OVERVIEW AND EXTENSIONS OF A SYSTEM FOR ROUTING DIRECTED GRAPHS ON SIMD ARCHITECTURES *

Sherryl Tomboulian

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center, Hampton VA 23665

ABSTRACT

Many problems can be described in terms of directed graphs that contain a large number of vertices where simple computations occur using data from adjacent vertices. A method is given for parallelising such problems on an SIMD machine model that uses only nearest neighbor connections for communication, and has no facility for local indirect addressing. Each vertex of the graph will be assigned to a processor in the machine. Rules for a labeling are introduced that support the use of a simple algorithm for movement of data along the edges of the graph. Additional algorithms are defined for addition and deletion of edges. Modifying or adding a new edge takes the same time as parallel traversal. This combination of architecture and algorithms defines a system that is relatively simple to build and can do fast graph processing. All edges can be traversed in parallel in time $O(T)$, where T is empirically proportional to the average path length in the embedding times the average degree of the graph. Additionally we present an extension to the above method which allows for enhanced performance by allowing some broadcasting capabilities.

Keywords: routing algorithm, SIMD architecture, parallel processing, graph embedding, interconnection network

INTRODUCTION

There are many problems that can be formulated as directed graphs. Such problems include circuit simulation, semantic networks and topography. Typically the *real-world* versions of these problems contain 100,000 vertices or more, and while the computations that occur at each vertex are simple, the size of the problems makes them computationally intensive. A natural way to parallelize these problems is a paradigm in which each processor is assigned a vertex in the graph, and there is some mechanism for realizing the arcs. This fine grained approach suggests the use of SIMD architectures, which can be built with many thousands of processors.

The problem with using SIMD architectures is that often they do not easily support generalized message passing schemes. This paper presents a method of embedding graphs in a class of SIMD architectures by using a special space-time labeling that supports message delivery and incremental addition of paths. The algorithms for this system are presented in (Refs. 16, 17). Basic concepts of the system will be reviewed, followed by an important generalization of the original method.

To maximize the number of processors which can be built, we choose the simplest hardware definition necessary to solve graph oriented problems. The machine model used is SIMD: there is a controller and a large number of slave processors which can execute the same instruction stream simultaneously. The processors have exclusively local memory, and they have no facilities for indirect addressing. The processors must be connected in a topology with the following requirements: (1) there must be some path between any two processors; (2) every neighbor link must be bi-directional, i.e. if A is a neighbor of B, then B must be a neighbor of A; and (3) the neighbor relations between processors must have a consistent invertible labeling. A more precise definition of the labeling requirements can be found in (Ref. 16). It suffices that most networks (Ref. 4) including grid, hypercube, cube connected cycles (Ref. 13), shuffle exchange (Ref. 14), and mesh of trees (Ref. 7) are admissible under the scheme. Additional requirements are that the processors be able to read from or write to their neighbors' memories, and that at least one of the processors acts as a serial port between the processors and the controller.

The Massively Parallel Processor (MPP) built by Goodyear Aerospace is an SIMD architecture with single bit processors arranged in a 128 by 128 processor grid (Ref. 2). The MPP is not the *perfect* machine for this algorithm since it is limited by its diameter being $\sqrt{N}$, but nevertheless is a good candidate.* A parallel machine design that fits our model well is the Boolean Vector Machine (BVM) being built at Duke University which is an SIMD machine that uses the cube connected cycles interconnection scheme (Ref. 13).

The Connection Machine, produced by Thinking Machines Corporation, is an SIMD architecture with 64K processors, each with 4K bits of memory, and complex routing hardware that supports arbitrary communication (Ref. 5). While a hardware router may be a preferred method for solving graph problems, many applications do not require this arbitrary communication facility and would profit by replacing the equivalent silicon area with more processors and using software for communication. By choosing a software alternative, problems that do not need generalized communication are more economical and graph oriented problems are still viable. Independent of the argument of whether one should or should not build an SIMD architecture with routing hardware, the fact remains that architectures such as the MPP are being built that do not have routing hardware, and routing software extends their usability.

THE METHOD

In this section we present the concept of *conflict-free space-time* labeling, henceforth referred to as CFST-labeling. Using this labeling scheme, we present a simple algorithm for data movement and an algorithm for generating CFST-labeling of a graph incrementally.

It is necessary to distinguish between the graph problem being attacked and the computer model being used. The graph being embedded will be referred to using standard graph terminology with regards to vertex, edge, and degree. The machine elements are called processors and wires. Each vertex will be assigned to a different processor. Each edge in the graph will be realized by a path in the physical network which is a list of consecutive wires joining adjacent processors. Each wire specification that is part of a path is referred to as a link.

Traversing all the edges of the embedded graph in parallel will take more than one step since messages cannot be sent instantaneously but rather must be passed along through successive neighbors. Traversing all edges in parallel, referred to as the delivery phase, will be considered an uninterruptible operation that takes T steps. In addition to the spatial characterization of a path, a path will also be characterized by a relative temporal offset within the delivery phase. Rules are provided governing paths. Algorithms are presented to create paths

*THIS WORK WAS SUPPORTED BY THE NATIONAL AERONAUTICS AND SPACE ADMINISTRATION UNDER NASA CONTRACT NO. NAS1-18107 WHILE THE AUTHOR WAS IN RESIDENCE AT ICASE.

*See (Ref. 3) for an implementation of sorting on the MPP.

incrementally and to transfer a collection of messages through paths simultaneously.

We will begin by defining the data structures which will be resident at each processor.

```
ALLOCATED ---- boolean flag indicates that processor
               is assigned a graph vertex
VERTEX_LABEL --- label of graph vertex
HAS_NEIGHBOR[1..neighbor_limit] --- flag indicates
                                    existence of wires
SLOTS[1..T] OF      edge path information
     START----------new edge starts here
     DIRECTION------direction to send
                   {1..neighbor_limit,FREE}
     END------------edge ends here
     ARC LABEL------label of edge
```

The ALLOCATED and VERTEX_LABEL fields indicate that the processor has been assigned a vertex in the graph. The HAS_NEIGHBOR field is used to indicate whether a physical wire exists in the particular direction (e.g. in a flat grid, edge processors only have 3 neighbors, and corner processors 2, while internal processors have 4); for a completely regular topology it is superfluous. The SLOTS data structure is the key to the routing system. It is used in the delivery algorithm to instruct the processor where to send a message and in the labeling algorithm to insure that paths are constructed so that no collisions will occur. The SLOTS array is used to tell the processors what they should do on each relative time position within the delivery phase.

One of the characteristics of this algorithm is that a fixed path is chosen to connect two processors and once chosen it is never changed. For example, consider the grid in Figure 1.

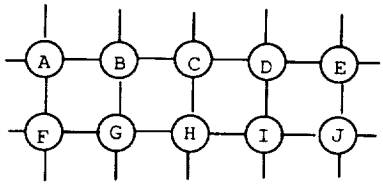

Figure 1.

If there is an edge between vertices in A and H, there are several possible paths: East-East-South, East-South-East, and South-East-East. Only one of these paths will be chosen between A and H, and that same path will always be used. For each edge, the corresponding path is not only fixed in space (i.e. the set of wires is constant), but is also fixed in time (the initial delay before the message starts down the path is constant). Once the starting time for the path has been fixed, it is never changed. Paths do not have to start on time 1, but can be scheduled to start at some relative offset within the delivery phase. Since there are no facilities for buffering, a message must proceed continuously along the specified directions without delay. For instance, if the path is of length 3 and it starts at time 1, then it will arrive at time 4; if it starts at time 2, it will be guaranteed to arrive at time 5. Further, it is necessary to place the paths so that no collisions occur; that is, no two paths can be at the same processor at the same instant in time. The rules for paths that fulfill these requirements are listed below.

- At most one link can enter a processor at a given time, and at most one link can leave a processor at a given time. It is possible to have both one coming and one going at the same time. Note that this does not mean that a processor can have only one link; it means that it can have only one link during a particular step in the delivery phase. It can have as many as T links going through it (since a delivery phase is length T by definition).
- Any path between two processors (u,v) representing an edge must consist of links at consecutive time steps. For example, if the path from processor u to processor v is {u,f,g,h,v}, then if the link from u-f is assigned time 1, f-g must use time 2, g-h time 3, and h-v time 4. Likewise if u-f occurs at time 5, then link h-v will occur at time 8.

When these rules are used to form paths, the SLOTS structure can be used to mark the paths. Each path goes through neighboring processors at successive time steps. For each of these time steps the DIRECTION field of the SLOTS structure is marked, telling the processor which direction it should pass a message if it receives it on that time slot. SLOTS serves both to instruct the processors how to send messages and to indicate that a processor is busy at a certain time slot so that when new paths are constructed it can be guaranteed that they won't conflict with current paths.

Consider the following example. Suppose we are given the directed graph with vertices A,B,C,D and edges $A \rightarrow C, B \rightarrow C, B \rightarrow D$, and $D \rightarrow A$ (Figure 2), and that vertices A,B,C, and D have been assigned to successive processors in a linear array. (A linear array is not a good network for this scheme but convenient for demonstration.) Initially all slots are free. We proceed to construct a CFST-labeling, placing each edge in the order it appears in the list above.

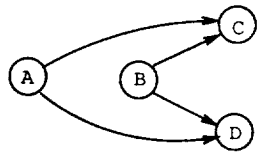

Figure 2.

```
A,B,C,D are successive members in a linear array

        1---2---3---4
        A---B---C---D
```

1. $A \rightarrow C$ can be completed with the map East-East, so Slots[A][1] = E, Slots[B][2]=E, End[C][2]=TRUE.
2. $B \rightarrow C$ can be done with the map East; it can start at time 1, since Slots[B][1] and End[C][1] are free.
3. $B \rightarrow D$ goes through C then to D; its map is East-East. B is occupied at time 1 and 2. It is free at time 3, so Slots[B][3]=E, Slots[C][4]=E, End[D][4]= TRUE.
4. $D \rightarrow A$ must go through C,B,A. using map West-West-West. D is free on time 1, and C is free on time 2, but B is occupied on time 3. The path can start from D on time 2. Slots[D][2]=W, Slots[C][3]= W, Slots[B][4]= W, End[A][4]=TRUE.

Every processor acts as a conduit for its neighbors' messages. No processor knows any message's source or destination, but each processor knows what it must do to establish the local connections.

Given that the paths satisfy the CFST-labeling rules, message delivery for graph problems is simple. The paths have been constructed so that there will be no collisions, and each path link uses consecutive time slots. The end of a path is specified by setting a separate bit that is tested after each message is received. A separate start bit in SLOT[k] indicates that a path starts at time k. The start bit is needed because the SLOTS array just tells the processors where to send a message, regardless of how that message arrived **. The start array indicates when a message originates, as opposed to arriving from a neighbor.

The following algorithm is basic to the routing system.

**Both the START and the STOP bits can be encoded as part of the DIRECTION field in SLOTS, but the presented method is simpler to explain and allows for more efficient execution.

```
PROCEDURE DELIVER

for i = time 1 to T
   FORALL processors
 /* if an edge starts or passes through at this time */
      if SLOT[i].START = 1 or active = 1
         for j=1 to neighbor-limit
         begin
            if SLOT[i].direction= j
               write message bit to
                   in-box of neighbor j;
               set active = 0;
         end
         FORALL processor that just received a message
         if end[i]
             move in-box to message-destination;
         else
            move in-box to out-box;
            set active bit = 1;
```

This code follows the method described above. The time slots are looped through, and the messages are passed in the appropriate directions as specified in the SLOTS array. Two bits, in-box and out-box, are used for message buffering.

The time complexity of data movement is $O(T \times neighbor\_limit)$. Since the number of neighbors is assumed to be a small constant for each network, the complexity is $O(T)$. This suggests that networks with fewer neighbors have advantages. For instance, the hypercube network has $logN$ neighbors, and the cube connected cycles network has 3 neighbors. Empirical results (Ref. 17) shows that while hypercube uses a smaller T, CCC actually has a faster overall delivery time.

Setting up Message Paths

One of the goals in developing this system was to have a method for adding new edges quickly. Paths are added so that they don't conflict with any old path. Once a path is placed it will not be re-routed by the basic placement algorithm; it will always start at the same spot at the same time. The basic idea of the method for placing a connection is to start from the source processor and in parallel examine all possible paths outward from it that do not conflict with pre-established paths. As the trial paths are flooding the system, they are recorded in temporary storage. At the end of this deluge of trial paths, if the destination processor has been reached, then a real path exists. Using the stored information a path can be backtraced and recorded. This is similar to the Lee-Moore routing algorithm (Refs. 6, 8) for finding a path in a system.

Suppose that the connection (u,v) is to be added. First it is assumed that processors for u and v have already been determined, otherwise (for now) assume a random allocation from a pool of free processors. It is necessary to find a path between u and v that does not conflict with any of the existing paths. The method for doing this is a type of flooding. A breadth-first search will be performed in parallel starting at the source processor. A record is kept of the trial paths resulting from this search. The paths must adhere to the CFST labeling rules, so a trial path must not conflict with paths that are already established. For instance, suppose a trial path starts at time 1 and moves to a neighboring processor, but that neighbor is already busy at time 1 (as can be seen by examining the DIRECTION-SLOT.) Since a path that would go through this neighbor at this time is not legal, the trial path would commit suicide, that is, it stops propagating itself. If the processor slot for time 2 was free, the trial path would attempt to propagate itself to that processor's neighbors at time 3.

Trial paths are recorded in a structure called TRIALSLOTS. A trial path knows if the next time slot is occupied by referring to the SLOTS data structure. If the destination processor is reached by a path, it will be a path that does not violate the rules. Therefore we can trace backwards from the destination processor using the markings in TRIALSLOTS and transfer this good path to the actual SLOTS structure.

PERFORMANCE

Adding an edge (assuming one can be added), deleting any set of edges, or traversing all the edges in parallel, all have time complexity $O(T \times neighbor\_limit)$. If it is assumed that $neighbor\_limit$ is a small constant then the complexity is $O(T)$. Since T is related both to the time and space needed, it is a crucial factor in determining the value of the algorithms presented. Some analytic bounds on T were presented in (Ref. 16), but it is difficult to get a tight bound on T for general interconnection networks and dynamically changing graphs. For the case where the graph is known *a priori* an upper bound of $O(log^2N)$ can be achieved on a hypercube. This is obtained by applying a result by Nassimi and Sahni (Ref. 10) in which they present a method for data broadcasting in SIMD computers which conforms to the CFST-labeling rules.

Of major interest is the on-line case, where edges are added and deleted dynamically. A simulator was constructed to examine the behavior of the algorithms. Besides the simulated data, the algorithms mentioned were actually implemented for the Connection Machine. The data presented by the simulator is consistent with that produced by the real machine. The major result is that the size of T appears proportional to the average degree of the graph times the average path length in the embedding.

This is a highly significant result. If it is assumed that the average number of connections and the neighbor-limit are bounded by small constants then the time for a parallel traversal operation, is, the worst case, $O(diameter)$. This indicates that the algorithm performs optimally for routing random communication graphs, since a random graph can have connections between processors that are diametrically opposed. If it assumed that the diameter is $O(logN)$ then C_T is $O(logN)$.

This bound indicates that the methods presented here are competitive with existing methods for parallel traversal in SIMD architectures. Some methods for SIMD parallel communication were mentioned in the introduction. Permutations can be done in $O(logN)$ time (Refs. 9, 10). Sorting can be done in essentially $O(log^2N)$ time, using (Ref. 12) or Batcher (Ref. 1) combined with Thompson (Ref. 15). These methods are restricted to permutations and sorting. One of the advantages of the method presented here is the ability to deal with graphs that are more general. Using the previous methods, if the connections specified a complete permutation, and additionally some processor also wanted to connect to two other processors, three entire permutations must be performed because the complexity is $diameter \times max\_number\_of\_connections$. In our system some items can have more connections than others without substantially increasing T. This result is achieved because the complexity of this method is based on the $diameter \times AVERAGE\_number\_of\_connections$, rather than the maximum number. The method used here also has the advantage that new connections can be added easily, unlike the other methods which require the entire set-up to be re-computed.

A further advantage of this method is the ability to exploit locality. Since the heuristic for T is dependent on the *average path length*, situations where the embedding can be arranged so that processors connect to those in some neighborhood will produce smaller values of T.

BRANCHING PATHS

There are many variations of this basic method that can be exploited. Among these are heuristics for picking shortest paths, assignment of node to processor, and choosing paths so as to avoid congestion. A discussion of these methods can be found in (Ref. 16).

A significant and important extension of the general CFST labeling rules involves a generalization that we refer to as *branching paths*. Branching paths is actually a form of broadcasting. Rather than having each arc represented by a different path, arcs starting from the same vertex can be combined for greater efficiency. However, this gain comes at the loss of two features.

The first is an obscure feature. In the basic CFST method presented, the system is identically free of conflicts whether it is run forwards or backwards. That is, rather than have an algorithm that starts at step 1 in the delivery cycle and initiates messages with the START bit marked, it is possible to write a *send backwards* algorithm which, using the same SLOTS structure, will start at time T initiating those links that *STOP* at the processors, and work backwards to time 1, reversing all the links, until the values get to the source. If one wished to implement a bi-directional graph rather than a directed graph, instead of actually putting in two sets of wires (from each pair $u \rightarrow v$ and then $v \rightarrow u$), one could just put in one set of wires, pass the values in the forward directions, then reverse it and pass the values back again.

The second and greater issue involves the nature of the messages that are passed along the arcs. If the application requires that a different message be passed along each arc, then it will not be possible to use this combining method. However, if all connecting processors are passed the same value, or the same value modified by a different constant, then this variation is viable. Many applications, including circuit simulation and neural networks fall into this class.

In the standard method, each arc in the original graph becomes a path in the embedding.

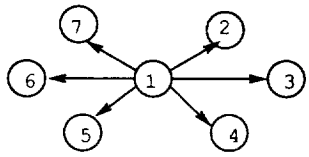

Figure 3.

For example, Figure 3 shows a graph in which one vertex is connected to all others. Mapped onto a linear arry, each arc becomes a path (Figure 4).

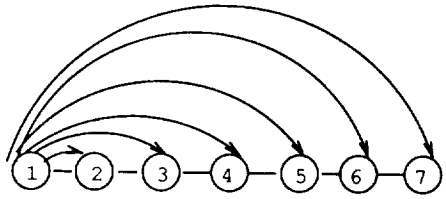

Figure 4.

The branching path method adheres to the CFST labeling rules, but alters the premise that each arc in the original graph is represented by a separate path. Instead, the set of arcs associated with each vertex can be represented by a set of branching paths. A branching path has two characteristics that differ from a standard path between two nodes. The first is that a path can "drop off" values at intermediate nodes in the path. For example, Figure 5 shows that a single path can be used to deliver messages to all intermediate vertices. Essentially, processors along the same route can share the path.

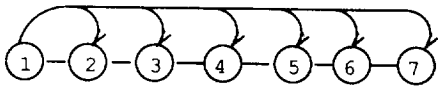

Figure 5.

In addition to path sharing, the other ability is to allow paths to branch out, or broadcast to their neighbors. For example, Figure 6 shows a grid in which processor A is connected to B and C. This is represented with one branching path.

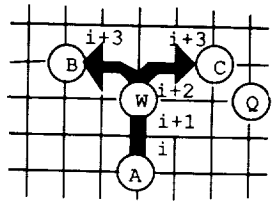

Figure 6.

More formally, as before, each vertex in the original graph is mapped to a processor in the network topology. The set of arcs associated with each vertex will be represented as a set of directed acyclic graphs (DAG). For each arc (u,v), there must be a DAG, D, whose root is u which has v as either a leaf node, or an internal node. For any leaf node w in the graph D with root u, (u,w) must be a an arc in the original graph. Further, in the family of DAGs which represent the arcs originating from the root u, each arc in the original graph can be represented only once. That is, if a graph has root u and leaf w, then no other graph with root u can have leaf w. Likewise, any internal node that is a destination must be marked to indicate that it is not just an intermediate node in the route, but rather a drop-off point. If a graph with root u has a specially marked internal node v representing arc (u,v), then no other graph with root u can have v has a specially marked node, although v can appear without a marking. As in the original method, each arc is only represented once, although paths can go through other nodes that are not part of their message destination. For example, in figure 6, A connects to B and C. Suppose that A were also connected to W. In this case W would have to be marked to indicate it not just an intermediate node in the graph. Further, if A were to connect to Q, this could be represented by a separate graph as a path going through W. But, W could only be a destination vertex in one of the two graphs.

Another aspect of this definition involves the nature of the messages. If all vertices that u connects to receive the same value message then the definition mentioned above is fine. If all receive the same value message multiplied by a different constant, it is also fine, because each constant can be stored at the destination node, so all connecting arcs will be passed the same value, and multiplication will occur at the time of delivery. However, if connecting values are truly different, then no two arcs (u,v), (u,w) that have different arc weights can be represented by the same DAG. Hence, in the original case, where we assumed that each arc receives a different dynamic value, each arc would have to be represented by a separate DAG, which would be a DAG with only one leaf, which is our definition of a path. So the original method is simply a subset of this method.

Having defined this family of graphs that represent the arcs in the new embedding, we proceed to label them according to an extension to the CFST labeling rules. The first rule, non-exclusion, originally said that at most one connection can enter a processor at a given time and at most one can leave. Now we allow more than one to leave provided that the connection is defined as part of the same DAG. The second rule, of contiguous time, still holds. When a path splits into two or more branches, each of the connections proceeds at the same time, as illustrated previously in Figure 6.

A small clarification: strictly speaking, the branching paths do not have to be DAGS, that is, they don't have to be acyclic. That is, when a connection reaches a processor, it does so at some time i, and it is possible for the path to loop back on itself, which will occur at some later time j. While usually such loops are inefficient, sometimes they are used to avoid jams in the system, and further, they can occur naturally as part of the algorithms which finds paths. In terms of the static algorithm, a way to avoid the conceptual messiness of cycles is

Algorithm Updates

The algorithms change surprisingly little. The algorithm for delivering the messages doesn't change at all. The loop which checks for the marked direction and passes the value in the appropriate direction may

simply see, for instance, that the North bit is on, and so passes it north, and then checks that the West bit is on, and passes likewise within the same time step. Since, because of the SIMD nature of the machine, it is already necessary to loop through the different directions, no extra work is done.

The algorithm for finding a new path requires a small modification. In the original method, paths propagate from the source through neighboring processors which are not already busy. To encorporate the branching path method, during the spreading method all connections which start from the source are activated, as they would be during the normal delivery cycle, and these active processors are included as part of the path spreading. That is, if a processor would normally be active at time i if it were sending a message from the source as part of its normal delivery cycle, then on time $i+1$ it will attempt to propagate a new path to its neighbors. If the shortest path heuristic is used (Ref. 17), so that each trial path has a length associated with it, then a new path that is formed as a branch or continuation off an old path starts at length 0 from that branch point, rather than starting at length 0 from the source. For problems that do not require unique values to be passed and that have multiple connections per vertex, these minor changes allow for a much more efficient message delivery system.

Analysis

The basic comments on analysis are the same as for the original case. A message delivery cycle still takes $O(T)$, where T is the number of slots, but by using the branching paths, T can be much smaller. As an example, we take the case of a fully connected graph embedded in a linear array. We know that the lower bound on T is the cutwidth of the resulting embedding (Ref. 17). When embedding the fully connected graph in a line using the original method, the maximum cutwidth is $O(N^2)$, and in the branching path method it is $O(N)$. Hence, in this case the savings between the two methods is substantial. While this is an unusual example, it gives an idea of the advantages. Some empirical results on the on-line random graph examples, of the type done in (Ref. 17), tend to show an improvement of a factor of 2 to 3 over the original method. These results are preliminary and haven't been statistically validated.

CONCLUSION

Some simple algorithms have been presented which allow arbitrary graphs to be embedded in SIMD architectures having a variety of topologies. The time for performing a parallel traversal and for adding a new connection appears to be proportional to the average path length in the embedding, times the average number of arcs in the graph being embedded. Since the average path length is no more than the diameter of the network, the method is competitive with existing methods for SIMD routing, with significant advantages for graphs that can exploit locality. Additional advantages are that there are no *a priori* requirements for the form of the data, the topological requirements are extremely general, and new arcs can be added without reconfiguring the entire system. The simplicity of the implementation and the flexibility of the method suggest that it could be an important tool for using SIMD architectures as graph processing machines.

REFERENCES

1. K. Batcher, "Sorting Networks and their Applications," *Proceedings of AFIPS 1968 SJCC*, pp. 307-314.
2. K. Batcher, "Design of a Massively Parallel Processor," *IEEE Trans on Computers*, Sept 1980, pp. 836-840.
3. J.E. Dorband, "Sort Computation and Conservative Image Registration", Ph.D. Thesis, Pennsylvania State University, Dec. 1985.
4. T. Feng, "A Survey of Interconnection Networks," *Computer*, Dec 1981, pp.12-27.
5. W. Hillis, "The Connection Machine," MIT Press, Cambridge, Mass., 1985.
6. C. Lee, "An algorithm for path connections and its applications," *IRE Trans. Elec. Comput.*, Vol. EC-10, Sept. 1961, pp. 346-365.
7. T. Leighton, "Parallel Computation Using Meshes of Trees," *Proc. International Workshop on Graph Theory Concepts in Computer Science*, 1983.
8. E. Moore, "Shortest path through a maze," *Annals of Computation Laboratory*, Vol. 30, Harvard Univ. Press, 1959, pp. 285-292.
9. D. Nassimi and S. Sahni, "Parallel Algorithms to Set-up the Benes Permutation Network," *Proc. Workshop on Interconnection Networks for Parallel and Distributed Processing*, April 1980.
10. D. Nassimi and S. Sahni, "Benes Network and Parallel Permutation Algorithms," *IEEE Transactions on Computers*, Vol. C-30, No. 5, May 1981, pp. 332-340.
11. D. Nassimi and S. Sahni, "Data Broadcasting in SIMD Computers,"*IEEE Transactions on Computers*, Vol. C-30, No. 2, Feb 1981, pp. 101-106.
12. D. Nassimi and S. Sahni, "Parallel Permutation and Sorting Algorithms and a New Generalized Connection Network," *JACM*, Vol. 29, No. 3, July 1982, pp. 642-667.
13. F. Preparata and J. Vuillemin, "The Cube Connected Cycles: a Versatile Network for Parallel Computation," *Comm. ACM*, Vol. 24, No. 5, May 1981, pp. 300-309.
14. H. Stone, "Parallel processing with the perfect shuffle," *IEEE Trans. Computers*, Vol. C-20, No. 2, Feb. 1971, pp. 153-161.
15. C. Thompson, "Generalized Connection Networks for Parallel Processor Intercommunication," *IEEE Tran. Computers* Vol. C-27, Dec. 1978, pp. 1119-1125.
16. S. Tomboulian, "A System for Routing Arbitrary Communication Graphs on SIMD Architectures," Doctoral Dissertation, 1986,Dept. of Computer Science, Duke University, Durham, NC.
17. S. Tomboulian, "A System for Routing Directed Graphs on SIMD Architectures", ICASE Report No. 87-14, NASA Langley Research Center, Hampton, VA (updated 1988).
18. R. Wagner, "The Boolean Vector Machine," *IEEE 1983 Conference Proceedings of the 10th Annual International Symposium on Computer Architecture*, pp. 59-66.