NASA Contractor Report 178265

ICASE REPORT NO. 87-14

ICASE

A SYSTEM FOR ROUTING ARBITRARY DIRECTED GRAPHS ON SIMD ARCHITECTURES

Sherryl Tomboulian

Contract No. NAS1-18107

March 1987

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

A System for Routing Arbitrary Directed Graphs on SIMD Architectures

Sherryl Tomboulian

ABSTRACT

There are many problems which can be described in terms of directed graphs that contain a large number of vertices where simple computations occur using data from connecting vertices. A method is given for parallelizing such problems on an SIMD machine model that is bit-serial and uses only nearest neighbor connections for communication. Each vertex of the graph will be assigned to a processor in the machine. Algorithms are given that will be used to implement movement of data along the arcs of the graph. This architecture and algoriths define a system that is relatively simple to build and can do graph processing. All arcs can be traversed in parallel in time $O(T)$, where T is empirically proportional to the diameter of the interconnection network times the average degree of the graph. Modifying or adding a new arc takes the same time as parallel traversal.

This work was supported under the National Aeronautics and Space Administration under NASA Contract No. NAS1-18107 while the author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23665.

1. Introduction

A class of problems can be described as directed graphs, where simple computations take place at the vertices based on the values of the neighboring vertices; examples include semantic networks, circuit design, and topographical problems. While the computation at each vertex is frequently trivial, these problems are often time consuming because the number of vertices in the problem is on the order of tens or hundreds of thousands.

One natural way to apply parallelism to this type of problem is to assign exactly one processor to each vertex in the graph and to have some mechanism for realizing the arcs. This small grain approach has a great deal of appeal. Since existing MIMD machines [7] have a few hundred or thousand processors, it is not be feasible to use them to parallelize large graph problems using one vertex per processor. SIMD architectures [7] support finer granularity, but often do not easily support generalized message passing schemes. This paper provides an introduction to a method of embedding graphs in a class of SIMD architectures. More complete details can be found in [18].

The method presented here assumes that the graphs have a sparse set of arcs. The average number of arcs that each vertex has is relatively small, bounded by some constant c where $c \ll$ *number of vertices*. Another requirement is that the graph be *semi-dynamic*, i.e., edges can be added or deleted; but the majority of graph edges remain fixed. This is the typical situation one encounters in an information acquisition and learning system, where new information is coming in and is being changed, but only a small portion of the existing information will be changed at any time. A third requirement is that the operations to be performed are fairly homogeneous. That is, in general the same (or nearly the same) computation is concurrently performed at all vertices.

Algorithms will be presented for arranging graphs in SIMD architectures that have a variety of network topologies. The basic idea is to create a mapping of the graph to the network in which message collision is impossible. The assignment of data to processors and the arrangement of connections are completely and automatically determined by the algorithm. Arcs can be added or modified easily.

1.1. Existing SIMD Routing Algorithms

Nassimi and Sahni have studied permutations in SIMD architectures. In [12], parallel algorithms for setting up a Benes permutation in $O(log^4N)$ for a hypercube or shuffle exchange network are presented. It has been shown [17] that many standard networks, such as the hypercube, can simulate a Benes permutation network in $O(log\ N)$ time, so once set up an SIMD machine using hypercube or shuffle exchange could then perform a permutation in $O(log\ N)$ time. In [14] they present an algorithm for permuting or sorting N data items on a hypercube or perfect shuffle network with $N^{1+1/k}$ processing elements, $(1 \leq k \leq \log N)$, in time $O(k \log N)$. Batcher presented a sorting network with $O(log^2N)$ delay and $O(N \log^2 N)$ switches in [1]. Using methods presented in [17], this hardware sorting scheme can be simulated on a hypercube in time $O(log^2N)$ for sorting N numbers.

The methods presented here are more general than the ones mentioned. Rather than just permutations or sorting, random graphs can be embedded. Another feature is that the algorithm works on a much wider variety of networks including those that are not completely regular. A third feature is that arcs can be changed without having to redefine the entire embedding. While the set-up time for graphs is longer in our system, the empirical time for parallel traversal (presented later) is competitive with the methods mentioned above.

2. Machine Model

The algorithms presented here apply to architectures having the following properties. Since the machine model used is SIMD, there is a single control unit and a large number of slave processors which can execute the same instruction stream simultaneously. It is possible to disable some processors so that only some execute the instruction stream, but it is not possible to have two processors performing different instructions at the same time. An *instruction* involves both a particular operation code **and** the local memory address. It is assumed that a processor cannot perform local indirect address operations. This restriction makes implementing certain structures, such as queues, difficult within individual processors. However, it allows the memory of all processors to be addressed (conceptually) by a single address driver, reducing the silicon area needed for these necessary parts of memory.

The processors have their own local memory. Neither indirect addressing nor global memory is assumed to be available. The memory is relatively small -- on the order of a few hundred or a few thousand bits. Information transfer between processors must be done through the neighbor connections (see below). Each processor has a unique *id number* stored in its local memory.

There exists some channel for getting information from the processors back to the control unit. The minimum requirement is that one processor acts as a serial port between the processors and the control unit. If one processor has a link to the controller, then with a network as described below it is possible to get data from any processor to the controller in time proportional to the *diameter* using software. Additionally, processors must be able to read from or write to their neighbors' memories (either facility is sufficient since one can simulate the other).

The processors are connected via a network. The following network requirements are sufficient to solve graph problems, given the algorithms presented below, and allow the construction of relatively cheap networks. (Note that these are the requirements for the topology of the physical network connections such as a grid or a hypercube, not the graph being embedded.)

(1) The connections between processors have labels (positive integers) associated with them that must satisfy certain requirements.

- The links must have an inverse (be full-duplex).
- The labels of wires entering and leaving a processor must be unique. (For example, in a grid, a processor has four connections conventionally labeled north,south,east,west; it would be unadmissible to have two wires labeled "north".)
- The name of a wire label and its associated inverse must be consistent for all processors. That is, if one processor has a wire labeled *i*, whose inverse is called *j*, then all processors that have an *i* connection must have the inverse called *j*. For instance in a grid, the inverse of 'north' is always "south". This does not mean that the same relative positions must have the same labels.

(2) The number of different neighbor labels, *neighbor–limit*, must be a small constant. For example, each node in a grid has four neighbors (north, south, east, and west).

(3) There must exist a path between every pair of processors, and the network diameter is assumed to be *small*, ideally $O(logN)$ where N is the number of processors.

Many networks [6] including grid, hypercube, cube connected cycles [15], shuffle exchange [16], and mesh of trees [10] are admissible under this scheme.

The Massively Parallel Processor (MPP) built by Goodyear Aerospace [2] is an SIMD architecture that is fairly well suited to the algorithm presented here. The MPP has single bit processors arranged in a 64 by 64 processor grid. The MPP is not the *perfect* machine for this algorithm since it is fairly small (speaking in a massively parallel sense) and since its interconnection scheme is a grid (of diameter $\sqrt{N}$). Nevertheless, the MPP is a good candidate. A parallel machine design that fits our model well is the Boolean Vector Machine (BVM) being built at Duke University [19]. It is an SIMD machine that uses the cube connected cycles interconnection scheme [15]. A design for an SIMD architecture that is suitable for semantic networks was presented by Scott Fahlman [3,4,5]. The architecture he proposed differs from our model in that it has a separate network for communication. The routing algorithm he uses, called hashnets [Fahlman 80b], has some similarities with the one presented here.

The Connection Machine [8], produced by Thinking Machines Corporation, is an SIMD architecture with 64K processors, each with 4K bits of memory. Communication is done by complex routing hardware. For many problems, the flexibility provided by such routing hardware is not justified since such hardware is expensive. This paper shows how with a simpler hardware arrangement, connections can be formed solely by software.

3. Algorithm

When discussing this algorithm, there is a high potential for ambiguity when referring to the physical arrangement of the processors in the computer architecture and the vertices of the graph being embedded. The graph being embedded will be referred to using standard graph terminology with regards to vertex, arc, and degree. Each vertex will be assigned to a different processor. Each arc in the graph will be realized by a path in the physical network which is a list of adjacent processors. The path consists of a series of links which specify the physical wire labels that connect the processors. In addition to the spatial characterization of a path, a temporal association will also be made. Rules are provided governing paths, and algorithms are presented to traverse all arcs simultaneously and to add a new arc.

Since the processors are connected to their neighbors in some predefined topology, a connection between two processors representing an arc in the graph can be formed by passing a message through neighboring processors until the message reaches the destination. There are two basic problems with this method. The first is that a processor must have some mechanism for determining which neighbor to pass a message to. The second is that there has to be some way to deal with collisions of messages. Collisions are particularly difficult to deal with in an SIMD environment without indirect addressing since the processors cannot keep individual stack pointers and reference the bottom of the stack indirectly; each position would have to be looped through serially. The same mechanism is used to solve both problems.

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

```
ALLOCATED ---- boolean flag indicating
  whether this processor is assigned
  a vertex in the graph
VERTEX LABEL --- label of graph vertex
HAS_NEIGHBOR[1..neighbor_limit] flag
  indicating the existence of neighbors
SLOTS[1..T] OF      arc path information
  START----------new arc starts here
  DIRECTION------direction to send
                {1..neighbor_limit,FREE}
  END-----------arc ends here
  ARC LABEL-----label of arc
```

The ALLOCATED and VERTEX_LABEL field indicates 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; for a completely regular topology it is superfluous and its use will be explained later. The SLOTS data structure is the key to the routing system. It is used to instruct the processor where to send a message and to insure that paths are constructed so that no collisions will occur.

SLOTS is an array with T elements. This value T is called the time quantum. 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 arcs in parallel will be considered an uninterruptible operation which will take T steps. The SLOTS array is used to tell the processors what they should do on each relative time position within the time quantum.

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.

```
  |  |  |  |  |
--A--B--C--D--E--
  |  |  |  |  |
--F--G--H--I--J--
  |  |  |  |  |
```

Fig. 1. Grid Example

If there is an arc between 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. Besides being invariant in space, paths are also invariant in time. As stated above, traversal is done within a time quantum T. Paths do not have to start on time 1, but can be scheduled to start at some relative offset within the time quantum. Once the starting time for the path has been fixed, it is never changed. Another requirement is that a message cannot be buffered; it must proceed along the specified directions without interruption so that 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 constructing 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 time quantum. It can have as many as T links going through it (since a time quantum is length T by definition).

- Any path between two processors (u,v) representing an arc must consist of links at contiguous 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 have 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 arcs {*A*→*C* , *B*→*C* , *B*→*D* , and *D*→*A*} (Figure 2), and that vertices A,B,C, and D have been assigned to successive processors in a linear array. (A linear array in not a good network for this scheme but is a convenient source of examples.)

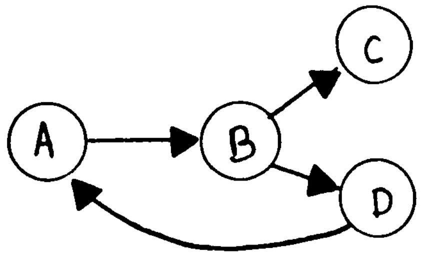

Fig. 2. Graph Example

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

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

First, A ->C can be completed with the map East-East, so
Slots[A][1] = E, Slots[B][2]=E, End[C][2]=1

B->C can be done with the map East; it can
start at time 1, since Slots[B][1] and End[C][1] are free.

B->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]=1.

D->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.
D is free on time 2, but C is occupied on time 3.
It can start from D at time 3,
Slots[D][3]=W, Slots[C][4]= W, Slots[B][5]= W, End[A][5]=1

Every processor acts as a conduit for its neighbors' messages. No processor knows where any message is going to or coming from, but each processor knows what it must do to establish the local connections.

The use of contiguous time slots is vital to the correct operation of the system. If all path-links are established according to the above rules, there is a simple method for making the connections. The paths have been restricted so that there will be no collisions, and paths' directions use 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 indicates when a path starts. 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.

```
for i = time 1 to T
      FORALL processors
            /* if an arc starts or is passing through at this time*/
            if SLOT[i].START = 1 or active = 1
                  for j=1 to neighbor-limit
                        if SLOT[i].direction= j
                           write message bit to in-box
                              of neighbor j;
            set active = 0;
            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 mentioned 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 $O(T \times neighbor\text{-}limit)$. The neighbor-limit is a critical factor in the time. The "machine model" section suggested that the number of neighbors be constant, in which case the complexity is $O(T)$. While the algorithm will function for networks with large neighbor-limit, the time for routing will increase proportionally.

3.1. Setting up Message Paths

One of the goals in developing this system was to have a method for adding new arcs quickly. The method used to construct new paths is essentially monotonic, with paths 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. (This does not refer to the ability to change arcs. The user can delete or modify an arc at any time, but the path for an arc is static.) Our algorithm adds arcs one at a time, not in parallel.

The basic idea of the method for placing a arc 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 [9] [11] for finding a path in a system.

Suppose that the arc (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 breath-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 *distributed path* 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 stops propagating itself. If the processor slot for time 2 is free, the trial path attempts to propagate itself to that processor's neighbors at time 3.

The HAS_NEIGHBOR bits are used for the new path algorithm. This field is checked to see if a physical neighbor connection exists before attempting to propagate a path in that direction. This allows irregular topologies and those with boundary conditions to be supported.

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.

The TRIALSLOTS structure is resident in each processor. Like SLOTS, TRIALSLOTS is an array with T elements. For each time slot there is one bit for each direction. The bit indicates whether a trial message reached that processor from that direction at that time. It is possible that more than one trial path can reach a processor coming from different directions on the same time-slot. Since each of these are possible paths, they can all be recorded.

The following algorithm provides the mechanism for finding new paths described above. Trial paths are constructed starting from the source processor. The formation of trial paths is attempted for each time step 1..T.

```
SEARCH FOR NEW PATH

Set Active Bit for Source processor;
Set Dest Bit for Destination processor;
for i = 1 to time T
begin
   FORALL processors with active bit set
       if SLOTS[i].direction is FREE
   begin
      for d = 1 to neighbor-limit
         if HAS_NEIGHBOR[d]
               mark direction d in TRIALSLOTS[i]for neighbor d;
               set next_active bit for neighbor d;
   end
   FORALL processors
   begin
     Move next-active to Active;
     Set source processor to Active;
     FORALL processors with Dest bit set {only 1 will be set}
        if the dest vertex is marked active
        begin
          if dest processor is not busy at that time
                mark the processor as reached;
          turn off its Active bit.
        end
   end
end
```

If the destination processor has been reached by the termination of the algorithm, then there exists a non-conflicting path. If it was not reached, there is no path possible without changing existing paths. If the destination processor has been reached, pick the time of the incident edge (if more than one, pick arbitrarily), then follow the marked edges backward until the source is reached, making the time-direction arrangements permanent by placing them in the SLOTS structure.

When the path searching algorithm is completed, all possible paths that emanate from the source processor have been tried and have been recorded if they were valid. The fact that *all* of the paths have been traversed is an important consideration. It means that if the destination is not reached, then it is not possible to reach it given the current configuration of the system using the the time quantum T.

Here is an example for spreading out for searching a linear array:

A----B----C----D----E----F----G

Suppose that there is already a path from B to E starting at time 1, and we want to add a path from C to A. Suppose the time quantum is restricted to be 4. (The time quantum could be set at any limit, and it is usually assumed that it is at least the diameter of the network. By limiting it to 4, it is not possible to make all connections since there are points that are more than 4 steps apart.)

Using time as the second dimension, figure 3 shows the line at each time step. The path from B to E starts at time 1, is at C at time 2, D at time 3, and arrives at E at time 4.

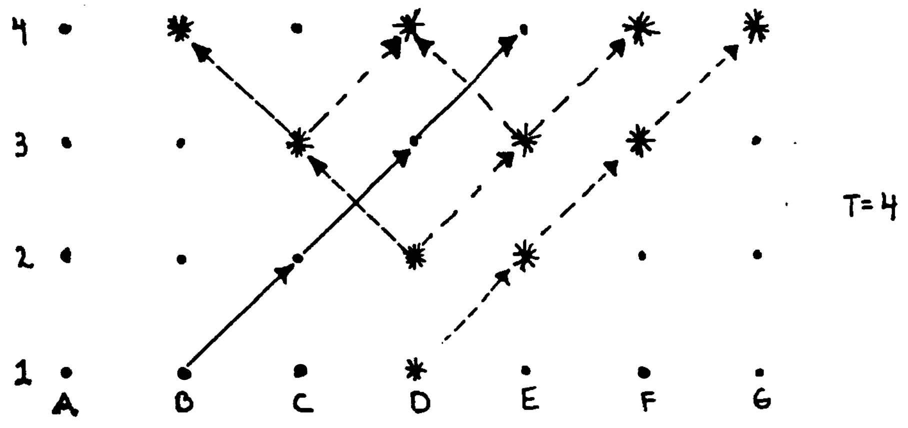

Fig. 3. Finding a Path in a Linear Array Configuration

We want to add a path from D to B. The *'s trace the spread of the trial paths from D. Note that processor C holds a message already at time 2, so D cannot propagate its message to C at time 1. However, D begins a new signal at time 2, which successfully reaches processor B by time 4. Note that, in this trivial system, the trial message heading right on the line cannot possibly reach A. However, in a more complex network, no simple rule is likely to allow some direction of search to be eliminated. Part of the beauty of the system is that the source processor does not "know" where the destination processor is located; it simply tries all possible paths outward, and when the spreading is finished the destination is checked to see if it has been reached.

3.1.1. Tracing Paths Backwards

The path finding algorithm just given requires that, in order to select and save a new path, the path must be traversed backwards. This is done by referencing TRIALSLOTS. Recall the way that

TRIALSLOTS was created. On time 1 if the source processor is not already occupied it will write to all its neighbors. Specifically, it will write to TRIALSLOTS for time 1, setting the bit that indicates the direction it came from. So, if the destination processor has its TRIALSLOTS marked for time t, it means that it was sent a message on time t. The destination processor should be set to save the message on time t, by marking END[t]= 1. By looking at TRIALSLOTS[t] the destination processor can tell which of its neighbors sent the message by taking the inverse of that direction. (Recall that the network requirements specified that each physical wire label have a unique inverse.)

If there is more than one valid link at a given time, the processor will select one of these. Since the processor knows that the message came from neighbor k, it will write the direction specified in its TRIALSLOTS[t] to SLOTS[t] of the neighbor in the inverse direction of TRIALSLOTS[t]. It will also set an activate bit in this neighbor. The process will then iterate, with each processor setting the appropriate direction in its neighbor's SLOTS array and then activating the neighbor and turning itself off. When the Source processor is reached, it marks the start of the message and terminates the process by turning its active_bit to 0.

4. Variations

The sketch of the algorithm presented in the previous sections leaves many loose ends. For instance, the processors that hold the vertices were chosen arbitrarily, and so were the paths connecting them. Many improvements can be made on this simplistic approach. A few are listed below.

4.1. Placements of the Data

Assignment of data to processors is an important problem. If an arc between vertices u and v is to be added, and both vertices have been mapped to processors, then there are no choices to be made. However, if only one is assigned or neither is assigned then there is some flexibility. The basic system will just pick free processors arbitrarily and connect them. This assignment of processors with no thought to connecting paths is an unnecessary constraint on the system.

If an arc is added between vertices u and v, it is only necessary that one of the processors be previously assigned. If u were assigned and v was not, the trial paths can be generated out from u, and all of the free processors that are reached are potential spots for v. Rather than trying to find a path between two specific processors, the system only fails if it can't find any path between the source processor and any free processor. If the destination processor was assigned but the source processor was not, it is still possible to use the same trick. To do this, a program which generates the paths backwards can be used. In this case the paths would start at the destination and spread out in the reverse direction starting at time T and going down to 1. If any of the reversed trial-paths reached a free processor, then u would be assigned to one of these and the new path marked in SLOTS.

4.2. Methods for picking paths

There are two major goals in picking paths. One is to minimize the length of the path, which corresponds to the number of slots used by that path. The second is to try to distribute the paths to avoid unnecessary congestion. Heuristics for picking paths are particularly important when combined with picking the position of the processors.

The basic method indicated that if more than one path reached the destination, the path be chosen randomly. This is not consistent with the desire to minimize path lengths. A system for choosing shortest path can be applied. This can be done by having the trial paths carry with them the length of their path so far. When two trial paths reach the same processor at the same time, the shorter one is chosen, and its length value is passed on as the path continues its quest. This will require carrying an extra log(T) bits, which implies needing a factor of log(T) more time.

If path lengths are provided, a variety of optimizations based on shortest path ideas exists. The most straightforward of these is to use shortest paths to help choose which processor to pick if there is more than one candidate destination.

4.3. Extensibility of T

When placing connections, it is desirable to have T small because it is both time and space. (T is proportional to the space needed to represent the arcs since the SLOTS structure has T elements.) However, T must be big enough so that the probability of making a connection is high. One way to deal with this problem is to have an expanding T. Initially, if the data base being used is small, T can be small. When a connection cannot be found, then T can be increased. Of course, this can only be done up to the limit of the number of slots that can fit in a processor's local memory.

4.4. Self-Loops

One way to introduce a primitive type of buffering is to have self-loops. When using self-loops, rather than having the number of directions each time be the number of neighbors, an extra direction called SELF can be added. This extra *direction* specifies that the value be sent to the sending processor on the next time step, i.e., it will simply remain at the processor. Since a self-loop adds to the length of the path without getting any closer to the destination, using the self-loop option would not be desirable in the general case. It is potentially useful in situations where the system is heavily congested, and a connection is trying to be formed between a and b, but there is no unblocked path; for example, suppose there is a path from a to x that arrives at x at time t, and there is a path from x to b that leaves x at time t+1. In this situation using a self-loop at x would complete the path. Additionally, self-loops are necessary in the situation where the graph to be embedded has self-looping arcs.

5. Error Recovery

An intrinsic feature of the routing algorithm is that the links are distributed over multiple processors. For a given path, no processor knows where the path comes from or goes to -- it only knows what is in its SLOTS array. Furthermore, if a processor were to fail, then there is the potential that all the messages that go through the processor could be lost. If a connecting wire were to fail, then the messages that go over that neighbor connection would be lost.

Clearly processor failure in such a system is of great concern. Fortunately it is possible to recover from many failures. When it is discovered that a processor is bad, or that a connection to a neighbor is bad, it is frequently possible to reconstruct the paths so that no data are lost. Correction is done by tracing the path to its source and destination, and then forming a new connection bypassing the faulty processors. The bad processor is marked accordingly, and the neighbor links are *disconnected* by setting the appropriate HAS_NEIGHBOR bits to 0 in the neighboring processors.

The easiest kind of processor failure to deal with is the situation where it is possible to read the direction-slots array. In this situation the bad processor is marked with a special bit; then starting at time step 1, the controller reads the memory to see which slots, if any, are occupied. Each path that goes through the bad processor must be reconstructed. To reconstruct a path, it must be traced backwards to its source and forward to its destination. Once the source and destination have been determined, the faulty path will be deleted and a new path with the same source and destination will be added. If it is possible to read the direction-slots array, it is easy to trace a path forward to the destination; just use the direction slots as in normal routing. To get back to the source, it is necessary to traverse the path backwards. A similar method was introduced in the path set-up algorithm. Since we know that the direction slot was busy at time i, the neighbors can be examined to see if they sent to the faulty processor at time i-1. This process is repeated until the source is reached. A bit can be set in the source processor and a different bit in the destination, and a path between the two can be added again using a variation of the edge connection routine which takes as input two bit vectors, one which is all 0s except for the source processor, and one all 0s except for the destination processor, instead of a source and destination label. Each path passing through a faulty processor has to be traced to its start and end points individually, and then re-added. Since there are up to T paths passing through any processor, the time of this operation is $O(T \times \textit{time–to–add})$, which is $O(T^2 \times \textit{neighbor–limit})$.

A slightly more difficult case is the situation in which it is impossible to read from the memory of the faulty processor. This can be handled if the neighboring processors to the faulty one are not themselves faulty. The method is fairly straightforward: first, at time t, the neighboring processors are examined to see if any write to the faulty one. If any does, then the neighbors are again examined to see if any of them receive a message at time t+2. It is possible to see if a processor received a message by seeing if it either has a stop bit set or if it is sending a message out that does not originate at that processor. If none of the neighbors are busy at time t+2, then the faulty processor must have been the destination of the message. If one or more of the neighbors are busy, then their neighbors are examined on time t, to see where the message came from. If none of the neighbors at time t+1 sent to the processor that was busy at time t+2, then the faulty processor must be the connector. Once all connections through the faulty processor are determined, tracing forwards and backwards can be done as above.

Neighbor connector failure can be repaired more easily than processor failure. If a link on a particular processor is found to be bad, the connection-slots array will be examined for paths which move in that direction. For each of those paths the destination processor will be found by tracing the path from that neighbor. The source can be found by tracing backwards. The HAS_NEIGHBOR bit is set to 0 in the two processors that are connected by the bad link. Once the endpoints have been found, the path can be re-entered. Since up to T paths may be affected, as many as T^2 operations will be needed.

6. Complexity and Empirical Bounds on T

Adding an edge (assuming one can be added), deleting any set of edges, or traversing all the edges in parallel, all have time complexity $O(T \times \textit{neighbor–limit})$. 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 [18], but it is difficult to get a tight bound on T for general interconnection networks and dynamically changing graphs. A simulator was constructed to examine the behavior of the algorithms. Besides the simulated data, the algorithms mentioned were 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 diameter of the interconnection network.

6.1. Simulation

Simulations were run on a serial machine. The largest machine size that was practical to simulate had 512 processors. In each experiment we specified a network, the type of graph to embed, methods for picking paths, limits on T, and so forth. For each experiment, 25 trials were run where each trial consisted of presenting a data graph, one edge at a time, to the simulator. The simulator was implemented for three interconnection networks: hypercube, cube connected cycles, and a toroidal grid.

Several types of graphs were generated for input to the simulator. Trees were the most extensively tested objects. The simulator had facilities for generating k-ary complete trees, or trees with a selected probability of having each child. X-trees were also examined. Random permutations (graphs of degree 1 formed by connecting vertices labeled 1..N with corresponding permuted values) were tested. Random graphs were also tested. To generate random graphs a limit on the number of out-edges of a vertex is specified. The processors are picked in a random order, and each processor has a random number of arcs between 0 and the limit specified each to a random vertex with uniform probability.

Heuristics can be used to determine the placement of vertices and paths. Rather than just picking vertices and paths arbitrarily, a combination of using shortest path and picking the destination processor based on this shortest path value was used.

The sizes for the hypercube simulation were 64, 128, 256, and 512 processors. For the grid the sizes were 64, 144, 256, and 559. Finally, for a cube connected cycles (ccc), the number of processors was 64, 160, 384, and 896. Thus the only size they all have in common is 64. The tables shown below (figures 4-9) display the experimental mean of the time quantum T for 25 trials of embedding a graph into different topologies. Each graph is embedded one edge at a time, and the number of slots needed to represent all the arcs in the path is recorded. Each trial using different random seeds for selection of placement and paths. The tables show the average T for these different placements. The student-t distribution was used to determine the 99% confidance intervals.

Complete binary trees were generated with the number of vertices in a tree equal to the number of processors, or as close as possible for a given sized network. For example, a tree of height 5 can be embedded in a hypercube of dimension 6. Results are given for trees presented depth first. (It may be noted that the size of T was ≤*diameter of the network*. Note that the diameter of the grid for the 4 sizes was 8,13,16,and 23; the ccc, 10,13,15, and 18; and the hypercube 6,7,8,and 9.)

Depth First Tree Time Quantum						
tree	grid		ccc		hyper	
height	N	T	N	T	N	T
5	64	5.6±.4	64	6.3±.4	64	4.3±.4
6	144	6.0±.4	160	6.6±.4	128	5.0±.4
7	256	9.0±.8	384	6.4±.5	256	5.2±.3
8	559	10.4±.7	896	7.1±.4	512	5.6±.4

Fig. 4. Time Quantum Results for Embedding Depth First Binary Trees

The same setup was used for X-trees. An X-tree has twice as many edges as a tree. While the size of T increased, it was less than twice the corresponding values for trees.

Depth First X-Tree Time Quantum						
tree	grid		ccc		hyper	
height	N	T	N	T	N	T
5	64	10.3±.4	64	12.8±.5	64	8.3±.6
6	144	13.7±.7	160	14.4±.6	128	9.5±.5
7	256	16.1±.8	384	16.6±.6	256	10.7±.6
8	559	21.8±.7	896	18.5±.7	512	11.6±.6

Fig. 5. Time Quantum Results for Embedding Depths First Binary X-Trees

The results for permutations were similar to the results for binary trees. Since there are N values being permuted, each processor has 1 arc. As above, the size of N listed in the table is the size for the hypercube. For all three networks, all of the processors were permuted. For instance, the second grid entry is actually of size 144, and so all 144 values were permuted. As with the trees, it may be noted that the size of T was less than or equal to the diameter of the topology. This is reasonable since there can be connections between two processors that are at the maximum distance from each other.

Permutation Time Quantum					
grid		ccc		hyper	
N	T	N	T	N	T
64	7.8±.3	64	9.7±.5	64	5.8±.5
144	12.1±.4	160	12.1±.4	128	6.8±.3
256	15.8±.4	384	15.4±.6	256	7.9±.4
559	23.0±.6	896	18.0±.4	512	9.0±.5

Fig. 6. Time Quantum Results for Embedding Permutations

The tests on random graphs were included to examine the performance of the algorithms on data that were not as regular as the previous cases. When examining the results for T on the previous data, it was conjectured that the size of T was proportional to the average number of arcs per processor times the diameter. For instance, trees and permutations have an average of one edge per processor and use about *diameter* slots. X-trees have twice as many edges and use less than 2 × *diameter*.

The experiments for random graphs were performed on all three networks using three different limits for the upper bound on the number of out-arcs per vertex in the directed random graph. The results corresponded almost exactly to the hypothesis. When the limit is 3, the average number of connections per processor is 2, and T is about 2 × *diameter* for all three networks. Likewise, when the limit is 5, the average number of connections is 3; T is about 3 × *diameter*. The consistency of this result is surprising, and suggest the existence of some underlying analytical principals.

The following three tables display the data for the experimental mean of T. Following each value, in parenthesis, is the diameter of the network times the average number of connections per vertex in the embedded graph . For the three tables this value is 2,3, and 4 times the diameter of the associated network. The relationship between the experimental value and the proposed bound can clearly be seen. As in previous experiments, confidence intervals are presented calculated using the student t distribution for a confidence of 99%.

Time Quantum, Random Graph, avg degree = 2								
grid			ccc			hyper		
N	T	est. T	N	T	est. T	N	T	est. T
64	15±.7	16	64	19±.8	20	64	11.4±.7	12
144	23.3±1.3	26	160	24.3±.9	24	128	13±.8	14
256	30±1.3	31	384	30.4±.8	30	256	14±.7	16

Fig. 7. Time Quantum Results for Embedding Graphs with avg degree = 2

Time Quantum, Random Graph, avg degree = 3								
grid			ccc			hyper		
N	T	est. T	N	T	est. T	N	T	est. T
64	22.3±0.7	24	64	27.4±0.9	30	64	16.4±0.9	18
144	33.4±1.5	39	160	35.1±1.2	36	128	18.4±0.9	21
256	43.1±1.7	48	384	44.7±1.3	45	256	20.8±1.0	24

Fig. 8. Time Quantum Results for Embedding Random Graphs with avg degree = 3

Time Quantum, Random Graph, avg degree = 4								
grid			ccc			hyper		
N	T	est. T	N	T	est. T	N	T	est. T
64	28.8±1.0	32	64	36.0±1.6	40	64	22.0±1.0	24
144	44.8±1.7	48	160	47.0±1.9	48	128	23.7±1.0	28
256	56.7±1.6	64	384	58.3±0.8	60	256	27.3±1.3	32

Fig. 9. Time Quantum Results for Embedding Random Graphs with avg degree = 4

Based on this data, we can assume with some degree of confidence that *T* is proportional to the average number of arcs per vertex times the diameter of the network. This is a highly significant result. If it is assumed that the average number of arcs and the neighbor-limit are bounded by small constants, then C_T, the time for a parallel traversal operation, is *O(diameter)*. If it assumed that the diameter is *O(log N)*, then C_T is *O(log N)*.

6.2. Connection Machine Implementation

The Connection Machine is an SIMD architecture with 64,000 bit serial processors. It differs from our machine model only in that it has extra hardware to perform the routing. The automatic routing provided can be ignored, and instead neighbor connections can be used for communication. The algorithms for connected information processing were implemented on the Connection Machine with the automatic routing disabled.

The implementation of the routing algorithms on the Connection Machine helped point out some of the practical details such as having an input-buffer and output-buffer at each processor so that a message that arrives at time t doesn't overwrite one that will leave at time t. It was also encouraging to note that only a few pages of code were needed to realize the appropriate routines and that the routines were written and debugged in a few days

One of the most interesting aspects of the implementation was dealing with the interconnection network. This was another area which demonstrated the flexibility of the routing algorithms presented here. As mentioned previously, the interconnection network is a hybrid between a grid and a hypercube. Each processor is configured as an element in a grid, so it has North, South, East, West neighbors, and every 12 out of 16 processors has a single hypercube wire. So, some processors have N,S,E,W, and H neighbors; some do not have the H. Additionally, the implementation here used the grid as a normal 2-d grid rather than a toroidal grid. Because of this some processors do not have all 4 of their NSEW neighbors. This slight irregularity in the network did not pose any problem for the algorithms. Each processor has 5 bits in their HAS_NEIGHBOR structure which were used to indicate the existence of N,S,E,W, and H connections. When the placement algorithm was executed, this bit was checked to insure that processors without a connection did not try to send or receive over a non-existent wire. The algorithm was implemented with the shortest path option.

Test data was collected running on a 16,000 processor segment of the Connection Machine. Complete binary trees and X-trees were tested ranging in height from 2 to 13. The results were surprisingly consistent. A complete binary tree with M vertex, *height=log(M)*, used about *log(M)* time slots; a tree of height 6 used 5 time slots; a tree of height 12 used 8 time slots. An X-tree with *height=log(M)* used about $2\times M+2$ time slots; an X-tree of height 6 used 16 slots; an X-tree of height 11 used 24 time slots. The results listed are exact numbers, not approximations. The consistency of the results for different sized trees was surprising. These results are consistent with the simulated data and with the observation that T is bounded by *diameter×average_number_connections*. Experimentation done on the Connection Machine was not as thorough or extensive as the simulator test due to time constraints. No data are available on the variance of the results.

7. Conclusion

Some simple algorithms have been presented which allow arbitrary graphs to be embedded in SIMD architectures with a variety of topologies. The time for performing a parallel traversal and for adding a new connection appears to be proportional to the diameter of the topology times the average number of arcs in the graph being embedded. In a system where the topology has diameter $O(logN)$, and where the degree of the graph being embedded is bounded by a constant, the time is assumed to be $O(logN)$. This makes it competitive with existing methods for SIMD routing with the advantages that there are no *apriori* requirements for the form of the data. Furthermore 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," *The Proceedings of AFIPS 1968 SJCC, 307-314*. pp. 307-314.

[2] K. Batcher, "Design of a massively parallel processor," *IEEE Trans on Computers*, Sept 1980, pp. 836-840.

[3] S. Fahlman, "NETL - A System for Representing and Using Real-World Knowledge," *MIT Press*, Cambridge, Massachusetts 1979.

[4] S. Fahlman, "Design sketch for a million-element NETL machine," *Proc. AAAI-80*, 1980.

[5] S. Fahlman, "The Hashnet Interconnection Scheme," Report # CMU-CS-80-125 Carnegie-Mellon University Dept. of Comp. Sc., 1980.

[6] T. Feng, "A Survey of Interconnection Networks," *Computer*, Dec 1981 pp.12-27.

[7] M. Flynn, "Some computer organizations and their effectiveness," *IEEE Trans Computers* Vol C-21, No 9, pp. 948-960.

[8] W. Hillis, "The Connection Machine," MIT Press, Cambridge, Mass., 1985.

[9] C. Lee. "An algorithm for path connections and its applications," *IRE Trans Elec Comput*, Vol. EC-10. Sept 1961, pp. 346-365.

[10] T. Leighton, "Parallel Computation Using Meshes of Trees," *Proc. International Workshop on Graph Theory Concepts in Computer Science*, 1983.

[11] E. Moore, "Shortest path through a maze," *Annals of Computation Laboratory*, Vol. 30, Cambridge, MA: Harvard Univ. Press, 1959, pp.285-292.

[12] 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.

[13] D. Nassimi and S. Sahni, "Benes Network and Parallel Permutation Algorithms," *IEEE Transactions on Computers* Vol C-30 No 5, May 1981.

[14] 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.

[15] 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.

[16] H. Stone, "Parallel processing with the perfect shuffle," *IEEE Trans. Computers* Vol C, No 20, Feb 1971, pp. 153-161.

[17] C. Thompson, "Generalized connection networks for parallel processor intercommunication," *IEEE Tran. Computers* Vol C, No 27 Dec. 78, pp. 1119-1125,

[18] S. Tomboulian, "A System for Routing Arbitrary Communication Graphs on SIMD Architectures," Doctoral Dissertation, 1986,Dept. of Computer Science, Duke University, Durham, NC.

[19] R. Wagner, "The Boolean Vector Machine," *IEEE 1983 Conference Proceedings of the 10th Annual International Symposium on Computer Architecture*

Standard Bibliographic Page

1. Report No. NASA CR-178265 ICASE Report No. 87-14	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle A SYSTEM FOR ROUTING ARBITRARY DIRECTED GRAPHS ON SIMD ARCHITECTURES		5. Report Date March 1987
		6. Performing Organization Code
7. Author(s) Sherryl Tomboulian		8. Performing Organization Report No. 87-14
9. Performing Organization Name and Address Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225		10. Work Unit No.
		11. Contract or Grant No. NAS1-18107
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, D.C. 20546		13. Type of Report and Period Covered Contractor Report
		14. Sponsoring Agency Code 505-90-21-01

15. Supplementary Notes

Langley Technical Monitor: J. C. South

Submitted to IEEE Trans. Comput.

Final Report

16. Abstract

There are many problems which can be described in terms of directed graphs that contain a large number of vertices where simple computations occur using data from connecting vertices. A method is given for parallelizing such problems on an SIMD machine model that is bit-serial and uses only nearest neighbor connections for communication. Each vertex of the graph will be assigned to a processor in the machine. Algorithms are given that will be used to implement movement of data along the arcs of the graph. This architecture and algorithms define a system that is relatively simple to build and can do graph processing. All arcs can be transversed in parallel in time O(T), where T is empirically proportional to the diameter of the interconnection network times the average degree of the graph. Modifying or adding a new arc takes the same time as parallel traversal.

17. Key Words (Suggested by Authors(s)) routing algorithm, SIMD architecture, parallel processing, graph embedding, interconnection network		18. Distribution Statement 61 - Computer Programming and Software Unclassified - unlimited	
19. Security Classif.(of this report) Unclassified	20. Security Classif.(of this page) Unclassified	21. No. of Pages 24	22. Price A02

For sale by the National Technical Information Service, Springfield, Virginia 22161

NASA Langley Form 63 (June 1985)