LOW DIAMETER REGULAR GRAPH AS A NETWORK TOPOLOGY IN DIRECT AND HYBRID INTERCONNECTION NETWORKS

BY

YOGESH A. MEHTA

B.E., University of Mumbai, 2003

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

Abstract

Performance of a parallel computer depends on the computation power of the processors and the performance of the communication network connecting them. With the increasing scale and compute power of today's parallel machines, interprocessor communication becomes the bottleneck. Communication performance depends on the network topology and routing scheme for packets. This master's thesis explores the use of low diameter regular (LDR) graph as a topology for interconnection networks. We generate graphs having same number of nodes and connections per node as the hypercube, a widely used network topology. These graphs have lower diameter and lower average internode distance than the corresponding hypercubes, which implies that on an average, packets travel for a lower number of hops. With a good routing scheme this would reduce the average message latency and lead to better communication performance. We run experiments with this new topology in a parallel simulation framework for interconnection networks, BigNetSim. We show that LDR graphs achieve better performance than equivalent hypercubes for standard network traffic patterns. We have also developed a framework for implementing hardware collectives and we compare collective communication performance for different topologies. We implement a hybrid topology of a fat-tree and a LDR graph and evaluate its performance in comparison with a hybrid of a fat-tree and a hypercube.

To my parents and my sister.

Acknowledgements

First and foremost, I would like to thank my advisor Prof. Laxmikant V. Kale for his guidance and encouragement in my two years at the Parallel Programming Laboratory at University of Illinois at Urbana-Champaign.

I would like to thank all members of the BigSim and BigNetSim project. Terry Wilmarth, who helped in understanding POSE, the simulation environment in which BigNetSim is developed. Sameer Kumar, for his informative assistance with interconnection networks and in particular, the hardware collectives. Praveen Kumar Jagadishprasad for his initial code walkthroughs of BigNetSim to get me started on this project. Gengbin Zheng and Eric Bohm for their help with the BigSim simulator. Nilesh Choudhury for his patience and effort in debugging and optimizing BigNetSim.

I would also like to thank all my friends at PPL for all the technical help, encouragement and fun during my presence at graduate school.

Finally, I express fullest gratitude for my parents and my sister who have been my pillars of strength and support and who have always been there for me whenever I needed them.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the recent years, there have been remarkable advances in the scale and compute power of parallel computers. New parallel computers with hundreds of thousands of processors that are capable of achieving hundreds of teraflops at peak speed have been built. For example, the BlueGene (BG/L) machine, which is being developed by IBM, when completed will have 128K processors and is expected to achieve 360 teraflops at peak speed. Research projects in varied application areas such as molecular dynamics, astronomy, genomics and engineering design have been undertaken to exploit this tremendous amount of computational power.

Porting existing applications and developing new applications for such large scale machines is a challenging task. If we can simulate the behavior of the application on a large machine, we might be able to improve the design of a machine even before it is built. The simulation could help in the development of algorithms which will scale well on such machines and thus enable efficient use of the machines. The BigSim [20, 21] project aims at developing a simulation framework that would facilitate the development of efficient scalable applications on very large parallel machines. In most cases, there is a significant time gap between the deployment of large scale machines and the development of applications to run on them. Performance prediction of applications using BigSim can allow for optimization of applications in advance, so that they are ready to run as soon as the machines become available. Even after the machines are built, there are often long waiting periods involved in acquiring large number of nodes on these machines. A simulator like BigSim can serve as a debugging and tuning environment which would be much more easily available than the actual machines.

Parallel applications involve a lot of interprocessor communication. The interconnection networks that connect different computers in a parallel machine are responsible for the communication performance and consequently for the overall performance of the application. For correctly simulating a parallel computing environment, it is necessary to accurately model the interconnection network. A network simulator BigNetSim [17], has been developed, which simulates the packet level communication on the detailed contention-based network models for large parallel computers. The size of data involved and the large compute power required makes sequential simulation impossible, hence we use parallel simulation for BigNetSim. For accurate simulation of the communication time, BigNetSim models in detail various entities of the network which include the switches, nodes, channels and network properties such as the topologies, routing algorithms and flowcontrol.

BigNetSim has been developed as a generic framework, so that new topologies and routing algorithms can be added and different types of networks can simulated. With the detailed network model, it can accurately simulate the interconnection networks in many of the widely used parallel computers today. Another application of this network simulator is to enable development of new topologies which might be better than the ones used today. While building and testing an actual network with the new topology can be difficult as well as impractical from point of view of time and money, the network simulator is a much more feasible alternative. Simulation can be used to compare these new ideas with currently used ones, tune them for performance, and then deploy them on actual networks.

One idea is to use low diameter regular (LDR) graphs as an interconnection network topology. We generate LDR graphs that have same number of connections per node as hypercubes but that have lower diameter and lower average internode distance than corresponding hypercubes. Message latency, i.e. the time taken for messages to travel from source to destination, is an important measure of the communication performance. With lower diameter and lower average internode distance, packets would travel a lower number of hops on average and cause reduced contention. Thus, we would expect LDR graphs to provide lower message latency thereby improving communication performance. We discuss the motivation and generation of these graphs in detail in Chapter 4. Also, two or more topologies could be combined in the same network to form a hybrid interconnection network. A simulator like BigNetSim can be used as a testbed for trying out new ideas for improving overall network performance.

1.1 Thesis Contribution

Principal contributions of this thesis are:

- Design and implementation of low diameter regular graph topology for interconnection networks.
- Developing and optimizing a shortest-path based routing algorithm for LDR graphs.
- Extending the hardware collective framework for hypercube, LDR graphs, and hybrid topologies for interconnection networks.
- Development of topology and routing scheme for hybrid networks of fat-trees and LDR graphs.

I also fixed and adapted the original LDR graph generation algorithm to generate input graph data for the LDR graph topology. I was involved in development of specific components of BigNetSim such as the traffic generator and hybrid network design. My contributions towards the debugging and optimization of BigNetSim, in part, have led to a much improved performance of the simulation and a more accurate modeling of interconnection networks.

1.2 Thesis Organization

Chapter 2 describes POSE , the parallel discrete event simulation environment used for developing the interconnection network simulator. Chapter 3 presents an overview of the interconnection networks used in parallel computers, their properties and entities, and how they are simulated in BigNetSim. We motivate the use of LDR graphs as a topology for interconnection networks in Chapter 4. This chapter also explains the generation of these graphs, routing schemes, and their implementation and performance. In Chapter 5, we discuss the design, implementation and performance of hybrid topologies for interconnection networks. Chapter 6 discusses the framework for hardware collectives and collective communication performance for different topologies. Chapter 7 presents some conclusions from our work and directions for future research.

Chapter 2

Parallel Discrete Event Simulation

We have implemented our network simulation using POSE [18] , a scalable general-purpose parallel discrete event simulation environment. POSE has been built in Charm++ [8] , a C++ based parallel programming system which supports the virtualization programming model. The following overview of Charm++ is based on detailed description in [8] and [7] and POSE overview is based on [18] and [19].

2.1 Charm++

Charm++ is an object-based, message-driven parallel programming environment. The basic unit of parallelism in Charm++ is a message driven C++ object known as chare. Methods can be invoked on a chare asynchronously from remote processors; these are known as entry methods.

Charm++ is based on the concept of virtualization [7]. Each chare is a separate execution component and the number of chares(N) is independent of the number of processors(P). In general, with N much greater than P, applications can run with millions of chares on a much smaller number of processors. With virtualization, user's view of the program is that of the chares and their interactions. The runtime system takes care of the mapping of chares to processors. This distinction between user's view and actual system implementation

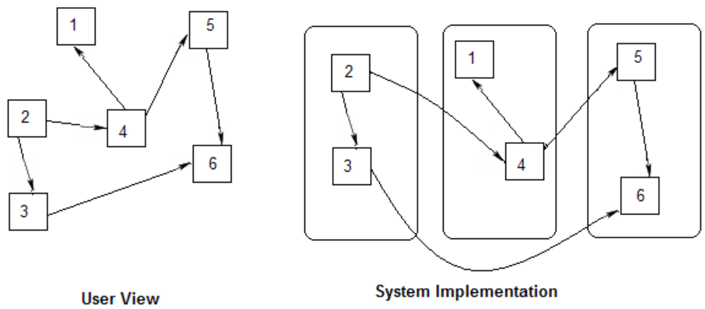


Figure 2.1. Charm++ Virtualization

is shown in Figure 2.1.

A dynamic Charm++ scheduler runs on each processor. The messages are stored in a queue which is sorted by a specific strategy. The scheduler picks the next message from the queue and invokes the corresponding method on the suitable object. As a result, no chare can hold the processor idle. Other chares can run while a particular chare is waiting for a message. This results in a good overlap of communication and computation and maximizes the degree of parallelism. On the basis of this virtualization model, Charm++ has been successfully used to simulate challenging applications like Molecular dynamics, Cosmology, and Rocket Simulation.

2.2 POSE

POSE stands for Parallel Object-oriented Simulation Environment. It has been developed by Terry Wilmarth, a member of the Parallel Programming Laboratory within the Department of Computer Science at the University of Illinois at Urbana-Champaign. POSE is a scalable parallel discrete-event simulation environment designed for simulation models with fine granularity of computation.

POSE encapsulates simulation entities in *posers*, which are equivalents of the chares

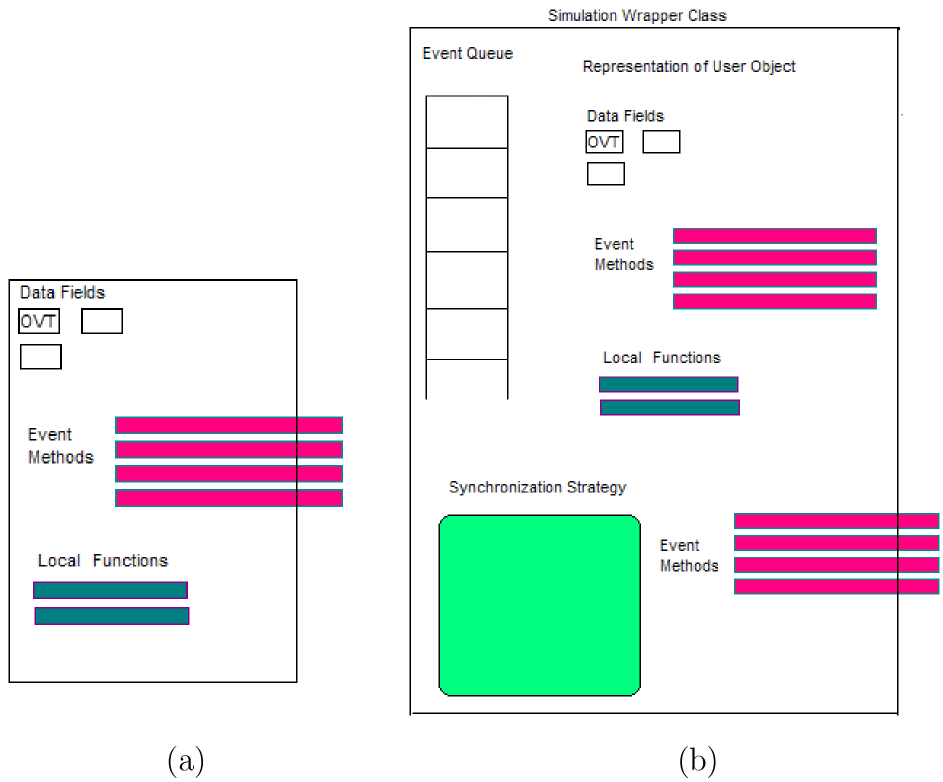


Figure 2.2. (a) User's view of a poser; (b) Internal POSE representation of a poser

in Charm++ . A structure of a poser is shown in Figure 2.2(a) . A poser stores it own virtual time known as Object Virtual Time(OVT). OVT is the virtual time that has passed since the start of the simulation relative to that object. Each poser has a set of event methods that are entry methods, they receive messages that have a timestamp. These entry methods capture incoming events, store them in a local event queue and invoke the local synchronization strategy on them. The event queue also stores checkpoints for the object state. This detailed internal representation of the poser is shown in Figure 2.2(b)

There are two ways in which a poser can advance its OVT. First is the elapse function. Calling an elapse with a number of time units passed as an argument advances the OVT of the poser by the time units specified. This indicates the time spent by the poser doing work. For example, in the context of network simulation, a channel poser can elapse time while it transmits a packet. Another way of advancing time on a poser is to invoke an event method on the poser with an offset. This offset is then added to the OVT of the poser, which is a way of indicating some activity performed in the future or to indicate the simulation time spent in transit. An equivalent example in the context of network simulation is when a packet is sent across a channel to a switch, the method to receive the packet is invoked on the switch with an offset equal to the time taken by the packet to transit the channel. The OVT of the switch poser will be appropriately advanced.

To develop an efficient application using POSE and to achieve good performance, it is important to decompose the problem into the smallest posers possible. This means that the degree of virtualization must be high. With smaller posers, the checkpoint and rollback overhead is less and object migration is easier. This also allows for better tuning of synchronization strategies to the object's behavior. An important drawback of higher degree of virtualization is that with more objects in the simulation, there is more frequent context-switching between entities for each event. Overhead of managing per-object information is also higher. We studied these tradeoffs [17] in the context of network simulation. We found that higher degree of virtualization has more pros than cons. For example, we observed that the 'switch' poser was too large and breaking it in to finer posers (making each port a separate poser) helped improve performance. Higher degree of virtualization also improved the scalability of our simulation.

We present a brief overview of the optimistic synchronization strategy used by POSE . The strategy is adaptive and can range from cautiously optimistic to highly optimistic. When the object receives an event it gets control of the processor and invokes the synchronization strategy to process events. The strategy performs necessary rollbacks and cancellations before beginning forward execution of events. Traditional optimistic approaches execute the earliest arriving event from a sorted list of events. POSE differs in that it maintains a *speculative window* which decides how far in the future beyond the current global virtual time (GVT) estimate an object may proceed. If there are events with timestamp $>$ GVT but within the window, then they are executed. All these events within the window are batched together and executed as a *multi-event*. This reduces the context-switching overhead and batching of events benefits from a warmed cache. These benefits outweigh the additional rollback overhead. The adaptive synchronization strategy and the multi-events, along with other features of POSE , are discussed in detail in [19].

Chapter 3

Interconnection Networks

Interconnection networks, as defined in [2] are programmable systems that transport data between terminals. Interconnection networks occur at a variety of scales from small-scale on-chip networks within a single processor to a large scale large-area or wide-area network. In the context of our work, we are concerned with networks which are used to connect different processors in a parallel computer system. With faster processors, we have faster computation and consequently, often communication becomes the bottleneck for the performance of a parallel computer. Better interconnection networks can help improve communication and thereby improve the performance of the entire system. Interconnection Networks can be broadly classified as direct networks and indirect networks.

3.1 Direct Networks

Each node in a direct network is connected to a router, so they are also called router based networks. The neighboring nodes can be connected by a pair of unidirectional or bidirectional channels. The function of a router can also be performed by a local processor, but dedicated routers are used in parallel computers for overlapping communication and computation. Every router has a certain number of input and output channels. Internal channels connect the local processor or memory to the router. External channels connect different routers.

3.2 Indirect Networks

For indirect networks the communication between any two nodes has to be carried out through switches. Every node has a network adapter that connects to a network switch. Each switch has a set of ports. Each port has an input and output link. A set of ports in each switch is connected to processors or connected to other switches. The interconnection of switches define various topologies. Transmitting a message in an indirect network from one node to another requires travelling to the switch of the first node, hopping across the network, reaching the destination node's switch, and then reaching the node itself.

3.3 Topology

Topology is the layout of connections of nodes in the network. The topology is important as it decides various important properties of the network such as bisection bandwidth, diameter, and average internode distance.

- The bisection bandwidth refers to the bidirectional capacity of a network between two equal-sized partitions of nodes. The cut across the network is taken at the narrowest point in each bisection of the network.

- Diameter refers to the length of the longest shortest path between any two nodes in a topology. It is the largest number of edges which must be traversed in order to travel from one node to another when paths which backtrack, detour, or loop are excluded from consideration.

- Average internode distance refers to the average of lengths of the shortest paths between all pairs of nodes in the topology.

We briefly discuss two common topologies here.

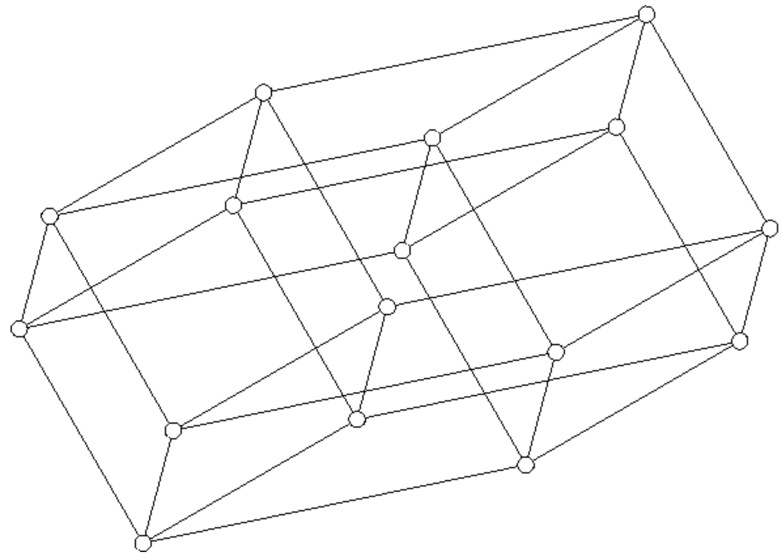


Figure 3.1. 2-ary 4-cube

3.3.1 Hypercube

Hypercube is a network with logarithmic complexity which has the structure of a generalized cube. In this topology, the nodes are placed at the vertices of a 2-ary M-cube, where M refers to the dimension. For example, a 2-ary 4-cube is shown in Figure 3.1.

For a hypercube of N nodes, the degree of each node is the same and is $log(N)$. The diameter of the hypercube is $log(N)$ and the average internode distance is $log(N)/2$. Hypercube is commonly used as a topology for direct networks.

3.3.2 Fat Tree

Fat-tree network [12] refers to the k-ary n-tree. The graph k-ary n-tree has been defined in [14] and [9]. It is a type of fat-tree which can be defined as follows:

Definition : A k-ary n-tree is a fat-tree that has two types of vertices: $P = k^n$ processing nodes and nk^{n-1} switches. The switches are organized hierarchically with n levels that have k^{n-1} switches at each level. Each node can be represented by the n-tuple $\{0, 1, ..., k-1\}^n$,

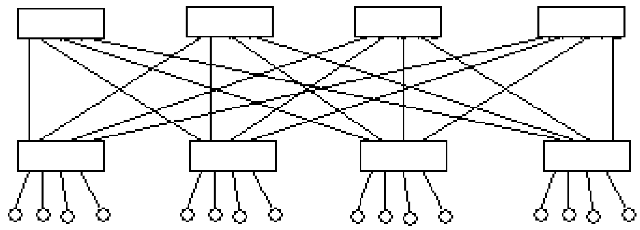


Figure 3.2. 16 Node fat-tree

while each switch is defined as an ordered pair $\langle w, l \rangle$ where $w \epsilon \{0, 1, ..., k-1\}^{n-1}$ and $l \epsilon \{0, 1, ..., n-1\}$. Here the parameter l represents the level of each switch and w identifies a switch at that level. The root switches are at level $l = n-1$, while the switches connected to the processing nodes are at level 0.

Fat-tree networks have various advantages, such as high bisection bandwidth, scalable topology, compact switches, and simple routing. They are used extensively in current generation high performance networks such as Quadrics and Infiniband. A Complete 16 node fat-tree is shown in Figure 3.2.

Both these topologies are implemented in BigNetSim. It also includes other topologies including the new topology based on low diameter regular graphs, which we discuss in detail in the next chapter.

3.4 Routing

A route is an ordered set of channels $a_1, a_2, a_3, \cdots, a_n$ where the output node of channel a_i is the input node of channel a_{i+1}. Depending on the type of network, there could be a single route or multiple routes between a source and a destination. A good routing algorithm balances the load uniformly across channels. There are two major classification of routing algorithms - fixed and adaptive.

3.4.1 Fixed Routing

Deterministic or fixed routing algorithms choose the same path between any two nodes, which is a function of the source and destination address. This can lead to load imbalance in the network for some load patterns. There can be increased contention in a specific part of the network, particularly in random traffic patterns. However, they are simple and inexpensive to implement. Deterministic algorithms are still prevalent today since designing a good randomized adaptive algorithm for irregular topologies is difficult.

3.4.2 Adaptive Routing

A routing technique is said to be adaptive if, for a given pair of source and destination, the path taken by a particular packet depends on dynamic network conditions, such as network contention, congested channels, or presence of faults. It provides fault tolerance to the system by introducing alternate paths since failure of a link will effectively leave the network disconnected in deterministic routing while the network will still remain connected in adaptive routing. Although the adaptive technique has clear advantages, it introduces a lot of complexity in the switch, which makes it costly.

3.5 Simulation model

The model is an effort to simulate the basic units of a network, namely switch, channel, network interface cards, and finally, nodes which inject messages into the network and receive messages intended to them. The conceptual model of BigNetSim is shown in Figure 3.3

Each of these entities are modeled as posers.

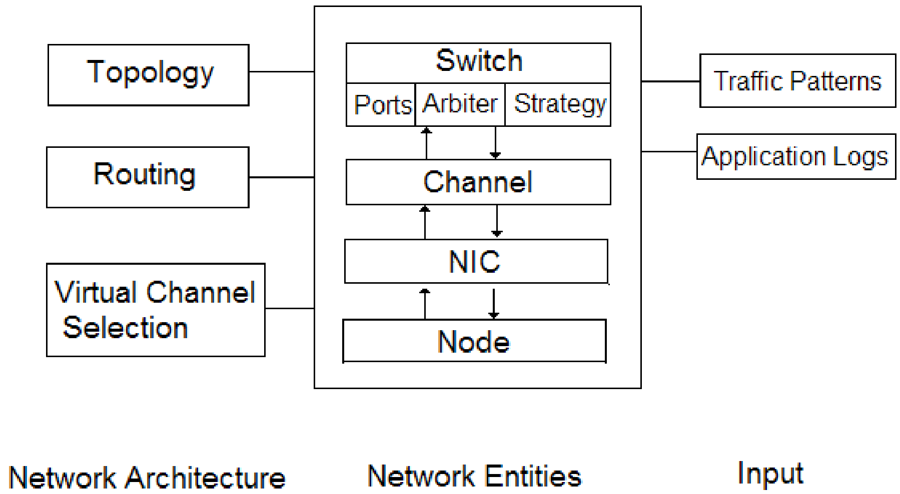


Figure 3.3. BigNetSim conceptual model

3.5.1 Switch

The switch assumes a packet switching strategy and uses virtual cut-through strategy to forward messages through the switches. Switches can be distinguished as:

- *Input Buffered (IB):* A packet in a switch is stored at the input port until the next switch in its route is decided and leaves the current switch if it finds available space on the next switch in the route.

- *Output Buffered (OB):* A packet in a switch decides beforehand about the next switch in its route and is buffered at the output port until space is available on the next switch along the route.

It has a simple and fair arbitration strategy which uses aging of packets to determine which packet competing for which port should get higher priority. We use credit based flow control in the network; the credits are equivalent to buffer space. A switch computes how many credits it has available on a specific downstream switch and based on the amount, it decides whether it can send a packet or not. The model also supports configurable strategies for input virtual channel selection and output virtual channel selection. The configurability of the switch provides a flexible design satisfying the requirements of a large number of networks.

3.5.2 Channel

The channel is a simple entity which receives a packet and delivers it to the next object it is connected to, which could be either a switch or a destination node. The channel models the delay equivalent to the time it would take for a packet to travel from one switch or node to another along that channel.

3.5.3 Network Interface Card

The network interface card divides a message into separate packets, based on the maximum transmission unit of the network, and sends them. It models DMA and HCA delays. The delays are categorized for small and large messages, then added to the message send times. It responds to excessive load with an injection threshold that models deteriorating caching effects as it gets overloaded. At the receiving end, the NIC assimilates the packets into the message and passes the data to the node.

3.5.4 Node and Traffic Patterns

The node generates the packets and injects them into the network. The traffic generator module can be used to generate different traffic patterns. Six different traffic patterns exist, which determine the destination node that it can generate:

- *k-shift*: address of the destination node for node i is $(i + k) mod(N)$
- *Ring*: equivalent to 1-shift

- *Bit transpose*: address of the destination node is a transpose of that of the source node i.e. $d_i = s_{(i+b/2)mod(N)}$
- *Bit reversal*: address of the destination node is a reversal of the bit address of the source node i.e. $d_i = s_{b-i-1}$
- *Bit complement*: address of the destination node is a bitwise complement of the address of the source node.
- *Uniform distribution*: This is a random traffic in which each node is equally likely to send to any of the other nodes.

The traffic generation time distribution can either be deterministic or it can follow a Poisson distribution.

3.5.5 Topologies and Routing Strategies

Implementation of a topology in our model involves defining the neighbors for a switch and the mapping of these neighbors to the port numbers on the current switch. Routing strategy decides the output port on which the packet is to be sent. Topologies and routing strategies can be created separately, and the architectures can be created to use these topologies and routing strategies. Various topologies have been implemented such as the hypercube, fat-tree and mesh3D topologies. Corresponding routing strategies such as hamming-distance routing for hypercubes; dimension-ordered and Torus routing for 3D-mesh topologies; and Up-Down routing for fat-tree topologies have also been implemented.

Chapter 4

Low Diameter Regular Graphs

4.1 Background

Communication performance of an interconnection network depends substantially on the topology. When large number of processors are connected together, the topology should be *dense*, i.e. the diameter must be small. Secondly, for a uniform treatment of all processors, the topology must be *regular*, i.e. each node in the topology graph must have the same degree. Low diameter regular (LDR) graph refers to a dense regular topology that can scale to a large number of nodes.

There has been extensive research in the construction and applications of regular graphs with small diameter. An upper limit on the number of nodes in a regular graph of degree $d > 2$ and diameter k is called the Moore bound [5] and it is given by

$$N(d,k) <= (d(d-1)^k - 2)/(d-2) \tag{4.1}$$

The Moore graph is a graph which achieves the Moore bound. These are complete graphs, polygon graphs (regular graphs of degree 2), as well as three others:

- (nodes, degree, diameter) = (10,3,2) : This is known as the Petersen graph. It is shown in Figure 4.1

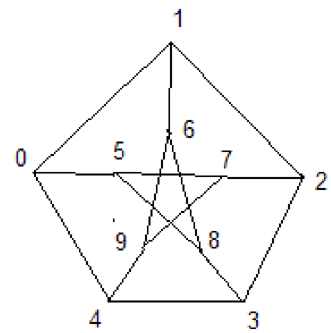


Figure 4.1. Petersen graph

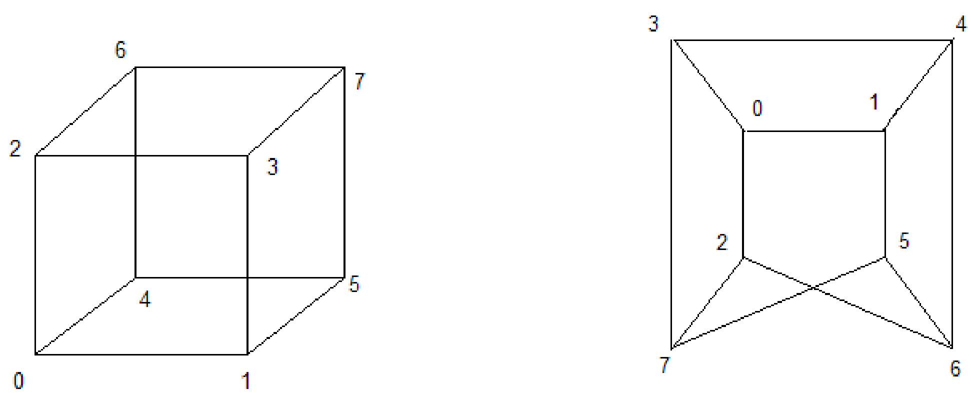
- (nodes, degree, diameter) = (50,7,2) : This is known as the Hoffman-Singleton graph[5]
- (nodes, degree, diameter) = (3250,57,2): This is a possible but yet undiscovered graph[1]

De Bruijn graphs[15] are a set of dense topologies which have been used in varied areas such as VLSI design and communication networks. Star graph[16] is another example of dense regular topology which is symmetric. D-Trees[6] is a class of tree-based dense interconnection network topologies which can interconnect more nodes than star graphs and n-cubes of comparable diameter.

4.2 Motivation

Low diameter and low average internode distance are desirable properties for an interconnection network topology. In routing, it implies low average packet hops and less average time spent by messages at intermediate nodes in the path. In addition to lower message latency, lower diameter also leads to larger communication neighbourhood for a particular node. The neighbourhood of a node refers to the set of vertices within a specific number of hops. A larger neighbourhood results in a more uniform load distribution which consequently leads to better utilization of processors.

In LDR graphs, each node has the same number of neighbors. If the number of neighbors is $log(N)$, where N is the number of nodes in the graph, then the number of connections is exactly the same as a corresponding hypercube of the same number of nodes. For example, for a 64 node network, we can build an LDR graph of connectivity 6 per node. We, consequently, get a graph of 192 edges. This is the same as the total number of connections in a hypercube of 64 nodes. The idea behind keeping the number of connections the same for both the LDR graph and the hypercube is that we can build the LDR graph with the same number of wires as is needed for a hypercube. Given this similarity with the hypercube, we want to see if we can get better performance with a newer topology. For this we need a well-defined procedure to generate an LDR graph, which is outlined in the next section.



(a) (b)

Figure 4.2. (a) 8 node Hypercube; (b) 8 node LDR graph

As an example, Figure 4.2 shows an 8 node LDR graph that we generated and a corresponding 8 node hypercube. Both have the same number of nodes and same number of connections per node. The diameter of the LDR graph is 2, as compared to the hypercube

Nodes	Conn.	HC dia.	LDG dia.	HC avg. i.n.d.	LDG avg. i.n.d.
8	3	3	2	1.5	1.375
16	4	4	3	2	1.77
32	5	5	3	2.5	2.11
64	6	6	4	3	2.45
128	7	7	4	3.5	2.65
256	8	8	4	4	2.87
512	9	9	5	4.5	3.09
1024	10	10	5	5	3.3
2048	11	11	5	5.5	3.48

Table 4.1. LDR graph v/s Hypercube

diameter of 3. The average internode distance (avg. i.n.d.) for the LDR graph is 1.375, which is less than 1.5 for the corresponding hypercube.

Table 4.1 is a list of example LDR graphs we generated and how they compare with corresponding hypercubes.

We observe that the LDR graph gives us a lower diameter and a lower average internode distance than the corresponding hypercube. The gains increase with increasing network size, which should make LDR graph an even better alternative for large network sizes. This gives us the motivation that the LDR graph can serve as a better alternative topology to a hypercube.

With this motivation we implement the LDR graph as a topology and compare its performance against the hypercube

4.3 Generating an LDR graph

In this section, we describe the procedure to generate a LDR graph. This was originally developed by Prof. Laxmikant V. Kale. The spanning tree is used as the basic structure to build the LDR graph. If the number of nodes of the LDR graph is 'V' and the connectivity is 'C', we initially build a spanning tree of V nodes wherein each node in the graph has

$C-1$ children. This assures a degree of V for non-leaf nodes which have 1 parent and $C-1$ children. An exception is the root which has no parent. To complete the LDR graph, we need to complete the connections for the root and other nodes with incomplete connections. Each new edge is added using the following procedure:

1. Pick a vertex 'A' with maximum incomplete connections.

2. Pick another vertex 'B' randomly from the remaining vertices.

3. Check if the two vertices are already connected, in which case pick another vertex at random.

4. Continue step 3, as long as there are no vertices remaining which satisfy the condition or we find a legitimate vertex.

5. In case we find a legitimate vertex B, then add an edge between A and B.

6. In case there is no legitimate vertex, we open some other edge $X-Y$ in the graph and then connect $A-X$ and $B-Y$. This enables us to avoid cycles of length 2 between 2 vertices in the graph.

The steps in construction of an 8 node LDR graph are illustrated in Figure 4.3

Note that a random number is used for the LDR graph generation. With a different choice of seed, we generate different LDR graphs with same number of nodes and connectivity and calculate the diameter and average internode distance for each LDR graph. The LDR graph with smallest average internode distance among a certain number of these generated graphs is chosen.

Once the graph is generated, the shortest paths between each pair of vertices in the graph are calculated. We use Dijkstra's shortest path algorithm[3] to calculate the shortest paths. This shortest path information is stored in files which can then be used as input in the BigNetSim framework.

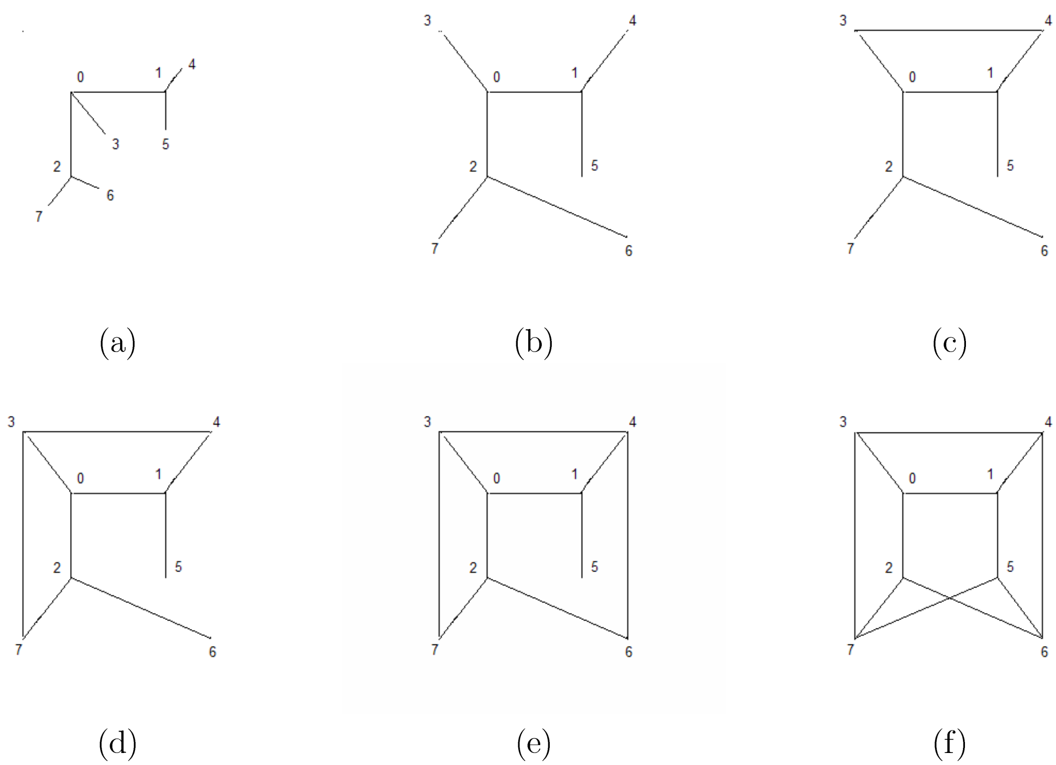


Figure 4.3. (a) Initial Spanning Tree; (b) Same Spanning Tree - A different layout; (c) Adding an edge; (d) & (e) Adding more edges one at a time; (f) Complete 8 node LDR graph

4.4 Implementation of LDR graph topology

To implement a topology in BigNetSim, we need to implement functions which find out the neighbors of a particular switch and functions which specify the next switch and channel given the current switch and port number. We have applied LDR graph as a topology for direct networks. In case of an LDR graph, when the graph is generated, we store the neighbors as well as the shortest path information in files. These files are read into memory in the initialization phase. This available neighbor data can be used to find the next switch and next channel for a particular port on the current switch.

4.5 Routing Algorithm for LDR graphs

For a topology to provide good and effective communication performance, we need to implement an efficient routing algorithm on it. In our motivation, we showed that LDR graphs achieve lower diameter and lower average internode distance than a hypercube. However, to lower the message latency, the routing scheme should minimize contention as network contention can significantly delay the packets.

To properly compare a hypercube and an LDR graph we need a good understanding of routing in a hypercube. We briefly discuss the hamming distance routing for hypercube implemented in BigNetSim.

Hamming Distance Routing: Hypercube is a symmetric topology. If we look at an example 8-node hypercube shown in Figure 4.2(a), we can notice the following property: the binary representation of every neighbor of a node differs by 1 bit from the binary representation of the node. Hamming distance refers to the number of bits that differ between the binary representations of two numbers. Hamming distance between the node and its neighbor is 1. Hamming distance between two nodes k hops away from each other is k. Moreover, a neighbor is connected at a port number which corresponds to the bit position at which it differs from the node. For example, a neighbor which differs in the least significant bit, i.e. bit position 0, is connected at port 0. Hamming distance routing algorithm for the hypercube is based on this property. The algorithm is as follows:

1. If the current switch is the last switch in the path, return the port number of the port connected to the node.

2. If not, find the exclusive OR (XOR) of current node and the destination. This will tell us which bit positions the current node and the destination differ in.

3. In order from least significant bit onwards, find the first bit position in the XOR which

is '1'. This corresponds to a neighbor which is on the path from the current node to the destination.

4. The bit position is the same as the port number of the port on which this particular neighbor is connected, so we return this port number.

Hypercube is a symmetric topology and this hamming distance routing leads to fairly uniform link utilization. Development of an equally effective routing algorithm for the asymmetric LDR graph is a challenging task.

Shortest path routing We have used the shortest paths idea as the basis for our routing algorithm. In an ideal scenario, with no contention, latency will be minimum if the packets are routed along the shortest paths from source to destination. We initially implemented a basic shortest path routing scheme and then optimized it to handle contention effectively.

Our basic LDR routing algorithm is as follows:

1. If the current switch is the last switch in the path, return the port number of the port connected to the node.

2. If not, use the shortest path information to find out the next switch on the shortest path from the current switch to the destination.

3. Find the port number of the port connected to this next switch and return this port number.

This is a simple algorithm which does not have any adaptivity. We compared its performance with a hypercube using hamming distance algorithm. We further optimized the algorithms to adapt better to network contention. We discuss the performance of the fixed routing and the incremental performance improvements with adaptive routing in the next section.

4.6 Performance

In this section we compare the communication performance of LDR graphs and equivalent hypercubes for different routing schemes as we incorporate adaptivity in routing based on feedback about what we learn. The basic routing algorithm is the shortest path based routing for LDR graph and Hamming distance routing for hypercube. A uniform random traffic pattern using a Poisson traffic generation frequency is used to generate packets in the network. This pattern was selected as it would closely resemble the behavior of an arbitrary collection of applications running on a supercomputer. The random selection of destinations result in a repeatable but random asymmetric load on the network. Each run involves each node in the simulated network generating 1000 packets.

Table 4.2 lists the network parameters we have used for our runs. These are derived from the interconnection network in the BlueGene/L machine [13].

Parameter	Value
Bandwidth	1.4 Gbps
Packet Size	256 bytes
Channel Delay	100 ns
Switch Delay	90ns
ASIC Speed	250 MHz
NIC Send Overhead	500 ns
NIC Recv. Overhead	500 ns

Table 4.2. Simulation Parameters

To evaluate performance, we plot the average message response time, i.e. the message latency as a function of increasing network load. The network load is modeled by a load factor.

Load Factor: The load factor refers to the ratio of the mean arrival rate of packets and the arrival rate that saturates a link. In the simulation, load factor is used to determine the mean of the arrival time distribution of packets. The mean is calculated as follows:

$$Mean = ((PacketSize)/(ChannelBandwidth))/(LoadFactor) \quad (4.2)$$

Ideally, if the entire bisection bandwidth is used, the network should hit congestion at a load factor of 1. However, since the destinations are chosen randomly, the entire bisection bandwidth is often not used up and in some cases we might see the network hitting contention at values of load factor greater than 1.

4.6.1 Performance with Fixed Routing

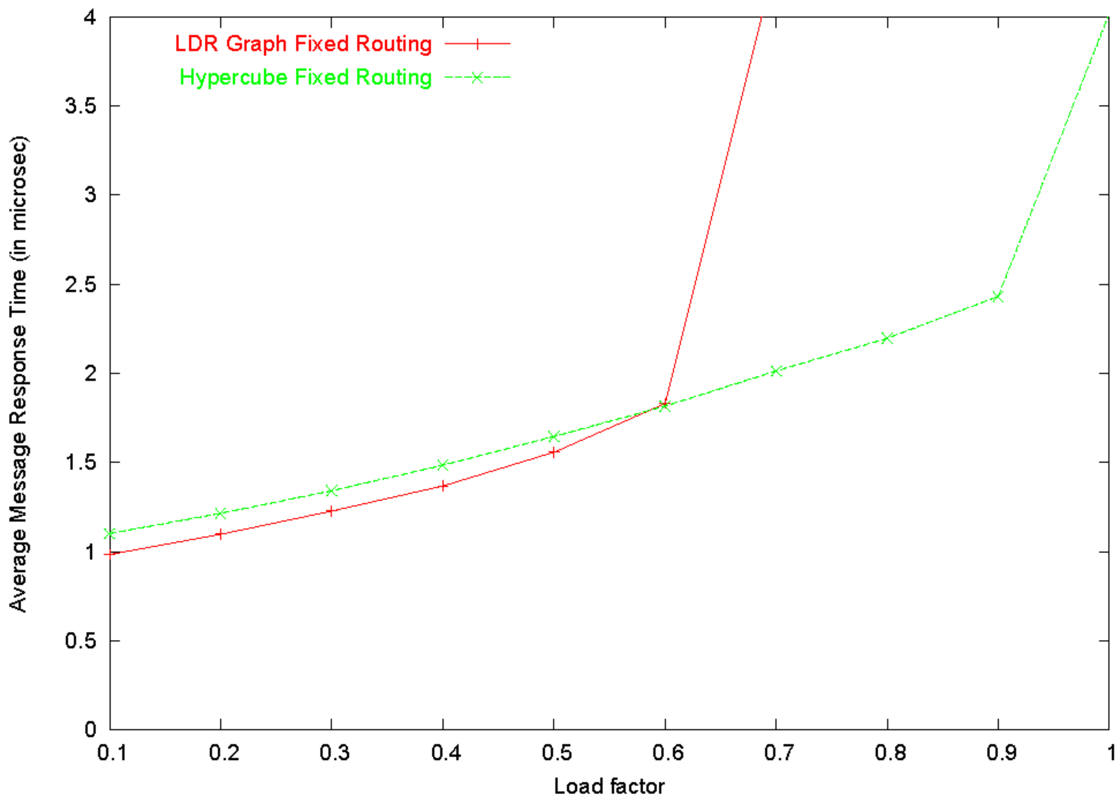


Figure 4.4. Message Response Time on a 64 node direct network with fixed routing

Figure 4.4 shows the plot of average message response time, i.e. average message latency against load factor for a 64 node direct network.

Here we see that the LDR graph has a slightly lower average message latency than the hypercube until a load factor of 0.6. This initial gain is a reflection of the lower average internode distance for a 64 node LDR graph as compared to the corresponding hypercube, noted in Table 4.1. However, as the load increases, the fixed shortest path LDR routing performs poorly. Since there is no adaptivity, the network hits contention at a much lower load as compared to the hypercube.

4.6.2 Performance with Oblivious Routing

We saw that in the fixed routing algorithm, all the packets going from the same current switch to the same destination would take the same route. An idea to improve on this would be to choose a different route at random instead of picking the same route every time. This would lead to more uniform distribution along all links connected to the switch. Therefore, we make a slight modification to the routing algorithms.

Instead of selecting the first port that could lead the current switch towards the destination, we find a set of output ports from the current switch, any of which could lead the current switch towards the destination. One of these output ports is chosen at random as the output port. Since this choice does not consider buffer size, link utilization, or any other network parameter, it is *oblivious* to the network condition. Hence, this routing is known as oblivious routing.

Oblivious Shortest Path LDR Routing :

1. If the current switch is the last switch in the path, return the port number of the port connected to the node.

2. If not, find the length of the shortest path from the current switch to the destination.

3. For all paths of the same shortest path lengths, find the corresponding next switch and the port on which that next switch is connected.

4. From these set of potential ports, choose a port at random and return it as the output port

Oblivious Hamming Distance Routing :

1. If the current switch is the last switch in the path, return the port number of the port connected to the node.
2. If not, find the exclusive OR (XOR) of the current node and the destination. This will tell us which bit positions the current node and the destination differ in.
3. Each of these bit positions correspond to a potential output port.
4. From this set of potential ports, choose a port at random and return it as the output port

Figure 4.5 shows the plot of average message response time, i.e. average message latency against the load factor for a 64 node direct network when oblivious routing is used.

We now see that LDR graphs perform better than hypercube over the entire range of network load. Adding some randomness to the routing improves the link utilization. This is advantageous to the LDR graphs because the shortest path tends to be biased towards particular links when fixed routing is used. However, in case of a hypercube, the fixed routing has a fairly uniform link utilization. In fact, random choosing might instead lead some links to be utilized more and the performance can even be worse than fixed routing.

We need an adaptive routing algorithm wherein the network conditions are considered while choosing the route. This is discussed in the next section.

4.6.3 Performance with Adaptive Routing

We need to use the buffer size at each of the ports as a parameter to decide which output port to route the packet on. This adaptive routing strategy is based on the minimal P-cube

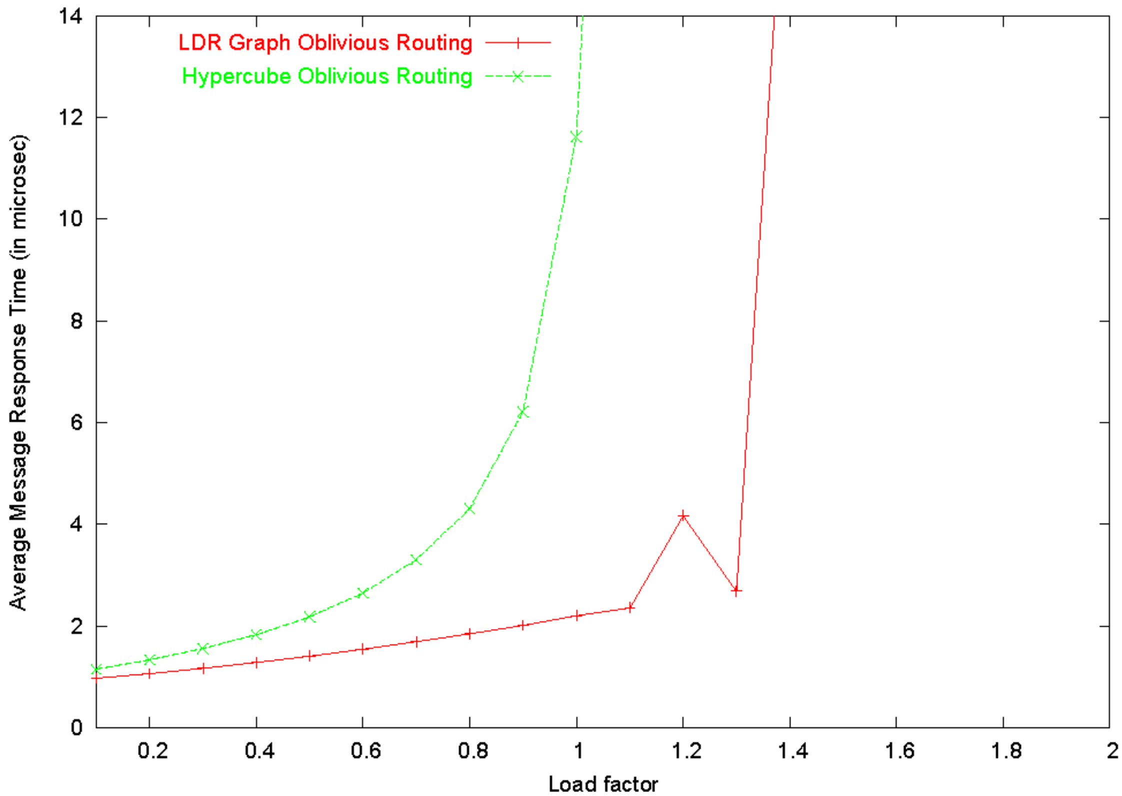


Figure 4.5. Message Response Time on a 64 node direct network with oblivious routing

routing presented in [4]. Minimal P-cube routing has been developed for hypercubes and we have adapted it for the LDR graphs as well.

The algorithms are similar to the corresponding oblivious routing algorithms but there is an improvement over the corresponding oblivious routing algorithms in the last step. The first 3 steps are the same until we find the set of potential ports. The step 4 differs and it is as follows:

4. From this set of potential ports, find the port which has the maximum available buffer size and return it as the output port.

For input buffered switches, we can additionally use the contention model in the switch to choose an output port that has minimum port contention along with maximum available buffer size. Port contention refers to the number of packets trying to go to a port.

Consider a simple example that illustrates how the 3 different routing algorithms work:

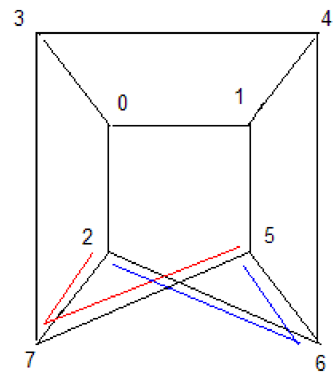


Figure 4.6. Routing on an LDR graph

A packet is currently at switch 2 and its destination is 5. There are two potential routes it can take $2-7-5$ or $2-6-5$. In fixed routing, always the route $2-7-5$ will be chosen (assuming that is the shortest path stored in the file). In oblivious routing we would pick one of 7 or 6 randomly. In adaptive, we look at the buffer sizes at the output ports of switch 2, which are connected to 6 and 7. Pick the route through a switch whose corresponding buffer has lower entries, i.e. maximum available buffer space.

Figure 4.7 shows the plot of average message response time, i.e. average message latency, against load factor for a 64 node direct network when adaptive routing is used. We see that with adaptive routing, both LDR graph and hypercube perform better; in addition network hits saturation for a higher load.

For lower load, the performance is almost the same, and the LDR graph shows some improvement in the section of higher load when the network has high congestion. In practice, most networks will use adaptive routing, as it gives best performance. Even with adaptive routing, we see that the LDR graphs offer some improvement in performance. On larger networks, we expect the performance gains to be more.

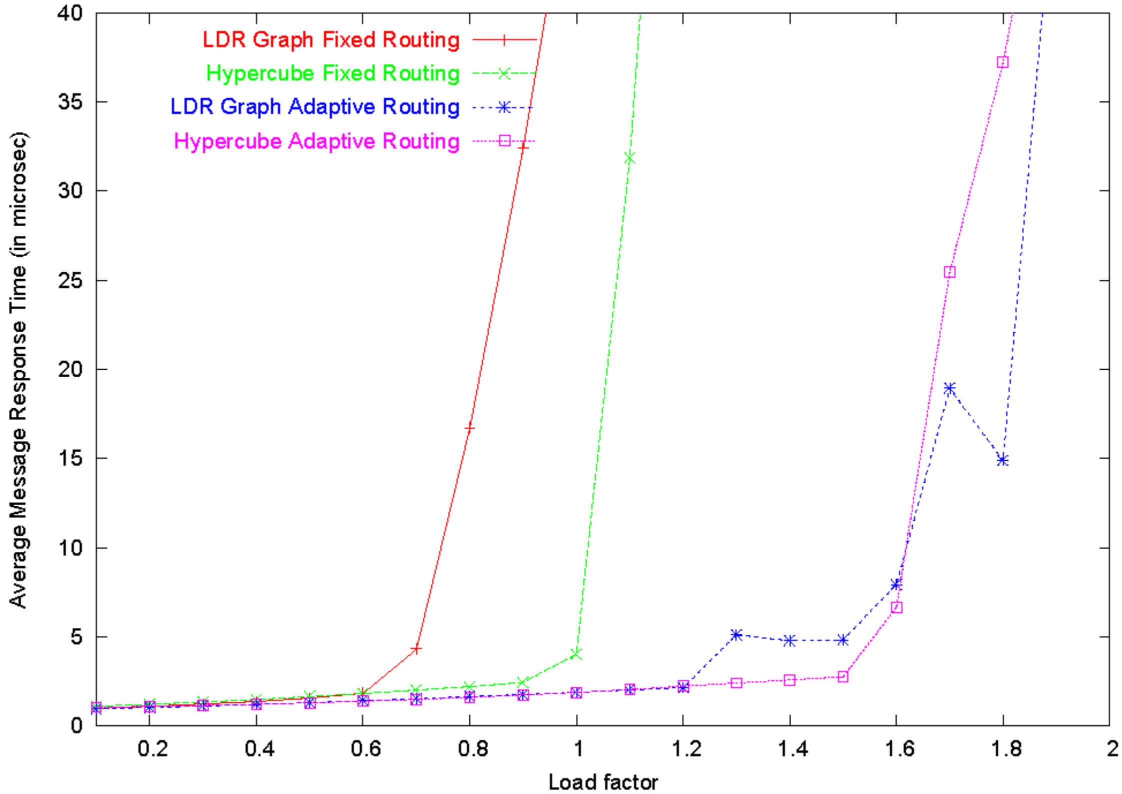


Figure 4.7. Message Response Time on a 64 node direct network with adaptive routing

From Figures 4.8 (a) and (b), we see that there is significant improvement in performance when a 2K node network is simulated with the same network parameters. With adaptive routing, LDR graphs have an average message response time that is almost 25-30% lower than the corresponding hypercube. Thus, our experiments show that LDR graph is indeed a better alternative to the hypercube.

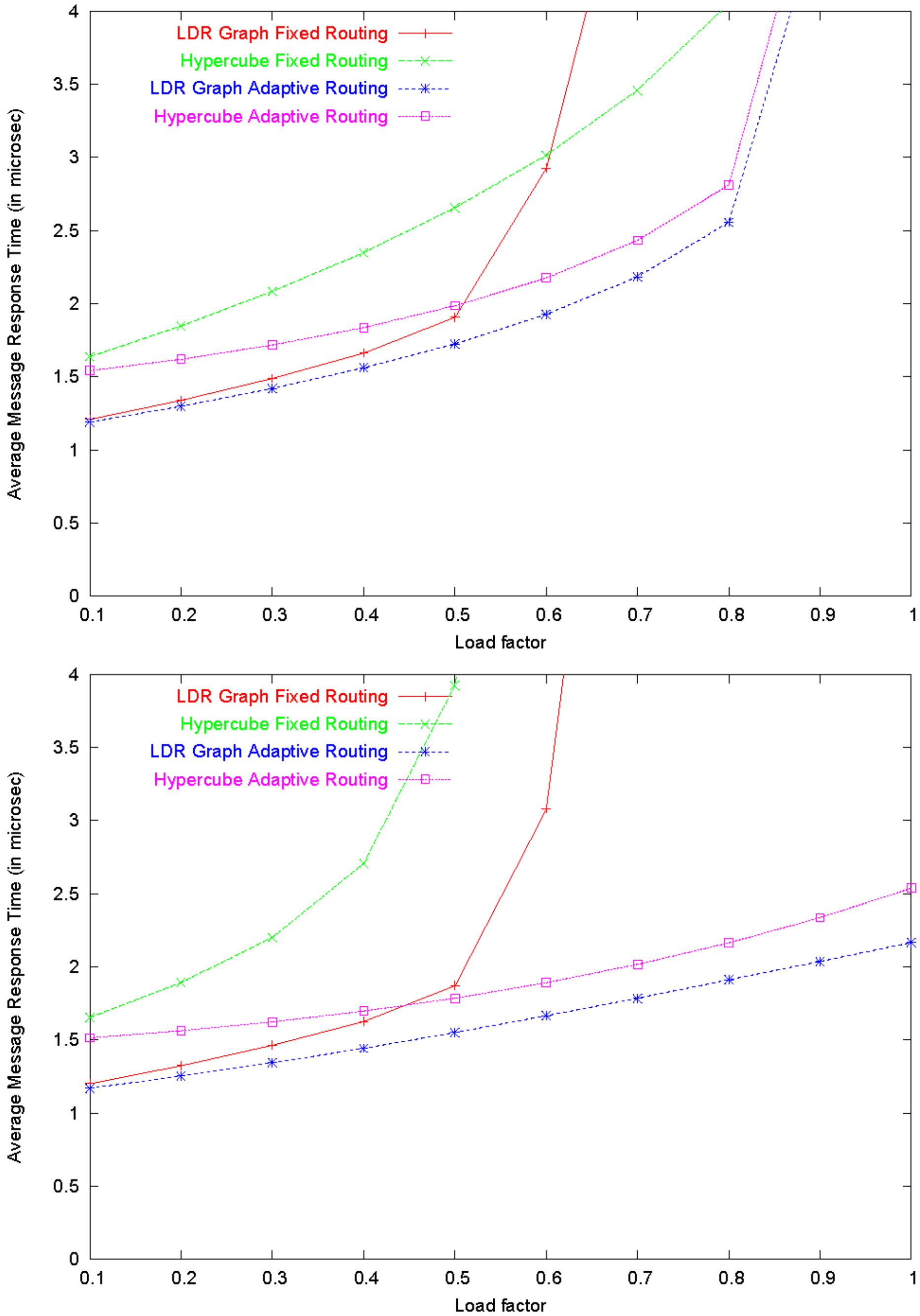


Figure 4.8. Message Response Time on a 2048 node network with (a) input buffered switches (top figure) and (b) output buffered switches (bottom figure)

Chapter 5

Hybrid Networks

Interconnection networks can be connected according to different topologies and we have discussed hypercube and fat-tree, the two popular topologies so far. Each has its distinct advantages, such as good bisection bandwidth, ease of routing and scalability in the case of a fat-tree and symmetry and simplicity of routing in case of a hypercube.

It is interesting to see if we can combine the advantages of two topologies by building a hybrid topology that has nodes connected using more than one topology. We call a network connected with such a hybrid topology a hybrid interconnection network.

In SGI Origin 2000 [11], a hierarchical fat hypercube topology is used. It is an interesting variation of the basic hypercube, which involves bristled node connections (two or more nodes connected to a switch) and plurality of n-dimensional hypercubes. Our idea is different, in that we have a number of fixed size hypercubes connecting the nodes and fat-trees connecting one switch from each hypercube. This is discussed in detail in the next section. We can also replace hypercube by any other topology and generate another hybrid. We replace hypercube by an LDR graph and see how this new topology effects the performance.

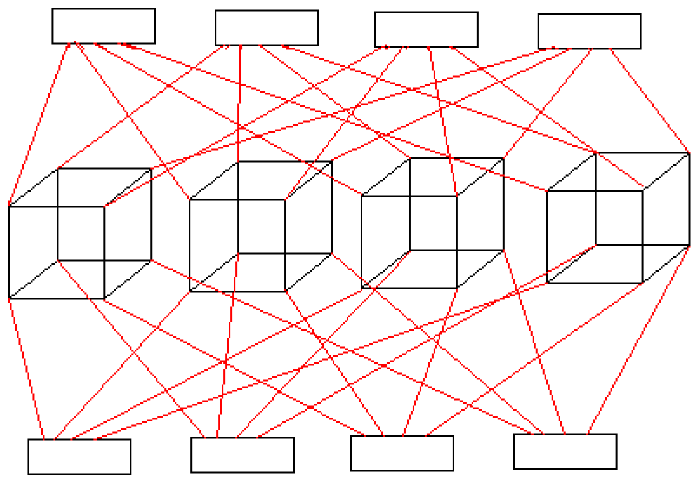


Figure 5.1. 32-node hybrid topology with 8-node hypercubes

5.1 Designing a hybrid of a hypercube and a fat-tree

We build the hybrid topology as follows.

1. Initially we define the size of the hypercube. This could be any power of 2 so that the direct network topology would correspond to a 2-ary N-cube.

2. Divide the nodes in sets, each of the size as defined for the hypercube.

3. Connect all these nodes in a direct network form so that we have multiple hypercube networks. If T is the total number of nodes and D is the number of nodes per hypercube, we would have $P = T/D$ hypercube networks.

4. Now we add the fat-tree in this as follows.

   (a) Consider the first switch in each hypercube network. We have P switches. Now we build a fat-tree using these P switches as the nodes. This gives us a fat-tree which connects all the hypercubes together.

(b) Similarly, we build fat-trees for the second switch in each hypercube, and so on.

(c) Consequently we have D such fat-trees connecting the hypercubes.

For example, for a 32 node network with each hypercube of 8 nodes, we have 4 such hypercubes, and then 8 different fat-trees connecting the hypercubes. The network would look like Figure 5.1

5.2 Designing a hybrid of a LDR graph and a fat-tree

Another topology we have implemented involves a hybrid of an LDR graph and a fat-tree. This is similar to the hybrid of hypercube and fat-tree. The only difference is that in this case, we replace the hypercube with an LDR graph of size D. The rest of the connections are similar.

5.3 Routing on an hybrid

The routing on a hybrid is straightforward because it is a combination of two separate topologies and the routing algorithm for each topology can be used here. Therefore, in the case of a hypercube and fat-tree hybrid we use Hamming Distance routing for the hypercube and the Up Down routing algorithm for the fat-tree. The packet is routed to the farthest node to which it can go, on that particular hypercube. It is then sent across the indirect network to the other hypercube and then forwarded to the destination node.

Similarly, routing can be done with the hybrid of a fat-tree and an LDR graph. The framework allows us to reuse the code used for individual topologies and corresponding routing algorithms in the hybrid implementation. With adaptive routing, a hybrid allows selection from multiple routes at two levels, at the direct network level as well as the indirect network level.

5.4 Performance

We present performance results of experiments on a hybrid network of fat-tree and hypercube and compare them with a hybrid of a fat-tree and an LDR graph.

We carried out the runs with the same network parameters as used in Chapter 4 for comparing a direct network of a hypercube and a LDR graph.

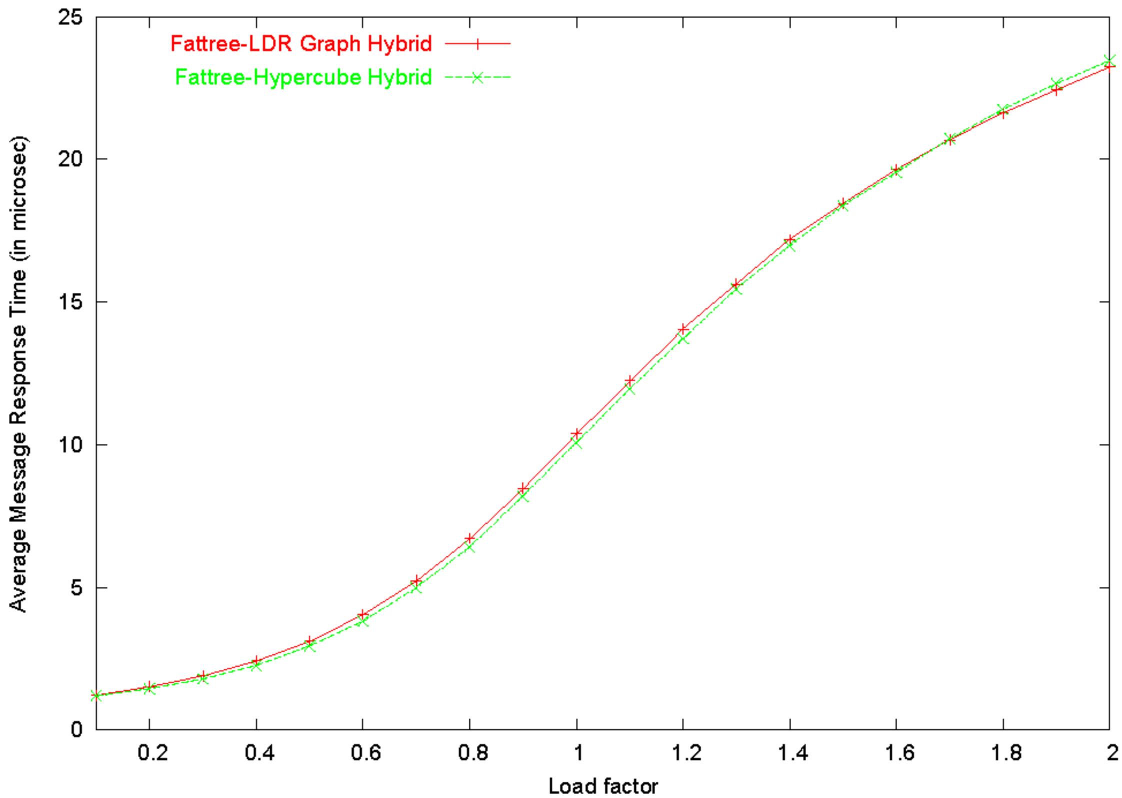


Figure 5.2. Message Response Time on a 1024 node hybrid network

We simulate a hybrid network of 1024 nodes having a 64-node direct network components. Each node in each direct network component, is further connected by a fat-tree and we have 1024/64, i.e. 16 such fat-trees. The results are plotted as shown in Figure 5.2. Here each node in the simulated network sends 50 packets with a uniform random traffic pattern.

From the plot, we can see that for a hybrid of fat-tree and an LDR graph, the message latency is initially higher, but improves as the network load increases. At high network load, the message latency is lower than that for a fat-tree-hypercube hybrid.

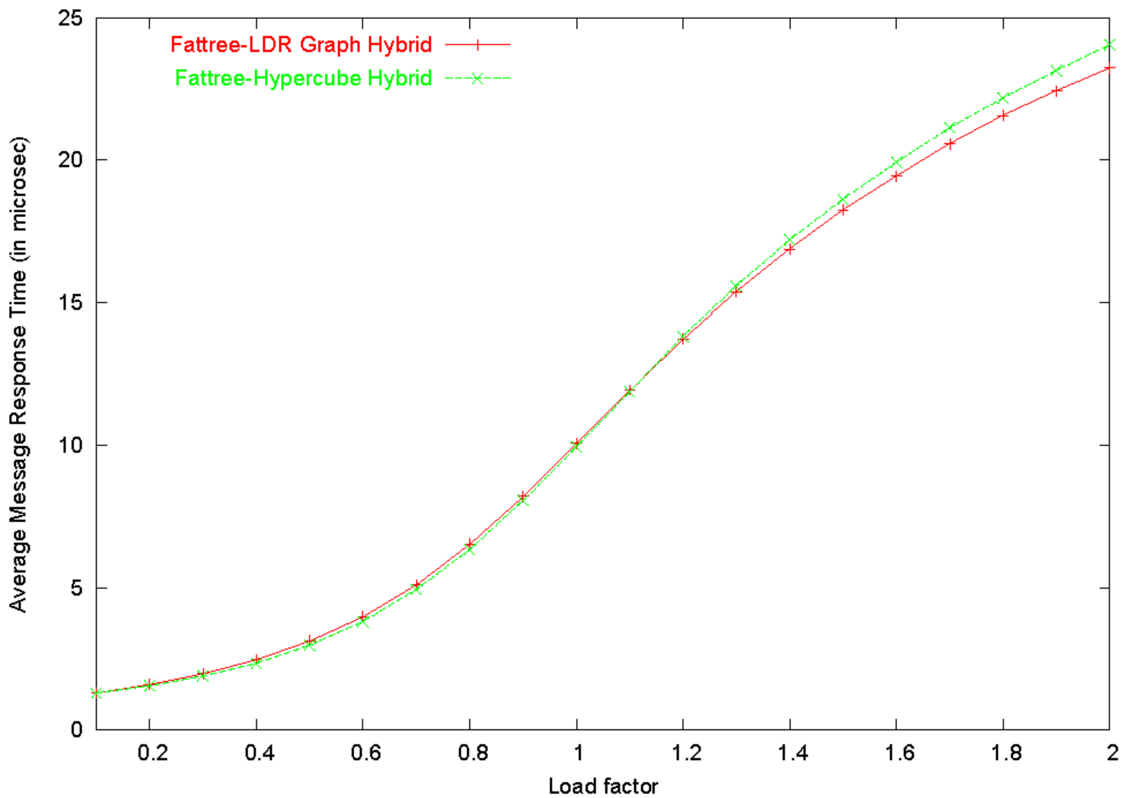


Figure 5.3. Message Response Time on a 4096 node hybrid network

Figure 5.3 shows the results of simulation on a 4096-node hybrid network with 256-node direct network components. Here also, we see that the message latency is slightly higher at low network loads but there is a greater improvement in performance at high network load. Thus, the gains in performance are higher for larger network sizes. Using LDR graph as the direct network component instead of the hypercube, the message latency is lower by about 4% for high network loads.

If we build a network with the same number of nodes and increase the size of the direct network component, we should see a greater improvement in performance. Figure 5.4 shows the results of simulation on a 4096-node hybrid network with 1024-node direct network

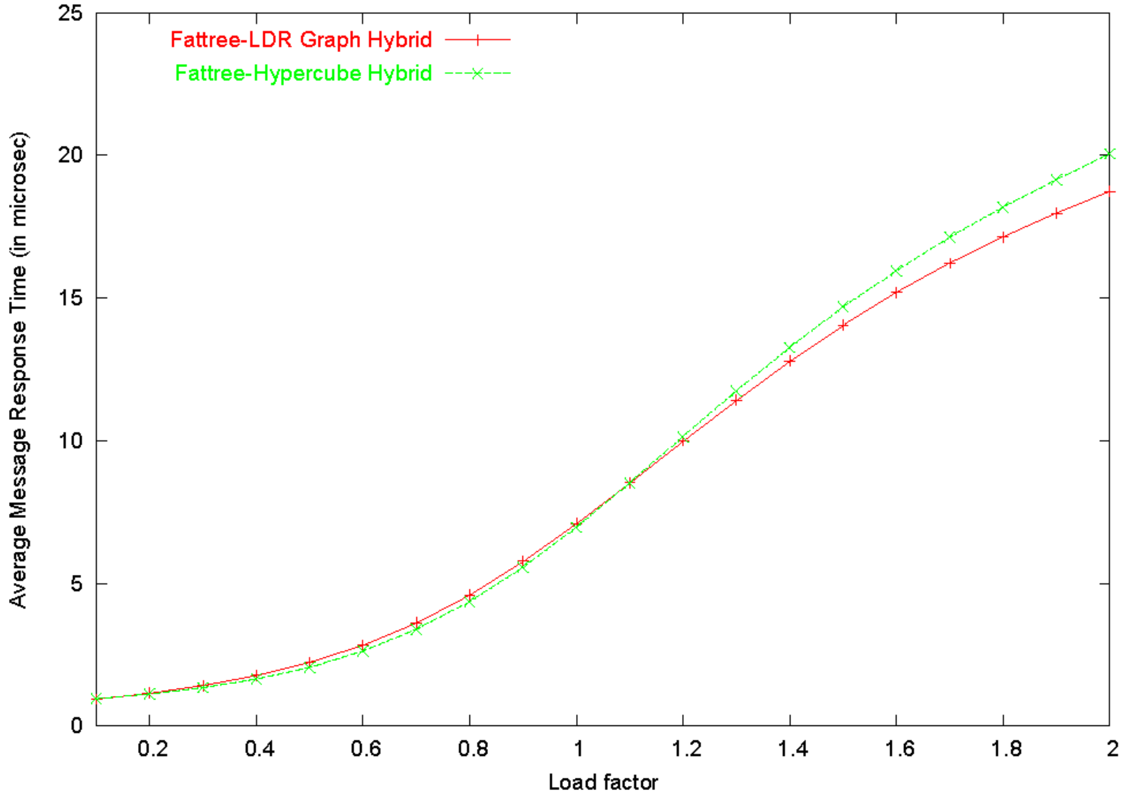


Figure 5.4. Message Response Time on a 4096 node hybrid network with a larger direct network component

components. Here we see that for the fat-tree-LDR graph hybrid, the message latency is lower than that for a fat-tree-hypercube hybrid by about 8% for high network loads. Thus, the LDR graphs give better performance when used as part of a direct component of a hybrid network.

Chapter 6

Collectives

Collective operations like broadcasts, multicasts and reductions are important communication operations which may involve some or all of the processors in a system. In [10] , different techniques for optimizing collective communication are discussed. Software collective optimization, which involves sending point-to-point messages along a tree, can be affected by delays at intermediate processors. Collective communication support in NICs is helpful but is limited by slow NIC hardware. A switch-based solution for optimizing collective communication is also presented in [10]. Support for collectives in the network requires the building of topology-specific spanning trees on the network.

Algorithms for building these collective spanning trees on fat-tree networks have been developed by Sameer Kumar and others as described in [10]. We have developed algorithms for building these spanning trees on other topologies such as Hypercube and LDR graph. Based on the algorithms for two different topologies, algorithms have also been developed for hybrid topologies. We now describe the algorithms in detail.

6.1 On Hypercube

Hypercube topology has been explained in Section 3.3.1. Hypercube has some interesting properties; for example, two adjacent nodes are different in only a single bit in the bit representation of the node numbers. In fact, this property is the basis of the Hamming Distance based routing used in a hypercube. We use the symmetric properties of the hypercube in the building of the spanning tree. In case of the multicast operation, we need to build the tree only on a set of nodes instead of on the entire network. For a broadcast operation, the tree has to be built over the entire network. In our approach, we have developed an algorithm to build a spanning tree over an entire hypercube. Multicast essentially is a broadcast over the hypercube enclosing the nodes in the multicast group. This brought forth the problem of determining the smallest hypercube enclosing the nodes in the multicast group; we came up with the following technique of implementing this.

Consider the bit representation of the nodes in the multicast group. By the definition of a hypercube, the number of nodes in the enclosing hypercube would be equal to 2^k where k is the total number of bits in which the nodes in the multicast group differ. For example, if 3 (0011), 5 (0101) and 7 (0111) are the nodes in the multicast group, then we see that they differ in 2 bits in totality and so the enclosing hypercube will have 2^2, i.e. 4 nodes. Also, the hypercube can be completely defined by ($0XX1$), i.e. retain the common bits and try out all combinations of the different bits (X=0 or 1). So in this example, the enclosing hypercube would contain $1, 3, 5, 7$. This technique can be efficiently implemented using bit operations like bitwise XOR and bitwise OR.

Subsequently, we build the tree and implement routing on this enclosing hypercube as we would do it on the entire hypercube in the case of a broadcast. This effectively allows us to support multicast on the hypercube. The tree building algorithm is as follows.

1. We iterate over the switches in this list which is identified as the enclosing hypercube.

2. For each switch i, we identify its children and mark this switch as the parent for each of its children.

3. A switch is chosen as a child if it is directly connected to the switch i, and it does not

have a parent already.

4. For the root switch of the tree, its parent is the corresponding node.

The TreeInfo data structure stores the information of this tree which is built by storing the parent and children for each switch in the switch list. Additionally, for all switches except the root switch, the corresponding node is also added as a child.

The next step is to determine the route from the root node to each of the switches in the tree; we use the hamming distance technique to determine this route. This is source routing because once the tree is built, we have a fixed route along the edges in the tree.

To abstract out these two common functions, building the tree and determining the route, for different topologies, we build a collective routing interface.

```
class CollectiveRoutingInterface {
public:
    //Build a multicast tree for collectives,
    //return list of switches in sw_list
    virtual void buildTree(CkVec<int> &sw_list, CkVec<TreeInfo> &pmap,
            vector<int> &group, int gsize)=0;
    //Find the source route to a switch from node
    virtual void routeFromNodeToSwitch(int src_node, int sw_id,
            unsigned char* route, int &numhops)=0;
    virtual ~CollectiveRoutingInterface() {}
};
```

The subnet managers for individual topologies can be derived from this interface.

6.2 On LDR Graphs

LDR graph topology has been explained in Chapter 4. The graph is described by a list of neighbours and the shortest paths between pairs of nodes in the graphs. We use this information to build the collective spanning tree on the LDR graph. The tree building algorithm is as follows:

1. We start with the root of the multicast group. For every switch i in the multicast group, we traverse the shortest path from the root to the switch i.

2. For every switch j which is traversed in this shortest path from the root, we mark its predecessor as its parent.

3. If a switch on the path was not already in the multicast group, it is added to the list.

4. For the root switch of the tree its parent is the corresponding node.

In the next step, to determine the source route from the root node to all the switches in the tree, we use the LDR graph routing which has been explained in Section 4.5

6.3 On Hybrid Networks

A hybrid network, as described in Chapter 5 consists of two topologies which are interconnected in a specific manner. To build a spanning tree on the hybrid network, we reuse the spanning tree building algorithms for the individual topologies.

1. We divide the nodes into bins. Each bin corresponds to a different direct network component.

2. For every node in the multicast group, we find the direct network component in which it is present and its position relative to that network component.

3. We mark this node, as well as nodes at the same relative position in all direct network components.

4. Once we are done with all nodes in the multicast group, we build trees on individual direct network components (hypercube or LDR graph) using the marked nodes on each component as the multicast group on that component.

5. We use the tree building algorithm for a fat-tree to build a spanning tree over a fat-tree that connects corresponding nodes in these direct network components.

Once the tree is built, we have to determine the source route from the root node to all the switches in the tree. For this purpose we make use of the Hybrid Routing algorithm which has been discussed in Section 5.3

6.4 Performance

We compared latency of a 10-packet broadcast on a direct network of a hypercube and an LDR graph. The network parameters are the same as listed in Table 4.2. The experiments were carried out on 64 node and 2048 node networks. The results are plotted as shown in Figure 6.1 and Figure 6.2.

We see that the LDR graph has lower message latency for the entire range of network load. The improvement in performance is greater for larger size networks. We use fixed routing and the gains are primarily due to the lesser number of links that packets have to travel. So we see that the difference in message latency is almost the same over the range of network load.

We also compared the broadcast performance on a 256 node hybrid network having a 16 node direct network component; the results are shown in Figure 6.3. There is a very small improvement in performance using LDR graphs. This is because the direct network

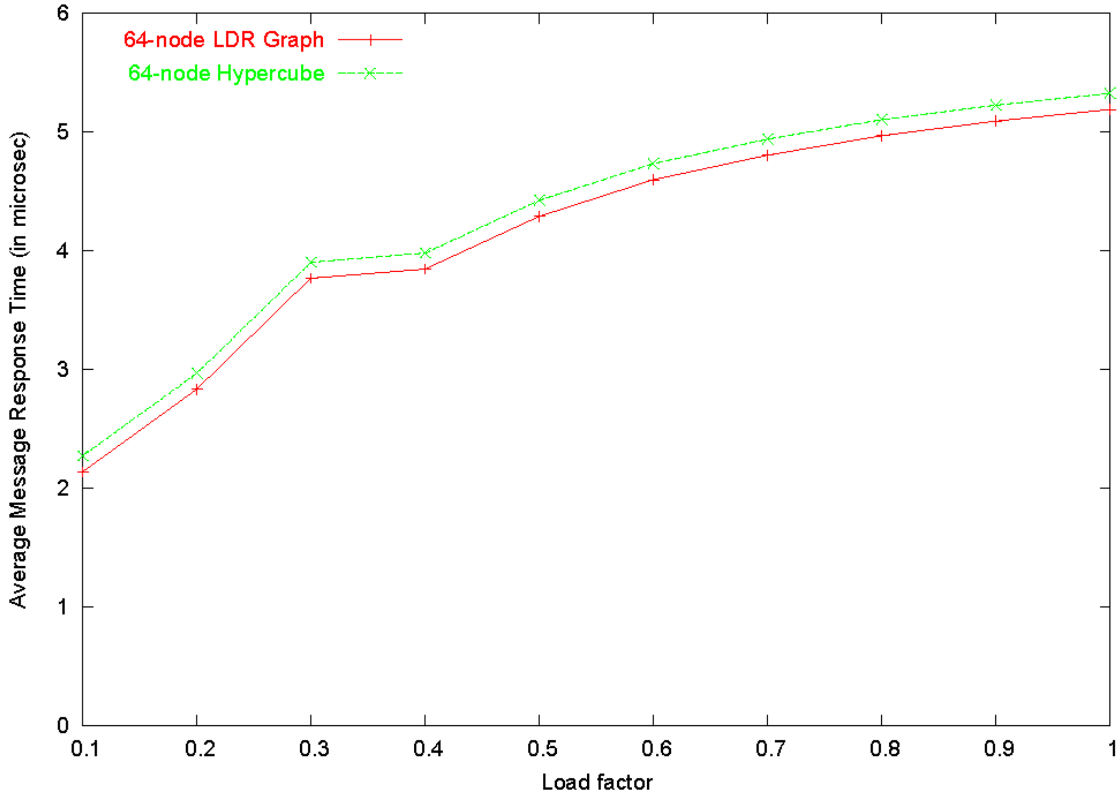


Figure 6.1. Message Response Time for broadcast on a 64 node direct network

component has just 16 nodes and there is small difference in the links to be travelled on the 16 node hypercube and the LDR graph. We still see some small gain, which is good, and larger networks would give greater performance improvement.

LDR graphs, thus, lead to better collective communication performance.

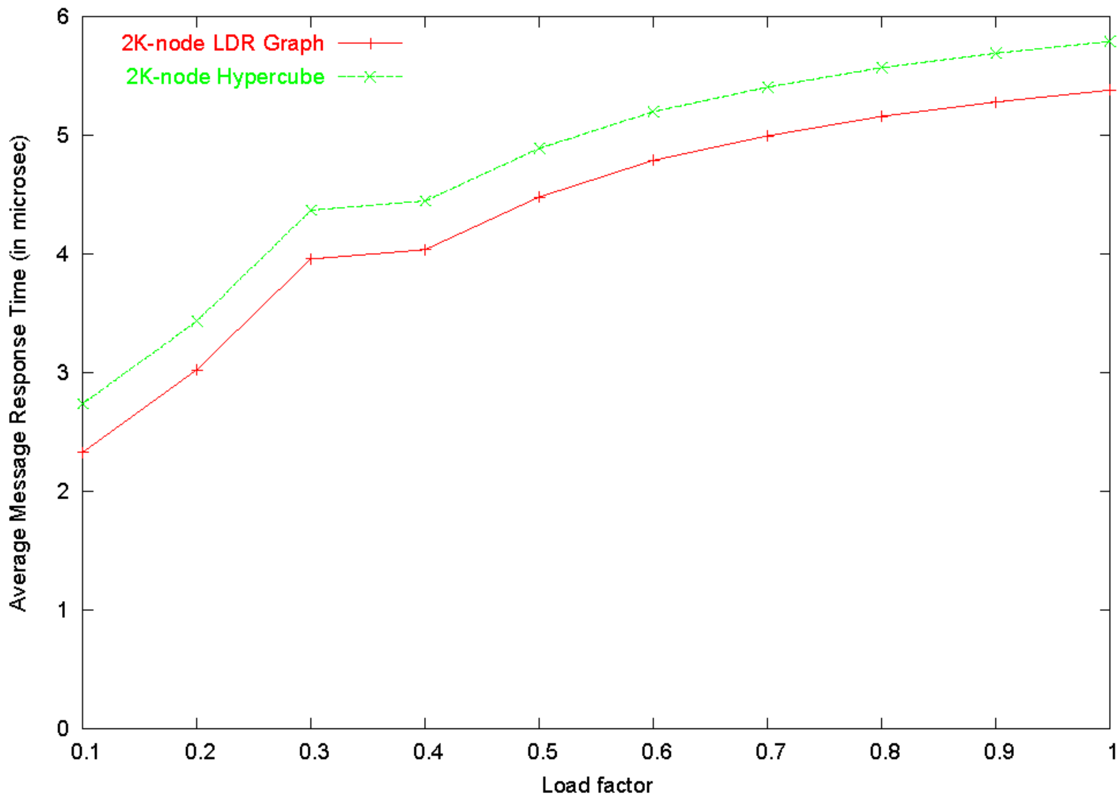


Figure 6.2. Message Response Time for broadcast on a 2048 node direct network

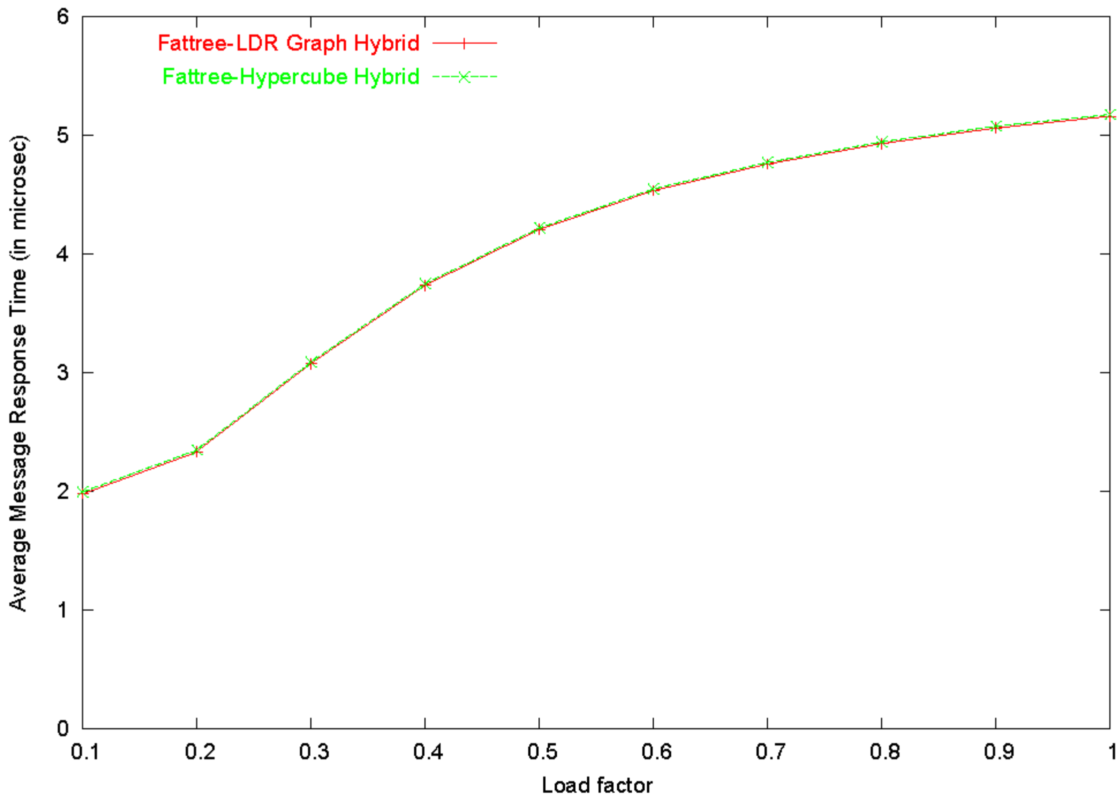


Figure 6.3. Message Response Time for broadcast on a 256 node hybrid network

Chapter 7

Conclusion and Future Work

This thesis describes the design, implementation and performance of an interconnection network topology based on low diameter regular graphs. We have compared this new topology with a hypercube having same number of nodes and same connectivity. Our simulations have shown that use of LDR graphs in 2048 node direct networks can lower the message latency by 25% as compared with the hypercube. In a 4096 node hybrid network, using LDR graph as the direct network component achieves a latency of about 8% lower than the corresponding hybrid network having hypercube as its direct network component. LDR graphs also achieve better collective communication performance.

Therefore, we see that lower diameter and lower average internode distance in the topology, along with an efficient routing algorithm, translates to a better communication performance on the network. Our experiments were done on a detailed contention-based network model. LDR graphs have proved to be an efficient and scalable network topology. Their performance makes a case for their use in the large-scale interconnection networks in the new and upcoming parallel machines.

There is a scope for improving the performance of LDR graphs further. It would be useful to incorporate a feedback during the generation of the graphs. We currently generate random graphs and pick the ones with the lowest diameter out of a few attempts. We could use feedback about link utilization to improve the graph structure from the communication perspective. For different random graphs, we could calculate a link utilization metric based on the number of times a link is used as part of shortest paths, and choose a graph that optimizes the metric.

The file interface for storing graph data limits the scalability. Replacing the entire path information by just the next node information allowed us to reduce file sizes and scale to up to $16K$ nodes. To scale to larger networks and run long parallel simulations, we could distribute the information so that each processor stores only parts of the graph instead of entire graph information.

With the generic nature of the BigNetSim framework, we could try different topologies which would optimize communication performance further.

References

[1] E. Bannai and T Ito. On finite moore graphs. In *J. Fac. Sci. Univ. Tokyo Sect. IA Math. 20*, pages 191–208, 1973.

[2] William James Dally and Brian Towles. *Principles and Practices of Interconnection Networks.* 2004.

[3] E.W. Dijkstra. A note on two problems in connection with graphs. In *Numerische Mathematik*, pages 1:269–271, 1959.

[4] Christopher J. Glass and Lionel M. Ni. The turn model for adaptive routing. *J. ACM*, 41(5):874–902, 1994.

[5] A. J. Hoffman and R. R. Singleton. On moore graphs of diameter 2 and 3. In *IBM Journal of Research and Development Vol. 4*, pages 497–504, 1960.

[6] L. V. Kalé and B. Ramkumar. D-trees: A class of dense regular interconnection topologies. In *Proceedings of the Frontiers of Massively Parallel Computation*, pages 207–210, Oct 1988.

[7] Laxmikant V. Kalé. The virtualization model of parallel programming : Runtime optimizations and the state of art. In *LACSI 2002*, Albuquerque, October 2002.

[8] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[9] Sameer Kumar and L. V. Kale. Scaling collective multicast on fat-tree networks. In *ICPADS*, Newport Beach, CA, July 2004.

[10] Sameer Kumar, Laxmikant V. Kale, and Craig Stunkel. Architecture for supporting hardware collectives in output-queued high-radix routers. Technical Report 05-02, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, Mar 2005.

[11] James Laudon and Daniel Lenoski. The sgi origin: a ccnuma highly scalable server. In *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*, pages 241–251, New York, NY, USA, 1997. ACM Press.

[12] C.E. Leiserson. Fat-trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Transactions on Computers*, 34(10), October 1985.

[13] M.Blumrich, D.Chen, P.Coteus, A.Gara, M.Giampapa, P.Heidelberger, S.Singh, B.Steinmacher-Burow, T.Takken, and P.Vranas. Design and analysis of the bluegene/l torus interconnection network. *IBM Research Report*, December 2003.

[14] Fabrizio Petrini and Marco Vanneschi. K-ary N-trees: High performance networks for massively parallel architectures. Technical Report TR-95-18, 15, 1995.

[15] Eric W. Weisstein From MathWorld-A Wolfram Web Resource. de bruijn graph. URL:http://mathworld.wolfram.com/deBruijnGraph.html.

[16] S.B.Akers, D.Harel, and B.Krishnamurthy. The star graph: An attractive alternative to the n-cube. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, 1987.

[17] Terry L. Wilmarth, Gengbin Zheng, Eric J. Bohm, Yogesh Mehta, Nilesh Choudhury, Praveen Jagadishprasad and Laxmikant V. Kale. Performance prediction using simulation of large-scale interconnection networks in pose. In *Proceedings of the Workshop on Principles of Advanced and Distributed Simulation*, page to appear, 2005.

[18] Terry Wilmarth and L. V. Kalé. Pose: Getting over grainsize in parallel discrete event simulation. In *2004 International Conference on Parallel Processing*, pages 12–19, August 2004.

[19] Terry L. Wilmarth. *POSE: Scalable General-purpose Parallel Discrete Event Simulation*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[20] Gengbin Zheng, Gunavardhan Kakulapati, and Laxmikant V. Kalé. Bigsim: A parallel simulator for performance prediction of extremely large parallel machines. In *18th International Parallel and Distributed Processing Symposium (IPDPS)*, Santa Fe, New Mexico, April 2004.

[21] Gengbin Zheng, Terry Wilmarth, Praveen Jagadishprasad, and Laxmikant V. Kalé. Simulation-based performance prediction for large parallel machines. In *International Journal of Parallel Programming*, number to appear, 2005.