Routing for Generalized Chordal Rings

Bruce W. Arden and Kit-Ming W. Tang

Department of Electrical Engineering
University of Rochester, Rochester NY 14627.

ABSTRACT

A recursive routing algorithm is presented for generalized chordal ring (GCR) graphs. This algorithm consists of two parts. The first part deals with an one-time establishment of a database, and the second part determines a path of length less than or equal to 2^l where l is the smallest integer that such a path exists. Note that $l \leq d$ where d satisfies $2^{d-1} < diameter \leq 2^d$. The inherent symmetry and the modular arithmetic connectivity of the GCR are exploited to achieve a parallel time complexity of $O(\log_2 diameter)$ and a serial time complexity of $O(diameter)$.

1 INTRODUCTION

One of the most active areas in computer architecture is the design of efficient interconnection networks. A processor or computer interconnection network can be modeled as a graph. The vertices of the graph correspond to the communication or processing nodes and the edges represent connections between nodes. Due to the limited number of connections that can be made to real chips, regular graphs of small degree, having a small maximum message path length, or diameter, are of great interest.

Moore provided in 1958 an upper bound for the number of vertices in a regular graph with certain degree and diameter [1]. This Moore bound is unattainable except for the cases where diameter is 2, and degree is 3, 7 (and possibly 57) [2]. The search for (δ,D) graphs that connect the maximum number of nodes with a degree δ and diameter D continues. Chudnovsky et al. have constructed the best $(\delta = 4, D)$ graphs for a range of D, from a family of graphs, called Cayley graphs [3]. In a recent report we have shown that these Cayley graphs can be represented as generalized chordal rings (GCR) [4]. In this paper we present a recursive routing algorithm for GCR graphs. The recursive algorithm is divided into two parts. The first part deals with the establishment of a database which is then replicated at every node. We emphasize that this part of the algorithm is an one-time computation. Once the database is established, the actual path determination, or routing, is carried out by the second part of the algorithm. The time complexities of this second part are $O(\log_2 diameter)$ and $O(diameter)$ for the parallel and serial case respectively. This paper is organized as follows: In section 2, we state the formal definition of GCR and provide an example. Section 3 presents the routing algorithm and section 4 provides an example to illustrate the algorithm. Finally in section 5, we summarize our results.

2 GENERALIZED CHORDAL RING (GCR)

In this section we provide the formal definition of GCR and present a specific example.

Definition 1 *A graph* **R** *is a generalized chordal ring (GCR) if vertices of* **R** *can be labeled with integers mod n, the number of vertices, and there is a divisor q of n such that vertex i is connected to vertex j iff vertex $i+q$ is connected to vertex $j+q$.*

As an example, we consider a GCR with divisor $q = 4$ and $n = 24$. In this case the graph has a vertex set, $\mathbf{V} = \{0, 1, \ldots, 23\}$. The connectivity of the graph is defined as: for any $i \in \mathbf{V}$, if i mod 4 =:

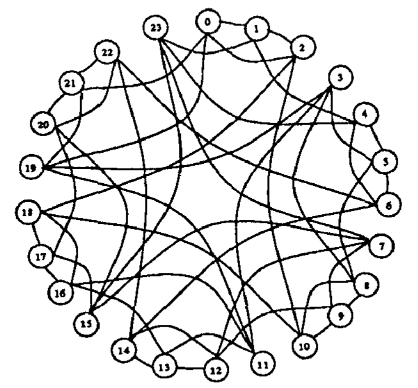

Figure 1: *A GCR with $q = 4$ and $n = 24$*

"0" : i is connected to $i+1$, $i+2$, $i-5$, $i-3$;
"1" : i is connected to $i+1$, $i-1$, $i+3$, $i-2$;
"2" : i is connected to $i-2$, $i-1$, $i+8$, $i-8$;
"3" : i is connected to $i+8$, $i-8$, $i+2$, $i+5$.

Such a GCR graph is illustrated in Figure 1.

3 RECURSIVE ROUTING ALGORITHM

In this section we develop a simple recursive routing algorithm with a parallel *time complexity* of O($\log_2$ *diameter*) which is capable of finding paths of any length $\leq 2^l$ where l is the smallest integer such that a path exists. Note that in the worst case the algorithm finds a path of length 2^d where d is an integer that bounds the diameter ($2^{(d-1)} <$ *diameter* $\leq 2^d$).

A GCR graph implies that nodes are labeled as $0, \ldots, n-1$. Our recursive routing algorithm exploits the inherent property of GCR: if i is connected to j, $i+mq$ is connected to $j+mq$ (modulo n) for any integer m. This property indicates that the connection pattern from any node can be represented by a single tree with a root node that is of the same class. There are q classes, where q is the chosen divisor of n. The connection pattern for a particular node can be obtained by an addition of some multiple of q to the address of every element in the corresponding tree. Furthermore, the mod n circular structure of the GCR means that nodes can be represented by an n-bit boolean-vector (or a compacted form of such a vector). Circular shifting this vector is tantamount to adding the same constant, the amount of shift, to each node address. In view of these observations, we divide the recursive routing algorithm into two parts. The first part involves the one-time generation of a database, which consists of d bit-vectors (represented in the algorithm as boolean-vectors) for every class. The identical database, dependent only on the chosen GCR structure, can be stored at every node for the purpose of path determination, or routing. Once the database is established, the actual routing between the source and destination pair is carried out by the second part of the algorithm.

We summarize this routing algorithm as follows:

3.1 Part 1: Database Generation

The objective of this part is to record, with repetition, the node labels at a distance $\leq 2^l$ ($l = 0, 1, \ldots, d-1$) from the root node of different classes. The results of this one-time computation are used later to determine the connection patterns for any node in the network. For each class of the GCR, a recursive algorithm "Half-Tree" is used to produce a *redundant* binary "half-tree", containing binary levels $2^0, 2^1, \ldots, 2^{d-1}$. During this process the nodes occurring at a distance $\leq 2^l$ from the root node are stored in the lth level in boolean vector form. That is, the mth bit of the lth boolean-vector indicates whether node labeled m first occurs at a distance $\leq 2^l$ from the root node. Notice that any node in the class can be made the root node of the corresponding half-tree. Subsequently we label the root node as "0" by subtracting the root node address from the address of every other node in the half-tree. For example, if node i_0 is the root node, we subtract i_0 from the address of all nodes in the binary half-tree. This representation allows us to rebuild the appropriate tree for an arbitrary node from the same class by a simple shifting of the boolean-vectors. In other words, if j_0 is a node of the same class as i_0, we can rebuild the half-tree with root node j_0 by adding j_0 to the address of every other node in the tree. Notice that i_0 and j_0 being in the same class implies that their difference is a multiple of q, the divisor of the GCR. The C-implementation of the recursive "Half-Tree" routine is shown in Table 1.

3.2 Part 2: Recursive Routing

This part of the algorithm is responsible for determining the path(s) between source and destination pairs. A recursive routine "Single-Path" is developed to find a path between a source and destination pair. We start the recursion at level $l = d-1$. From the database established in Part 1, we first check if the current source and destination (symmetrically viewed as two sources) are root and leaf in the same 2^l tree. If so, there is a path with length $\leq 2^l$. We continue the recursion by decreasing the level l by one. If not, that means the shortest path between the two nodes has length $> 2^l$ and $\leq 2^{l+1}$. We then check if

```
/********************************************************************* /
/* To generate redundant binary level half tree * /
/********************************************************************* /

Half_Tree(value,index,dist,Cons,root,xd,xn,xk,xdeg,S_Tree)
int         value,index,dist,*Cons,root,xd,xn,xk,xdeg,*S_Tree;
{int        level,i,class,name;

            if (dist <= Pow(2,xd-1))                    /* distance from root * /
            {   class = value%xk;
                if ((dist != 0) && (value != root))
                 {   level = Gnt(dist,2);               /* dist <= 2**level * /
                     name = Mod(value-root,xn);
                     i=level;
                     while (i<xd)                       /* assign to all levels > i* /
                     { *(S_Tree+i*xn+name) = TRUE; ++i; }
                 }                                      /* go through generators * /
                for (i=0; i<xdeg; ++i)
                   if ((i != Mod(index+(xdeg /2),xdeg)) || (dist==0))
                        Half_Tree(Mod(value+*(Cons+class*xdeg+i),xn),i,
                           dist+1,Cons,root,xd,xn,xk,xdeg,S_Tree);
            }
}
```

Table 1: *Half-Tree Recursive Routine*

a common node exists at level l of the source half-tree and at level $l-1$ of the destination half-tree, assuming $l > 0$. If such a common node is not found, we locate a common node at level l of both the source and destination half-trees. Such a common node always exists because the graph is connected and the diameter is bounded by $2^{d-1} < diameter \leq 2^d$. Once a common node is found, either at level $l-1$ or l of the destination half-tree, we split the problem into two, each with a level one less than the current value and the identified common node as the destination or source.

By passing along the source and destination to see if they are in the same half-tree, this algorithm finds a path of length $\leq 2^s$ where s bounds the shortest path, s-path ($2^{s-1} <$ s-path $\leq 2^s$). For instance if there are two paths, one has length 3 and another has length 4, the algorithm finds the one with length 3. But if the shortest path has length 4 and there is another path with length 5, the algorithm may find the one with length 5. The fact that two levels are checked for a common node increases the probability of finding the shortest path within the range 2^{s-1} and 2^s without increasing the complexity of the algorithm. As a matter of fact, this strategy allows us to find a path of length $\leq 1.5\ (2^{s-1})$, if it exists. When the diameter of the graph is large, more levels of comparison can be considered. The routine "Single-Path" is included in Table 2. A simple augmentation of this routine will produce all paths of length $\leq 2^s$, $2^{s-1} \leq$ s-path $\leq 2^s$.

Regarding the complexity of this part of the algorithm, the time for "shifting" and the "AND" operation are, in principle, constant with respect to n. Thus with parallelism, the algorithm has a time complexity of $O(log_2\ D)$; and with serial computation, it is of $O(D)$, where D represents the diameter. For space complexity, each node has to store qd n-bit boolean vectors, which results in an $O(n^2\ q\ log_2\ D)$ complexity for all nodes in terms of bits. However, if the boolean-vectors are sparse, and n is large, various coding schemes for the one-bit-locations can be used instead of the boolean-vectors, but at a price of greater time complexity.

4 AN EXAMPLE

In this section we use the 24 nodes GCR graph described in section 2 as an example to illustrate the computation of our recursive algorithm. In this case both the diameter and divisor are 4. That is, we have $d = 2$ and $q = 4$. Suppose the source and destination nodes are 0 and 8, and both are of class 0. There are two shortest paths, each has length 3 and four more paths with length 4 between these two nodes. They are:

path 1 :	0, 19, 3, 8;
path 2 :	0, 2, 10, 8;
path 3 :	0, 1, 4, 5, 8;
path 4 :	0, 19, 11, 3, 8;
path 5 :	0, 2, 18, 10, 8;
path 6 :	0, 2, 10, 9, 8.

According to the GCR constants, the database for class 0 is as follows:

Level 0 :	1, 2, 19, 21;
Level 1 :	1, 2, 3, 4, 10, 11, 18, 19, 20, 21, 22, 23.

Note that the nodes at level 0 are projected to level 1. Since node 8 belongs to class 0, the half-tree for the destination node can be obtained by a modular shift of 8 bits:

```
struct xnode { int id; struct xnode *next;};                    /* path structure */
/********************************************************************/
/*    Find a single path between source and destination.           */
/********************************************************************/

Single_Path(path,s1,s2,S_Tree,xd,xn,xk,level)
struct xnode **path;                                   /* s1, s2: source & dest */
int s1,s2,*S_Tree,xd,xn,xk,level;                      /* S_Tree: half-trees */
{          int mid;                                    /* xd, xn, xk: d, n, k */
           int *S_vec1, *S_vec2;                       /*S_vec:vector after shifting*/

           if (level < 0) Push(path,s2);
           else                                        /* shift s1's half-tree */
           { S_vec1 = (int *) malloc((unsigned) (xn*sizeof(int)));
             Shift(S_vec1,level,s1,S_Tree+(s1%xk)*xd*xn,xd,xn);
             if (*(S_vec1+s2))                         /* check if s2 in half tree of s1 */
              {  (void) free((char*) S_vec1);
                 Single_Path(path,s1,s2,S_Tree,xd,xn,xk,level-1);
              }  else                                  /* find common point */
              {  S_vec2 = (int *) malloc((unsigned) (xn*sizeof(int)));
                 mid = xn;
                 if (level>0)                          /* at level l-1 of s2 half-tree */
                 {Shift(S_vec2,level-1,s2,S_Tree+(s2%xk)*xd*xn,xd,xn);
                  And(S_vec1,S_vec2,&mid,xn,s2);}
                 if (mid == xn)                        /*if not found, check level l */
                 {Shift(S_vec2,level,s2,S_Tree+(s2%xk)*xd*xn,xd,xn);
                  And(S_vec1,S_vec2,&mid,xn,s2);}
                 (void) free((char*) S_vec1);  (void) free((char*) S_vec2);
                                                         /* split into 2 problems */
                 Single_Path(path,s1,mid,S_Tree,xd,xn,xk,level-1);
                 Single_Path(path,mid,s2,S_Tree,xd,xn,xk,level-1);
              }
           }
}
```

Table 2: *Single-Path Recursive Routine*

Level 0 : 3, 5, 9, 10;
Level 1 : 2, 3, 4, 5, 6, 7, 9, 10, 11, 12, 18, 19.

We start the recursion at level $d-1=1$. Since node 8 is not a node at level 1 of the source half-tree, we proceed to find a common node at level 1 of the source half-tree and at level 0 of the destination half-tree. There are two such nodes: node 3 and node 10. We choose one of these nodes, say node 3, as our common node and split the problem into two. One with a source node $s1=0$, destination node $s2=3$; and the other has a source node $s1=3$ and destination node $s2=8$. Both recursive calls are set at level 0. Since node 3 belongs to class 3, the database for this class after shifting 3 bits is:

Level 0 : 5, 8, 11, 19;
Level 1 : 0, 4, 5, 6, 8, 9, 10, 11, 13, 16, 19, 21.

For the first half with $s1=0$, $s2=3$, level$=0$, we compare the half-tree of node 0 to that of node 3 at level 0 and find node 19 as the common node. The next recursion split this half into two problems: $s1=0$, $s2=19$ and $s1=19$, $s2=3$ at level -1, which means that node 19 and 3 will be pushed into the path vector.

For the second half with $s1=3$, $s2=8$, level$=0$, we find that node 8 is a node at level 0 of the s1 half-tree. The recursion thus proceed with $s1=3$, $s2=8$ but at level -1, which again means that node 8 will be pushed into the path vector. At this point, the entire path 1 is found. Should we have chosen node 10 as the first common node at level 1, we would have found path 2, which is another shortest path.

As stated earlier, this recursive algorithm requires a parallel time complexity of O($\log_2$ *diameter*) and a serial complexity of O(*diameter*). Also a path with length $\leq 2^s$ can be found between any two nodes that has a shortest path, s-path ($2^{s-1}<$ s-path $\leq 2^s$).

5 CONCLUSIONS

In this paper we presented a recursive routing algorithm composed of two parts. The first part deals with the establishment of a database at every node. Computation involved in this part of the algorithm is carried out once and for all. Part 2 of the algorithm is responsible for the actual determination of a path between a source and destination pair. By exploiting the inherent symmetry of the GCR, this algorithm finds a path of any length $\leq 2^s$ ($2^{s-1}<$ shortest path $\leq 2^s$) where $s\leq d$ and d bounds the diameter of the graph by $2^{d-1}<$ diameter $\leq 2^d$.

References

[1] A.J. Hoffman and R.R. Singleton. "On Moore Graphs with Diameters 2 and 3". *IBM Journal*, 30:497–504, November 1960.

[2] J.C. Bermond and C. Delorme. "Strategies for Interconnection Networks: Some Methods from Graph Theory". *Journal of Parallel and Distributed Computing*, 3:433–449, 1986.

[3] D.V. Chudnovsky, G.V. Chudnovsky, and M.M. Denneau. Regular graphs with small diameter as models for interconnection networks. Technical Report RC 13484(60281), IBM Research Division, February 1988.

[4] B.W. Arden and K.W. Tang. Representation and Routing of Cayley Graphs. Technical Report EL-89-02, Department of Electiral Engineering, University of Rochester, 1989.