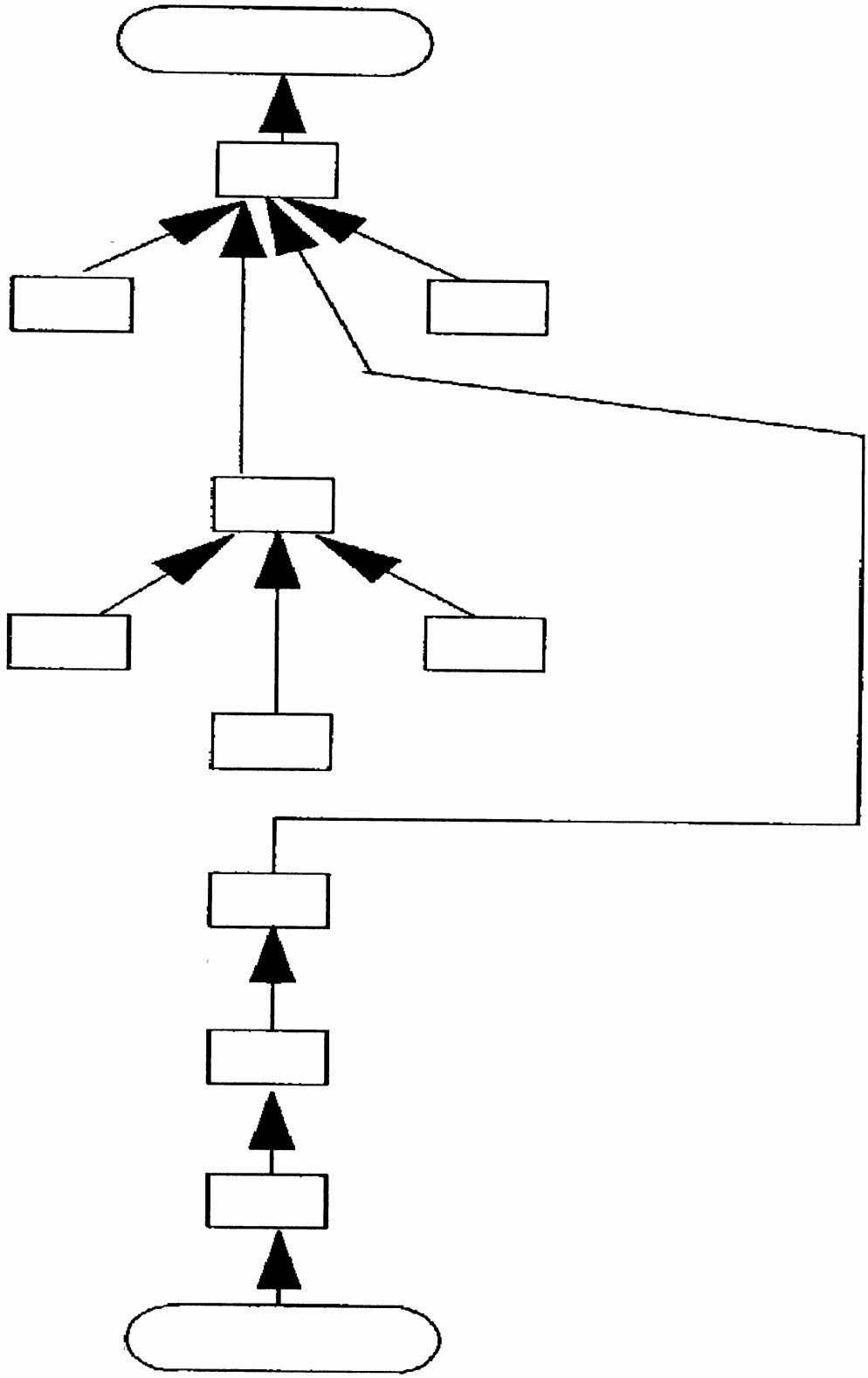

Figure 3. Dominance relations with each block pointing to its immediate predominator

The dominators problem is relatively new and not much extensive study has been done in this area. Lengauer and Tarjan have developed a fast algorithm for finding dominators in a flowgraph using one of the useful tools in graph theory, the "backtracking technique", namely the depth-first search technique [TARJ72], a technique which not only gives the vertices reachable (See definition in Section 2.1) from the start vertex of the search, but also enough information about the connectivity (See definition in Section 2.1) structure of the graph to efficiently determine the dominators [LENG79].

Concurrent (parallel) programming has become important in recent years because of its attractive feature of speeding up program execution [GEHA88]. Aggarwal, Anderson and Kao [AGGA90] have provided the parallel depth-first search algorithm for general directed graphs.

This thesis involves the comparative analysis of the fast algorithm by Lengauer and Tarjan and the parallel algorithm in which case the depth-first search in the fast algorithm is replaced by the parallel depth-first search by Aggarwal, Anderson and Kao, both of which rely upon a graph-theoretic matrix-based approach.

CHAPTER II

LITERATURE REVIEW

2.1 Graph Theory Preliminaries

This section introduces the graph theory preliminaries used throughout this thesis. It is essentially a compilation of all the graph-theoretic terminology used in this document.

DIGRAPH (DIRECTED GRAPH): A digraph is an ordered pair (V,E) where V is a finite set of vertices, and E is a relation on V. The elements of E are called the edges of the digraph. For every pair of vertices $u,v \in V$, the set of edges E will contain at most one edge (u,v) from u to v, and at most one edge (v,u) from v to u. If $(u,v) \in E$, we say that u precedes v or is an antecedent of v [SKVA86].

STRONG COMPONENT: The set of vertices in a digraph D can be partitioned into equivalence classes, and by giving each equivalence class all the vertices connected to one another, the connected subgraphs of a graph, called its components, can be constructed [SKVA86].

If u is a point in a digraph D then the set of vertices that belong to the equivalence class of u is called the component (or, alternatively, a strong component) of u, which is symbolized by C(u). Since components are

equivalence classes, the components defined by two points are either the same or have no points in common [ROBI80].

STRONGLY CONNECTED GRAPH: A digraph with one strong component is called strongly connected.

STRONGLY CONNECTED COMPONENTS: Graphs $G_i = (V_i, E_i)$ are strongly connected components of a directed graph $G = (V,E)$, where V is partitioned into equivalence classes V_i, $1 \leq i \leq r$, such that vertices v and w are equivalent iff there is a path from v to w and a path from w to v and E_i, $1 \leq i \leq r$, the set of edges connecting the pairs of vertices in V_i.

SUBGRAPH: A graph $G_1 = (V_1,E_1)$ is a subgraph of G if $V_1 \subseteq V$ and $E_1 \subseteq E$.

FLOWGRAPH: $G = (V, E, r)$ is a directed graph (V, E) with a distinguished start vertex r such that for any vertex $v \in V$ there is a path from r to v.

SPANNING TREE: T if $G = (V, E)$ is a graph and $T = (V',E',r)$ is a tree such that (V',E') is a subgraph of G and $V = V'$.

DOMINATOR: A vertex v is the dominator of another vertex $w \neq v$ in a flowgraph $G = (V,E,r)$, r being the start vertex, if every path from r to w contains v.

IMMEDIATE DOMINATOR: Vertex v is the immediate dominator of w, if v dominates w and every other dominator of w dominates v.

SEMIDOMINATOR: is $\min\{v \mid \text{there is a path } v = v_0,v_1,\ldots,v_k = w \text{ such that } v_i > w \text{ for } 1 \leq i \leq k-1\}$.

REACHABLE: A vertex w is reachable from vertex v if there is a path from v to w.

<u>CONNECTIVITY</u>: There is a path between any two vertices.

<u>ADJACENCY MATRIX</u>: Two nodes v_1, $v_2 \in V$ in the digraph $D = (V,E)$ are adjacent if there exists either of the two edges: (v_1, v_2) or $(v_2, v_1) \in E$. Either a digraph D, its adjacency matrix A(D), is defined by

$$A(D) = [a_{ij}]; \qquad i, j = 1, 2, \ldots, n,$$

$$\text{where } a_{ij} = \begin{cases} 1, & \text{if } (v_i, v_j) \in E \\ 0, & \text{otherwise} \end{cases}$$

2.2 Fast Algorithm for Dominators

There have been several attempts made for finding dominators in directed graphs.

Aho and Ullman [AHO72] came up with the algorithm for finding dominators by deleting each vertex v in turn from G (a directed graph) and testing which vertices are reachable from s (start vertex), thus showing that any reachable vertex is not dominated by v. Their algorithm required O(V(V+E)) time if the problem graph had V vertices and E edges.

Purdom and Moore [PURD72] had the same time bound as the Aho and Ullman's algorithm. The algorithm by Tarjan [TARJ74] used depth-first search and efficient algorithms for computing disjoint set unions and manipulating priority queues to achieve a time bound of O(V log V + E) if V is the number of vertices and E is the number of edges in the graph.

Lengauer and Tarjan [LENG79] developed a fast algorithm using depth-first search for finding dominators in a flowgraph running in O(m log n) time, where m is the number of edges and n is the number of vertices in the problem graph. Given a arbitrary flowgraph as shown in Figure 4 below, the algorithm constructs a dominator tree as shown in Figure 5 (page 12).

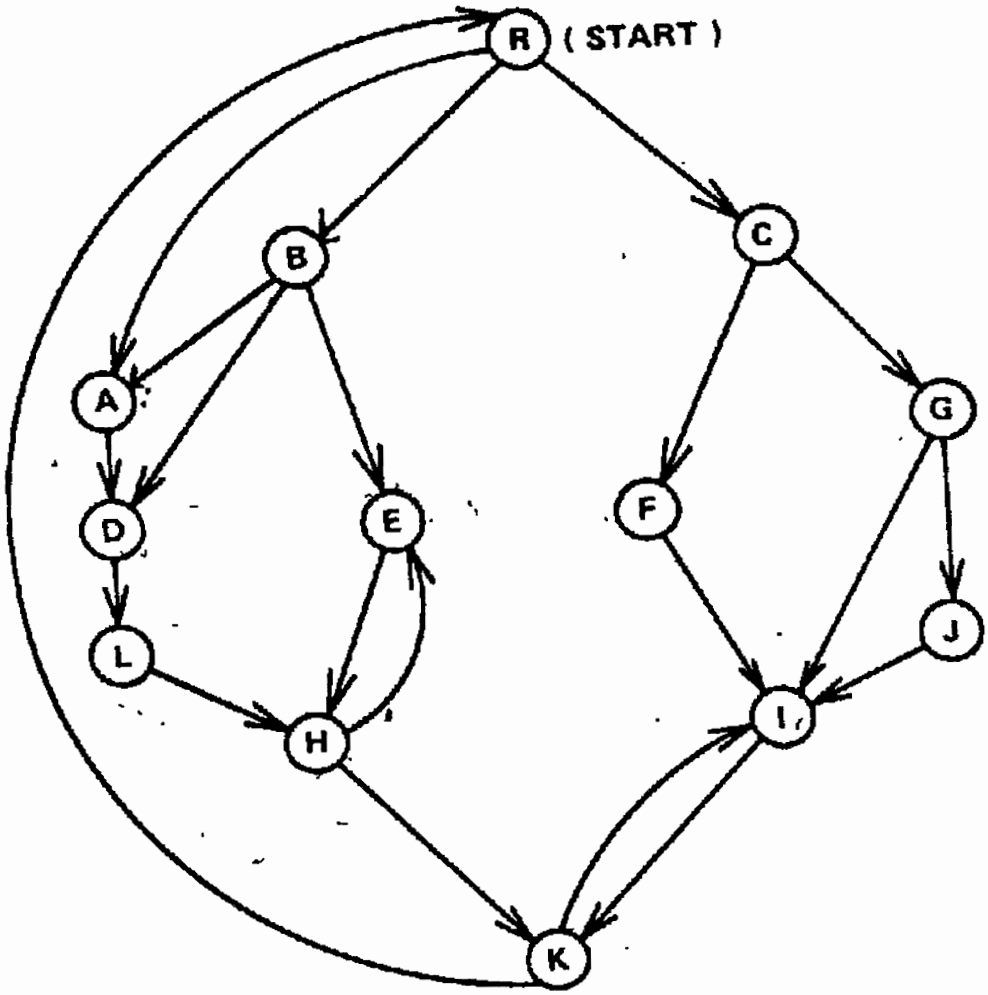

Figure 4. A flowgraph

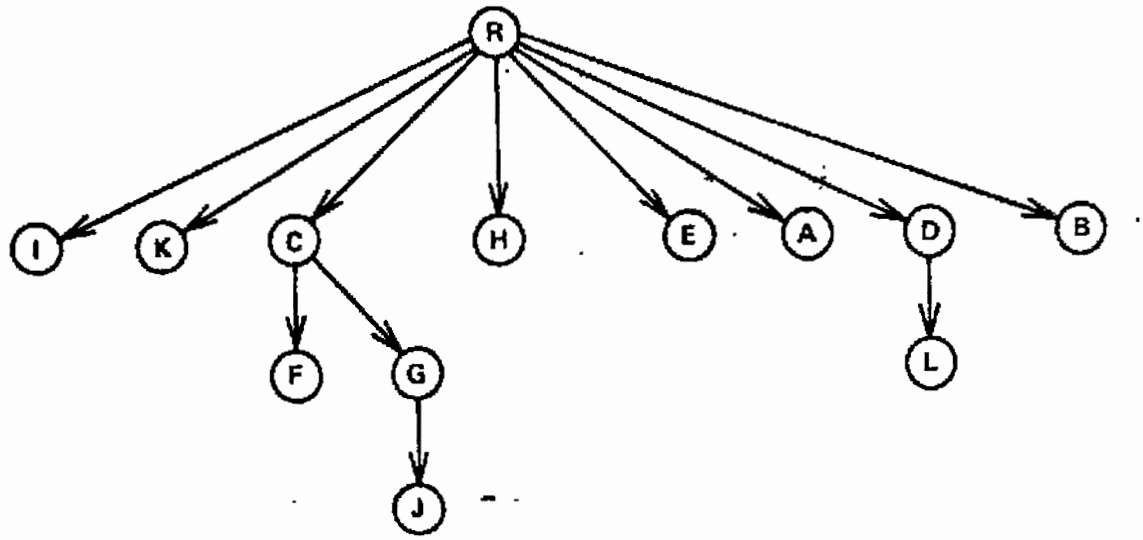

Figure 5. Dominator tree of flowgraph in Figure 4

The fast algorithm carries out a sequential depth-first search of the problem graph, i.e. the construction of a depth-first spanning tree numbering the vertices as they are reached during the search, followed by the computation of the semidominators of all the vertices in decreasing order by number. Then the immediate dominator of each vertex is implicitly defined followed by the explicit definition of the immediate dominator of each vertex carrying out the computation vertex by vertex in increasing order by number.

2.3 Parallel Depth-First Search in General Digraphs

Depth-First Search or the "backtracking technique" is one of the most useful tools in graph theory. In the

setting of parallel computation, various studies were conducted on this technique.

For lexicographic depth-first search, Ghosh and Bhattacharjee provided an algorithm [GHOS84]. For unordered depth-first search, Smith [SMIT86] provided with an algorithm for undirected graphs. He and Yesha [HE88] came up with an algorithm for undirected graphs. Aggarwal and Anderson [AGGA88] provide an algorithm for general undirected graphs.

Aggarwal, Anderson and Kao [AGG90] have presented a general directed depth-first search algorithm which uses a "divide-and conquer" strategy which is similar to that used by Aggarwal and Anderson [AGGA88] for general undirected depth-first search. The concept of "directed cycle separators" defined by Kao [KAO88] is used in this algorithm.

At the highest level, the algorithm finds and removes a portion of a depth-first search tree of a directed graph. The algorithm then recurses on strongly connected components as well as certain weakly connected subgraphs of the resulting graph. The parallel computation model used for the algorithm is the EREW PRAM model, i.e., no two processors are allowed to simultaneously read from or write into the same memory cell.

CHAPTER III

THE FAST ALGORITHM

3.1 The Fast Algorithm Preliminaries

This chapter focuses on the graph-theoretic, matrix -based approach to study the fast algorithm by Lengauer and Tarjan to find dominators in a flowgraph.

The approach used in this thesis makes the following assumptions:

1. For a given program we can draw a directed graph (known as the program control flow graph) with unique entry and exit vertices;
2. Each vertex in the graph corresponds to a block of code in the program with the flow within each block being sequential;
3. Each edge in the directed graph corresponds to the branches taken in the program; and
4. Each vertex can be reached from the entry vertex and each vertex can reach the exit vertex.

3.2 The Fast Algorithm

This algorithm is aimed at construction of the dominator tree of an arbitrary flowgraph which represents a

program, from the adjacency matrix of its control flow graph. The algorithm is outlined below.

1. Develop the directed graph representation (i.e., the control flow graph) of a given program.
2. Develop the adjacency matrix of the control flow graph. The adjacency matrix is the input.
3. Carry out depth-first search of the problem graph. Number the vertices from 1 to n as they reached during search. Initialize the variables used in succeeding steps. This generates a spanning tree rooted at the start vertex with the vertices numbered in preorder.
4. Compute the semidominators of all vertices. Carry out the computation vertex by vertex in decreasing order by number.
5. Implicitly define the immediate dominator of each vertex.
6. Explicitly define the immediate dominator of each vertex, carrying out the computation vertex by vertex in increasing order by number.

The implementation of the algorithm uses the following arrays:

Input

succ(v): The set of vertices w such that (v,w) is an edge of the graph.

Computed

parent(w): The vertex which is the parent of vertex w in the spanning tree generated by the

search.

pred(w): The set of vertices v such that (v,w) is an edge of the graph.

semi(w): A number defined as follows:

(i) Before vertex w is numbered, semi(v) = 0.

(ii) After w is numbered but before its semidominator is computed, semi(w) is the number of w.

(iii) After the semidominator of w is computed, semi(w) is the number of the semidominator of w.

vertex(i): The vertex whose number is i.

bucket(w): A set of vertices whose semidominator is w.

dom(w): A vertex defined as follows:

(i) After step 3, if the semidominator of w is its immediate dominator, then dom(w) is the immediate dominator of w. Otherwise dom(w) is a vertex v whose number is smaller than w and whose immediate dominator is also w's immediate dominator.

(ii) After step 4, dom(w) is the immediate dominator of w.

The following is the complete listing of the Algol-like version of the fast algorithm:

```
procedure DOMINATORS(integer set array succ(1::n);integer
    r,n;integer array dom(1::n));
  begin
    integer array parent, ancestor, vertex(1::n);
    integer array label, semi(0::n);
    integer set array pred, bucket(1::n);
    integer u, v, x;
  procedure DFS(integer v);
    begin
      semi(v) := n := n + 1;
      vertex(n) := label(v) := v;
      ancestor(v) := 0;
      for each w ∈ succ(v) do
        if semi(w) = 0 then parent(w) := v; DFS(w) fi;
        add v to pred(w) od
  end DFS;
  procedure COMPRESS(integer v);
    if ancestor(ancestor(v)) = 0 then
       COMPRESS(ancestor(v));
       if semi(label(ancestor(v))) < semi(label(v)) then
          label(v) := label(ancestor(v)) fi;
       ancestor(v) := ancestor(ancestor(v)) fi;
  integer procedure EVAL(integer v);
    if ancestor(v) = 0 then EVAL := v
       else COMPRESS(v); EVAL := label(v) fi;
```

```
  procedure LINK(integer v,w);
    ancestor(w) := v;
step1:  for v := 1 until n do
          pred(v) := bucket(v) := 0; semi(v) := 0 od;
        n := 0;
        DFS(r);
        for i := n by -1 until 2 do
          w := vertex(i);
step2:  for each v ∈ pred(w) do
        u := EVAL(v);
        if semi(u) < semi(w) then semi(w) := semi(u) fi od
        add w to bucket(vertex(semi(w)));
        LINK(parent(w),w);
step3:  for each v ∈ bucket(parent(w)) do
        delete v from bucket(parent(w));
        u := EVAL(v);
        dom(v) := if semi(u) < semi(v) then u
                     else parent(w) fi od od;
step4:  i := 2 until n do
           w := vertex(i);
           if dom(w) = vertex(semi(w))
              then dom(w) := dom(dom(w)) fi od;
           dom(r) := 0;
end DOMINATORS;
```

The algorithm uses path compression (the technique which changes the structure of the tree during a find

operation by moving vertices closer to the root) to improve its performance greatly [TARJ79].

The application of the fast algorithm to an example graph from McCabe's work [MCCA76] appears in Appendix A and the performance of the algorithm is seen in an graphical representation in the Figures 7 and 8 in Appendix B.

CHAPTER IV

THE PARALLEL ALGORITHM

4.1 The Parallel Algorithm Preliminaries

This section deals with the preliminaries required for the discussion of the parallel algorithm. The algorithm follows the same assumptions made for the fast algorithm in Section 3.1.

The parallel algorithm makes use of the Sequent's (Sequent Symmetry S81 with 24 80386 processors running at 20Mhz each with the Dynix/ptx 1.3 as the operating system) support for parallelism and its characteristics [GUID85]. The algorithm makes use of some elements of parallel programming such as creation and termination of multiple processes, creation of shared and private data, scheduling, the division of computing tasks among parallel processes, task synchronization and mutual exclusion. The algorithm involves multitasking which is a programming technique that allows a single application to consist of multiple processes executing concurrently. The data partitioning multitasking programming method is used and involves creating multiple identical processes and assigning a portion of the data to each process. Dynamic scheduling for scheduling the tasks among processes is used by the algorithm because of its

feature that each process checks for tasks at run time by examining a task queue or a "do-me-next" array-index and thus provides dynamic load balancing: all processes keep working as long as there is work to be done and since the work is evenly distributed among the processes, the work can be completed sooner. Thus dynamic scheduling has an advantage over static scheduling which provides static load balancing: since the division of tasks is statically determined, several processes may stand idle while one processor completes its share of job. The dynamic scheduling algorithm is:

1. Wait until some tasks appear.
2. Remove the first task from the list and do it.
3. If there are any more tasks, go to step 2. Otherwise go to step 1.

To protect code sections that contain dependent variables to yield correct results, thus providing mutual exclusion, locks (a semaphore which ensures that only one process at a time can access a shared data structure or execute a critical region of code) are used. Synchronization of processes i.e, a process waits at a barrier (A synchronization point) after finishing its job for the other processes to come and join, is done by the algorithm. Since a fork operation involves a lot of CPU overhead (Time and computation not spent in calculating the result of a program) time, the child processes were parked

and then released whenever needed by the algorithm and only killed when the parallel depth first search was done.

Reasonable typical model of parallel processing is considered [ECKS77]. There are k identical processors, each with a CPU capable of performing typical operations such as arithmetic, comparisons, and boolean operations and each with a label between 1 and k which identifies. A single arbitrary large memory is available to all the processes for manipulation of data. Different processors are not allowed to read from the same memory location simultaneously, may write into different memory locations but must not attempt to write into the same memory simultaneously. A global control unit must be capable of synchronizing the various processes. The code is delineated syntactically as:

```
instruct processor(i); 1 <= i <= j;
    sequence of instructions;
end_instruction;
```

and has j <= k processors executing simultaneously. Execution cannot resume after the end_instruction until all the processors have completed execution of the delineated sequence of instructions.

4.2 The Parallel Algorithm

This algorithm is aimed at computing the dominators of a structured program from the adjacency matrix of its control flow graph.

The algorithm implements the depth first search in a parallel form [AGGA90]. The vertices of a graph G are represented by the integers 1 to n. An adjacency list matrix representation of G is constructed from the adjacency matrix, and is a n x (n-1) matrix ALM such that 1 <= i <= n, row i consists of the list of vertices that are heads of edges with tail i. Associated with the adjacency list matrix is an n-vector of end markers EM where EM(i) contains the index j of the last vertex in the ith row of the adjacency list matrix. This setup helps different processors to simultaneously examine successive vertices to see whether they are "unvisited" or not. An "unvisited" adjacency list U(v) is created which lists all the vertices adjacent to v and are still labeled "unvisited". As soon as a vertex w is "visited", it is removed from the adjacency lists U(v) for all v adjacent to w. All the "visited" vertices are added to the ARC_LIST list and the "unvisited" vertices are added to the FROND_LIST list. The deletion of a newly "visited" vertex v, from the lists U(w) for all w adjacent to v are performed in parallel.

The Algol-like version code of the algorithm (PMDFS) is outlined below:

```
begin
for each v ∈ V do initialize ARC_LIST(v)
and FROND_LIST(v) as null lists;
mark every vertex "unvisited";
v = start vertex;
```

```
FATHER(v) = 0;
NUMB_VERTICES_VISITED = 0;
pmdfs(v);
procedure pmdfs(v);
  begin
  comment v is the vertex being searched from;
  mark v "visited";
  NUMB_VERTICES_VISITED = NUMB_VERTICES_VISITED + 1;
  NUMBER(v) = NUMB_VERTICES_VISITED;
  instruct_processor(i); 1 <= i <= k;
  for j = 1 to floor(EM(v)/k) do
    if (k * (j - 1) + i) <= EM(v)
    then begin
         w(i) = ALM(v,k * (j - 1) + i);
         delete v from U(w(i));
         if w(i) is "unvisited"
         then add v to FROND_LIST(w(i));
         end;
  end_instruction;
  for w ∈ U(v) do
   begin
   FATHER(w) = v;
   add w to ARC_LIST(v);
   remove v from the end of FROND_LIST(w);
   pmdfs(w);
   end;
  end;
```

<u>end</u>

Thus the above algorithm replaces the sequential depth first search strategy in the fast algorithm by Lengauer and Tarjan. The start vertex is identified as the directed cycle separator since it is a cycle of length zero and the removal of this vertex separates the graph to start with.

The implementation of the algorithm uses the same arrays as the fast algorithm given in Section 3.2.

The application of the parallel algorithm to an example graph from Mccabe's work [MCCA76] appears in Appendix A and the performance of the algorithm running on different processors as well as the comparison of the algorithm with the fast algorithm is seen in the Figures 7, 8, 9 and 10 in Appendix B.

CHAPTER V

THE RANDOM GENERATION ALGORITHM

5.1 The Random Generation Preliminaries

This chapter focuses on the graph-theoretic, matrix-based approach to generate random flowgraphs and use the generated flowgraphs to run the fast algorithm by Lengauer and Tarjan and the parallel algorithm developed using the parallel depth-first search algorithm by Aggarwal, Anderson and Kao to get comparative results. These comparative results are then graphically represented as shown in Appendix B.

5.2 The Connectivity Algorithm

The input to the fast and parallel algorithms is a connected graph. The Connectedness Algorithm [AHO74] needs for its input a directed graph G = (V,E) and labeling function l which is defined as

$$l(v,w) = \begin{cases} 1, & \text{if}(v,w) \text{ is an edge} \\ 0, & \text{if not} \end{cases}$$

and is the adjacency matrix for the given graph. For the connectedness of the given graph, the reflexive-transitive

closure of the graph has to be calculated. The output is the calculation of $c(v_i,v_j)$ which is the sum over all the paths from v_i to v_j of the label of the path. The algorithm will return $c(v_i,v_j)$ to be equal to 1 for all i and j between 1 and n if the graph is connected.

The algorithm is as follows :

```
begin
    for i = 1 until n do C^0_ii = 1 + l(v_i,v_j)
    for 1 <= i, j <= n and i = j do C^0_ij = l(v_i,v_j)
    for k = 1 until n do
      for 1 <= i, j <= n do
        C^k_ij = C^(k-1)_ij + C^(k-1)_ik . C^(k-1)_kj
    for 1 <= i, j <= n do c(v_i,v_j) = C^n_ij
```

5.3 The Random Generation Method

The approach used in the algorithm is the generation of the random adjacency matrices which has as its contents 0's and 1's. These 0's and 1's are randomly obtained by running the random generator [PARK88]. Then the adjacency matrices are tested for the property of connectedness using the connectedness algorithm described in Section 4.1. Only connected graphs are generated. Then using these adjacency matrices the fast and the parallel algorithms are run with the variable parameters - the adjacency matrix, the number of vertices, the start vertex and the number of processors asked for by the user (in the case of the parallel algorithm).

CHAPTER VI

RESULTS

Experiments were performed in order to compare the performance of the fast algorithm with that of the parallel algorithm. The fast algorithm Algol version was translated into C. The parallel algorithm was developed by using the parallel depth-first search approach by Aggarwal, Anderson and Kao in the fast algorithm and translated in C. Both the programs were separately tested out initially on the flowgraphs given in the Mccabe's paper [MCCA76].

Rigorous testing was done by the development of an algorithm which generated 10 random flowgraphs (connected) per vertex for vertices ranging from 5 to 150 in steps of 5, in form of adjacency matrices. These matrices were then used to run the fast and the parallel programs (for the parallel program the number of processors varied from 1 to 16 in powers of 12) and the processing times were formed in a tabular form. Tables III and IV and Figures 7, 8, 9, 10 in Appendix B illustrate the results.

TABLE IV was formed from TABLE III recording the average processing times. TABLE IV in Appendix B was then plotted into four graphs. Figure 7 shows the graphs of fast algorithm and the parallel algorithm running on one

processor in the vertex range of 5 to 100. Figure 8 shows the graphs of fast algorithm and the parallel algorithm running on one processor in the vertex range of 50 to 150. Figure 9 shows the performance of the parallel algorithm running on number of processors = 1, 2, 4, 8, 16 with the vertex range between 5 to 100. Figure 10 shows the performance of the parallel algorithm running on number of processors = 1, 2, 4, 8, 16 with the vertex range between 50 to 150.

Figures 7 and 8 show that the fast algorithm has a better performance than the parallel algorithm for comparatively smaller graphs. As the number of vertices increase and the graphs become larger, the parallel algorithm beats the fast algorithm.

Figures 9 and 10 show that the performance of the parallel algorithm improves with the number of processors increasing.

Therefore for number of processors = 16, the performance of the parallel algorithm is the best.

CHAPTER VII

SUMMARY AND CONCLUSIONS

The main theme of this thesis was the comparative analysis of the dominators fast algorithm by Lengauer and Tarjan and the parallel algorithm developed by using the algorithm by Aggarwal, Anderson and Kao in the fast algorithm, using a graph-theoretic matrix-based approach. The approach used in this thesis relies upon the following assumptions:

1. For a given program we can draw a directed graph (known as the program control flow graph) with unique entry and exit vertices;
2. Each vertex in the graph corresponds to a block of code in the program with the flow within each block being sequential;
3. Each edge in the directed graph corresponds to the branches taken in the program; and
4. Each vertex can be reached from the entry vertex and each vertex can reach the exit vertex.

Essentially, these assumptions convey the notion that the algorithms developed as part of this thesis apply only to structured programs.

The parallel algorithm approach proved to be the improved version of the fast algorithm. As the number of processors were increased, the parallel program performed even better. Looking at the trends which are seen in the graphs in Appendix B, the fast algorithm has a better performance than the parallel algorithm for smaller graphs but as the number of vertices increase, the performance of the parallel algorithm is better.

Therefore it can concluded that the parallel depth-first search strategy by Aggarwal, Anderson and Kao improved the performance of the fast algorithm by Lengauer and Tarjan.

BIBLIOGRAPHY

[AGGA88]
A. Aggarwal and R. J. Anderson, "A Random NC Algorithm For Depth First Search", Combinatorica, Vol. 8, pp. 1-12, 1988.

[AGGA90]
Alok Aggarwal, Richard J. Anderson and Ming-Yang Kao, "Parallel Depth-First Search in General Directed Graphs", SIAM Journal on Computing, Vol. 19, No. 2, pp. 397-409, April 1990.

[AHO72]
A. V. Aho and J. D. Ullman, The Theory of Parsing, Translation, and Compiling, Vol II: Compiling, Prentice-Hall, Englewood Cliffs, N.J., 1972.

[AHO74]
A. V. Aho, J. E. Hopcroft and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading MA, 1974.

[ANDE87]
R. Anderson, "A Parallel Algorithm For The Maximal Path Problem", Combinatorica, Vol. 7, No. 4, pp. 315-326, 1987.

[COPP87]
Don Coppersmith and Shmuel Winograd, "Matrix Multiplication Via Arithmetic Progressions", Proc. 19th Anuual ACM Symposium on Theory of Computing, Association For Computing Machinery, pp. 1-6, 1987.

[ECKS77]
Denise M. Eckstein and Donald A. Alton, "Parallel Graph Processing Using Depth-First Search", Proceedings of a Conference on Computer Science, pp. 21-29, 1977.

[EVEN75]
Shimon Even and R. E. Tarjan, "Network Flow And Testing Graph Connectivity", SIAM Journal on Computing, Vol. 4, No. 4, pp. 507-518, December 1975.

[GAZI87]
Hiller Gazit and Gary L. Miller, "A Parallel Algorithm For Finding A Separator In Planar Graphs", IEEE Symposium on Foundations of Computer Science, pp. 238-248, 1987.

[GEHA88]
Narain Gehani and Andrew D. McGettrick, Concurrent Programming, AT&T Bell Laboratories, 1988.

[GHOS84]
Ratan K. Ghosh and G. P. Bhattacharjee, "A Parallel Search Algorithm For Directed Acyclic Graphs", BIT, Vol. 24, pp. 134-150, 1984.

[GUID85]
Guide To Parallel Programming On Sequent Computer Systems, Prentice Hall, Englewood Cliffs, New Jersey, Third Edition, 1985.

[HE88]
Xin He and Yaacov Yesha, "A Nearly Optimal Parallel Algorithm For Constructing Depth First Spanning Trees in Planar Graphs", SIAM Journal on Computing, Vol. 17, pp. 486-491, 1988.

[HECH75]
Matthew S. Hecht and Jeffrey D. Ullman, "A Simple Algorithm For Global Data Flow Analysis Problems", SIAM Journal on Computing, Vol. 4, No. 4, pp. 519-532, December 1975.

[HO84]
Hon S. Ho, "Graph Theoretic Modeling and Analysis in Software Engineering", Handbook of Software Engineering, Van Nostrand Reinhold Company, pp. 26-37, 1984.

[KAO88]
M. Y. Kao, "All graphs have cycle separators and planar directed depth-first search is in DNC", Proc. 3rd Aegean Workshop on Computing, Corfu, Greece, J. H. Reif, ed.; Lecture Notes in Computer Science 319, Springer-Verlag, Berlin, New York, pp. 53-63, 1988.

[KARP86]
R. M. Karp, E. Upfal and A. Wigderson, "Constructing A Perfect Matching Is In Random NC", Combinatorica, Vol. 6, No. 1, pp. 35-48, 1986.

[LENG79]
Thomas Lengauer and Robert Endre Tarjan, "A Fast Algorithm for Finding Dominators in a Flowgraph", ACM Transactions on Programming Languages and Systems, Vol. 1, No. 1, pp. 121-141, July 1979.

[LIPT77]
Richard J. Lipton and Robert E. Tarjan, "A Separator Theorem For Planar Graph", Proceedings of a Conference on Computer Science, pp. 1-10, 1977.

[LOVA85]
L. Lovasz, "Computing Ears And Branchings", IEEE Symposium on Foundations of Computer Science, pp. 464-467, 1985.

[LOWR69]
Edward S. Lowry and C. W. Medlock, "Object Code Optimization", Communications of the ACM, Vol. 12, No. 1, pp. 13-22, January 1969.

[MCCA76]
Thomas J. McCabe, "A Complexity Measure", IEEE Transactions on Software Engineering, Vol SE-2, No. 4, pp. 308-320, December 1976.

[MULM87]
Ketan Mulmuley, Umesh V. Vazirani and Vijay V. Vazirani, "Matching Is As Easy As Matrix Inversion", Combinatorica, Vol. 7, No. 1, pp. 105-113, 1987.

[PARK88]
Stephen K. Park and Keith W. Miller, "Random Number Generators: Good Ones Are Hard To Find", Communications of the ACM, Vol. 31, No. 10, pp. 1192-1201, October 1988.

[PURD72]
Paul W. Purdom and Edward F. Moore, "Algorithm 430: Immediate Predominators in a Directed Graph", Communications of the ACM, Vol. 15, No. 8, pp. 777-778, August 1972.

[REGH78]
E. Reghbati and D. Corneil, "Parallel Computations In Graph Theory", SIAM Journal on Computing, Vol. 7, No. 2, pp. 230-237, May 1978.

[REIF77]
John H. Reif, "Code Motion", Proceedings of a Conference on Computer Science, pp. 11-20, 1977.

[ROBI80]
D. F. Robinson and L. R. Foulds, Digraphs: Theory and Techniques, Gordon and Breach Science Publishers, New York, 1980.

[SKVA86]
Romualdas Skvarcius and William B. Robinson, Discrete Mathematics with Computer Science Applications, The Benjamin/Cummings Publishing Company, Inc., 1986.

[SMIT86]
Justin R. Smith, "Parallel Algorithms For Depth-First Searches I. Planar Graphs", SIAM Journal on Computing, Vol. 15, pp. 814-830, 1986.

[TARJ72]
Robert Tarjan, "Depth-First Search And Linear Graph Algorithms", SIAM Journal on Computing, Vol. 1, No. 2, pp. 146-160, June 1972.

[TARJ74]
Robert Tarjan, "Finding Dominators In Directed Graphs", SIAM Journal on Computing, Vol. 3, No. 1, pp. 62-89, March 1974.

[TARJ79]
Robert Endre Tarjan, "Applications Of Path Compression On Balanced Trees", Journal of the Association for Computing Machinery, Vol. 26, No. 4, pp. 690-715, October 1979.

[TARJ84]
R. E. Tarjan and U. Vishkin, "Finding Biconnected Components And Computing Tree Functions In Logarithmic Parallel Time", 25th Annual Symposium on Foundations of Computer Science, pp. 12-20, 1984.

APPENDIXES

APPENDIX A

EXAMPLES FOR THE FAST AND PARALLEL ALGORITHMS

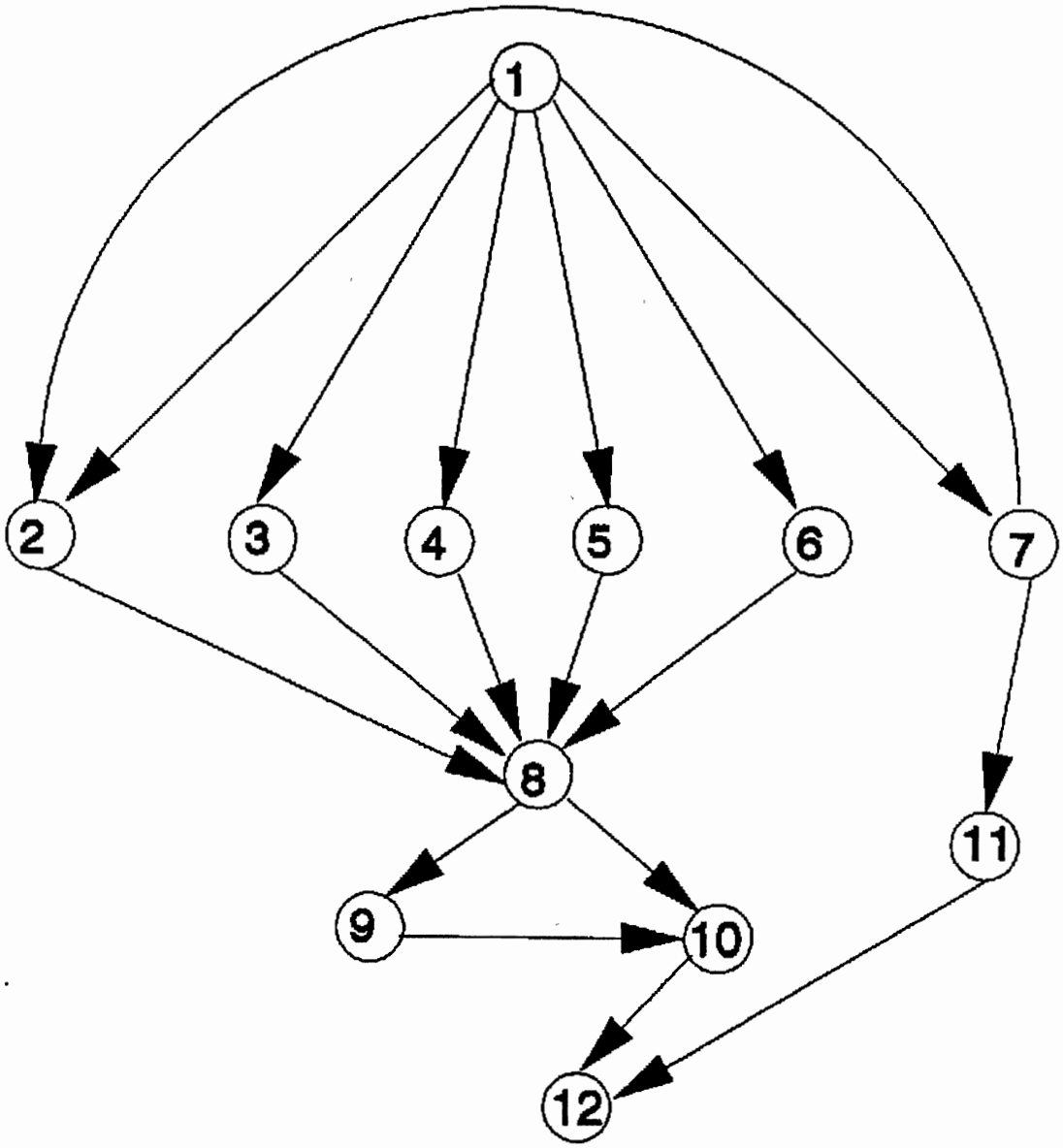

Figure 6. Control Flow Graph [MCCA76] for # of vertices = 12

TABLE I

ADJACENCY MATRIX FOR FIGURE 6

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	1	1	1	1	1	1	0	0	0	0	0
2	0	0	0	0	0	0	0	1	0	0	0	0
3	0	0	0	0	0	0	0	1	0	0	0	0
4	0	0	0	0	0	0	0	1	0	0	0	0
5	0	0	0	0	0	0	0	1	0	0	0	0
6	0	0	0	0	0	0	0	1	0	0	0	0
7	0	1	0	0	0	0	0	0	0	0	1	0
8	0	0	0	0	0	0	0	0	1	1	0	0
9	0	0	0	0	0	0	0	0	0	1	0	0
10	0	0	0	0	0	0	0	0	0	0	0	1
11	0	0	0	0	0	0	0	0	0	0	0	1
12	0	0	0	0	0	0	0	0	0	0	0	0

row labels represent vertex numbers
column labels represent vertex numbers

TABLE II

DOMINATOR TABLE FOR FIGURE 6

Vertex	Dominator
1	0
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	8
10	8
11	7
12	1

APPENDIX B

RESULTS

TABLE III

ANALYSIS OF THE FAST AND PARALLEL
ALGORITHMS PROCESSING TIMES
IN SECONDS

Verts	Seq Algo	Par Algo # of processors 1	2	4	8	16
5	0.02	0.41	0.100	0.030	0.010	0.010
	0.03	0.41	0.100	0.030	0.010	0.010
	0.03	0.41	0.100	0.030	0.020	0.010
	0.03	0.41	0.100	0.030	0.020	0.010
	0.03	0.41	0.100	0.030	0.020	0.010
	0.03	0.41	0.100	0.030	0.020	0.010
	0.03	0.41	0.100	0.030	0.020	0.010
	0.04	0.41	0.100	0.030	0.020	0.010
	0.04	0.41	0.100	0.030	0.020	0.010
	0.04	0.41	0.100	0.040	0.020	0.010
10	0.05	0.42	0.100	0.040	0.020	0.010
	0.05	0.42	0.110	0.040	0.020	0.010
	0.06	0.42	0.110	0.040	0.020	0.010
	0.06	0.42	0.110	0.040	0.020	0.010
	0.06	0.42	0.110	0.040	0.020	0.010
	0.06	0.42	0.110	0.040	0.020	0.010
	0.07	0.42	0.110	0.040	0.020	0.010
	0.07	0.43	0.110	0.040	0.020	0.010
	0.07	0.43	0.110	0.040	0.020	0.010
	0.07	0.43	0.110	0.040	0.020	0.020
15	0.11	0.43	0.110	0.040	0.030	0.020
	0.12	0.43	0.110	0.040	0.030	0.020
	0.12	0.43	0.110	0.040	0.030	0.020
	0.12	0.43	0.110	0.040	0.030	0.020
	0.12	0.43	0.110	0.040	0.030	0.020
	0.12	0.43	0.110	0.040	0.030	0.020
	0.12	0.44	0.110	0.040	0.030	0.020
	0.12	0.44	0.110	0.040	0.030	0.020
	0.12	0.44	0.120	0.040	0.030	0.020
	0.12	0.44	0.120	0.040	0.030	0.020
20	0.17	0.44	0.120	0.050	0.030	0.030
	0.18	0.45	0.120	0.050	0.030	0.030
	0.18	0.45	0.120	0.050	0.040	0.030
	0.18	0.45	0.120	0.050	0.040	0.030
	0.18	0.45	0.120	0.050	0.040	0.030
	0.18	0.45	0.120	0.050	0.040	0.030
	0.18	0.45	0.120	0.050	0.040	0.030
	0.19	0.46	0.130	0.050	0.040	0.030
	0.19	0.47	0.130	0.060	0.040	0.030
	0.19	0.47	0.140	0.060	0.040	0.030
25	0.27	0.48	0.140	0.070	0.050	0.050

TABLE III (Continued)

Verts	Seq Algo	Par Algo # of processors 1	2	4	8	16
	0.27	0.48	0.140	0.070	0.050	0.050
	0.27	0.48	0.150	0.070	0.050	0.050
	0.27	0.48	0.150	0.070	0.050	0.050
	0.27	0.49	0.150	0.070	0.050	0.050
	0.27	0.49	0.150	0.070	0.050	0.050
	0.28	0.49	0.150	0.070	0.050	0.050
	0.28	0.50	0.150	0.070	0.050	0.050
	0.28	0.50	0.150	0.070	0.050	0.050
	0.28	0.50	0.160	0.070	0.050	0.050
30	0.37	0.50	0.160	0.080	0.070	0.070
	0.37	0.50	0.160	0.080	0.070	0.070
	0.37	0.50	0.160	0.080	0.070	0.070
	0.38	0.51	0.160	0.090	0.070	0.070
	0.38	0.51	0.160	0.090	0.070	0.070
	0.38	0.51	0.170	0.090	0.070	0.070
	0.38	0.51	0.170	0.090	0.070	0.070
	0.38	0.52	0.170	0.090	0.070	0.070
	0.38	0.52	0.170	0.090	0.070	0.070
	0.38	0.53	0.180	0.090	0.070	0.070
35	0.49	0.53	0.180	0.110	0.090	0.090
	0.49	0.53	0.180	0.110	0.090	0.090
	0.49	0.53	0.190	0.110	0.090	0.090
	0.49	0.53	0.190	0.110	0.090	0.090
	0.50	0.53	0.190	0.110	0.100	0.090
	0.50	0.54	0.190	0.110	0.100	0.090
	0.50	0.54	0.190	0.110	0.100	0.090
	0.50	0.54	0.190	0.110	0.100	0.090
	0.50	0.56	0.190	0.110	0.100	0.090
	0.50	0.57	0.210	0.110	0.100	0.090
40	0.62	0.58	0.210	0.130	0.130	0.120
	0.63	0.58	0.210	0.130	0.130	0.120
	0.63	0.58	0.220	0.130	0.130	0.120
	0.63	0.58	0.220	0.130	0.130	0.120
	0.63	0.59	0.220	0.130	0.130	0.120
	0.63	0.59	0.220	0.130	0.130	0.120
	0.63	0.59	0.220	0.130	0.130	0.120
	0.63	0.60	0.220	0.130	0.130	0.120
	0.64	0.62	0.220	0.130	0.130	0.120
	0.64	0.62	0.240	0.130	0.130	0.120
45	0.77	0.62	0.250	0.170	0.160	0.150
	0.78	0.62	0.250	0.170	0.160	0.150
	0.78	0.62	0.250	0.170	0.160	0.150
	0.78	0.62	0.250	0.170	0.160	0.150
	0.78	0.63	0.250	0.170	0.160	0.150
	0.78	0.63	0.250	0.170	0.160	0.150

TABLE III (Continued)

Verts	Seq Algo	Par Algo # of processors 1	2	4	8	16
	0.78	0.63	0.250	0.170	0.170	0.150
	0.79	0.63	0.260	0.170	0.170	0.150
	0.79	0.63	0.260	0.170	0.170	0.150
	0.79	0.64	0.280	0.170	0.170	0.150
50	0.95	0.66	0.290	0.210	0.190	0.180
	0.95	0.67	0.290	0.210	0.190	0.180
	0.95	0.67	0.290	0.210	0.190	0.180
	0.95	0.67	0.290	0.210	0.190	0.180
	0.95	0.68	0.290	0.210	0.190	0.180
	0.95	0.68	0.290	0.210	0.190	0.180
	0.95	0.68	0.290	0.210	0.190	0.180
	0.95	0.68	0.290	0.210	0.190	0.180
	0.96	0.69	0.290	0.210	0.190	0.190
	0.97	0.69	0.330	0.210	0.200	0.190
55	1.13	0.74	0.330	0.260	0.230	0.230
	1.13	0.74	0.330	0.270	0.230	0.230
	1.14	0.74	0.330	0.270	0.240	0.230
	1.14	0.74	0.330	0.270	0.240	0.230
	1.15	0.74	0.330	0.270	0.240	0.230
	1.15	0.74	0.330	0.270	0.240	0.230
	1.15	0.74	0.330	0.270	0.240	0.230
	1.15	0.77	0.340	0.270	0.240	0.230
	1.15	0.77	0.370	0.270	0.240	0.230
	1.16	0.79	0.370	0.270	0.260	0.250
60	1.34	0.79	0.380	0.320	0.280	0.270
	1.34	0.80	0.380	0.320	0.280	0.270
	1.34	0.80	0.380	0.320	0.280	0.270
	1.34	0.80	0.380	0.320	0.280	0.270
	1.34	0.81	0.380	0.320	0.280	0.270
	1.35	0.82	0.380	0.320	0.280	0.270
	1.35	0.82	0.380	0.320	0.280	0.270
	1.35	0.82	0.390	0.320	0.280	0.270
	1.37	0.83	0.420	0.320	0.280	0.270
	1.38	0.84	0.420	0.330	0.320	0.270
65	1.57	0.86	0.430	0.390	0.330	0.320
	1.57	0.86	0.430	0.390	0.330	0.320
	1.57	0.86	0.430	0.390	0.330	0.330
	1.57	0.87	0.430	0.390	0.330	0.330
	1.58	0.87	0.430	0.390	0.330	0.330
	1.58	0.88	0.430	0.390	0.330	0.330
	1.58	0.88	0.430	0.390	0.330	0.330
	1.58	0.88	0.440	0.390	0.330	0.330
	1.59	0.89	0.490	0.390	0.330	0.330
	1.60	0.95	0.490	0.390	0.330	0.330
70	1.79	0.95	0.490	0.480	0.390	0.380

TABLE III (Continued)

Verts	Seq Algo	Par Algo # of processors 1	2	4	8	16
	1.81	0.95	0.490	0.480	0.390	0.380
	1.81	0.96	0.490	0.480	0.390	0.390
	1.83	0.96	0.490	0.480	0.390	0.390
	1.83	0.96	0.500	0.480	0.390	0.390
	1.83	0.96	0.500	0.480	0.390	0.390
	1.83	0.96	0.500	0.480	0.390	0.390
	1.83	0.97	0.500	0.480	0.390	0.390
	1.83	0.98	0.520	0.480	0.400	0.390
	1.85	1.02	0.590	0.480	0.400	0.390
75	2.08	1.06	0.590	0.570	0.470	0.460
	2.09	1.06	0.590	0.570	0.480	0.460
	2.09	1.06	0.590	0.570	0.480	0.460
	2.09	1.07	0.590	0.570	0.480	0.460
	2.09	1.07	0.590	0.570	0.480	0.460
	2.09	1.07	0.590	0.570	0.480	0.460
	2.09	1.07	0.590	0.580	0.480	0.460
	2.10	1.07	0.590	0.580	0.480	0.460
	2.10	1.09	0.590	0.580	0.480	0.460
	2.11	1.13	0.660	0.580	0.480	0.460
80	2.36	1.13	0.660	0.620	0.530	0.510
	2.36	1.14	0.660	0.620	0.530	0.510
	2.37	1.15	0.660	0.630	0.530	0.510
	2.37	1.15	0.660	0.630	0.530	0.510
	2.37	1.15	0.660	0.630	0.530	0.510
	2.37	1.16	0.660	0.630	0.530	0.510
	2.38	1.16	0.670	0.630	0.530	0.510
	2.38	1.16	0.670	0.630	0.530	0.510
	2.38	1.18	0.670	0.640	0.540	0.520
	2.38	1.19	0.790	0.660	0.540	0.520
85	2.65	1.24	0.790	0.710	0.620	0.590
	2.65	1.25	0.790	0.710	0.620	0.590
	2.65	1.26	0.790	0.710	0.620	0.590
	2.65	1.26	0.790	0.710	0.620	0.590
	2.65	1.26	0.790	0.710	0.620	0.590
	2.66	1.27	0.790	0.710	0.630	0.590
	2.66	1.27	0.790	0.710	0.630	0.590
	2.66	1.27	0.790	0.710	0.630	0.600
	2.66	1.29	0.790	0.720	0.630	0.600
	2.67	1.34	0.890	0.720	0.630	0.600
90	2.96	1.35	0.890	0.780	0.700	0.650
	2.96	1.36	0.890	0.780	0.700	0.660
	2.96	1.36	0.890	0.780	0.700	0.660
	2.96	1.36	0.890	0.790	0.700	0.660
	2.96	1.36	0.890	0.790	0.700	0.660
	2.96	1.36	0.890	0.790	0.700	0.660

TABLE III (Continued)

Verts	Seq Algo	Par Algo # of processors 1	2	4	8	16
	2.96	1.37	0.890	0.790	0.700	0.660
	2.97	1.37	0.890	0.790	0.700	0.660
	2.97	1.38	0.890	0.790	0.710	0.660
	2.97	1.39	1.000	0.830	0.710	0.680
95	3.27	1.50	1.050	0.890	0.820	0.760
	3.28	1.51	1.050	0.900	0.820	0.760
	3.29	1.51	1.050	0.900	0.820	0.760
	3.29	1.51	1.050	0.900	0.820	0.760
	3.29	1.51	1.050	0.900	0.820	0.760
	3.30	1.51	1.050	0.900	0.820	0.760
	3.30	1.51	1.050	0.900	0.820	0.760
	3.30	1.51	1.050	0.910	0.820	0.770
	3.31	1.53	1.050	0.910	0.820	0.770
	3.31	1.55	1.050	0.910	0.820	0.770
100	3.61	1.61	1.150	0.970	0.890	0.830
	3.61	1.61	1.160	0.980	0.890	0.830
	3.62	1.62	1.160	0.980	0.890	0.830
	3.62	1.62	1.160	0.980	0.900	0.830
	3.62	1.62	1.160	0.980	0.900	0.830
	3.62	1.62	1.160	0.980	0.900	0.840
	3.62	1.62	1.160	0.980	0.900	0.840
	3.63	1.63	1.160	0.980	0.900	0.840
	3.63	1.63	1.160	0.980	0.900	0.840
	3.65	1.64	1.170	0.990	0.900	0.840
105	3.80	1.77	1.310	1.080	1.020	0.930
	3.80	1.78	1.330	1.080	1.020	0.940
	3.80	1.78	1.330	1.080	1.020	0.940
	3.80	1.78	1.330	1.090	1.020	0.940
	3.80	1.79	1.330	1.090	1.020	0.940
	3.81	1.79	1.330	1.090	1.020	0.940
	3.81	1.79	1.330	1.090	1.020	0.940
	3.81	1.79	1.330	1.090	1.020	0.950
	3.81	1.80	1.340	1.090	1.030	0.970
	3.82	1.81	1.360	1.100	1.040	0.970
110	4.36	1.90	1.510	1.200	1.150	1.050
	4.36	1.90	1.510	1.200	1.150	1.050
	4.36	1.91	1.510	1.200	1.150	1.050
	4.37	1.91	1.510	1.200	1.150	1.050
	4.37	1.91	1.510	1.200	1.150	1.050
	4.37	1.91	1.510	1.200	1.150	1.050
	4.38	1.91	1.520	1.200	1.150	1.050
	4.38	1.92	1.520	1.200	1.150	1.060
	4.38	1.92	1.520	1.200	1.160	1.060
	4.39	2.07	1.520	1.210	1.160	1.060
115	4.75	2.07	1.720	1.330	1.300	1.180

TABLE III (Continued)

Verts	Seq Algo	Par Algo # of processors 1	2	4	8	16
	4.75	2.07	1.720	1.330	1.300	1.180
	4.76	2.07	1.720	1.330	1.300	1.180
	4.76	2.07	1.720	1.340	1.300	1.180
	4.76	2.08	1.720	1.340	1.300	1.180
	4.77	2.08	1.730	1.340	1.310	1.180
	4.77	2.08	1.730	1.340	1.310	1.180
	4.77	2.08	1.730	1.340	1.310	1.180
	4.78	2.08	1.730	1.340	1.310	1.180
	4.78	2.27	1.730	1.340	1.310	1.190
120	5.15	2.28	1.980	1.490	1.470	1.330
	5.16	2.28	1.980	1.490	1.470	1.330
	5.16	2.29	1.980	1.500	1.470	1.330
	5.17	2.29	1.980	1.500	1.480	1.340
	5.17	2.29	1.980	1.500	1.480	1.340
	5.19	2.29	1.980	1.500	1.480	1.340
	5.19	2.29	1.980	1.500	1.480	1.340
	5.20	2.30	1.980	1.500	1.480	1.340
	5.21	2.30	1.980	1.500	1.480	1.340
	5.21	2.44	1.980	1.510	1.490	1.340
125	5.61	2.44	2.180	1.620	1.620	1.460
	5.61	2.44	2.180	1.630	1.620	1.460
	5.61	2.45	2.180	1.630	1.630	1.460
	5.61	2.45	2.180	1.630	1.630	1.460
	5.62	2.45	2.180	1.630	1.630	1.460
	5.62	2.46	2.180	1.640	1.630	1.460
	5.63	2.46	2.180	1.640	1.630	1.460
	5.63	2.48	2.180	1.640	1.630	1.470
	5.65	2.48	2.190	1.640	1.630	1.470
	5.65	2.58	2.190	1.640	1.640	1.470
130	6.05	2.59	2.360	1.750	1.730	1.560
	6.06	2.59	2.360	1.750	1.740	1.570
	6.06	2.60	2.360	1.750	1.740	1.570
	6.07	2.60	2.360	1.750	1.740	1.570
	6.08	2.60	2.360	1.750	1.740	1.570
	6.08	2.60	2.360	1.750	1.740	1.570
	6.09	2.61	2.360	1.750	1.740	1.570
	6.10	2.61	2.360	1.760	1.740	1.570
	6.12	2.61	2.360	1.760	1.740	1.580
	6.13	2.63	2.360	1.760	1.750	1.580
135	6.54	2.87	2.750	2.020	1.970	1.790
	6.55	2.88	2.750	2.020	1.970	1.790
	6.55	2.89	2.750	2.020	1.980	1.790
	6.56	2.89	2.750	2.020	1.980	1.790
	6.56	2.89	2.760	2.030	1.980	1.800
	6.57	2.89	2.760	2.030	1.980	1.800

TABLE III (Continued)

Verts	Seq Algo	Par Algo # of processors 1	2	4	8	16
	6.59	2.90	2.760	2.030	1.980	1.810
	6.59	2.90	2.760	2.030	1.980	1.810
	6.62	2.90	2.760	2.030	1.990	1.810
	6.62	2.92	2.760	2.030	2.000	1.850
140	7.03	3.03	2.930	2.140	2.080	1.900
	7.03	3.04	2.930	2.150	2.080	1.910
	7.03	3.04	2.930	2.150	2.090	1.910
	7.03	3.04	2.930	2.150	2.090	1.910
	7.03	3.05	2.930	2.150	2.100	1.910
	7.05	3.05	2.940	2.150	2.100	1.910
	7.07	3.06	2.940	2.150	2.100	1.910
	7.07	3.06	2.940	2.160	2.100	1.910
	7.08	3.06	2.940	2.160	2.100	1.910
	7.08	3.15	2.940	2.160	2.100	1.920
145	7.53	3.15	3.070	2.260	2.170	1.990
	7.53	3.15	3.070	2.260	2.180	1.990
	7.54	3.15	3.080	2.270	2.180	1.990
	7.54	3.16	3.080	2.270	2.180	2.000
	7.54	3.16	3.080	2.270	2.180	2.000
	7.55	3.18	3.080	2.270	2.190	2.000
	7.55	3.19	3.080	2.270	2.200	2.000
	7.55	3.19	3.080	2.270	2.210	2.000
	7.58	3.22	3.080	2.290	2.240	2.010
	7.60	3.53	3.100	2.370	2.270	2.010
150	8.06	3.54	3.570	2.590	2.480	2.290
	8.06	3.54	3.570	2.600	2.480	2.290
	8.07	3.54	3.580	2.600	2.480	2.300
	8.07	3.54	3.580	2.600	2.480	2.300
	8.07	3.57	3.580	2.600	2.480	2.300
	8.09	3.68	3.580	2.600	2.490	2.300
	8.11	3.79	3.580	2.600	2.490	2.300
	8.11	3.92	3.580	2.600	2.490	2.300
	8.12	3.99	3.580	2.610	2.510	2.310
	8.13	4.12	3.590	2.610	2.510	2.320

TABLE IV

ANALYSIS OF THE FAST AND PARALLEL ALGORITHMS AVERAGE PROCESSING TIMES IN SECONDS

Verts	Seq Algo	Par Algo # of processors				
		1	2	4	8	16
5	0.032	0.410	0.100	0.031	0.018	0.010
10	0.062	0.423	0.109	0.040	0.020	0.011
15	0.119	0.434	0.112	0.040	0.030	0.020
20	0.182	0.454	0.124	0.052	0.038	0.030
25	0.274	0.489	0.149	0.070	0.050	0.050
30	0.377	0.511	0.166	0.087	0.070	0.070
35	0.496	0.540	0.190	0.110	0.096	0.090
40	0.631	0.593	0.220	0.130	0.130	0.120
45	0.782	0.627	0.255	0.170	0.164	0.150
50	0.953	0.677	0.294	0.210	0.191	0.182
55	1.145	0.751	0.339	0.269	0.240	0.232
60	1.350	0.813	0.389	0.321	0.284	0.270
65	1.579	0.880	0.443	0.390	0.330	0.328
70	1.824	0.967	0.507	0.480	0.392	0.388
75	2.093	1.075	0.597	0.574	0.479	0.460
80	2.372	1.157	0.676	0.632	0.532	0.512
85	2.656	1.271	0.800	0.712	0.625	0.593
90	2.963	1.366	0.901	0.791	0.702	0.661
95	3.294	1.515	1.050	0.902	0.820	0.763
100	3.623	1.622	1.160	0.980	0.897	0.835
105	3.806	1.788	1.332	1.088	1.023	0.946
110	4.372	1.926	1.514	1.201	1.152	1.053
115	4.765	2.095	1.725	1.337	1.305	1.181
120	5.181	2.305	1.980	1.499	1.478	1.337
125	5.624	2.469	2.182	1.634	1.629	1.463
130	6.084	2.604	2.360	1.753	1.740	1.571
135	6.575	2.893	2.756	2.026	1.981	1.804
140	7.050	3.058	2.935	2.152	2.094	1.910
145	7.551	3.208	3.080	2.280	2.200	1.999
150	8.089	3.723	3.579	2.601	2.489	2.301

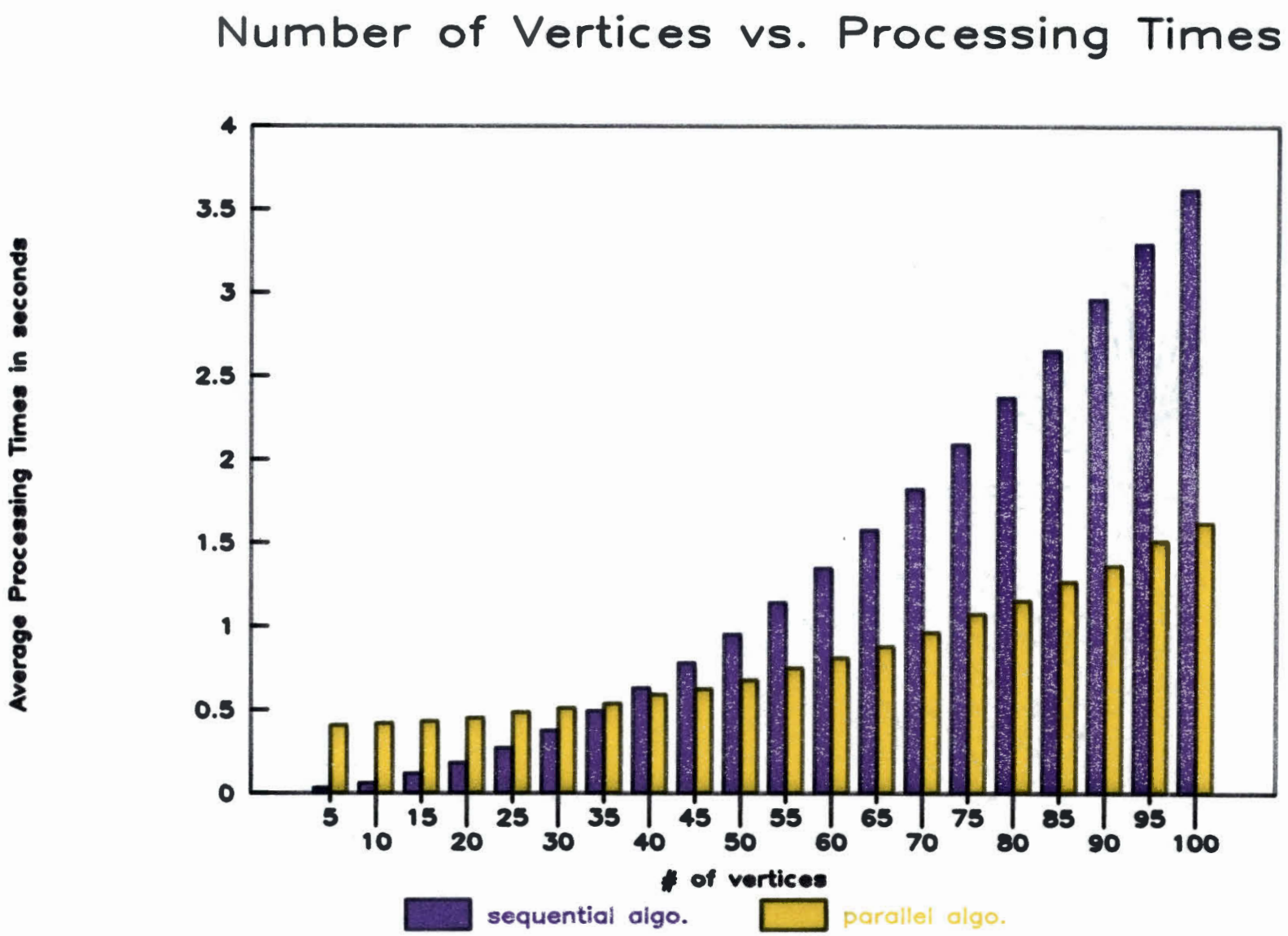

Figure 7. Number of Vertices vs. Processing Times for the vertex range 5-100

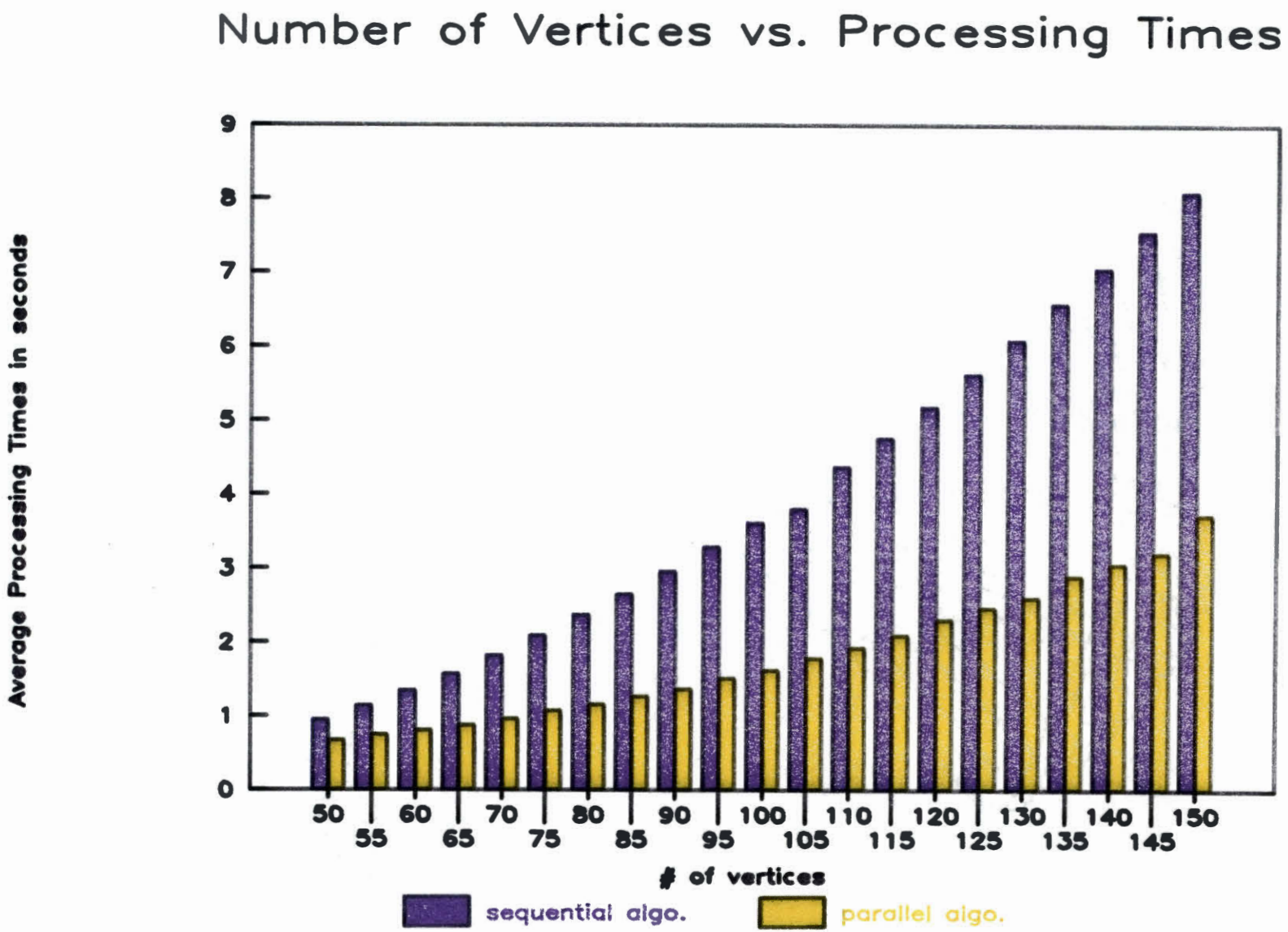

Figure 8. Number of Vertices vs. Processing Times for the vertex range 50-150

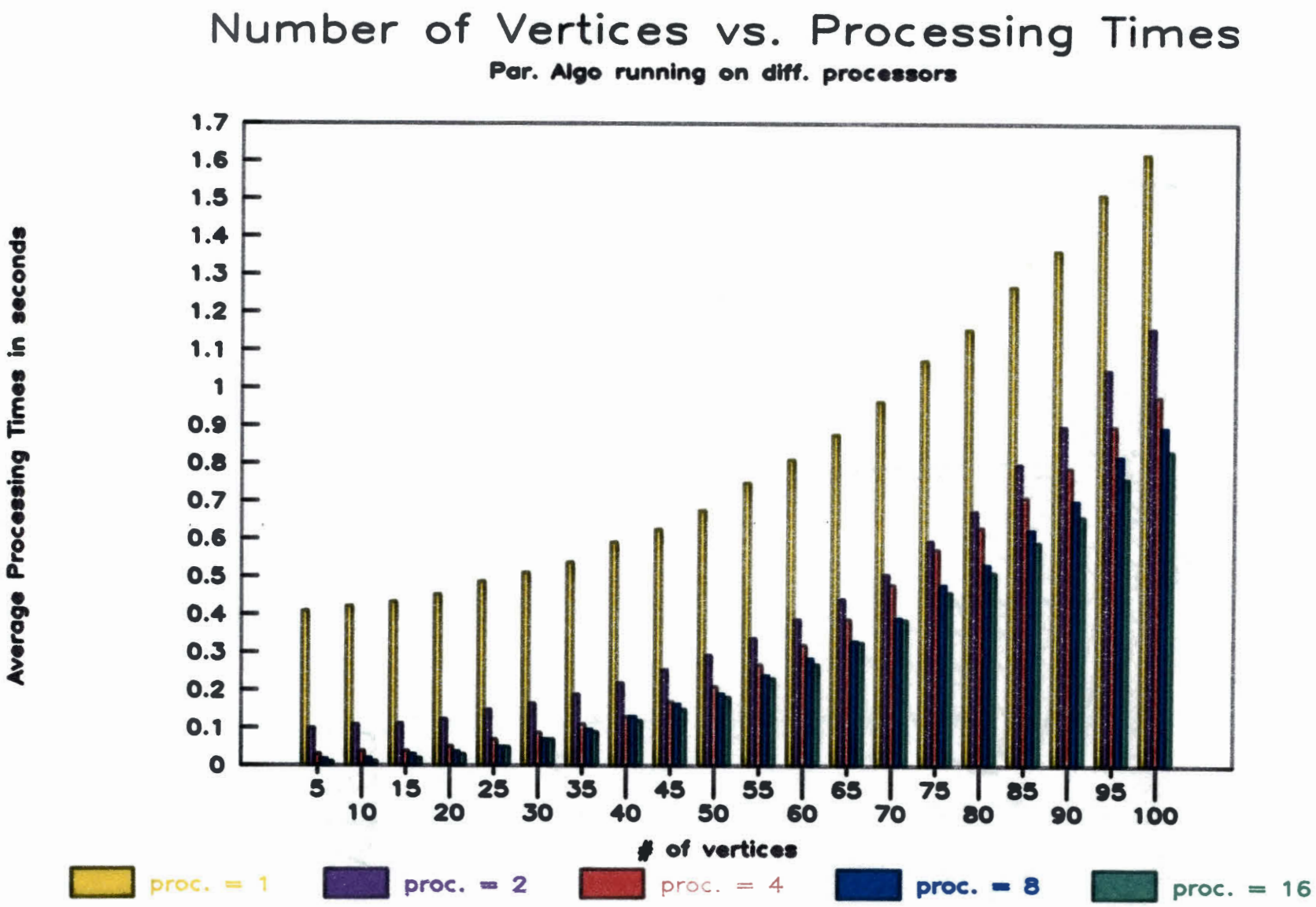

Figure 9. Number of Vertices vs. Processing Times for the vertex range 5-100 for the parallel algorithm

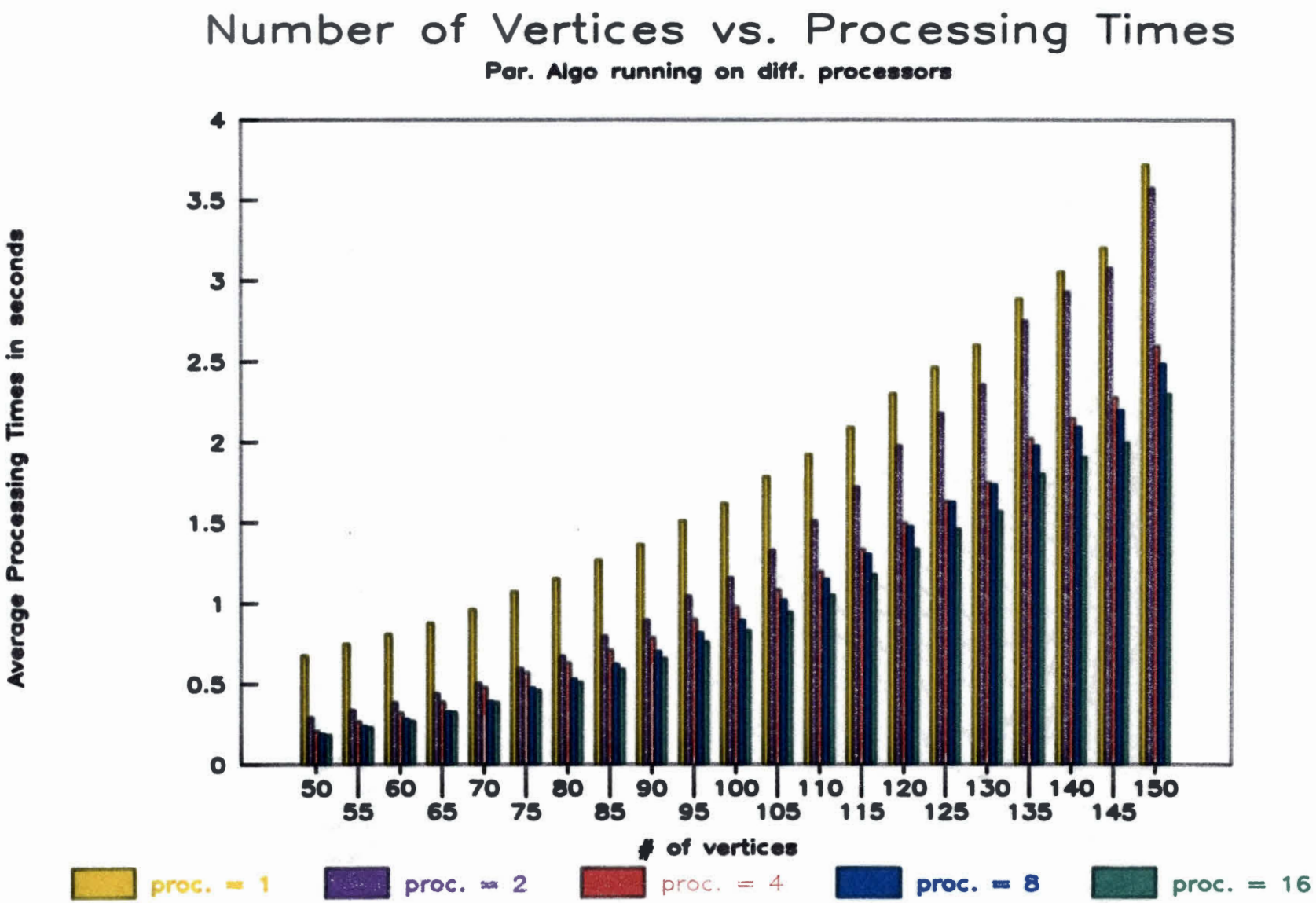

Figure 10. Number of Vertices vs. Processing Times for the vertex range 50-150 for the parallel algorithm

APPENDIX C

FAST ALGORITHM PROGRAM LISTING

```
/* program = domfast.c */
/*********************************************************/
/*                                                       */
/*     Dominators Fast Algorithm Program Listing         */
/*                                                       */
/*********************************************************/
/*                                                       */
/*     Author:  Sharmila Shankar                         */
/*     Date:    02/20/92                                 */
/*     Class:   COMSC 5000 - Thesis                      */
/*     Adviser: Dr. Blayne Mayfield                      */
/*                                                       */
/*********************************************************/
/* This is the fast algorithm for finding dominators in a */
/* flowgraph. The algorithm uses depth-first search and   */
/* an efficient method of computing functions defined on  */
/* paths in trees                                         */
/*                                                        */
/* The implementation of the algorithm uses the following */
/* arrays                                                 */
/* Input                                                  */
/* succ(v):     The set of vertices w such that (v,w) is  */
/* an edge of the graph                                   */
/*                                                        */
/* Computed                                               */
/* parent(w):   The vertex which is the parent of vertex w*/
/* in the spanning tree generated by the search           */
/*                                                        */
/* pred(w):     The set of vertices v such that (v,w) is  */
/* an edge of the graph                                   */
/* semi(w):   A number defined as follows:                */
/*    (i)   Before vertex w is numbered, semi(v) = 0      */
/*    (ii)  After w is numbered but before its semi-      */
/*          dominator is computed, semi(w) is the number  */
/*          of w                                          */
/*    (iii) After the semidominator of w is computed,     */
/*          semi(w) is the number of the semidominator of */
/*          w                                             */
/* vertex(i):   The vertex whose number is i              */
/* bucket(w):   A set of vertices whose semidominator is w*/
/* dom(w):      A vertex defined as follows:              */
/*    (i)  After step 3, if the semidominator of w is its */
/*         immediate dominator, then dom(w) is the imme-  */
/*         diate dominator of w. Otherwise dom(w) is a    */
/*         vertex v whose number is smaller than w and    */
/*         whose immediate dominator is also w's immediate*/
/*         dominator                                      */
/*    (ii) After step 4, dom(w) is the immediate dominator*/
/*         of w                                           */
/*********************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int **succ, **pred, **bucket, *dom;
```

```
int *parent, *ancestor, *vertex, *label, *semi;
int r,n,u,v,x,w,i,j, start_time, end_time, exec_time;
FILE *fp, *fopen();
char fname[20];

/* beginning of the main program */

main(argc, argv)
int argc;
char *argv[];
{
  start_time = clock();
  printf("The adjacency matrix file name: ");
  strcpy(fname, argv[1]);
  printf("%s\n",fname);
  if((fp = fopen(fname, "r")) == NULL)
    {
     printf("CANNOT OPEN FILE...PROGRAM ABORTED\n\n");
     exit(0);
    }
  printf("The number of vertices: ");
  n = atoi(argv[2]);
  printf("%d\n",n);
  printf("The start vertex: ");
  r = atoi(argv[3]);
  printf("%d\n",r);

/* allocate pointer arrays : set succ, pred, bucket to
address of newly allocated matrices */
/* allocate data arrays : set first element of succ, pred,
bucket to address of first element of newly allocated data
arrays */
/* initialise pointer arrays : set each element of succ,
pred, bucket to address of corresponding element of data
arrays */

  succ = (int**)malloc((n+1)*(sizeof(int*)));
  succ[0] = (int*)malloc((n+1)*(n+1)*(sizeof(int)));
  for (i = 1; i <= n; i++)
     succ[i] = succ[0] + ((n+1) * i);

  pred = (int**)malloc((n+1)*(sizeof(int*)));
  pred[0] = (int*)malloc((n+1)*(n+1)*(sizeof(int)));
  for (i = 1; i <= n; i++)
     pred[i] = pred[0] + ((n+1) * i);

  bucket = (int**)malloc((n+1)*(sizeof(int*)));
  bucket[0] = (int*)malloc((n+1)*(n+1)*(sizeof(int)));
  for (i = 1; i <= n; i++)
     bucket[i] = bucket[0] + ((n+1) * i);

  dom  = (int *)malloc((n+1) * (sizeof(int)));
  parent  = (int *)malloc((n+1) * (sizeof(int)));
  ancestor  = (int *)malloc((n+1) * (sizeof(int)));
```

```
  label  = (int *)malloc((n+1) * (sizeof(int)));
  vertex  = (int *)malloc((n+1) * (sizeof(int)));
  semi  = (int *)malloc((n+1) *(sizeof(int)));

/* read in the adjacency matrix */

  for(i = 1; i <= n; i++)
   for(j = 1; j <= n; j++)
    fscanf(fp, "%d", &succ[i][j]);

  printf("\nThe adjacency matrix for n = %d vertices is :
\n\n",n);
  for(i = 1; i <= n; i++)
   {
    printf("%2d   ",i);
   for(j = 1; j <= n; j++)
    printf("%d ", succ[i][j]);
    printf("\n");
   }

  /* step 1 */
  /* This uses the recursive procedure DFS below to carry
out the depth-first search */

  for(v = 1; v <= n; v++)
   {
    semi[v] = 0;
    for(w = 1; w <= n; w++)
     {
       pred[v][w] = 0;
       bucket[v][w] = 0;
     }
   }

  x = n;
  n = 0;
  DFS(r);

  for(i = n; i >= 2; i--)
   {
    w = vertex[i];

  /* step 2 */

    for (v = 1; v <= x; v++)
      {
        if(pred[w][v] == 1)
         {
           u = EVAL(v);
           if(semi[u] < semi[w])
             semi[w] = semi[u];
         }
       }
    bucket[vertex[semi[w]]][w] = 1;
```

```
    LINK(parent[w], w);

  /* step 3 */

    for (v = 1; v <= x; v++)
     {
       if(bucket[parent[w]][v] == 1)
        {
          bucket[parent[w]][v] = 0;
          u = EVAL(v);
          if(semi[u] < semi[v])
            dom[v] = u;
          else
            dom[v] = parent[w];
        }
     }
    }

   /* step 4 */

   for(i = 2; i <= n; i++)
     {
       w = vertex[i];
       if(dom[w] != vertex[semi[w]])
          dom[w] = dom[dom[w]];
     }
   dom[r] = 0;

  printf("\nThe Dominators of the Flowgraph are \n\n");
  for(i = 1; i <= x; i++)
    printf("(%d, %d) \n", i, dom[i]);

  fclose(fp);

  /* free all allocated memory */

 free(succ);
 free(pred);
 free(bucket);
 free(dom);
 free(parent);
 free(ancestor);
 free(label);
 free(vertex);
 free(semi);
  end_time = clock();
  exec_time = end_time - start_time;
  printf("\n The execution time is %2.2f
\n\n",(float)(exec_time)/1000000);

 }

/*************************************************************/
/*                                                           */
```

```
/* DFS                                                          */
/* ---                                                          */
/* This procedure conducts the depth-first search               */
/****************************************************************/

DFS(v)
int v;
{
 int w;
 semi[v] = n = n+1;
 vertex[n] = label[v] = v;
 ancestor[v] = 0;
 for(w = 1; w <= x; w++)
  {
   if(succ[v][w] == 1)
    {
     if(semi[w] == 0)
      {
        parent[w] = v;
        DFS(w);
      }
     pred[w][v] = 1;
    }
  }
 }/* end of DFS */

/****************************************************************/
/*                                                              */
/* COMPRESS                                                     */
/* --------                                                     */
/* This procedure carries out path compression                  */
/****************************************************************/

COMPRESS(v)
int v;
{
 if(ancestor[ancestor[v]] != 0)
  {
    COMPRESS(ancestor[v]);
 if(semi[label[ancestor[v]]] < semi[label[v]])
   label[v] = label[ancestor[v]];
 ancestor[v] = ancestor[ancestor[v]];
 }
 } /* end of COMPRESS */

/****************************************************************/
/*                                                              */
/* EVAL                                                         */
/* ----                                                         */
/* This procedure returns v if v is the root in the forest*/
/* Otherwise it returns any vertex u not equal to r(the   */
/* root of the tree in the forest) of minimum semi(u) on  */
/* the path from r to v                                         */
/****************************************************************/
```

```
int EVAL(v)
int v;
{
 if (ancestor[v] == 0)
    return v;
 else
    {
      COMPRESS(v);
      return(label[v]);
    }
 } /* end of EVAL */

/*************************************************************/
/*                                                           */
/* LINK                                                      */
/* ----                                                      */
/* This procedure adds the edge (v,w) to the forest          */
/*************************************************************/

LINK(v,w)
int v, w;
{
 ancestor[w] = v;

}/* end of LINK */
```

APPENDIX D

PARALLEL ALGORITHM PROGRAM LISTING

```
/* program = dompar.c */
/*********************************************************/
/*                                                       */
/*      Parallel Algorithm Program Listing               */
/*                                                       */
/*********************************************************/
/*                                                       */
/*     Author:  Sharmila Shankar                         */
/*     Date:    02/20/92                                 */
/*     Class:   COMSC 5000 - Thesis                      */
/*     Adviser: Dr. Blayne Mayfield                      */
/*                                                       */
/*********************************************************/
/* This is the parallel algorithm for finding dominators  */
/* in a flowgraph. The algorithm uses the parallel depth  */
/* first search strategy by Aggarwal, Anderson and Kao    */
/*                                                        */
/* The implementation of the algorithm uses the following */
/* arrays                                                 */
/*                                                        */
/* Input                                                  */
/* succ(v):  The set of vertices w such that (v,w) is an  */
/* edge of the graph                                      */
/*                                                        */
/* ALM(v): The list of vertices which are heads of the    */
/*         edges with tail v                              */
/*                                                        */
/* U(v): The list of vertices which are adjacent to v and */
/*       and are still unvisited                          */
/*                                                        */
/* arc_list:    The list of visited vertices              */
/*                                                        */
/* frond_list:   The list of unvisited vertices           */
/*                                                        */
/* Computed                                               */
/* parent(w):   The vertex which is the parent of vertex  */
/* w in the spanning tree generated by the search         */
/* pred(w):     The set of vertices v such that (v,w) is  */
/* an edge of the graph                                   */
/* semi(w):  A number defined as follows:                 */
/*    (i)   Before vertex w is numbered, semi(v) = 0      */
/*    (ii)  After w is numbered but before its semi-      */
/*          dominator is computed, semi(w) is the number  */
/*          of w                                          */
/*    (iii) After the semidominator of w is computed,     */
/*          semi(w) is the number of the semidominator of */
/*          w                                             */
/* vertex(i):   The vertex whose number is i              */
/* bucket(w):   A set of vertices whose semidominator is w*/
/* dom(w):      A vertex defined as follows:              */
/*    (i)  After step 3, if the semidominator of w is its */
/*         immediate dominator, then dom(w) is the imme-  */
/*         diate dominator of w. Otherwise dom(w) is a    */
/*         vertex v whose number is smaller than w and    */
```

```
/*          whose immediate dominator is also w's immediate*/
/*          dominator                                      */
/*     (ii) After step 4, dom(w) is the immediate dominator*/
/*          of w                                           */
/*********************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <parallel/microtask.h>
#include <parallel/parallel.h>

int **succ, **pred, *dom, *parent, *ancestor, *vertex;
int *label, **bucket, flr();
shared int *semi, **arc_list, **frond_list, **U, *EM, *el;
shared sbarrier_t *barrier;
shared int **ALM,n,x;
shared slock_t magiclock, *lp = &magiclock;
int r,u,v,w,i,j,k,start_time, end_time, exec_time;
FILE *fp, *fopen();
char fname[20];
int nprocs, m_rele_procs(), m_park_procs();

/* beginning of the main program */

main(argc,argv)
int argc;
char *argv[];
{
  void main_process();
  char *shmalloc();
  printf("The adjacency matrix file name: ");
  strcpy(fname,argv[1]);
  printf("%s\n",fname);
  if((fp = fopen(fname, "r")) == NULL)
    {
     printf("CANNOT OPEN FILE...PROGRAM ABORTED\n\n");
     exit(0);
    }
  printf("The number of vertices: ");
  n = atoi(argv[2]);
  printf("%d\n",n);
  printf("The start vertex: ");
  r = atoi(argv[3]);
  printf("%d\n",r);
  printf("Number of processors available:
%d\n",cpus_online());
  printf("The number of processes asked for: ");
  nprocs = atoi(argv[4]);
  printf("%d\n",nprocs);

/* shared memory allocation */
/* allocate pointer arrays : set succ, pred, bucket to
address of newly allocated matrices */
```

```
/* allocate data arrays : set first element of succ, pred,
bucket to address of first element of newly allocated data
arrays */
/* initialise pointer arrays : set each element of succ,
pred, bucket to address of corresponding element of data
arrays */

  start_time = clock();

  succ = (int**)shmalloc((n+1)*(sizeof(int*)));
  succ[0] = (int*)shmalloc((n+1)*(n+1)*(sizeof(int)));
  for (i = 1; i <= n; i++)
     succ[i] = succ[0] + ((n+1) * i);

  ALM = (int**)shmalloc((n+1)*(sizeof(int*)));
  ALM[0] = (int*)shmalloc((n+1)*(n+1)*(sizeof(int)));
  for (i = 1; i <= n; i++)
     ALM[i] = ALM[0] + ((n+1) * i);

  pred = (int**)shmalloc((n+1)*(sizeof(int*)));
  pred[0] = (int*)shmalloc((n+1)*(n+1)*(sizeof(int)));
  for (i = 1; i <= n; i++)
     pred[i] = pred[0] + ((n+1) * i);

  bucket = (int**)malloc((n+1)*(sizeof(int*)));
  bucket[0] = (int*)malloc((n+1)*(n+1)*(sizeof(int)));
  for (i = 1; i <= n; i++)
     bucket[i] = bucket[0] + ((n+1) * i);

  arc_list = (int**)shmalloc((n+1)*(sizeof(int*)));
  arc_list[0] = (int*)shmalloc((n+1)*(n+1)*(sizeof(int)));
  for (i = 1; i <= n; i++)
     arc_list[i] = arc_list[0] + ((n+1) * i);

  frond_list = (int**)shmalloc((n+1)*(sizeof(int*)));
  frond_list[0] = (int*)shmalloc((n+1)*(n+1)*(sizeof(int)));
  for (i = 1; i <= n; i++)
     frond_list[i] = frond_list[0] + ((n+1) * i);

  U = (int**)shmalloc((n+1)*(sizeof(int*)));
  U[0] = (int*)shmalloc((n+1)*(n+1)*(sizeof(int)));
  for (i = 1; i <= n; i++)
     U[i] = U[0] + ((n+1) * i);

  el = (int *)malloc((n+1) * (sizeof(int)));
  EM = (int *)shmalloc((n+1) * (sizeof(int)));
  dom  = (int *)malloc((n+1) * (sizeof(int)));
  parent  = (int *)shmalloc((n+1) * (sizeof(int)));
  ancestor  = (int *)shmalloc((n+1) * (sizeof(int)));
  label  = (int *)shmalloc((n+1) * (sizeof(int)));
  vertex  = (int *)shmalloc((n+1) * (sizeof(int)));
  semi  = (int *)shmalloc((n+1) *(sizeof(int)));

  for (i = 1; i <= n; i++)
```

```
      for (j = 1; j <= n; j++)
        { ALM[i][j] = 0;
          EM[i] = 0;
        }

  for(i = 1; i <= n; i++)
     {
       for(j = 1; j <= n; j++)
       {
       fscanf(fp, "%d", &succ[i][j]);
       U[i][j] = succ[i][j];
       if(succ[i][j] == 1)
        {
          k = 1;
          while (1)
           {
             if (ALM[j][k] == 0)
               {
                 ALM[j][k++] = i;
                 break;
               }
             else
                k++;
           }
            EM[j] = k - 1;
         }
     }
  }

  printf("\nThe adjacency matrix for n = %d vertices is :
\n\n",n);
  for(i = 1; i <= n; i++)
   {
    printf("%2d   ",i);
   for(j = 1; j <= n; j++)
    printf("%d ", succ[i][j]);
    printf("\n");
   }

  /* step 1 */
  /* This step conducts the depth-first search */

  for(v = 1; v <= n; v++)
   {
    semi[v] = 0;
    for(w = 1; w <= n; w++)
     {
       pred[v][w] = 0;
       bucket[v][w] = 0;
       arc_list[v][w] = 0;
       frond_list[v][w] = 0;
     }
   }
  x = n;
```

```
/* NUMB_VERTICES_VISITED IS n */

/* set number of processes and initialize the barriers */

  m_set_procs(nprocs);
  s_init_barrier(&barrier,nprocs);
  n = 0;
  parent[r] = 0;
  PMDFS(r);
  m_kill_procs();   /* kill the child processes */

  for(i = x; i >= 2; i--)
   {
    w = vertex[i];

  /* step 2 */

    for (v = 1; v <= x; v++)
      {
        if(pred[w][v] == 1)
         {
           u = EVAL(v);
           if(semi[u] < semi[w])
             semi[w] = semi[u];
         }
       }

    bucket[vertex[semi[w]]][w] = 1;
    LINK(parent[w], w);

  /* step 3 */

    for (v = 1; v <= x; v++)
     {
       if(bucket[parent[w]][v] == 1)
        {
          bucket[parent[w]][v] = 0;
          u = EVAL(v);
          if(semi[u] < semi[v])
            dom[v] = u;
          else
            dom[v] = parent[w];
        }
     }
    }

   /* step 4 */

   for(i = 2; i <= n; i++)
     {
       w = vertex[i];
       if(dom[w] != vertex[semi[w]])
          dom[w] = dom[dom[w]];
     }
```

```
   dom[r] = 0;

  printf("\nThe Dominators of the Flowgraph are \n\n");
  for(i = 1; i <= x; i++)
    printf("(%d, %d) \n", i, dom[i]);

  fclose(fp);

  /* free the shared memory allocation and the other
allocations */

  shfree(succ);
  shfree(ALM);
  shfree(pred);
  free(bucket);
  free(dom);
  shfree(arc_list);
  shfree(frond_list);
  shfree(U);
  shfree(parent);
  shfree(ancestor);
  shfree(label);
  shfree(vertex);
  shfree(semi);
  shfree(EM);
  free(el);

  end_time = clock();
  exec_time = end_time - start_time;
  printf("\nThe execution time is: %2.2f
\n",(float)(exec_time)/(1000000));

 }

/*********************************************************/
/* PMDFS                                                 */
/* -----                                                 */
/* This procedure carries out the parallel depth-first   */
/*  search                                               */
/*********************************************************/

PMDFS(v)
int v;
{
 int w;
 semi[v] = n = n + 1;
 vertex[n] = label[v] = v;
 ancestor[v] = 0;

/* release if any parked child processes */
  m_rele_procs;
  m_fork(main_process,v);
/* park the child processes for future use */
  m_park_procs;
```

```
  for(w = 1; w <= x; w++)
    {
     if(U[v][w] == 1)
     {
         parent[w] = v;
         arc_list[v][w] = 1;
         frond_list[w][v] = 0;
         PMDFS(w);
     }
         pred[w][v] = 1;
    }

 }/* end of PMDFS */

/*************************************************************/
/* COMPRESS                                                  */
/* --------                                                  */
/* This procedure carries out path compression               */
/*************************************************************/

COMPRESS(v)
int v;
{
 if(ancestor[ancestor[v]] != 0)
  {
    COMPRESS(ancestor[v]);
 if(semi[label[ancestor[v]]] < semi[label[v]])
   label[v] = label[ancestor[v]];
 ancestor[v] = ancestor[ancestor[v]];
 }
} /* end of COMPRESS */

/*************************************************************/
/* EVAL                                                      */
/* ----                                                      */
/* This procedure returns v if v is the root in the forest*/
/* Otherwise it returns any vertex u not equal to r(the   */
/* root of the tree in the forest) of minimum semi(u) on  */
/* the path from r to v                                      */
/*************************************************************/

int EVAL(v)
int v;
{
 if (ancestor[v] == 0)
    return v;
 else
    {
      COMPRESS(v);
      return(label[v]);
    }
 } /* end of EVAL */
/*************************************************************/
```

```
/* LINK                                                          */
/* ----                                                          */
/* This procedure adds the edge (v,w) to the forest              */
/*****************************************************************/

LINK(v,w)
int v, w;
{
 ancestor[w] = v;

} /* end of LINK */

/*****************************************************************/
/* main_process                                                  */
/* ------------                                                  */
/* This procedure carries out the parallel search and            */
/* deletions from the unvisited list matrix in parallel.         */
/* Dynamic Scheduling multitasking is adopted                    */
/*****************************************************************/

void main_process(v)
int v;
{
 int procs;
 int i, j,base,top;

  procs = m_get_numprocs(); /* number of processors */

 while((base = 1 * (m_next() - 1)) < x)
 {
    top = base + 1;
    if(top >= x) top = x - 1;
    for (i = base; i < top; i++);
   {
    for (j = 1; j <= flr(EM[v]/procs); j++)
     {
       if(((procs * (j - 1)) + i) <= EM[v])
         {
           m_lock();
           el[i] = ALM[v][(procs * (j - 1)) + i];
           U[el[i]][v] = 0;
           if(semi[el[i]] == 0)
             frond_list[el[i]][v] = 1;
           m_unlock();
         }
       }
}
}
 s_wait_barrier(&barrier);  /* synchronization point */

} /* end of main_process */

/*****************************************************************/
/* flr                                                           */
```

```
/* ---                                                          */
/* This procedure returns the floor of a number                 */
/*************************************************************/

int flr(num)
int num;
{
  return(num + 1);
 } /* end of flr */
```

APPENDIX E

RANDOM GENERATION PROGRAM LISTING

```
/* program = rand_flow.c */
/*************************************************************/
/*                                                           */
/*       Random Generation of Flow Graphs Driver Listing     */
/*                                                           */
/*************************************************************/
/*                                                           */
/*               Author:  Sharmila Shankar                   */
/*               Date:    04/20/92                           */
/*               Class:   COMSC 5000 - Thesis                */
/*               Adviser: Dr. Blayne Mayfield                */
/*                                                           */
/*************************************************************/
/* This program generates 10 random flow graphs for nodes */
/* 5 to 150 in steps of 5 and is the driver routine for   */
/* the execution of the fast algorithm and the parallel   */
/* algorithm. Here Node means vertex                       */
/*************************************************************/
#include <stdio.h>
#include <string.h>
#define LOW_LIM_NODE 5
#define HIGH_LIM_NODE 100
#define NODE_STEP 5
#define MAX_PROCS 16
#define MAX_FLOW_GRAPHS_PER_NODE 10
float seed = 1.0;
float rand_num_generator();

main()
{
 int procs = 0, node = 0, count = 0, node_count = 0,
line_count = 0;
 int rand_numb = 0;
 FILE *fp, *fopen();
 char fname[20], temp[4], faststr[40], parstr[40];
 for (node = LOW_LIM_NODE; node <= HIGH_LIM_NODE; node =
node + NODE_STEP)
 for(count = 1; count <= MAX_FLOW_GRAPHS_PER_NODE; count++)
         {
           strcpy(fname,"");
           strcpy(fname,"adj");
           strcpy(temp,"");
           sprintf(temp,"%d",node);
           strcat(fname,temp);
           strcat(fname,"__");
           strcpy(temp,"");
           sprintf(temp,"%d",count);
           strcat(fname,temp);

    /* continue generating till a connected graph is got */

           while (1)
           {
           fp = fopen(fname,"w");
```

```
            node_count = 1;
            line_count = 1;
            while(line_count <= node)
             {
              rand_numb = rand_num_generator() * node;
              if((rand_numb % 2) == 0)
                fprintf(fp,"1 ");
              else
                fprintf(fp,"0 ");
              if((node_count % node) == 0)
               {
                fprintf(fp,"\n");
                line_count++;
               }
              node_count++;
              }
              fclose(fp);

              /* testing of connectivity */

          if(conn(fname,node) == 1)
           break; /* graph is connected, so exit from loop */
             }
              strcpy(faststr,"");
              strcpy(faststr,"domfast ");
              strcat(faststr,fname);
              strcat(faststr," ");
              strcpy(temp,"");
              sprintf(temp,"%d",node);
              strcat(faststr,temp);
              strcat(faststr," 1 ");
              system(faststr);
        for(procs = 1; procs <= MAX_PROCS; procs = procs * 2)
                {
                  strcpy(parstr,"");
                  strcpy(parstr,"dompar ");
                  strcat(parstr,fname);
                  strcat(parstr," ");
                  strcpy(temp,"");
                  sprintf(temp,"%d",node);
                  strcat(parstr,temp);
                  strcat(parstr," 1 ");
                  strcpy(temp,"");
                  sprintf(temp,"%d",procs);
                  strcat(parstr,temp);
                  system(parstr);
            }
          }
}

/*********************************************************/
/* procedure : rand_num_generator()                      */
/* This procedure returns a random number                */
/*********************************************************/
```

```
float rand_num_generator()
{
  float a,q,r,m,value,lo,test;
  int hi;
  a = 16807;
  m = 2147483647.0;
  q = 127773.0;
  r = 2836.0;

  hi = seed/q;
  lo = seed - q * hi;
  test = a * lo - r * hi;
  if(test > 0.0)
    seed = test;
  else
    seed = test + m;
  value = seed/m;
  return value;
 }/* end of rand_num_generator */

/*************************************************************/
/*                                                           */
/* conn                                                      */
/* ----                                                      */
/* This procedure tests out the connectivity of a given      */
/* adjacency matrix and returns a flag                       */
/*************************************************************/
conn(fp,n)
char fname[20];
int n;
{
int **l, **c, i,j,k,flag;
struct Cost {
              int **succ;
              } *C;

  FILE *fp, * fopen();
  strcpy(fname, fp);
  printf("The name of the adjacency matrix: %s",fname);
  if((fp = fopen(fname, "r")) == NULL)
   {
     printf("CANNOT OPEN FILE...PROGRAM ABORTED\n\n");
     exit(0);
    }
  printf("The number of vertices: ");
  printf("%d\n",n);

  C = (struct *)malloc((n+1) * sizeof(struct));
  C.succ = (int**)malloc((n+1)*(sizeof(int*)));
  C.succ[0] = (int*)malloc((n+1)*(n+1)*(sizeof(int)));
  for (i = 1; i <= n; i++)
     C.succ[i] = C.succ[0] + ((n+1) * i);

  l = (int**)malloc((n+1)*(sizeof(int*)));
```

```
  l[0] = (int*)malloc((n+1)*(n+1)*(sizeof(int)));
  for (i = 1; i <= n; i++)
     l[i] = l[0] + ((n+1) * i);

  c = (int**)malloc((n+1)*(sizeof(int*)));
  c[0] = (int*)malloc((n+1)*(n+1)*(sizeof(int)));
  for (i = 1; i <= n; i++)
     c[i] = c[0] + ((n+1) * i);

  for(i = 1; i <= n; i++)
   for(j = 1; j <= n; j++)
    fscanf(fp, "%d", &l[i][j]);

  printf("\n The adjacency matrix for n = %d vertices is :
\n\n",n);
  for(i = 1; i <= n; i++)
   {
    printf("%2d   ",i);
   for(j = 1; j <= n; j++)
    printf("%d ", l[i][j]);
    printf("\n");
   }

for(i = 1; i <= n; i++)
   {
  C[0].succ[i][i] = 1 + l[i][i];
 }

for(i = 1; i <= n; i++)
   for (j = 1; j <= n; j++)
    if (i != j)
      C[0].succ[i][j] = l[i,j];

for (k = 1; k <= n; k++)
  for(i = 1; i <= n; i++)
   for(j = 1; j <= n; j++)
    C[k].succ[i][j] = C[k - 1].succ[i][j] +
                          C[k - 1]succ[i][k] * C[k -
1].succ[k][j];

for(i = 1; i <= n; i++)
   for (j = 1; j <= n; j++)
      c[i][j] = C[n].succ[i][j];

flag = 1;
for(i = 1; i <= n; i++)
   for (j = 1; j <= n; j++)
     if(c[i][j] != 1)
       flag = 0;
fclose(fp);
return flag;
} /* end of conn */
```

APPENDIX F

USER MANUAL

USER MANUAL

The fast, parallel and the random generation programs were run on the Sequent. The following abbreviations are used. Adj Mat is the adjacency matrix, vertices are the number of vertices, vertex is the start vertex and the procs is the number of processors asked for.

Part 1: Fast Program

At the Sequent prompt type

domfast <adj mat> <vertices> <vertex> <Enter>

The program will display the adjacency matrix, the number of vertices, the start vertex and the pairs of the dominators in the form (vertex, its dominator) on the screen, in that order.

Part 2: Parallel Program

At the Sequent prompt type

dompar <adj mat> <vertices> <vertex> <procs> <Enter>

The program will display the adjacency matrix, the number of vertices, the start vertex, the number of processors available, the number of processors asked for and the pairs of the dominators in the form (vertex, its dominator) on the screen, in that order.

Part 3: Random Generation Program

At the Sequent prompt type

rand_flow <Enter>

The program will display the adjacency matrices, the number of vertices, the start vertex, the number of processors available, the number of processors asked for and the pairs of the dominators in the form (vertex, its dominator) on the screen, in that order for the respective programs being run.

VITA

Sharmila Shankar

Candidate for the Degree of

Master of Science

Thesis: PARALLELIZATION OF THE FAST ALGORITHM FOR COMPUTATION OF DOMINATORS IN A FLOWGRAPH

Major Field: Computer Science

Biographical:

Personal Data: Born in Agra, Uttar Pradesh, India, August 4, 1964, the daughter of Ashish Kumar Mookerjee and Kamala Mookerjee and the wife of Shankar Narayanaswamy.

Education: Graduated from St. Helena's High School, Poona, India, in June 1982; received Bachelor of Science Degree in Mathematics from University of Poona at Poona in June 1985; received Master of Science Degree in Mathematics with specialization in Computer Science from Indian Institute of Technology, Bombay, India in July 1987; completed requirements for the Master of Science degree at Oklahoma State University in July 1992.

Professional Experience: Programmer, M.N. Dastur & Co. Ltd., Bombay, India, July 1987 to August 1988. Systems Executive, Mahindra British Telecom Ltd., Bombay, India and Cardiff, United Kingdom, September 1988 to August 1990. Graduate Assistant, Computer Services, Department of Agricultural Economics, Oklahoma State University, April 1991 to August 1992.