Crosspoint Switches for Reconfigurable Networks

Edgar Pan

Department of Electrical and Computer Engineering

McGill University, Montreal

December 2020

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Master of Engineering

Table of Contents

Abstract

Network topology is an important factor in the performance of any distributed computing systems, such as data centers or supercomputers. Because supercomputers are expected to be reprogrammable and fulfill multiple tasks, their topologies are likewise typically general purpose, favouring no one task. However, certain applications could benefit from more specialized network topologies. One solution could be to have dynamic networks that can be reconfigured into more task-specific topologies. One way to achieve this is by introducing crosspoint switches, such as Reflex Photonics's Hybrid Optical Bridge, into the network. However, determining the placement of these switches is a nontrivial task. In this thesis, I wrote a program that determines how to connect crosspoint switches to the various nodes to effectuate a desired reconfiguration. The program follows a pipeline architecture that is divided into three modules: 1) the link delta acquisition process, 2) the chaining process, and 3) the selection process. In the link delta acquisition process, the program determines what reconfigurations are necessary. In the chaining process, the program creates a list of all possible switch configurations that satisfies the necessary reconfiguration. The selection process selects the most economical combination of potential switch configurations to achieve the desired reconfiguration. By making it possible to tune networks to more specific tasks, this could increase the computational efficiencies of servers as well as their power efficiency.

Résumé

La topologie d'un réseau est un facteur important dans la performance de tout système de calcul distribué, comme les centres de traitement de données ou les superordinateurs. Parce que les superordinateurs doivent être reprogrammables et doivent accomplir des tâches variées, leurs topologies sont également à usage général, sans favoriser aucune tâche. Cependant, certaines applications pourraient grandement bénéficier d'une topologie de réseau plus spécialisée. Une solution serait d'avoir un réseau dynamique qui pourrait être reconfiguré avec des topologies spécialisées pour chaque tâche. Cela pourrait être accompli en introduisant des commutateurs de point de croisement, tels que le Pont Optique Hybride de Reflex Photonics, dans le réseau. Cependant, déterminer les emplacements idéaux pour ces commutateurs est une tâche non négligeable. Dans cette thèse, j'ai écrit un programme qui détermine comment connecter les commutateurs de point de croisement aux différents nœuds pour accomplir la reconfiguration désirée. Le programme a une architecture de type pipeline divisée en trois modules : 1) la procédure d'acquisition de delta lien, 2) la procédure d'enchaînement, et 3) la procédure de sélection. La procédure d'acquisition de delta lien détermine quelles reconfigurations sont nécessaires. La procédure d'enchaînement compose une liste de toutes les configurations de commutateurs possibles qui satisfont les reconfigurations nécessaires. La procédure de sélection choisit la combinaison de commutateur potentiel la plus économique qui accomplit la reconfiguration désirée. En rendant possible d'accorder les réseaux à leurs tâches, cela devrait augmenter l'efficacité de calcul des serveurs et de leur efficacité énergétique.

Acknowledgement

Prof. Odile Liboiron-Ladouceur for her supervision and feedback over the course of the last three years, and for her help reviewing this thesis.

Edwin Pan for answering some questions on software architecture.

Prof. Michael Rabbat for reviewing and providing feedback on this thesis.

Dr. Edward Pan for his help in data analysis and for reviewing this thesis.

Contribution of Author

All of the work presented in this thesis is my own except where explicitly stated.

1 Introduction

One of the backbone elements of all large-scale internet system is the network interconnection topology. For all the performance improvements a single hardware device can have, its potential would be stifled should the data flow be bottlenecked by an inadequately configured network. This can be seen when one views the results of supercomputer performances on Top500, when one compares the theoretical maximum performance to the actual performance achieved. The top supercomputer at the time of writing, IBM's Summit, only achieved 148,600 TFlops/s compared to its theoretical 200,794.9 TFlops/s operation speed [1].

Given the importance of the network topology, the question now becomes which one to use? Ideally, networks would use a fully connected configuration, where every single node is connected to every other node, ensuring the only bottleneck is the capacity of the channel itself. However, when dealing with networks where nodes numbers in the hundreds to thousands, having a direct connection between all nodes is impractical. Thus, the exercise of selecting one of the many proposed network topologies becomes a question of trade-offs based on a set of restrictions.

The trouble arises when one considers the demand. Certain applications work better on certain topologies. One research showed that Distributed Machine Learning applications works better on a BCube topology than the typical Fat-Tree topology [2]. The purpose of the Fat-Tree, meanwhile, is essentially to avoid the issues of bottlenecking by ensuring multiple alternate paths are available and that all hosts have the same distance, ensuring a relatively robust network [3].

So how would one choose? One could take the BCube to do very well with Machine Learning applications, but the network would sacrifice its robustness and performance when it runs other applications. Would the sacrifice be worth it?

This thesis wishes to offer an alternative: "Why not both?" A network that can be reconfigured between two topologies would enable a more efficient use of computational

resources by matching application to a more appropriate topology. Introducing crosspoint switches, such as Reflex Photonic's Hybrid Optical Bridges (HOBs), into a network is one way to make it reconfigurable. However, determining where to place these switches in a network is nontrivial.

In this thesis, I describe a program that will determine where to place these reconfiguration switches. The program was developed in Matlab, and the code can be found both in Appendix 6.1 and on Github [4]. It takes as input two matrices describing the two desired configurations of a network and outputs how the reconfiguration switches should be connected, such that they can effectuate the transition between the two network topologies. The program functions under certain restrictions, which will be discussed in more detail later. The key two being that all nodes must maintain their degree and the crosspoint switches must not have any empty ports. Failure to meet these restrictions will cause the program to crash.

Before discussing the technical details of the program, this thesis will review the fundamental concepts behind network topologies along with a few popular examples. We will then discuss how different applications and synthetic traffic patterns perform under different topologies before we briefly discuss the HOB. With the background covered, this thesis will then discuss the program and its modules in detail, including their performance, known issues, and how they may be resolved in the future.

2 Background

In this chapter, we will first cover a few fundamental concepts of network topologies and their graph models. We shall then explore popular topologies commonly used in datacenters and supercomputers, their canonical structure, and primary benefits. We shall then cover the performance of those network topologies under different traffic patterns. Finally, we shall briefly cover the HOB as the means to allow for easy network reconfiguration between multiple topologies.

2.1 Key Concepts

A network topology is the pattern of physical interconnection between hardware resources, whether those resources are processing cores on a chip or server towers in a data center [5].

A network topology can be modeled with a mathematical structure known as a **graph**, denoted as $G(V, E)$, which consists of a set of **nodes (or vertices)** V interconnected by a set of **links (or edges/channels)** E. The elements of E consist of pairs (u, v) of distinct nodes $u, v \in V$. Translated into a real-life network, the nodes V typically represent the switches or routers in a network while the edges E represent the cables.

The **order** of a graph is the number of vertices a graph has, and the **size** of a graph is the number of edges. The number of edges originating from a single node denotes the node's **degree** (or a switch's **radix**). The **distance** between two nodes indicates the minimum number of links a signal needs to traverse (also referred to as **hops**) for the two nodes to communicate. The **diameter** of a graph indicates the longest distance in the graph. A **subgraph** is a subset of G, V, and E.

It is also important to note whether a graph is **directed,** meaning there is at least one link that can only be traversed in one direction, or **undirected**, meaning all links go both ways. Given the nature of telecommunication networks, it is safe to assume that all graphs are undirected, and thus any pair of connected nodes may communicate back and forth freely.

Two nodes or links are **adjacent** if they share a single link or node between them, respectively. We can therefore map out the full interconnection of a network through an **Adjacency Matrix** A. In an adjacency matrix, the rows represent the source (or ingress) node of a link and the columns represent the destination (or egress) node of that link. As such, the entries A_{ij} is equal to 1 if there exists an edge starting from vertex i and ending in vertex j, or 0 if there does not. In an undirected graph, the adjacency matrix is symmetric. The sum of row i represents the degree of node i: d_i.

There are a few notable kinds of graph, particularly the regular graphs and the bipartite graphs. If all vertices in a graph share the same degree d, then the graph is a **regular graph**, specifically, a d-regular graph. If two sets of nodes V_1 and V_2 only have links interconnecting between the two sets, but none intra-connecting within either set, as shown in the figure below, then these creates a **bipartite graph**.

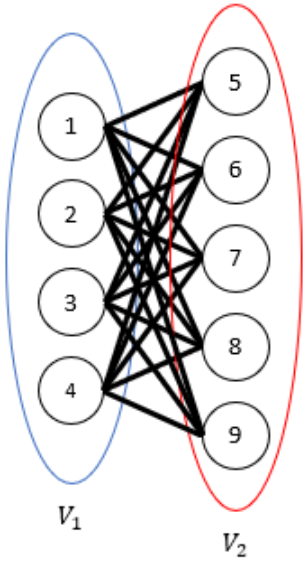

Figure 1: Example of a group of four nodes connecting to a group of five nodes in a complete bipartite graph.

2.2 Network Topologies

In this subsection, we will go over four popular networks topologies: the Hypercube, the Torus, the Fat-Tree, and the Dragonfly.

A network topology is usually described by a set of parameters and properties. For example, a Fat-Tree network is describe based on the switches' radix k, which then determines how large the network can be [3]. In practice, however, implemented networks seldom follow their mathematical definition. This can be due to a variety of reasons, including maintenance, hardware failure, or simply a desired change by the network administrators. All that said, this thesis shall assume that all topologies are implemented in their canonical forms. That means the topologies will abide to their definition as described in their respective publications as much as possible.

2.2.1 Hypercube and Torus

The Hypercube topology and Torus topology are two relatively basic regular network topologies in the field. Each node represents a router where servers are connected.

The Hypercube takes the corners of an n-dimensional (hyper)cube as nodes in a network. The number of nodes increases exponentially (2^n) while the diameter increases linearly with every axis (value of n) added to the network, leading to a highly connected network with a relatively low diameter, allowing for faster and more reliable communication [6, 7]. The nodes of a Hypercube network can in fact be represented by a binary sequence of length n, each bit representing one axis, as shown in the following figure.

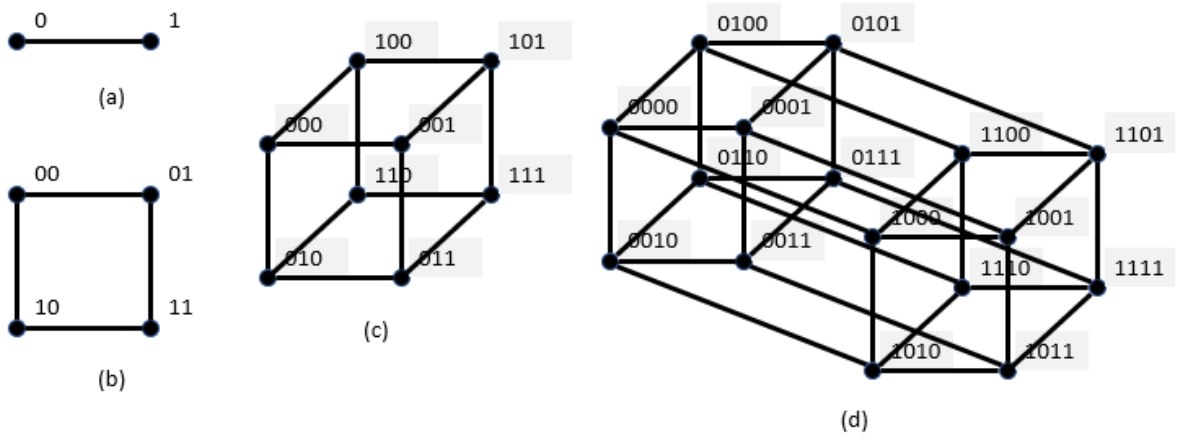

Figure 2: Graphs representing the n-D Hypercube with binary notation. (a) 1-D. (b) 2-D. (c) 3-D. (d) 4-D.

The main disadvantage of the Hypercube is that it is not scalable, nor is it practical to expand an already existing Hypercube network. The order of the network is *strictly* dependent on the degree of each nodes, meaning in order to expand a network from a radix-4 16-node network to a radix-5 32-node network, one would need to either replace all 16 radix-4 switches with 32 radix-5 switches, or unplug a host from each switches in order to re-assign the port for switch-to-switch communication. At which point, one may as well create a new network.

The Torus network is another relatively simple topology. The torus network can be described as a k-dimensional wrap-around mesh or grid network, where all nodes typically have two neighbors in each of the k dimensions [8]. Contrary to the Hypercube, the length of the Torus in each of the dimensions is arbitrary, yet uniform, meaning one needs not change the rest of the network to expand further in any direction.

The two most common variants of the Torus network seen in the top 20 supercomputers are the 5D Torus, which was used in the Sequoia, Mira, and JUQUEEN supercomputers developed by IBM, and the 6D Torus, which was used in the K Computer and SORA-MA supercomputers by Fujitsu [9].

The Hypercube and Torus network adjacency matrices can both be described by a line of Kronecker products and sums. The Kronecker product is a matrix operation where the values of one of the matrices is multiplied and inserted into the other, such that:

$$A_{m\times n} \otimes B_{p\times q} = \begin{bmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{bmatrix}_{pm\times qn} \tag{1}$$

where a_{mn} are the elements of matrix A. The Kronecker sum is a line or Kronecker products such that:

$$A_{m\times m} \oplus B_{p\times p} = A_{m\times m} \otimes I_p + I_m \otimes B_{p\times p} \tag{2}$$

where I_p and I_m are identity matrices of sizes p and m respectively. Let us define Q_n as the adjacency matrix of an n-dimensional hypercube. The Hypercube network adjacency

matrix runs on an induction sequence for however many dimensions the Hypercube will have. First, we define the 1-D "cube" (where $n = 1$) with the adjacency matrix:

$$Q_1 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{3}$$

That will serve as a basis upon which the matrix expands. By observing the adjacency matrix of consecutive-dimension hypercubes, we can see that the previous value of n reappears in the next value, as seen the in equations below.

$$Q_2 = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} & \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\ \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} Q_1 & I_2 \\ I_2 & Q_1 \end{bmatrix}_4 = Q_1 \oplus Q_1 \tag{4}$$

$$Q_3 = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} & \begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix} \\ \begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix} & \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} & \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & 0 \\ \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} & 0 & \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\ \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & 0 & \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} & \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\ 0 & \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \end{bmatrix} \tag{5}$$

$$Q_3 = \begin{bmatrix} Q_1 & I_2 & I_2 & 0 \\ I_2 & Q_1 & 0 & I_2 \\ I_2 & 0 & Q_1 & I_2 \\ 0 & I_2 & I_2 & Q_1 \end{bmatrix} = I_4 \otimes Q_1 + Q_2 \otimes I_2 \tag{6}$$

$$Q_3 = Q_2 \oplus Q_1 \tag{7}$$

As such, we find the recursive formula:

$$Q_n = Q_{n-1} \oplus Q_1 \tag{8}$$

The discovered formula has been implemented in the GenHND.m function found in appendix 6.1.2.

The Torus adjacency matrix will be represented by T and can be broken down into a set of basis matrices, B_n, where n is the dimension or axis. The value of B_n is the adjacency matrix of the nodes in that specific axis, best described as a wrap-around line. For instance, if a network has five nodes in the second axis, then we would denote the basis matrix as:

$$B_2 = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{9}$$

More generally, the elements of a basis matrix B_n of size $p_n \times p_n$ can be described as:

$$B_{n,(i,j)} = \begin{cases} 1, & i = 1, j = p_n \\ 1, & i = p_n, j = 1 \\ 1, & |i - j| = 1 \\ 0, & else \end{cases} \qquad \text{where } i, j \in 1,2, \dots, p_n \tag{10}$$

With each of the axis length and their corresponding basis matrices defined, then the basis will be Kronecker summed, forming the adjacency matrix.

$$T = B_1 \oplus \dots \oplus B_N \tag{11}$$

As an example, the 3×4 2D Torus basis matrices and adjacency matrix are the following.

$$B_1 = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}, B_2 = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \tag{12}$$

$$T = B_1 \oplus B_2 = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix} \quad (13)$$

This formula has been implemented in the GENTor.m function found in appendix 6.1.3.

2.2.2 Fat-Tree

The Fat-Tree as a data center network topology was initially proposed by Al-Fares, *et al.* [3], which was based on Leiserson's Fat-Tree designed to interconnect the processors of a general-purpose parallel supercomputer [10].

The original Fat-Tree takes the concept of a binary tree network and improves upon it. The binary tree is a topology where, from an originating "root" node, two nodes "branch" out from the root, and from each of those, more pairs of nodes "branch" out, every parent node forming two descendent nodes in each layer or generation. The problem occurs when two nodes simultaneously try to communicate with two other nodes on the other side of the tree due to bandwidth limitations. When the two nodes try to communicate up through the tree, there is only enough bandwidth in the cabling wire to handle one such communication, meaning that, at best, the performance is reduced by half. The Fat-Tree, therefore, resolves that by "thickening the trunk", by adding more wires in each links the closer to the root the topology gets, such that a parent node can freely transmit the information provided by both descendent nodes. This is illustrated in figure 3(a).

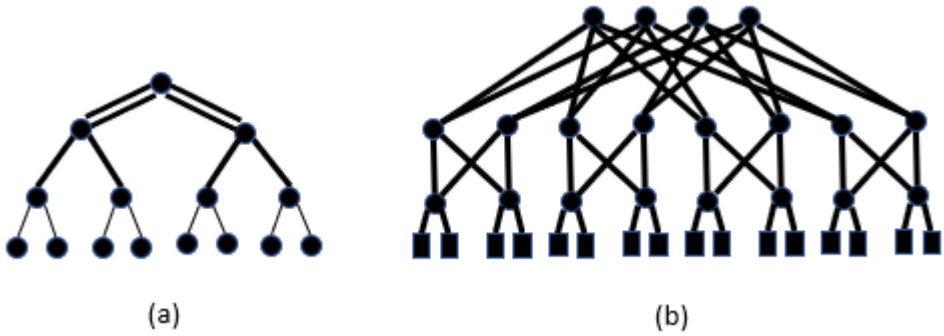

Figure 3: The two different types of Fat-Tree. (a) A 15-processor network. Note that the closer to the root it gets, the thicker or more numerous the wires are per channel, increasing bandwidth. (b) A typical 4-ary Fat-Tree network supporting 16 terminals at the bottom.

While it may be possible to reconfigure the nodes of a classic Fat-Tree, this was not explored. As such, we move on to the datacenter variant of the Fat-Tree proposed by Al-Fares, *et al. [3]*. Technically, the Fat-Tree is a "special instance" of a Clos network. The Clos topology is named after Charles Clos, who initially proposed a multi-stage and multi-leveled approach to configure switches for telephone networks [11]. Telephone networks rely on constant uninterrupted signals rather than the scattering of data packets across a network, meaning that most networks based on Clos's network are robust and consistent.

The premise behind the Clos version of the Fat-Tree is that instead of "thickening" the wires to a single parent node, the network splits that parent node into multiple "core" nodes. The cores are still connected "fully" to the individual nodes in the next layer in a bipartite graph.

Formally, the Clos Fat-Tree is defined as a k-ary Fat-Tree, where k is the radix of the individual switches or nodes of the network. The topology is split into three layers: the Core layer and the Pod layer, which is subdivided into the Aggregation layer, and the "Edge" layer which includes the terminals, as shown in figure 3(b). For the sake of clarity, in this subsection, "edge" will refer to this layer, rather than a topology's channel or link.

The Clos Fat-Tree consists of k pods containing two layers of $\frac{k}{2}$ nodes each connected in a bipartite subgraph. For the edge layer's switches, $\frac{k}{2}$ ports connect to the hosts while $\frac{k}{2}$

ports connect to the aggregation layer. In the aggregation layer, $\frac{k}{2}$ ports are connected to the edge layer while $\frac{k}{2}$ ports connect to the core layer outside the pods. The core layer consists of $\left(\frac{k}{2}\right)^2$ nodes, each connecting to one of the aggregation nodes of every pods. In the figure below, we see an example of a 6-ary Fat-Tree.

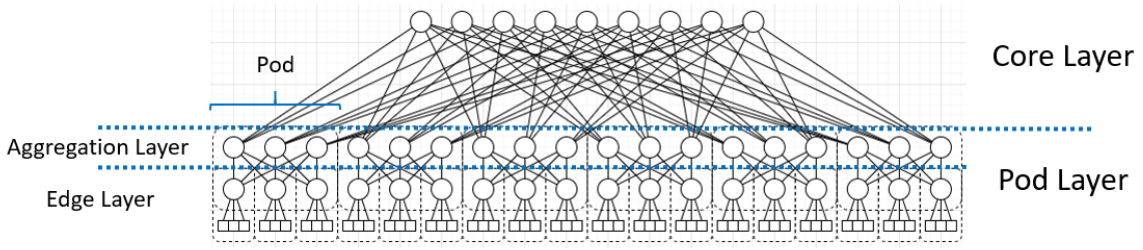

Figure 4: A 6-ary Fat Tree supporting 54 terminals (represented by squares). Note that there are 6 pods, each containing 6 routers (represented by circles) connected in a bipartite subgraph.

This results in a topology with a diameter of six hops, an order of $\frac{5}{4}k^2$, and capable of supporting $\frac{k^3}{4}$ hosts or terminals. This means that while the topology makes use of a fairly large number of routers, it should have very reliable throughput performance in exchange.

In summary, the Binary Tree topology consists of a parent node splitting into two (or more) descendent nodes at every "generation" layers. The "classic" Fat-Tree "thickens the trunk" by adding more bandwidth in the older generations, typically by adding more wires between nodes. The Clos Fat-Tree returns to the uniform inter-nodal connection of the binary tree by splitting the Fat-Tree's single core into multiple core nodes, forming the backbone of the topology.

2.2.3 Dragonfly and its Variants

The original Dragonfly topology had been proposed by Kim, *et al.* [12] in 2008. The purpose of the dragonfly topology is to address two issues in optical networks: the limited radix capabilities of optical switches and the high cost of optical cables required in a system, as shown in the following figure.

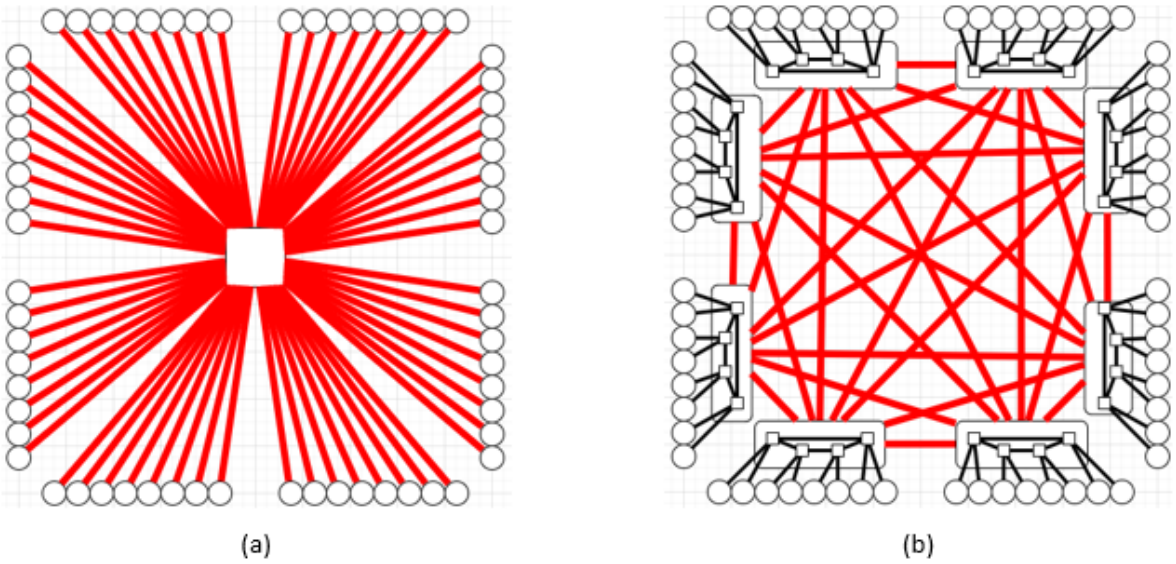

Figure 5: Two 64-node networks. (a) is an ideal network with each node separated by 2 hops. One can see how there are therefore 64 long global channels which can get expensive. (b) The Dragonfly network is a more practical implementation of the desired network, where a cluster of shortrange cables and routers creates the 8 “supernodes” and reduces the number of global channels needed to 28.

At present, optical switches typically work by cascading several dynamic optical couplers (effectively 2-by-2 switches for optics) in a manner to allow for switching at a higher effective radix. This means that the larger the radix, the more switches are cascaded. Because these switches use thermal-optic effects, each coupler can experience switching times in the microseconds, which when cascaded would be unsuitable for switching on short packet timescales. Thus, the maximum radix of an optical switch is widely considered 16 [13].

As such, the Dragonfly topology seeks to resolve that by creating high-radix *virtual routers* by grouping up low-radix routers. In other words, using a group of small routers to effectively create large “virtual” routers. For example, with four radix-8 switches, one can effectively form a single radix-20 node by reserving 3 ports on each of the switches for intra-group connection, as seen in figure 6. The other five may be used either connect to hosts or other such clusters. This results in a “virtual” fully connected network of a much smaller order.

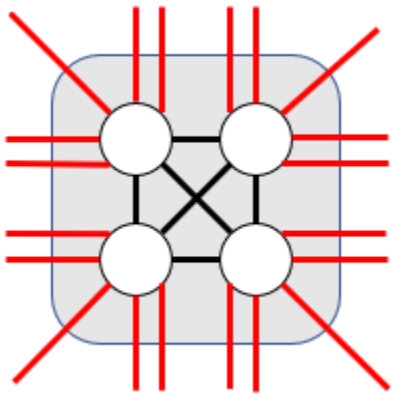

Figure 6: A Virtual Radix-20 Node created by 4 radix-8 switches. Black lines are short range intra-connection while Red lines are global inter-connections. While no port is dedicated to terminals in this example, one could easily have reallocated one of the global channels to terminal connections.

The Dragonfly topology is divided into three hierarchical levels. First, we have the individual routers with a radix of k. Those ports are divided into three purposes: p ports are reserved for the hosts/terminals, $(a-1)$ ports are reserved for the local, short-ranged channels, and h ports are reserved for global channels, such that $k = p + h + a - 1$.

One level higher, we have the *Group*, which consists of a routers. A group is connected to ap hosts/terminals and has ah global channels. This results in an effective *virtual radix* of $k' = a(p+h)$, where $k' \gg k$. This is the key property of a given dragonfly network, because it defines how the smaller routers (nodes) creates the larger virtual routers (supernodes). Canonically, the routers within the groups are fully connected, meaning that for a terminal to reach the appropriate global channel, only a single small hop is needed.

Finally, there is the *System* level. Here, the Groups typically form a fully connected global network, resulting in a global diameter of 1. In a canonical Dragonfly, there are $g = ah + 1$ groups, with only one connection between groups, which can support up to $N = ap(ah+1)$ terminals.

It should be noted that while the original proposal uses electrical cables for short-distance communications, namely in the intra-group communications, short optical cables

can still be used. Even if this negates the benefit of reducing the cost of optical cables, it still addresses the issue of the limited radix of an optical switch.

While Kim, *et al.* recommend enforcing the restrictions of $a \geq 2h$ and $2p \geq 2h$ in order to ensure traffic performance does not suffer, in 2017, Teh, *et al.* have conducted a study on the effect of varying the parameters of a Dragonfly network [14].

One notable variant of the Dragonfly topology was Shpiner, *et al.'s* Dragonfly+. While in the original Dragonfly, each level is fully connected, in the Dragonfly+, the intra-group connection takes a Clos-like topology [15].

The Dragonfly+ can best be summed up by a Fat-Tree topology but without a core layer. Rather than have a fully connected uniform cluster of routers within each group, they are now organized in two bipartite layers: spine and leaf. This resembles the pods of a Fat-Tree topology. As such, one could even reconfigure a Fat-Tree to a Dragonfly+ topology for the sake of bypassing or decreasing the load on the core routers in lower density traffic and even disabling the core routers for maintenance purposes as shown in the following figure.

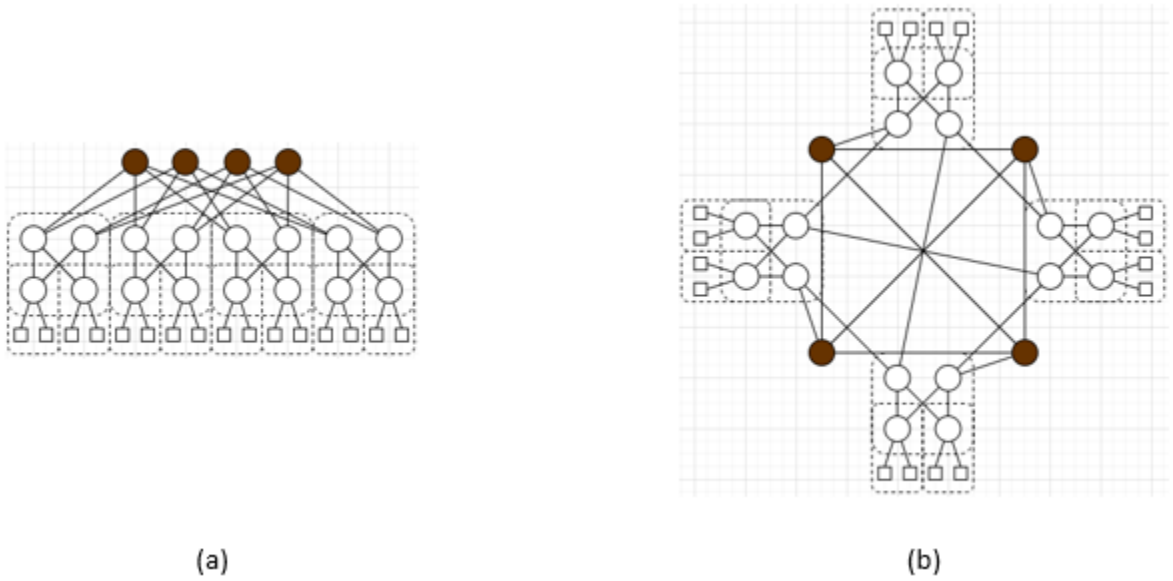

Figure 7: A 4-ary Fat-Tree network (a) reconfigured to a pseudo-Dragonfly+ network in (b). The Core routers are highlighted.

2.3 Traffic Patterns and Applications in Various Topologies

In this section, we shall examine the recorded operational performance in the November 2017 Top500 supercomputer competition, which is compiled in appendix 6.2. This contest was significant because it was the first time an alternate supercomputer benchmark test was also used. We examine the top 20 supercomputers of each list, noting their performance in each benchmark test. We then note whether a particular topology does better than the average in one test while another does better than the average in another test. Afterwards, we examine Jyothi, *et al.'s* experiments and their results for the relative performance of various topologies in data centers.

2.3.1 Performance of Supercomputer Processors

Supercomputers are typically identified by the sheer power and number of processing cores they possess, and have their performance measured by the number of Floating-point Operations they can perform per second (Flops/s).

At present, there are three high-performance (HP) benchmark tests executed on general-purpose supercomputers: Linpack, Congruent Gradient, and Green.

The High-Performance Linpack (HPL) benchmark test was the original standard supercomputer test for the Top500 supercomputer list since its creation in June 1993 [16]. It tests the performance of supercomputers when solving general dense matrix problem $Ax = b$ for three problem sizes: 100 by 100 (inner loop optimization), 1000 by 1000 (full program with three loop optimization), and a scalable parallel problem [17]. The loop refers to the process in which the large matrix is broken down and Basic Linear Algebra Subroutines (BLAS) are called to solve them in more manageable forms [18]. In other words, the Linpack primarily tests the supercomputer's ability to factor and solve a large dense system of linear equations using Gaussian Elimination with partial pivoting [19]. Dongarra, *et al.* refer to the traffic pattern of the HPL, with its relatively low need to access data, as a Type 2 pattern.

The High-Performance Conjugate Gradient (HPCG) test serves a similar purpose as the Linpack test; however, it was meant to better reflect modern real-life applications by

focusing more on Partial Differential Equations (PDEs), which better model aspects of the physical world. To solve those problems, the HPCG program uses iterative methods, which slowly approaches a solution, assuming it converges for a given initial value. Specifically, the program primarily measures how quickly it executes Krylov subspace solvers on distributed memory hardware [19]. Key to our purposes was that it was meant to serve as a complementary benchmark test to the HPL with different demands on the system. Many important scientific calculations have low computation-to-data-access ratios, meaning that most scientific calculations require a lot of data access (Dongarra, *et al.* refer to this traffic pattern as Type 1). In other words, there are more frequent network demands in the HPCG test while HPL focuses more on the individual core's ability to execute floating point operations [19]. Despite the difference in emphasis, both test programs still distribute their operations throughout the network. With its different demand, it would result in a different traffic pattern. As such, one may find a correlation between a type of topology and the relative performance.

The Green test evaluates the power efficiency of a supercomputer, namely the number of Flops it can perform for every Watt of power. While important, the Green test shall be mostly disregarded in this thesis because we focus more on the effect of the network topology on the performance. While it may be possible for topologies to be a factor in power consumption, that has not been examined.

The importance of a good topology becomes very evident when one examines the performance of some of the top supercomputers. In 2017, the Sunway TaihuLight supercomputer possesses 10,649,600 cores distributed over 40,960 CPUs (each with 260 cores) with a theoretical maximum performance of 125,436 TFlops/s, yet it only managed to achieve 93,014.6 TFlops/s in the HPL test (74.2% efficiency) and 481 TFlops/s in the HPCG test (0.4% efficiency). The average performance in the rest of the top 10 supercomputers of 2017 was 65.3%, ranging from 50.3% to 93.2%. As such, much of Sunway's position as top supercomputer of 2017 can be attributed to the sheer quantity of processing power the system possessed and not its overlap performance. The significant difference in performance can

also be attributed to the tendencies of supercomputer engineers to design their supercomputers around the HPL test, where it primarily exhibits the Type 2 traffic pattern while having little of the Type 1 traffic patterns, and thus will not account for any programs that do exhibit Type 1 traffic patterns in their designs. The issue is further exacerbated with the development of accelerators, which makes CPUs extremely effective with Type 2 patterns, but only barely support Type 1. Dongarra, *et al.* brought up the Oak Ridge National Laboratory's Titan system passing its floating point operations to its GPU as an example of the use of accelerators [19].

What remains is determining how much of an impact the different topologies have on the relative performance of the HPL and HPCG tests. To determine this, I took the top 20 supercomputers in both the HPL list and HPCG list of 2017 and noted their topology archetype, the number of CPUs (nodes) in the system, the theoretical maximum processing speed (Rpeak), and the achieved processing speed (Rmax or R). The performance efficiency is determined by how much of the theoretical maximum speed the system has achieved in percentage. The compiled data obtained from the Top500 website [9] are shown in Appendix 6.2.

The average efficiency of the HPL test is 70.3% with a standard deviation of 16.1% while the average efficiency of the HPCG test is 1.7% with a deviation of 1.0%. Due to the high disparity between the performance of supercomputers running the HPL test versus the HPCG test, it is a bit more difficult to compare the results. As such, we compared the relative performance to the test average, which is determined by the efficiency compared to the average efficiency. Based on that, we determined whether a particular topology is better for a particular benchmark test. The following graphs shows the efficiencies and the average values.

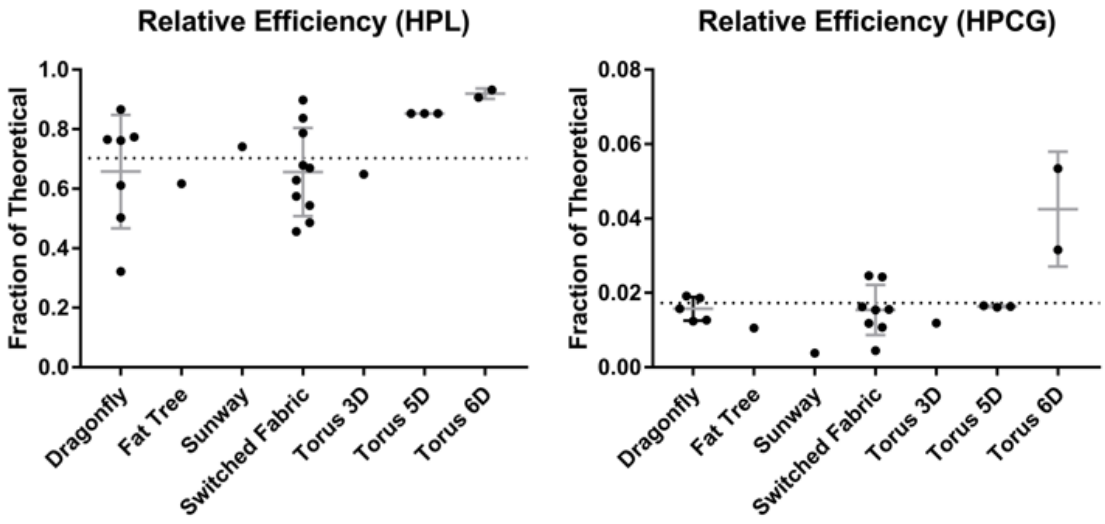

Figure 8: Relative Efficiencies of Top 20 supercomputers with respect to their maximum Flops/s performance under HPL and HPCG benchmark test with average line and standard deviation marked.

Unfortunately, no clear correlation between topology types and benchmark test could be established. In fact, one could even make the determination that the Torus 6D topology simply outperforms *all* others. The average for most topologies maintains its position relative to the average performance. While the Sunway has a significant drop when going from the HPL to HPCG test, its topology is unique (or at the very least, unknown since no publication detailing its structure could be found) so we cannot make any empirical determination from it.

Likely, the sample is also too small to truly determine whether the topology was the main factor or if some other property of the supercomputer itself was detrimental to its ability to process either benchmark test. It is also possible that despite the differences in tasks the HPL and HPCG benchmark tests each have, their traffic demand may be similar enough to experience relatively no difference in performance efficiency from a topological perspective.

2.3.2 Performance in Data Center Networks

Jyothi, *et al.* conducted a study on the throughput performance of various network topologies under different traffic patterns [20]. They were attempting to find a standardized method to measure all network topologies. One of their main challenges was the lack of

access to data from real-life data center networks, specifically on the type of traffic and the structure.

As such, the authors simulated various synthetic traffic patterns on multiple topologies to create a "near-worst-case" traffic matrix that will yield the lowest throughput. Similar to the adjacency matrix, the Traffic Matrix T (TM) defines a traffic demand, where in $T(i,j)$, node i requests a certain amount of flow to node j. The TM is usually normalized such that $\forall i, \sum_j T(i,j) \leq 1$ and $T(j,i) \leq 1$. Note that the capacity of a single link is 1. As such, the throughput is the maximum value t for which $T \cdot t$ is feasible in the network graph. This is usually formulated and solved as a Linear Program, an optimization problem.

The three synthetic traffic pattern the authors used were the All-to-All, where all nodes tries to communicate with all other nodes at the same time; Random Matching, where each node tries to communicate with another randomly assigned node such that every node has one input and one output signal; and Longest Matching, assigns to each node whichever node has the longest distance. The results of their simulations, shown in the table below, are of interest to us because they demonstrate our premise.

Table 1 Relative Throughput at the Largest Size tested under Different TMs (recreated from [20].)

Topology Family	All-To-All	Random Matching	Longest Matching
BCube (2-ary)	73%	90%	51%
BCell (5-ary)	93%	**97%**	79%
Dragonfly	**95%**	76%	72%
Fat-Tree	65%	73%	**89%**
Flattened BF (2-ary)	59%	71%	47%
Hypercube	72%	84%	51%

The relative throughput is meant to represent the throughput quality compared to a random graph with similar resources to nullify the basic advantage of simply having a higher connectivity within a graph. In other words, they generated a random graph with the same number of nodes and each of them with the same number of links [20]. The key results here

are the Dragonfly under All-to-All traffic, 5-ary BCell under Random Matching traffic, and Fat-Tree under Longest Matching traffic. They show that under very different traffic demands, certain topologies are more efficient for certain traffic patterns, or applications.

The structure of BCell and Fat-Tree networks both have restrictions in the server placement [20], and to a lesser extent, so does the Dragonfly (specifically, the Dragonfly+ case). Because both have server placement restrictions, reconfiguring from one to the other should still be practical to achieve.

2.4 The HOB Device

For the most part, this thesis does not concern itself with the technical details of the devices to be utilized, only in its operation and how it may affect the topological interconnection of a system. Nevertheless, it is still important to have a general understanding of the devices involved, including how it can be expanded upon in the future.

The Hybrid Optical Bridge (HOB) is a 12-by-12 optical signal regenerator and redistributor developed by Reflex Photonics. In other words, it is a 12-by-12 optical crosspoint switch, as shown in the following figure. However, it is worth keeping in mind that it is also a signal regenerator, meaning that it converts an optical signal to electric signals and then recreates it as a fresh, retimed non-degraded optical signal. That means variations with higher or lower numbers of ports and capacity are entirely possible.

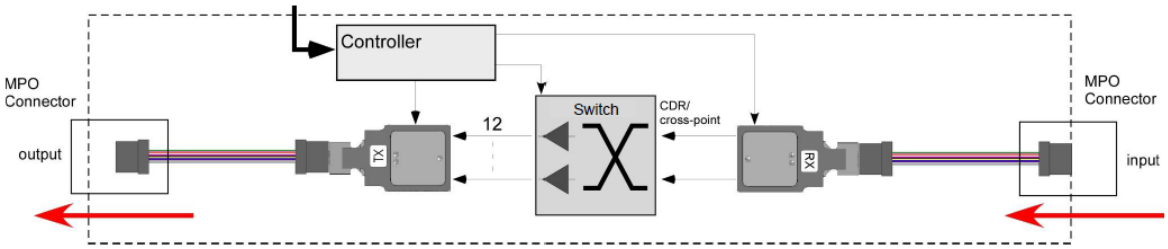

Figure 9: Schematic of the HOB base design provided by Reflex Photonics [21]. Optical signals are converted into electrical signals before being sent through a crosspoint switch. Then the signal is regenerated and transmitted as fresh, retimed non-degraded optical signals.

The configuration of the crosspoint switch is controlled via either a USB or WiFi controller, which would allow multiple switches to be controlled simultaneously from a single control terminal. As such, an operator or a centralized controller can remotely

reconfigure the network to better suit their purposes. These purposes can be to better accommodate a task for supercomputer networks, to better accommodate traffic demands in a data center network, or even to temporarily reroute traffic and isolate specific routers for the sake of maintenance.

The HOB had a throughput of 336 Gbps across the 12 channels back in 2017 [22], meaning each channel has a capacity of 28 Gbps. In order to achieve higher capacities, multiple channels can be bundled into a single larger channel. In fact, Reflex Photonics proposed a 3-port configuration where a port consists of four channels providing 112 Gbps of throughput between any two ports, as shown in the following diagram [21].

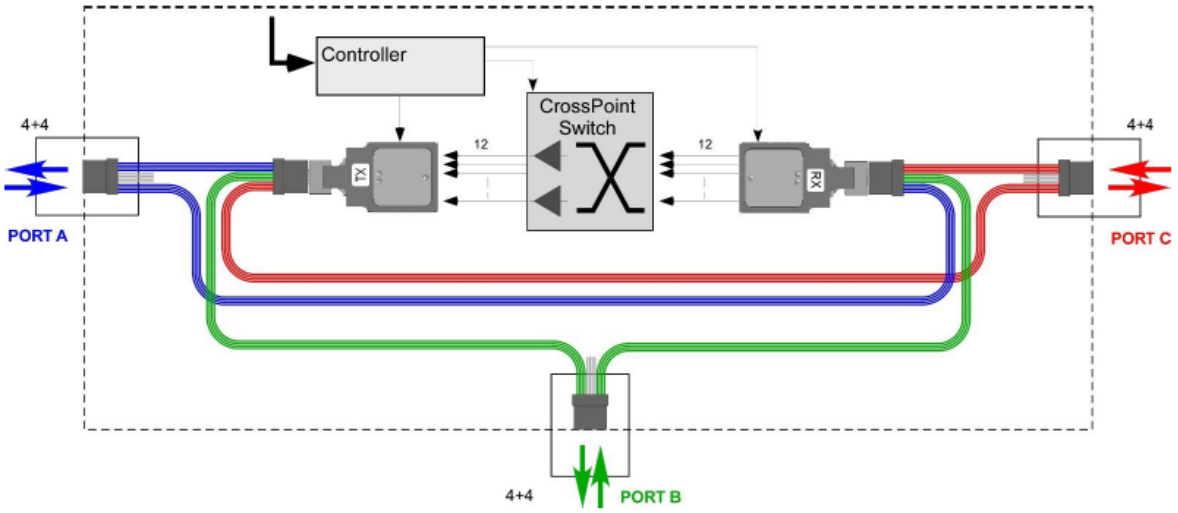

Figure 10: Reflex Photonic's proposed 3-port HOB configuration [21]. Four channels are reserved for each port and can be freely interconnected to any ports.

Such a configuration would allow for a much simpler "Port A connects exclusively either to Port B or Port C" setup. In other words, one would only need to place the HOB at every reconfiguration event (places where reconfiguration would occur) to achieve topological reconfiguration. However, this would leave four channels idle, wasting 112 Gbps of its potential throughput. The goal then is to configure the switch in such a way that all channels will always be used.

As such, while this may decrease the bandwidth of the individual connections to three channels (84 Gbps), we would recommend at least a 4-port (or radix-2) configuration, such

that Port A and Port B can freely connect between Port C and Port D, as shall be discussed later in this thesis.

One thing that should be clarified, the earlier diagram of the 3-port configuration implies that the crosspoint switch within the HOB is used unidirectionally. In other words, each port has to connect on both sides of the crosspoint switch in order to have both an input and output. However, by most conventions, crosspoint switches are bidirectional meaning that even when diagrams show only one channel, there are two, one for each direction.

In this thesis, we shall assume that all crosspoint switches are bidirectional, as this is a requirement to the algorithm that will be presented. However, even if unidirectional switches are used, it is possible to emulate bidirectional switches by reserving half the ports of a unidirectional crosspoint switch for backwards communications which effectively halves the radix, as shown in the following figure. In 11(a) all lines are bidirectional while in 11(b), communication can only occur from left to right, or right to left, but still allows for the same connectivity as in 11(a). A consequence of emulating bidirectional switches with unidirectional switches is that they could allow for connection of nodes that in a bidirectional switch are located on the same side. For our purposes, this will be disregarded. Since bidirectional switches have no true input or output ports, we shall refer to either side of the switches simply as the left or the right side.

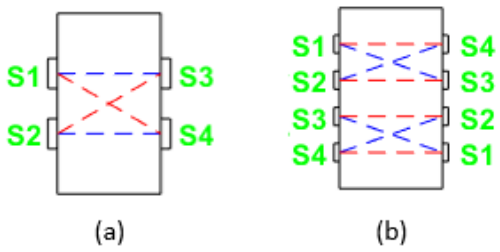

Figure 11: Comparison of (a) two-way and (b) one-way crosspoint switch configuration.
Bundled channels are shown as only one link in diagram.

Finally, while Reflex Photonic's HOB currently only possesses 12 channels, it is entirely possible to resize the HOB to a larger radix. Recall that one of the main limitations to optical switching was the relatively slow switching speed of the thermal-optic effects. With

the electrical crosspoint switch at the core of the HOB, we remove the optical limitations and thus allow for a greater number of channels to be connected at once, giving us more options.

As such, we shall assume the following: the reconfiguration switches can be of any size, the reconfiguration switches will be bidirectional, and the two sides of the switch will be interconnected in a bipartite manner.

3 Implementation of the Node Group Synthesis Program

To reconfigure a network from one network topology to another, the HOB or similar crosspoint switches can be used. To determine where best to place the reconfiguration switches, I have written a program that will analyze a set of adjacency matrices representing the desired topology configurations and return a list of **Node Group Pairs** (**NGPs**, shorthanded as **Group Pairs** or **GPs**). Each GPs represents a crosspoint switch and a group of nodes connected into each side.

The program follows a pipeline architecture, where the program is divided into independent and replaceable parts or **modules** arranged in a chain such that the output of a preceding module acts as the input of the following module. In this case, the program is divided into three modules: 1) the Link Delta Acquisition Process, where multiple adjacency matrices of the same order are evaluated to determine the potential and necessary changes in links to effectuate a reconfiguration from one described topology to another; 2) the Chaining Process, where the link deltas from the previous step are iteratively chained together to form a list of every possible non-redundant Group Pairs; and 3) the Selection Process, where a final set of Group Pairs is selected based on the amount of necessary link contributions and non-overlapping links each Group Pairs contribute. The program also has an optional fourth and final stage helpful in the visualization of the final reconfiguration, where the selected Group Pairs are drawn on both a system-wide basis and the individual switch configurations.

The operational conditions at this point of the program's development are as follows:

1. There can only be two topologies. The current methodology the program uses can only handle systems swapping between two topologies. This is due to the implementation of the program, which will be discussed in more details in section 3.2. While it may be possible to achieve network reconfigurations comprising of more than two topologies by having reconfiguration switches interconnect with other reconfiguration switches, this feature is not yet supported. At present, the program treats the HOB crosspoint switches as an invisible

component in the network—a component meant only to reconfigure a network. It does not, therefore, handle situations where the HOB devices themselves are nodes in the network. These would only occur in situations where a group of nodes must be able to freely interconnect with another group of nodes, such as in reconfigurable optical dragonfly proposed by Samadi, *et al.*, where any node in a supernode can connect to any node in another supernode [23], as shown in the following figure.

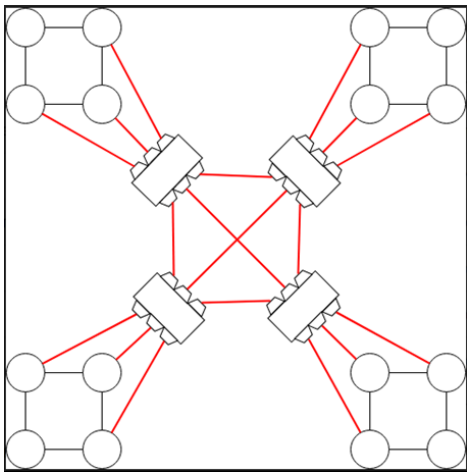

Figure 12: In Samadi, *et al.'s* reconfigurable dragonfly, the central four switches connect to each other, and thus would likely need to be mapped as nodes of the network.

2. All nodes must maintain their degree. All used ports on a router or server must be used and remain used, and therefore all links must go somewhere when reconfigured. In other words, if a server was connected to two other servers in the first topology, it must be connected to two servers in the second topology. This reflects both the reality that routers have a finite and constant number of ports and the practicality of making use of all available resources; there is no point in leaving a cable or port idle simply to fit the canonical description of a specified topology, excepting scenarios where one splits a larger network into sub-networks as described earlier.

3. All relevant edges must be defined. There is no point in defining a network by the likelihood that there is a link between two nodes. We are dealing with manually reconfiguring a network between two specific topologies and the necessary link reconfigurations to make that happen.

4. All GPs must be full. More specifically, a user must use a radix value such that it can be filled without leaving any empty ports. This is a consequence of the implementation of the program, where the Chaining Process would deem any GP combination incapable of reaching the desired radix unviable and thus not forward it to the next module.

There is also a fifth non-critical condition, in that the program would still function, but the result becomes significantly less efficient. The number of necessary link changes (or link deltas as we will call them later) must be a multiple of the radix of the switch. For example, if a system requires 16 reconfiguration events (link deltas, which will be defined in the next section), reliable radix values are 2, 4, 8, and 16.

3.1 Link Delta Acquisition

The first step is the Link Delta Acquisition Process, implemented in GenConnList.m (appendix 6.1.4). We define a **link delta** (dL) as a pair of adjacent links where one is unique to topology A and one is unique to topology B relative to their common node (referred to as source node). For instance, suppose that we have a network with nodes $\{1,2,3,4\}$ as shown in the following figure. In topology A, we have links $(1,2)$ and $(3,4)$. In topology B, we have links $(1,4)$ and $(3,2)$. The link delta would then be notated as $(1, 2 \leftrightarrow 4)$ where source node 1 links to destination node 2 in topology A and destination node 4 in topology B, and $(3, 4 \leftrightarrow 2)$ where node 3 links to node 4 in topology A and node 2 in topology B. Note that we can also define the link deltas relative to all source nodes. For example, we can also describe $(2,1 \leftrightarrow 3)$, which is also a valid link delta.

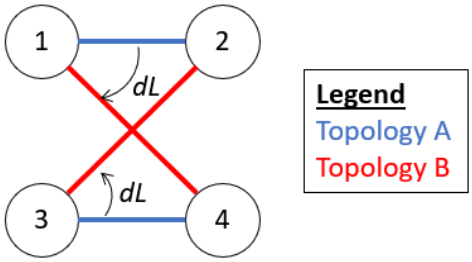

Figure 13: Topology reconfiguration of four nodes with links (1,2) and (3,4) being reconfigured to (1,4) and (2,3)

It is worth clarifying that a link delta merely describes a potential way for a link to be reconfigured for every node. It does not necessarily refer to the exact reconfiguration that occurs in the final setup. In cases where each node requires multiple link reconfigurations, there will most likely be more link deltas than necessary to effectuate the network reconfiguration. To demonstrate, suppose we have a network with nodes {1,2,3,4,5} as shown in the next figure. In topology A, we have links (1,2) and (1,4), while in topology B, we have links (1,3) and (1,5). As such, there are four possible link deltas: $(1, 2 \leftrightarrow 3)$, $(1, 2 \leftrightarrow 5)$, $(1, 4 \leftrightarrow 3)$, and $(1, 4 \leftrightarrow 5)$, even though only two are necessary to go from Topology A to Topology B.

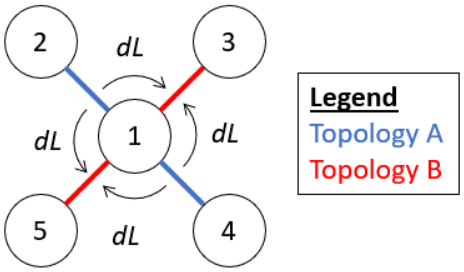

Figure 14: A demonstration of all possible link deltas *dL*, even if only two reconfigurations would occur.

At this point in the process, we only concern ourselves with generating a comprehensive list of potential link deltas dL for every node of a topology. The specific selection of which to use will be decided at a later stage.

The program code is included in appendix 8.1.4. It takes a three-dimensional matrix as input, each layer representing the adjacency matrix of each desired network configuration. The process then identifies and extracts the links common to all topologies and therefore do not require any reconfigurations. This allows the process to then identify the links exclusive to each topology.

Once every link exclusive to each topologies have been identified, we then scan through each node, or rows or columns in the adjacency matrices, and identify their respective exclusive links. Each topology has a set of destination nodes, and we create a set product (or cartesian product) that provides every possible combination of link deltas.

The set product or cartesian product is a set theory operation where the elements of two (or more) sets are combined into every possible ordered pairs (or n-tuple). In mathematical notation, $A \times B = \{(a, b)|a \in A, b \in B\}$.

While the setprodcell.m function (appendix 6.1.5) can cross-product a multitude of sets (i.e., possible configurations), the program should only take two adjacency matrices as input, otherwise it will fail to give an appropriate output.

Finally, GenConnList.m outputs a list of potential link deltas (*dL*) and necessary links (*Nec*). The necessary links list is a list of every link in every link delta. This serves as a list the program will check against to ensure that every needed links has been effectuated in case the program needs to compromise and create new links that have not been defined explicitly. This was originally meant to remove the necessity of explicitly defining all links and allow the program to autonomously create "dummy" links, which would allow the program to direct incomplete link deltas that have nowhere to go. Such a feature was unfortunately not implemented, but we shall still cover the intent.

Suppose we take the earlier subset of nodes {1,2,3,4,5} illustrated in figure 13, but this time, we do not define (1,5) as a link explicitly part of Topology B, as shown in the figure below. In other words, the link (1,5) is not a necessary link in this topology. That means that when it comes time to select link deltas, we will be forced to select both $(1, 2 \leftrightarrow 3)$ and $(1, 4 \leftrightarrow 3)$, which is redundant since both contributes the link (1,3). As such, we can still create a "dummy" link (a link that does not exist in the formal definition of either topologies, but does exist as a fully functional link in practice in order to satisfy the second condition) of (1,5) which allows us to create the "dummy" link deltas of $(1,2 \leftrightarrow 5)$ and $(1,4 \leftrightarrow 5)$ which will only be selected if it allows for a better selection at the end.

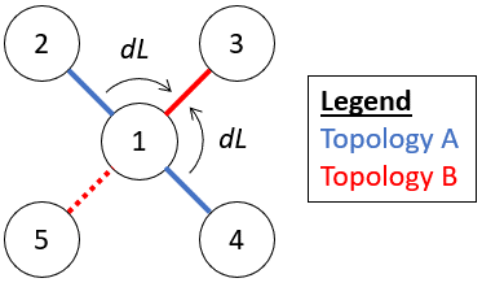

Figure 15: Two link deltas colliding when link (1,5) is absent.
Could be resolved by adding a dummy link between 1 and 5.

As the program is currently, the necessary link component only serves to verify the results. However, the data is available for future expansion. Additionally, this process can only reliably manage two topologies. While the function is capable of handling more, the program has not yet been designed to handle such a scenario.

3.2 Chaining Process

The second step is the Chaining Process implemented in ChainPairs2.m (appendix 6.1.6). This is where a comprehensive list of every possible Node Group Pairs is generated. Recall, the **Node Group Pair** refers to the set of nodes connected to each sides of a reconfiguration crosspoint switch (i.e., the HOB), such that the left Node Group refers to the nodes connected to the left side of a switch and the right Node Group refers to the nodes connected to the right side of a switch. A GP will be called **full** when every port on both sides are connected to a node.

The goal is to create a comprehensive list of every possible full GPs that will contribute to effectuating the network reconfiguration, a master set. One can then select a subset from this master set to create a solution set based on specified restrictions or requirements.

The function generates each GPs by chaining a collection of link deltas. By following the ‘flow’ of changing links, one can chain together a sequence of link deltas that eventually loops back to the ‘initial’ link, as shown in figure 16. The idea is to configure crosspoint switches that need only shift its link over by one port to effectuate the desired network reconfiguration. In other words, in Topology A, all ports on the left side of the crosspoint

switch would connect to the ports across them horizontally, and in Topology B, they shift their connection to the next port, except for the bottom left port, which would loop back around to connect to the top right port. The program will arrange the order of the nodes chosen on both sides of the switch such that all switches only need to follow this connection shift pattern to effectuate the network reconfiguration.

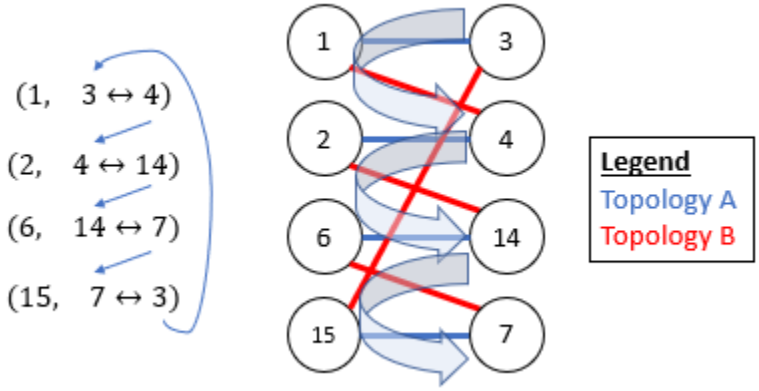

Figure 16: A Node Group Pair with its chaining sequence highlighted. Note how when the network reconfigures from Topology A (in blue) to Topology B (in red), all of the blue links are "shifted" by one node to become the red links in this set, the bottom right node (node 7) looping back around to the top of the right set (node 3).

The function works as follows. Given switches with a radix N and a full list of link deltas, the function picks a starting node, designated as the *Root* (usually node 1). It then finds all link deltas with a source (or starting) node matching the Root node, and those potential link deltas (called Root dL) are compiled into a Root List. The function will then pick one of the Root dL and note the two destination nodes, designating topology A (or the old topology) as the Head and topology B (or the new topology) as the current Tail. It then searches the link delta list for any link deltas where topology 1 matches the current Tail. This creates a second list of prospective "links" in the chain. The process selects one of these chains and, much like it did with the Root, designates a new Tail. Note that if the new Tail matches the Head, then the Group Pair so far is saved as a "minichain", meaning that it was possible to create a Group Pair using a smaller radix switch. The process repeats until the chain reaches a length of N-1. At which point, it will search for prospective links with the addendum that the Tail must also match with the Head to close the chain. If there are more

than one link delta that can close the chain, then both are recorded and added to the final GP set. Figure 17 demonstrates one such cycle.

We begin by picking node 1 as the Root, bringing up a full list of link deltas whose source node matches the Root as a prospective Root List (first column in the figure). We will eventually go through all of them, but we start with the first entry of the list, $(1,3 \leftrightarrow 4)$, and note the Head of the chain, 3, and the current Tail of the chain, 4. As such, we compile all link deltas that starts with 4 in their destination pairs (i.e., connects to 4 in topology A) into a list (second column in the figure). Like the Root List, we will eventually go through all of them, but we begin with the first link delta of column 2, $(2,4 \leftrightarrow 3)$. Here, we technically completed a chain because the new Tail matches the Head of the chain (node 3), but we have not achieved the desired radix. As such, we record this minichain in a separate location and continue to the third column.

In the third column, we experience the effects of a few filters that will be explained in more details later, but in short, they serve to prevent redundancies within the prospective GP. In this case, when we compiled a list of all link deltas with node 3 in their destination pairs, we found that $(1,3 \leftrightarrow 4)$ and $(1,3 \leftrightarrow 13)$ have a redundant link, specifically (1,3) which can be found in the link delta we started with in column 1, $(1,3 \leftrightarrow 4)$. As such, those two link deltas are discarded, and we move on with the rest of the list. We select link delta $(11,3 \leftrightarrow 7)$.

At this point, we have a chain of length 3. To form a GP for a radix-4 switch, our next link delta *must* close out the chain in a loop. In other words, the link delta must start with node 7 and end with node 3 in its destination pair. Here, the only link delta that satisfies this condition is $(15,7 \leftrightarrow 3)$. We have thus formed a link delta chain consisting of $(1,3 \leftrightarrow 4)$, $(2,4 \leftrightarrow 3)$, $(11,3 \leftrightarrow 7)$, and $(15,7 \leftrightarrow 3)$. We record this completed GP and move on back a column to explore the rest of its list, namely $(11,3 \leftrightarrow 10)$. Once that is complete, we reach the end of the list in column 3 and move on to the next entry in column 2, $(2,4 \leftrightarrow 14)$. This cycle continues iteratively until all branching lists have been explored.

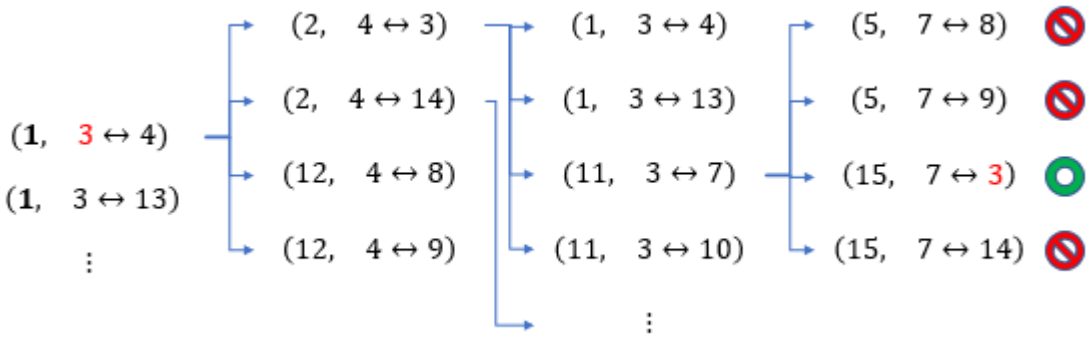

Figure 17: The chaining sequence the Root link (1, 3↔4) would undergo to create a GP of Radix 4. Note that two link deltas are skipped in the third column due to overlap of link contribution.

Once we have found all GPs that originates from the current Root node, we move on to the next Root and start over until all Root nodes have been explored. The entire process is summarized in the flow chart that follows.

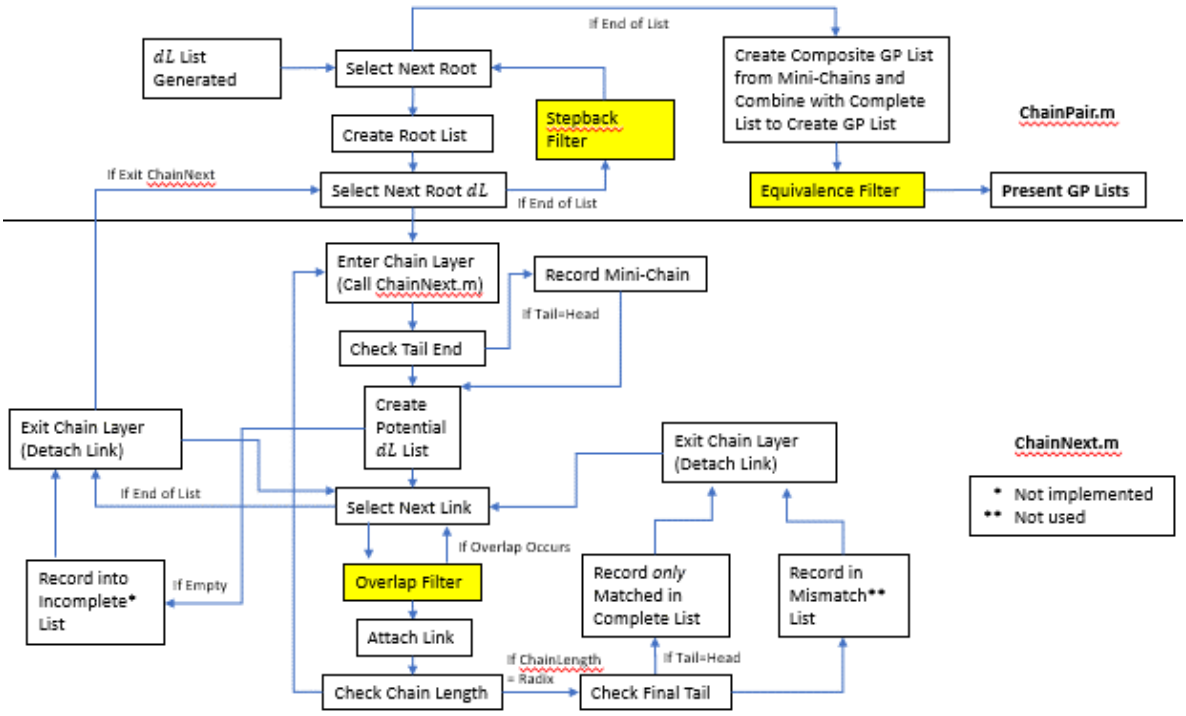

Figure 18: Chaining Process Flow Chart with filter blocks highlighted in yellow.

The process is refined through filters that catches GPs that would be redundant with previously generated GPs. There are two in-process filters, designated Stepback Filter and Overlap Filter, and one post-process filter, designated Equivalence Filter. It should be noted

that two features in the flow chart are either not implemented or not used. They were meant to assist in the creation of GPs possessing dummy links, however, the feature has not been implemented.

3.2.1 Stepback Filter

The Stepback Filter is the first filter in the system. It ensures that no *Roots* designated in the previous section reappear in later iteration of the scanning process. In other words, once the program has finished using node 1 as the Root node to create potential Group Pairs, node 1 will never again reappear in later iterations, whether as a source or destination. This is to prevent essentially creating the same group from the tail end. This is most obviously exemplified in the radix-2 case, as seen in the following figure. However, even in larger radices, one can recognize the redundancy if one compares the elements of each Node Groups.

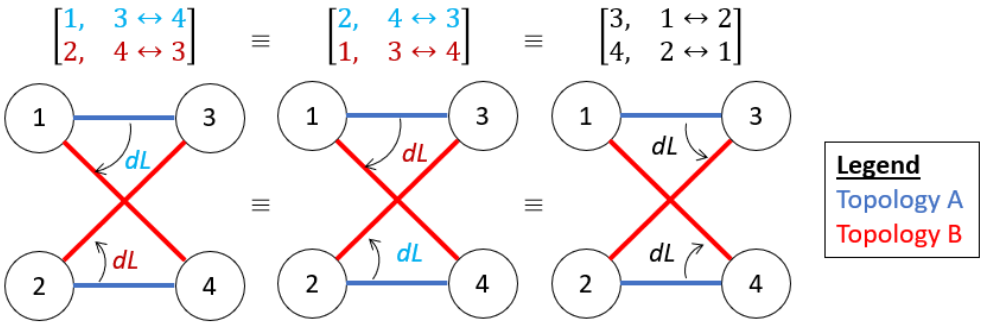

Figure 19: Example of three Radix-2 GPs generated with no Stepback Filter. Each expressed differently yet creating identical networks. The left GP would be the original. The middle has node 1 reappearing as a source node. The right has node 1 reappearing as a destination node in their link deltas.

3.2.2 Overlap Filter

The Overlap Filter was originally two separate filters: The Looping Filter and the Semi-Overlap Filter. The Looping Filter was a basic filter that detected instances of repeated link delta. For example, in the example demonstrated in figure 16, it would only catch the repeated instance of (1,3 ↔ 4) in column 3, because it only searches instances where a link delta is used multiple times in a chain. The Semi-Overlap Filter was intended to find instances of redundant link contributions in every GP. Specifically, it finds instances where a source and destination node are "flipped". For example, suppose for some reason, a GP contains the link

deltas (1,3 ↔ 4) and (3,1 ↔ 15), it would realize that (1,3) from the first link delta and (3,1) from the second link delta are the same link. Due to their different intended purposes, it was not immediately obvious on initial design that the Semi-Overlap Filter would achieve the same task as the original Looping Filter, and thus they were combined into a single filter.

Integrating the Overlap Filter to run in-process required a change in methodology. Rather than scanning the completed Group Pair on a column-wise basis, the program instead compares the prospective Link delta to be added to the at-this-point incomplete Chain and checks for redundancies. It is more efficient this way because with this in-process filter in place, it can be assumed that all prior chained Link deltas are not redundant. Should a redundancy be detected, then the current chain is aborted, and the process moves on to the next prospective Link delta (demonstrated in column 3 of Figure 17).

3.2.3 Equivalence Filter

The Equivalence Filter is the final filter for the chaining process. It is a post-process filter that scans through the potential solution set and detects all equivalent elements. In other words, it searches for two GPs that possess the same node groups. Unfortunately, all attempts at integrating this filter as an in-process subroutine to cut down on computational time have failed to yield a complete potential solution set.

The only way for such a filter to be in-process would be to be able to recognize a redundant pattern early during the Chaining Process. For example, given a GP chain with eight *dL*s, we may start another GP chain that turns out is the same as the first GP but backwards. The in-process filter should be able to recognize that pattern and stop it early, saving up a lot of computational time. Unfortunately, on testing the in-process vs. the post-process filters, there was a discrepancy where the in-process would abort too many chains too early, leading to missing GPs compared to the more comprehensive list having this filter post-process would provide.

It was eventually determined that further attempts at trying to refine the process or otherwise tune the sensitivity would be a poor investment of time when the other filters do a sufficient job at reducing the program's runtime.

3.3 Selection Process

With a comprehensive list of potential Group Pairs, it is now time to choose a set that will effectuate the necessary changes in links to reconfigure a network. Here, we enter the Selection Process (appendix 6.1.9). As an aside, it should be noted that the following method is more easily comprehensible from a visual and therefore "human" perspective, however, it is probably very inefficient from a computational or analytical point of view. The main challenge is to select the smallest number of GPs that can effectuate the reconfiguration of the network. That means that there should be as few redundant bridging links amongst the selected GPs as possible. As such, it is important to then know which GPs can and cannot be taken simultaneously. To do that, we create a similarity or overlap matrix. This is similar to the adjacency matrix, except this time, the "nodes" correspond to their respective elements in the potential GP set. The similarity matrix will thus list out the relationship between every GP element and indicate the number of common essential link contributions.

Once a similarity matrix is fully established, we create an adjacency matrix by establishing a tolerance value, which indicates how many overlaps in nodal links provided by each GPs will be permitted before a similarity link is established. Suppose we take a tolerance of zero, then there is a link between GP nodes that has any common link contribution as shown in the following figure. As such, we create a similarity graph where all sufficiently overlapping GPs are interconnected.

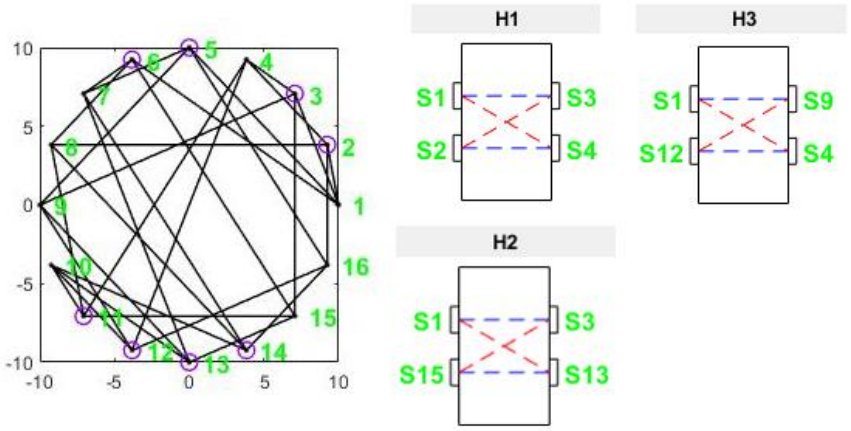

Figure 20: The Similarity Graph of a 16-node Hypercube-Torus system with Radix-2 reconfiguration switches, where all nodes with at least a single overlapping link contribution is interconnected. The selected GPs are circled, indicating a set of GPs that could be selected without any link contribution overlap. For example, GPs H2 and H3 were selected, while H1 has a single link overlap with either other two (H1-H2 both contributes link (1,3) while H1-H3 both contributes (1,4)).

The goal then is to choose a set of nodes where each node is **not** interconnected with another member of the selected set. From a graph perspective, that means all nodes must have an even number of hops in between each other.

To accomplish this computationally, it is necessary to create a distance matrix. From that, to increase the likelihood the final selection will not miss any necessary links, we start with a random GP with the greatest number of viable nodes. Viable nodes being nodes an even number of hops away from the reference. We select the next node by likewise ensuring that the number of overlapping viable nodes is maximized. This continues until all viable nodes have been collected or rejected.

Once a selected set is established, then the GPs the selected nodes correspond to represent the selection and configurations of the HOB devices needed to reconfigure the network from topology A to topology B.

4 Program Performance Analysis

In this section, we shall examine the program's performance in converting a 16-node 4D Hypercube (H4D) to a 4×4 Torus network (T4×4). For radices ranging from 2 to 16, we shall evaluate the accuracy and the resource efficiency of the solution the program provides and measure the computation time at which this was found.

4.1 Preliminary Examination

This scenario was selected for its overall simplicity and small size. I was able to manually solve the radix-2 and radix-4 case and thus would be able to verify the accuracy of the program's results. It also served as a good starting point for the development of this program, because it possessed favourable properties. I decided that only once the program can reliably handle the baseline easy scenario will development move on to more complex cases.

An important property to note about the 4D Hypercube and the 4×4 Torus is that they are uniformly isomorphic. In other words, by simply rearranging the node designation without changing the interconnection, a Hypercube can become a Torus. Conversely, by rearranging the relevant links, one can achieve the desired reconfiguration. It should be noted that isomorphism by itself is insufficient to ensure the program would work; we need to ensure that each node has a constant degree. As such, while we do not necessarily need to stick to a traditional mapping of a topology (i.e. node 1 does not necessarily need to connect to node 2) it is important to keep track of the node designation in order to ensure the degree is preserved. In this case, the three conditions for the use of the program are satisfied: We have only two topologies of interest. All nodes maintain a constant degree of 4. All links are defined.

Of the four links each node possesses, two of them need reconfiguration, yielding a total of 64 potential link deltas, with four link deltas associated with every node. This conveniently uniform distribution allows us to predict the maximum number of GP combinations since, at every layer of the iteration, there are certainly four potential link deltas

to branch off to. In other words, with no filters, the number of GPs the program could theoretically generate can be estimated by the following formula:

$$P = L \times l^{r-2} \tag{14}$$

where P is the unfiltered number of Pairs of Node Groups the system may generate, L is the number of link deltas dL the system has, l is the number of link deltas per nodes, and r is the radix of the reconfiguration crosspoint switch. The reason for the -2 in the l's power is due to the fact that it starts with all potential link deltas and that when you reach the end of the chain, there is usually only one possible link delta that could be used to complete the chain.

While this formula has not been exhaustively verified, this gives us an idea of the scale of the work the program needs to run through. The number of potential chain combinations could reach upwards to 17.2 billion for the radix-16 case with an estimated 22.9 billion calls on the ChainNext.m function, yet there is only a *single* non-redundant solution. The table below demonstrates why the redundancy filters are critical to run this program with a reasonable memory requirement.

Table 2: Estimated Number of Unfiltered GPs and ChainNext.m calls vs Filtered values (no composite GP).

Radix	No Filter GPs Est.	Filtered GPs	No Filter Calls Est	All Filters Calls
2	64	16	64	24
4	1,024	80	1,344	214
6	16,384	176	21,824	1,388
8	262,144	233	349,504	7,172
10	4,194,304	176	5,592,384	26,542
12	67,108,864	88	89,478,464	68,517
16	17,179,869,184	1	22,906,492,224	157,966

4.2 Speed of Solution

The primary purpose of the filters is to weed out redundant outcomes. That can theoretically be easily achieved by allowing the program to run and then check the results. However, the average iteration of the ChainNext.m function has a duration of around 3 ms,

thus a system that needs to over 17 billion iterations would be computing for more than 1.6 years. As such, the real purpose of those filters is to reduce the number of necessary iterations by catching or preventing redundancies early.

With the reduction in place, we can now run the program and measure the time elapsed. For reference, the program was executed on my personal computer, which runs on an AMD Ryzen 7 1700 (8 cores, 16 threads), 32GB of RAM and an EVGA GeForce GTX 1080 graphics card. The runtimes measured are compiled in Table 3. The most obvious feature of the timed table is the expected scaling of the Chaining Process, with it taking up more and more time the larger the radix gets.

Table 3: Timed Program Execution with percentage distribution (leftover time goes to all other process). Also noted are the number of composite (comp) GPs in the total GP count.

Radix	GP (comp)	Full Time	dL Acquisition	Chain Process	Selection Process	Draw Time
2	16 (0)	1.141s	0.029s (2.54%)	0.088s (7.71%)	0.223s (19.54%)	0.787s (68.97%)
4	120 (40)	5.908s	0.029s (0.49%)	0.990s (16.76%)	4.357s (73.75%)	0.519s (8.78%)
6	288 (112)	29.614	0.027s (0.09%)	4.700s (15.87%)	23.860s (80.57%)	1.014s (3.42%)
8	365 (132)	61.769s	0.029s (0.05%)	22.537s (36.49%)	38.783s (62.79%)	0.407s (0.66%)
10	244 (68)	98.906s	0.025s (0.03%)	81.029s (81.93%)	17.439s (17.63%)	0.398s (0.40%)
12	116 (28)	219.195s	0.027s (0.01%)	214.375s (97.80%)	4.094s (1.87%)	0.677s (0.31%)
16	2 (1)	485.574s	0.055s (0.01%)	485.041s (99.89%)	0.096s (0.02%)	0.360s (0.07%)

The dL Acquisition process is relatively steady because it is unaffected by the inputted radix. Instead, the dL Acquisition process scales quadratically with the order of the

network ($O(N^2)$) because it needs to compare the link status between every pair of nodes in the network.

The Chaining Process has an exponential complexity $O(l^r)$, where l is the number of link deltas per nodes and r is the radix value, due to how the problem grows by the same factor at every layer of the chain the process needs to run through. While the filters do reduce the runtime, I do not believe it reduces the complexity of the problem.

The selection process meanwhile grows before hitting a peak at the median radix and falling. This pattern is matched by the number of GPs generated by the Chaining Process. This implies that the problem grows about linearly with the number of GPs inputted into the process ($O(GP)$). This is consistent with the fact that the most computationally heavy part of this process is the generation of the graph and the GP comparison to generate the similarity matrix. The comparison aspect scales quadratically with both the radix ($O(r^2)$) and the number of GPs inputted ($O(GP^2)$) for similar reasons as the dL Acquisition process; it needs compare each pairs of GPs for their similarity.

The draw time is the least important section to examine because it does not contribute to finding a solution set. That said, there are still a few key observations to make. The drawing program simply takes the final list of GPs and draws both a wide graph showing each node connected to their corresponding reconfiguration switches as well as the configurations for each of the individual reconfiguration switches. Consequently, the more GPs the program needs to draw, the longer it takes. As such, the larger the radix, the less GPs the system will need to effectuate the desired reconfigurations. This means that the Drawing Process scales linearly with the number of GPs inputted ($O(GP)$). What is worth noting then is that the time recorded does not necessarily reflect that (specifically radix 6 takes longer than radix 2). This has to do with the quality of the solution, which will be discussed in the next section.

4.3 Quality of Solution

The primary function relevant to the quality of the solution is the Selection Process. Due to how the program functions, it is not possible to have unused ports on a reconfiguration

switch in the final solution set. As such, the quality of the solution is determined by whether it uses as few crosspoint switches as possible, which consequently will minimize the number of wasted ports or links if any, and whether it effectuates all link deltas necessary for the reconfiguration.

The smallest chain that can be achieved with these topologies is a radix-2 chain, suggesting that the best radix to use would be some factor of 2. Empirically, it was determined that for the 16-node Hypercube-Torus conversion, the best radices to use are 2, 4, 8, and 16 which uses 8, 4, 2, and 1 switch(es) respectively. In these cases, the link deltas fit perfectly with no redundant or missing links.

As shown in Table 2 though, the pattern does not necessarily hold true in the case of radices 6, 10, and 12. That is because in its current state, the program's decision is based simply on the "spacing" with no account to actual content of GP. Should the preliminarily selected solution not cover all necessary links, the similarity graph would then be adjusted by loosening the tolerance value, such that a similarity link may need more GP similarity to exist. The idea was that by allowing a limited amount of overlaps between the selected GP (which would inevitably occur when using imperfect radix), one would then get the next best solution.

Unfortunately, due to not attempting to maximize the contribution of the next randomly selected GP, it may not contribute all necessary links to then finalize the selection. The program then mistakenly believes there is no solution at the current tolerance level and loosens it even further. Consequently, due to the random selection involved, the result is also inconsistent. As such, the selection process will need to be significantly revised and improved in order to ensure reliable and robust results.

4.4 Potential Improvements

Due to time constraint and a particular focus on ameliorating the Chaining Process such that it could even finish, there are a series of features and improvements that could not be implemented. With that said, there are still plenty of improvements the program could

have. The main weakness to the Chaining Process is its exponential time factor. Unfortunately, there are very little additions that can be made to Chaining Process in its current form, in that I could not find any ways of adding more in-process filters to reduce the computational time. As such, any improvements would require a fundamental change in the process's structure. For instance, instead of initializing the ChainNext inner looping for each individual link delta, one could attempt to parallelize the process by scanning through all link deltas simultaneously, identify equivalent tail ends (i.e. all link deltas ending with 4 and so on) and then simultaneously attach the next link delta. While at each layer of the chain, there will be an exponential number of link deltas to evaluate, the number of chaining to occur at each layer should be reduced to, at most, the order of the system (in this case 16). This may be incompatible with certain filters in their current implementation, but should it work correctly, this may reduce the problem size from $O(l^r)$ to $O(r^2)$. Additionally, one can consider transferring this process to C or another programming language that may perform this task faster than MatLab.

Another issue with the Chaining Process is its inability to give a good output when given a radix value that does not resolve the chain. More specifically, in the Hypercube Torus case, it will not have any GPs listed if a user enters a radix of 3 because no chain of that length exists. As such, the proper solution here would be to either deliberately leave a port empty (i.e. not connected) on both sides of the switch or to have the program automatically correct the impossible parameter. Either way, adjusting the program to be capable of handling this will make the program more robust.

With the runtime and robustness issues mostly resolved, it is time to resolve the solution accuracy problem. The primary difficulty in designing the selection process is that it needs a "whole picture" view. It needs to be able to start from a point that will not prevent an optimal solution. To use an oversimplified example, suppose we have a set of pairs: $\{(1,2), (2,3), (3,4), (4,5), (5,6)\}$. The program needs to be able to recognize that starting from $(2,3)$ would prevent it from acquiring 1, and 4 or 6, since the only pairs that would have no overlap would be $(4,5)$ or $(5,6)$. As such, unless one were to replace the similarity

graph with another method of getting the "whole picture," any future developments will have to focus on the specifics of how the process decides which GP to choose.

In the program's current iteration, the selection process starts with and subsequently selects the GP with the maximum number of other GPs that are an even number of hops away. This is because those GPs are more likely to be on the "edge" of any chains. To use the earlier example, (1,2) would have two other GPs while (2,3) would only have one. Sadly, the principle no longer applies when there are no other GPs that are two hops away under the zero-overlap setting, such as would be the case for radix-10. The radix-6 case would reach a similar situation when it acquires two GPs.

The main issues then are that the selection is otherwise blind and that the program loosens its tolerance value when the process encounters failure without attempting the alternative solutions. While the idea of loosening tolerance is a requirement to allow for imperfect radix fits has merit, the method is flawed.

A feature that was missing is the ability to bias the selection, to evaluate the individual GPs and assign a score to them. The original "maximum options" property will instead add to those GP's score rather than be the sole factor and it would allow for other factors to be considered, such as the proximity of the nodes in a GP, or that a GP has a favourable characteristic that would decrease the chance of inefficient coverage. Finally, it can evaluate the prospective GP's link delta contribution such that it will cover any leftover links when a preliminary list has been established. In other words, it would be better to remove the blind, random element of the selection process and instead use an evaluative approach. Future developers will have to determine whether it is sufficient to evaluate once at the start and once to finish up, or if evaluating after adding any GP to the solution list will not slow down the program by an unreasonable degree.

Alternatively, any future developers can change the approach altogether and formulate this as a discrete optimization problem. It has unfortunately never occurred to me to try this approach nor was I familiar enough with optimization to attempt in the time I had left.

Additionally, the program should be more extensively tested on other topology combinations, with efforts to reduce or even eliminate the necessary conditions.

4.5 Results

For the sake of not overloading this section with results, we shall focus only on the radix-8, and radix-12 cases. One for when all conditions are satisfied, and one where the fifth condition is ignored. The similarity graphs are too visually dense to glean anything meaningful and thus will not be presented here, however, the Group Pair Relations (*GR*) are included in the ChainResults_20200915.mat file uploaded with the rest of the program.

4.5.1 Radix-8 Results

The Radix-8 switches satisfy all conditions of the program, and thus the results are consistent and reliable. Out of the 365 possible GPs, two are always selected, and they are random enough to not easily yield the same result which is consistent with the intended design.

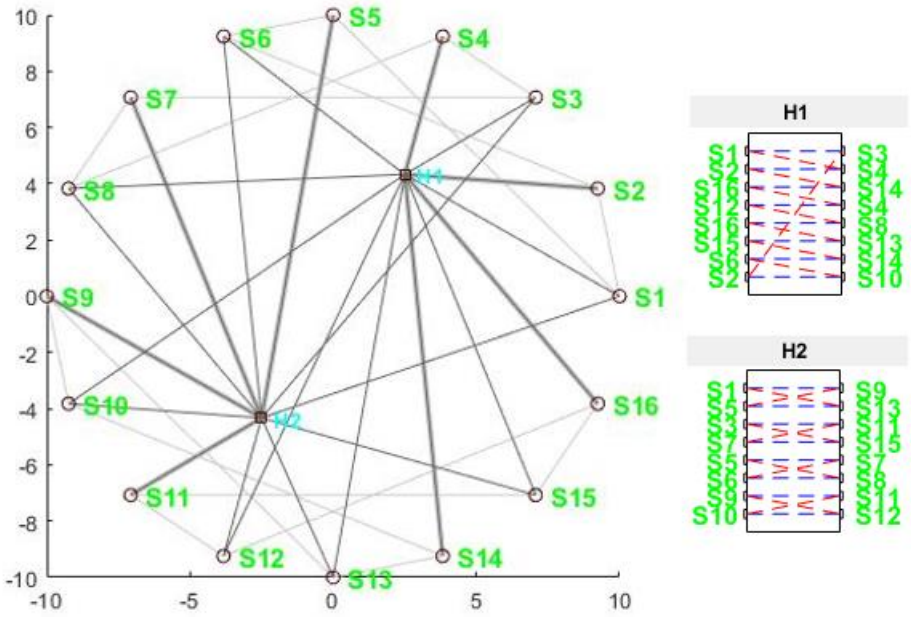

Figure 21: H4D (blue lines) to T4×4 (red lines) system placed in a circular pattern and how their servers would connect to the radix-8 switches (H1 and H2) to each other. Also included are the internal configurations of the individual switches Thicker lines represents multiple connections in between nodes while faded lines represent non-reconfiguring links.

Presented in Figure 21 on the previous page are the results the program would generate for radix-8 switches.

4.5.2 Radix-12 Results

The Radix-12 switch does not satisfy the fifth condition, seeing as there are 16 link deltas needed to reconfigure a 16-node Hypercube to a 4×4 Torus. At this moment, however, the program struggles to handle situations where GPs are not filled up perfectly.

Optimally, the solution should pick two GPs out of the 116 possible combinations. However, due to the current approach of loosening the tolerance uniformly and allowing for more overlap to occur per GP, the program will end up selecting four GPs, as shown in the following figure.

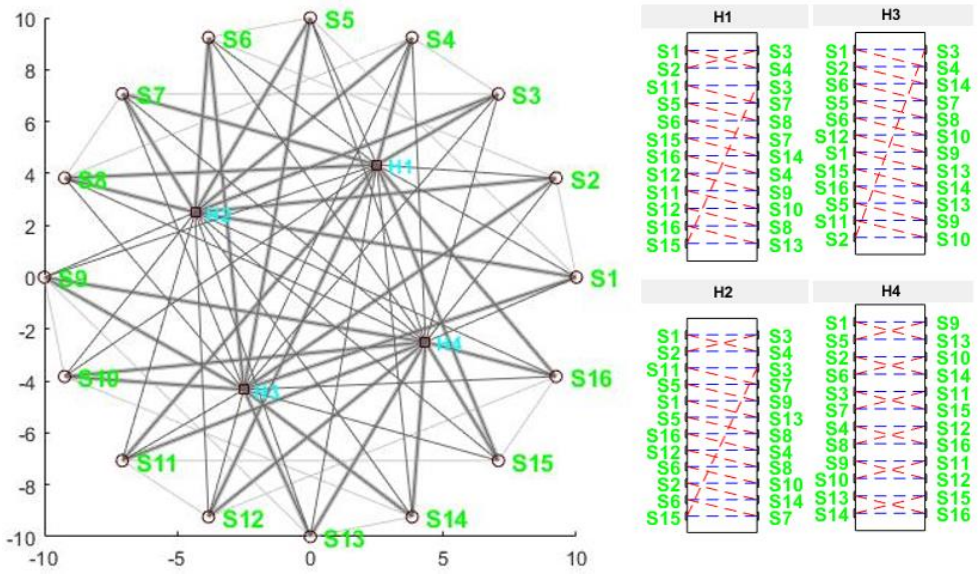

Figure 22: Suboptimally selected radix-12 configuration. Follows same line convention as previous figure. Note that there are several redundancies in the link contributions; $(1, 3 \leftrightarrow 4)$ & $(2, 4 \leftrightarrow 3)$ appears multiple times.

The less efficient solution provided when the fifth condition is not satisfied has two main consequences. The program will execute more slowly because it spends more time trying to select GPs that may not provide the maximum link contribution. This leads to the second consequence where the solution will be more expensive because it introduces unnecessary switches.

5 Conclusion

In this thesis, we have covered a few of the fundamentals of network topologies and graph models, followed by four examples of popular topologies: Hypercube, Torus, Fat-Tree, Dragonfly.

With the context in place, we then examined the performance of various application and traffic patterns on various network topologies. We began by examining the two benchmark tests typically used on supercomputers, the Linpack and Conjugate Congruent tests. Unfortunately, no clear conclusions could be drawn from the test results of the top 20 supercomputers. As such, we moved on to Jyothi, *et al.'s* examination of synthetic traffic pattern on various network topologies. Their results demonstrated that Dragonfly had the best result for All-to-all traffic, the 5-ary BCell had the best result for Random Matching, and the Fat-Tree had the best result for Longest Matching.

Having established that specific traffic patterns (or applications) may benefit from certain topologies, we address how to implement a reconfigurable network. While the Hybrid Optical Bridge (HOB) was brought up as the crosspoint switch to use both for its remote capabilities and for its lack of radix limit, the principles offered in this thesis can apply to any crosspoint switch.

Thus, I proposed my program to evaluate two desired topologies and offer a solution to the placement of crosspoint switches to allow for reconfiguration. The process was divided into four modules: link delta acquisition process, the chaining process, the selection process, and the presentation process. The modularity of the designs allows for the individual alteration or amelioration of the individual parts without needing to alter the others, assuming compatible input and output format.

The main work in the proposed process is ensuring that the program would finish in reasonable time via the use of filters in the chaining process. Consequently, the quality of the solution was not sufficiently refined to be wholly robust. As such, there is much that still needs to be done.

5.1 Future Works

Any future work building upon this thesis will most likely be building upon the program or researching traffic patterns various applications may have and how different topologies handle them.

The link delta acquisition and chaining process can be revamped to allow the program to handle three or more topologies. That way, a three-way reconfiguration between the BCell, Fat-Tree, and Dragonfly+ may become feasible. This would require a much more elaborate methodology than to only shift the switches' configuration one port down to effectuate the topology reconfiguration since 2 topologies may share a link while the third does not.

Similarly, works can be directed towards loosening or even removing some of the conditions the program have in order to run. The program can be made to be able to handle undefined links, or handle situations where the node degree is not maintained as is currently required.

Despite the filters, the chaining process is still the slowest part in this program. One potential approach in improving the computation speed is to modify the currently sequential process to run in parallel, such that all GPs are generated simultaneously. The filters are still compatible since they only limit potential options for a given Root link delta or prevent overlap. Should a method be found, this will reduce the current exponential growth of the problem size to be linearly proportional to the radix.

The selection process will likely be the core of any future development though. The process currently uses a blind, semi-random approach to finding a solution set. It does not truly "know" what makes a good GP to use. As such, an examination of how best to optimize GP selection would prove beneficial. A system to bias or otherwise add weight to specific desirable GPs would also greatly ameliorate the program.

On the traffic-topology side, one could conduct a survey on the full 500 supercomputers of a top500 list and to determine whether certain topologies are associated with higher performance efficiency in certain benchmark tests.

6 Appendix

6.1 Codes

6.1.1 TestScript.m

```
% Test Script for Node Group Synthesis Algorithm
% Edgar Pan
%%
%Settings - ONLY CHANGE VALUES HERE
%Topology Data
H4D = GenHND(4);
T2D = GENTor(4,4);
M(:,:,1) = H4D;
M(:,:,2) = T2D;
%ChainPairs Setting
radix = 2; %Be aware that increasing this increases runtime exponentially
filters = [1 1 1]; %Not recommended to deactivate in higher radix
%Selection Process
ForceTolerance = 0;
tolerance = 0;
%Drawing Process
DrawGraphs = 1;
DrawCommon = 1;
ActiveTopology = 0;
HOBLabel = 1;
%Run Profiler
RunProfiler = 1;
%%
%Start Timer
if RunProfiler
    profile on
end
tic
%%
%Generates Connectivity (Link Delta) List
[dL,Nec,Common] = GenConnList(M);
%%
%ChainPairs
if radix > 0
    [CL,filt_met] = ChainPairs2(dL, radix, filters);
else
```

```
    [CL2,filt_met2] = ChainPairs2(dL, 2, filters);
    [CL4,filt_met4] = ChainPairs2(dL, 4, filters);
    [CL6,filt_met6] = ChainPairs2(dL, 6, filters);
    [CL8,filt_met8] = ChainPairs2(dL, 8, filters);
    [CL10,filt_met10] = ChainPairs2(dL, 10, filters);
    [CL12,filt_met12] = ChainPairs2(dL, 12, filters);
    [CL16,filt_met16] = ChainPairs2(dL, 16, filters);
end
%%
%Selection Process
if ForceTolerance
    tol = tolerance;
else
    tol = [];
end
if radix > 0
    [Selected,SelectedIndex] = SelectionProcess(CL,Nec,tol);
else
    [Selected2,SelectedIndex2] = SelectionProcess(CL2,Nec,tol);
    [Selected4,SelectedIndex4] = SelectionProcess(CL4,Nec,tol);
    [Selected6,SelectedIndex6] = SelectionProcess(CL6,Nec,tol);
    [Selected8,SelectedIndex8] = SelectionProcess(CL8,Nec,tol);
    [Selected10,SelectedIndex10] = SelectionProcess(CL10,Nec,tol);
    [Selected12,SelectedIndex12] = SelectionProcess(CL12,Nec,tol);
end

%%
%Find Missing Links
if radix > 0
    Missing = FindMissing(Nec,Selected);

    if ~isempty(Missing)
    %     CL = FillMissing(CL,Missing);
        disp('Warning: Not all Necessary Links have been completed.')
        disp('Recommend not forcing the tolerance.')
    end
end
%%
%Drawing Process
if DrawGraphs && radix > 0
    GraphOrder = size(M,1);
    if DrawCommon
```

```
        DrawHobSystem(GraphOrder,Selected,ActiveTopology,HOBLabel,Common)
    else
        DrawHobSystem(GraphOrder,Selected,ActiveTopology,HOBLabel)
    end
end
%%
%Measure Elapsed Time
ElapsedTime = toc;
if RunProfiler
    p = profile('info');
    profile viewer
end
```

Published with MATLAB® R2019a

6.1.2 GenHND.m

```
function [ Aj ] = GenHND( N )
%GENHND
%   Author: Edgar Pan (edgar.pan@mail.mcgill.ca)
%   Generates a Hypercube of N dimensions
%Base Adjacency
Q1 = [0 1 ; 1 0];
Aj = Q1;
if N >= 2
    for n=2:N
        Q = Aj;
        Aj=kron(Q,eye(2)) + kron(eye(2^(n-1)),Q1);
    end
end
end
```

Published with MATLAB® R2019a

6.1.3 GENTor.m

```
function [ T_Aj ] = GENTor( X, varargin )
%GENTor Generate a Torus Function
%   Given a set of dimensions, generates a Torus graph.
%   Does not work for any dimensions lower than 2.
%--------------------------
% Define the adjacency matrix of the n-D Torus.
%   Input length of each dimension separated by commas.
%   Example: GENTor(4,4,2)
%
%   Additional options - Add specific tags after list of dimensions
%       'noloop' - Creates a mesh matrix with no wraparound
%       'sglloop' - Creates a torus matrix with only a maximum of single
%            link during wraparound.
%       'dblloop' - Default. Creates Torus matrix with maximum of double
%            loop. Only applicable for any dimension of length 2.
%   Example
%       GENTor(4,4,2,'noloop') - 4x4x2 Mesh with no looping
%
%
%   Written by Edgar Pan
%   Version 2.0.0
%   Created 2019-05-10
    %%
    %Parses the inputs
    if any([cellfun('isclass',varargin,'cell')
cellfun('isclass',varargin,'struct')])
        error(' GENTor only supports numeric/character arrays ')
    end
    N = nargin; %Full n arg in
    %Finds where options arguments, if any, begins
    optIdx = cellfun(@ischar,varargin);
    if any(optIdx)
        optStart = find(optIdx,1);
        vars = varargin(1:optStart-1);
        opts = varargin(optStart:N-1);
        n = optStart;
        %Number of numeric entries, including X. The index shift cancels.
    else
        vars = varargin;
        opts = [];
```

```
 40          n = N;
 41      end
 42      %Catches input errors, any vectors/matrices.
 43      if any([length(X)>1 cellfun(@(x) length(x)>1,vars)])
 44          error('GENTor: Please do not enter any matrices. Separate dimensions
with commas.')
 45      end
 46      %Catches input errors, anything with 0-length dimension
 47      if (X < 1) || any(cellfun(@(x) x<1,vars))
 48          error ('GENTor: Please input dimensions for an existing graph.');
 49      end
 50      %%
 51      %Parses the options
 52      optLoop = -1;
 53      if ~isempty(opts)
 54          if any(cellfun(@(x) strcmpi(x,'noloop'),opts))
 55              optLoop = 0;
 56          elseif any(cellfun(@(x) strcmpi(x,'sglloop'),opts))
 57              optLoop = 1;
 58          elseif any(cellfun(@(x) strcmpi(x,'dblloop'),opts))
 59              optLoop = 2;
 60          end
 61      end
 62      %%
 63      %Start Compiling all the numbers in.
 64      T_Aj = GetTorusBasis(X,optLoop);
 65      for d = 2:n
 66          if vars{d-1} > 1
 67              Y = GetTorusBasis(vars{d-1},optLoop);
 68  %             T_Aj = kronSum(T_Aj,Y);
 69                      T_Aj = kronSum(Y,T_Aj); %Keeps original numbering
orientation.
 70          else
 71              disp('GENTor: WARNING - Dimension of length 1 detected and
ignored.')
 72          end
 73      end
 74      %%
 75      %Internal Functions
 76      function B = GetTorusBasis(y,optLoop)
 77          B = zeros(y);
 78          B(2:y+1:end) = 1;
```

```
 79          B(y+1:y+1:end) = 1;
 80          if nargin < 2
 81              optLoop = 2;
 82          end
 83          %Asks whether double looping allowed.
 84          if optLoop == 0
 85              %Does nothing
 86          elseif optLoop == 1
 87              B(y,1) = 1;
 88              B(1,y) = 1;
 89          else
 90              B(y,1) = B(y,1) + 1;
 91              B(1,y) = B(1,y) + 1;
 92          end
 93
 94      end
 95
 96      function KS = kronSum(A,B)
 97          if size(A,1) ~= size(A,2) || size(B,1) ~= size(B,2)
 98              error('GENTor - kronSum: Invalid Input. Must be square matrices')
 99          end
100          KS = kron(A,eye(length(B))) + kron(eye(length(A)),B);
101      end
102  end
```

Published with MATLAB® R2019a

6.1.4 GenConnList.m

```
function [ NC, Nec, Common ] = GenConnList( M )
%GenConnList Creates list of switching connections between topologies.
%   Author: Edgar Pan (edgar.pan@mail.mcgill.ca)
%     The program scans through the adjacency matrices of the various
%     network configurations. The adjacency matrices are listed as a single
%     3D matrix, with each pages representing a specific configuration.
%     The program is capable of handling more than two configurations,
%     however, that is not recommended, since it has not been fully tested
%     nor explored (i.e. useful output format?)
%
%   Input:
%       M - 'Adjacency Matrices' describing the topologies.
%                   3 Dimensional Array. [Rows, Columns, Topology]
%                           Every layer of array indicates a new topology
%                           Square matrix
%                           Mn: Number of Matrices (minimum 2)
%    Output:
%            NC - "Node Connection" (Note: Old term for Delta Link)
%                   1 + Mn columns, ? rows
%                           [SrcNod (Column/Row), Topo1Node, Topo2Node, …
TopoMnNode]
%       Nec - "Necessary Connections"
%           Lists all node connections necessary to fully describe all
%           topologies
%       Common - "Common Links"
%           List of all Common links filtered out due to being irrelevant
%           to the process.
    %%
    %Basic dimension data
    sizeM = size(M);

    %%
    %Ensures there are multiple topologies entered.
    if sizeM(3)<2
        NC = 0;
        Nec = [];
        Common = M;
        disp('No changing connection required for a single topology')
        return
    end
```

```
40
41      %%
42      %Common Link Filtering Process
43
44      Common = all(M,3);
45
46      %Allocating Switching Link space
47      dM = zeros(sizeM);
48
49      %Filters out all common links from individual layers
50      for k = 1:sizeM(3)
51          dM(:,:,k) = xor(M(:,:,k),Common);
52      end
53
54      %%
55      %Finding maximum radix of every columns
56      %By first going through each topologies and finding how many links
57      %each columns have. Then comparing every column's value and
58      %then picking the highest one of all of them.
59      max_radix = max(sum(dM),[],3);
60
61      %Preallocating NC space
62      NC = zeros(sum(max_radix.^sizeM(3)),1+sizeM(3));
63      %row = highest radix to the power of the number of topologies
64      %column = number of topologies + 1 to indicate source
65
66      %The reasoning behind squaring the maximum radix is to have room to
67      %place 0's in order to represent Loose/Ghost Links.
68
69      for j = 1:sizeM(2)
70          %%
71          %Version 2 (variable topology compatibility)
72          %Allocates cell space for the connection data
73          xCell = cell(1,sizeM(3));
74
75          for k = 1:sizeM(3)
76              y = find(dM(:,j,k));
77              xCell{k} = y;
78          end
79          z = setprodcell(xCell);
80
81          %%
```

```
        %Compilation process. Puts the returned cross products into a
        %single list.
        sizeZ = size(z);
        z = cat(2,j*ones(sizeZ(1),1),z); %Creates a column for source idx
        shift = sum(max_radix(1:(j-1)).^sizeM(3));
        NC(1 + shift : sizeZ(1) + shift,:) = z;
    end

    %%
    %Section for Listing out all Necessary Connections
    dM_flat = any(dM,3);
    [a,b] = find(dM_flat);
    Nec = [a,b];
    Nec = Nec(a<b,:);
end
```

Published with MATLAB® R2019a

6.1.5 setprodcell.m

(Original program written by Mukhtar Ullah, adapted for our purposes here by Edgar Pan)

```
function C = setprodcell(X)
% SETMATPROD product of multiple columns of a matrix.
%
%   This version of the code setprod takes a Cell array directly.
%
%   For X = {A, B, C}
%   C = setprodcell(X) returns the cartesian product of the sets
%   A,B,C, etc, where A,B,C, are numeric or character arrays.
%
%   Example: A = [-1 -3 -5];   B = [10 11];   C = [0 1];
%
%   X = SETPROD(A,B,C)
%   X =
%
%      -5    10     0
%      -3    10     0
%      -1    10     0
%      -5    11     0
%      -3    11     0
%      -1    11     0
%      -5    10     1
%      -3    10     1
%      -1    10     1
%      -5    11     1
%      -3    11     1
%      -1    11     1
% Mukhtar Ullah
% mukhtar.ullah@informatic.uni-rostock.de
% September 20, 2004
% Adapted into Cell version by Edgar Pan
% edgar.pan@mail.mcgill.ca
% January 18, 2019
args = X;
if any([cellfun('isclass',args,'cell') cellfun('isclass',args,'struct')])
    error(' SETPROD only supports numeric/character arrays ')
end
% n = nargin;
n = length(args);
```

```
39 [F{1:n}] = ndgrid(args{:});
40 for i=n:-1:1
41     G(:,i) = F{i}(:);
42 end
43 C = unique(G , 'rows');
```

Published with MATLAB® R2019a

6.1.6 ChainPairs2.m

```
function [ CL, filter_meta ] = ChainPairs2( dL, radix, filters )
%ChainPairs v2 Synthesizes list of potential node groups by chaining
%connection pairs. At present only handles 2 topology systems.
%   Author: Edgar Pan (edgar.pan@mail.mcgill.ca)
%   Initially written: 2019-12-18
%     In essence, this program takes a list of potential link
%     reconfigurations and "chains" them into the two sides of a crosspoint
%     switch. It does this by first selecting a "Root" Index, determined by
%     the source node of a particular reconfiguration pair (dL).
%     Next, designating the Destination Node for Topology 2 as the "Tail",
%     it finds a dL pair with the "Head" or Destination Node for Topology 1
%     that matches the Tail of the previous dL.
%   Input:
%       dL - Changes in Links list.
%           Formerly "Node Connection",
%           most likely generated by GenConnList.m
%               Format: [SourceNode Topology1Node Topology2Node ...]
%       radix - determines the number of ports on a one side of a switch.
%       filters - Manually selects which filters to use. Default: All
%           active.
%               Format: [Stepback Overlap/Looping Equivalence]
%               crosspoint switch.
%               -Stepback - Prevents process from scanning through prior
%               columns/rows.
%               -Overlap/Looping - Scans for contributed Link redundancies
%               -Equivalence - Scans for GPs that are equivalent
%   Output:
%       CL - "Chain List"
%           Cell Array
%           {1} = Completed GroupPair List
%           {2} = Minimum Chain List
%           {3} = Split Chain-Composite List
%               {1} Chains that are perfect fit for given Radix size.
%               {2} Composite Blocks composed of Minimum Chain List
%           {4} = Incomplete Chain List
%       filter_meta - Metadata generated by the process filters.
%           {1} = Record Metadata. If no variable records the metadata in
%           output, then metadata will not even be processed. Will be
%           equals to [1] if filter_meta exists as an output variable.
%           {2} = Metadata for Stepback Filter
```

```
41 %               Rows correspond to the completed Root_Index node.
42 %               Column 1 corresponds to the number of completed chains
43 %                   compiled so far.
44 %               Column 2 corresponds to a list of Delta Links at end of
45 %                   corresponding Root_Index.
46 %           {3} = Metadata for Equivalence Filter
47 %               {1} list - full unfiltered list
48 %               {2} ia - Index kept from original list.
49 %                       i.e. filtered_list = list(:,:,ia).
50 %               {3} ic - Index to recreate original list.
51 %                       i.e. list = filtered_list(:,:,ic)
52 %                   {4} filtered_index - List of indices that has been
removed.
53     %%Initialization
54     %Current iteration of acceptable Delta Links
55     dL_List = dL;
56
57     %Number of vertices in the graph (Order). Used to set Loop Limit
58     Graph_Order = max(max(dL));
59
60     %Initializes Connection Chain as empty sets.
61     CL = cell(4,1);
62
63     %%Filters
64     %Development constant: How many filters were implemented
65     Filters_Constant = 3;
66
67     %Activation of Composite Group Pair generation (manual activation)
68     CompGPActive = true;
69
70     %initialize the filters' meta data
71     if nargout < 2
72         filter_meta = {0};
73     else
74         filter_meta = {1, cell(Graph_Order,2) ,[]};
75     end
76
77     if nargin < 3 || length(filters) ~= Filters_Constant
78         use_filters = ones(1,Filters_Constant);
79         disp('ChainPairs2.m: Default Filter used')
80     else
81         use_filters = filters;
```

```
 82      end
 83
 84      %%
 85      %Scans through Node Connection List
 86      for j = 1:Graph_Order
 87          %Generate Potential Root List (Starting Points)
 88          Root_Index = dL_List(:,1)==j; %Creates logic array
 89          Root_List = dL_List(Root_Index,:); %Which then this scans faster
 90          Root_Length = size(Root_List(:,1),1);
 91
 92          %Cycle through Root List
 93          for i = 1:Root_Length
 94              %Creates a template with the Chain's current Progress
 95              ChainProgress = Root_List(i,:);
 96              %Feeds the Chain template into the recursive system.
 97              [CL,filter_meta] = ChainNext2(dL_List,radix,...
 98                  CL,ChainProgress,...
 99                  use_filters,filter_meta);
100          end
101
102          %%
103          %Stepback Filter
104          if use_filters(1)
105              %Filter Meta Data
106              %Saves the removed value in the debug output
107              if filter_meta{1}
108                  %Saves length of potential GP list at end of
109                  %each Root_Index.
110                  %In other words, "when" the stepback has occurred.
111                  filter_meta{2}{j,1}=size(CL{1},3);
112
113                  %Saves removed value into metadata
114                  if isempty(filter_meta{2}{j,2})
115                      filter_meta{2}{j,2} = dL_List(any(dL_List==j,2),:);
116                  else
117                      filter_meta{2}{j,2} = ...
118                          cat(1, filter_meta{2}{j,2},...
119                          dL_List(any(dL_List==j,2),:));
120                  end
121              end
122
123              dL_List = dL_List(~any(dL_List==j,2),:);
```

```
124             %Finds every instance of the first column and the first row
125             %And removes them from the list
126         end
127     end
128
129     %%
130     %Reorganize CL{1}
131     CmpltChains = CL{1};
132
133
134     %%
135     %Create Composite GroupPairs
136     if CompGPActive && ~isempty(CL{2})
137         Comp = CreateCompGP(CL{2},radix,filters);
138         CompMat = cell2mat(reshape(Comp,1,1,[]));
139         FullList = cat(3,CmpltChains,CompMat);
140     else
141         FullList = CmpltChains;
142     end
143
144     %%
145     postprocequivfil = filters(3);
146
147     %Post-Processing Equivalence Filter
148     if postprocequivfil
149         %Finds the amount of possible group pairs and creates a cell array
for
150         %it. Also creates a reordered version of the list.
151         list = FullList;
152         sizeList = size(list,3);
153
154         %Limit of when Chains end and Composite Groups start
155         sizeSplit = size(CmpltChains,3);
156         %Method: Take every group pair pages, line it up into a single row.
157         %Then reorder it from smallest to largest. Then find any duplicates
158         %with unique().
159         sortedList = sort(reshape(list,[],sizeList))';
160             %In this case, lists out all Group Pairs in an array of
161             %columns. Then transposes the list so that each ROWS represents
162             %a group pair
163         [~, ia, ic] = unique(sortedList,'rows','stable');
164
```

```
        %insert filter metadata
        filter_meta{3}{1} = list;
        filter_meta{3}{2} = ia;
        filter_meta{3}{3} = ic;
        filter_meta{3}{4} = setdiff(1:sizeList,ia); %What was removed
        %Uses the index for unique values returned from uniqueness scan
        %To select which group pairs to keep.
        FullList = list(:,:,ia);
        CmpltChains = list(:,:,ia(ia<=sizeSplit));
        CompMat = list(:,:,ia(ia>sizeSplit));
    end
    CL{1} = FullList;
    CL{3} = {};
    CL{3}{1} = CmpltChains;
    CL{3}{2} = CompMat;
end
```

Published with MATLAB® R2019a

6.1.7 ChainNext2.m

```
function [ CL, filter_meta_update ] = ChainNext2( dL, radix, List, Chain, ...
    filter, filter_meta )
%ChainPairs Synthesizes list of potential node groups by chaining
%connection pairs. At present only handles 2 topology systems.
%   Author: Edgar Pan (edgar.pan@mail.mcgill.ca)
%   Input:
%       NC - "Node Connection" list, most likely generated by GenConnList.m
%               Format: [SourceNode Topology1Node Topology2Node ...]
%       radix - determines the number of ports on a one side of a
%               crosspoint switch.
%       List - The Completed Chain List that's been fed in.
%       Chain - The Node Group Template Fed in. i.e. Current Chain.
%       filter - Settings input for filter activation
%       filter_meta - Metadata for filters.
%   Output:
%       CL - "Chains List" A returned list of the node groups

    %Default Safety Response
    CL = List;
    filter_meta_update=filter_meta;

    %Check Tail end of current Chain
    Tail = Chain(end);
    Head = Chain(1,2);

    %Generate Potential Links List
    Link_Index = dL(:,2)==Tail; %Creates logic array
    Link_List = dL(Link_Index,:); %Which then this scans faster
    Link_Length = size(Link_List(:,1),1);

    %The amount of topologies system will switch between. Should be 2.
    Modes = size(Chain,2)-1;

    %Preallocation
    %Allocating Reference to store all Mismatched Combinations
    Mismatch = [];

    %Variable for checking any occurrence of a Complete List
    Disable_Mismatch = false;

```

```
    %List Data
    sizeList = size(List{1});

    %Filter Settings
    OverlapFil = filter(2);
    EquivFil = filter(3);

    %Catching Minimum Chains, puts them in a cell array for variable sizes
    %Put in this early stage such that it will not account for "Completed"
    %Chains
    if Head == Tail
        %Number of Minimum GP cases found
        minCases = length(CL{2});
        CL{2}{minCases+1} = Chain;
    end

    %Cycle through Link List
    for i = 1:Link_Length

        skip = 0;

        %Checks for Overlap Filter
        if OverlapFil
            for m = 2:Modes+1
                if any(ismember(Chain(:,[1 m]),Link_List(i,[1 m]),'rows'))
                    %Assumption: A specific link will only happen in
                    %specific topology mode.
                    skip = 1;
                    break;
                elseif any(ismember(Chain(:,[1 m]),...
                        fliplr(Link_List(i,[1 m])),'rows'))
                    %Checks also for cases where a backwards link also
                    %overlaps
                    skip = 1;
                    break;
                end
            end
            if skip
                continue;
            end
        end

```

```
 83
 84          %Next Link Selected and Inserted
 85          %Updates template with the Chain's current Progress
 86          ChainProgress = cat(1,Chain,Link_List(i,:));
 87          %Checks the Chain Length
 88          Chain_Length = size(ChainProgress,1);
 89
 90          %Finds the minimum radix for the minimum Chain cases
 91          minChainLength = 0;
 92          if ~isempty(CL{2})
 93              minChainLength = min(cellfun('size',CL{2},1));
 94          end
 95
 96          if EquivFil && Chain_Length > radix - minChainLength &&
~isempty(List{1})
 97              %In-process equivalence filter framework (doesn't do anything)
 98
 99              %Halfway through the chain, if we notice that it's basically a
100              %pre-existing chain, but backwards, skip.
101
102              %The process is relatively straightforward. Just check the
103              %first two columns (how to handle larger cases then?).
104              %Checking only for each Links (i.e. 1 3; 2 4). Then just check
105              %whether similar patterns occur.
106              %The real challenge is doing that for all entries in the list
107              %without needing to iterate through it every time.
108
109              %That was the intent, at least. It had not worked as intended.
110              %As such, the "continue" line is never reached, but to avoid
111              %bugs, this section was left in.
112
113              if length(sizeList)<3 || sizeList(3) < 2
114                  equiv = [ismember(ChainProgress(:,[1 2]),...
115                      List{1}(:,[1 2]),'rows');
116                      ismember(Tail,List{1}(:,2))];
117              else
118                  %currently brute force solution
119                  equiv = zeros(2,sizeList(3));
120                  for eqIdx = 1:sizeList(3)
121 %                       ChainProgress(:,[1 2])
122 %                       List{1}(:,[1 2],eqIdx)
123 %                       equiv(:,eqIdx) = [ismember(ChainProgress(:,[1 2]),...
```

```
124 %                           List{1}(:,[1 2],eqIdx),'rows');
125 %                           ismember(Tail,List{1}(:,2))];
126                 end
127             end
128 
129             if any(all(equiv))
130                 continue
131             end
132 
133         end
134 
135         if Chain_Length < radix
136             %Feeds the Chain template into the recursive system.
137             [CL,filter_meta] =
ChainNext2(dL,radix,CL,ChainProgress,filter,filter_meta);
138         else
139             %If we've reached or surpassed the limit of the radix
140             if Link_List(i,3) == ChainProgress(1,2)
141                 %Case of Complete List, where the last Tail and the first
142                 %Head matches up
143                 CL{1} = cat(3,CL{1},ChainProgress);
144 
145                 %Disables mismatched case because we know matched exists.
146                 Disable_Mismatch = true;
147                 Mismatch = [];
148             elseif ~Disable_Mismatch
149                 Mismatch = cat(3,Mismatch,ChainProgress);
150             end
151         end
152     end
153 end
```

Published with MATLAB® R2019a

6.1.8 CreatCompGP.m

```
function [CLComp] = CreateCompGP(minChains,radix,filters)
%CreateCompGP Compiles Minimum Chains into composite GP blocks
%   Author: Edgar Pan (edgar.pan@mail.mcgill.ca)
%   Basically, this function checks every combination of minimal GPs and
%   records every resulting composite GP that fills up a switch of the
%   indicated radix.
%   Input:
%       minChains - Full list of minimal chains. Essentially GPs that were
%       complete before reaching the desired radix.
%       radix - The radix we want these GPs on.
%       filters - Default ON. Settings for filter. Function only uses the
%       Overlap filter which checks for any instance of redundant links in
%       the generated compGP.
%   Output:
%       CLComp - Chain List Composite
%           The compiled list of Composite Group Pairs.
%
%   ASSUMPTION: Minimal Chains creates can also compose higher order
%   minimum chains as well, hence only deal with Minimal (lowest radix).
%   Only keeps primes.

    %Filter Settings
    if nargin < 3
        OverlapFil = true;
    else
        OverlapFil = filters(2);
    end

    %Finds the list of sizes
    GPlengths = cellfun('size',minChains,1);

    %Finds the types of lengths in list
    LengthTypes = unique(GPlengths);

    keepTypes = true(1,length(LengthTypes));
    %Keeping only Prime Radices
    for L = 1:length(LengthTypes)-1
        if ~keepTypes(L)
            continue;
        end
```

```
41          for J = L+1:length(LengthTypes)
42              if keepTypes(J)
43                  f = factor(LengthTypes(J));
44                  if ismember(LengthTypes(L),f)
45                      keepTypes(J) = false;
46                  end
47              end
48          end
49          if ~any(keepTypes(L+1:end))
50              break;
51          end
52
53          nextIdx = find(keepTypes(L+1:end),1) + L;
54          if ~isempty(nextIdx)
55              if nextIdx > length(LengthTypes)-1
56                  break;
57              end
58              L = nextIdx - 2;
59          end
60      end
61
62      %Acquires Reduced List
63      KeepIdx = ismember(GPlengths,LengthTypes(keepTypes));
64
65      ReducedList = minChains(KeepIdx);
66      RedGPLength = GPlengths(KeepIdx);
67
68      %Gonna brute force the solution
69      maxIdx = length(ReducedList);
70
71      %Creates an array of binary numbers counting from 1 to however many
72      %minimum Chains combos there are, representing the use of a particular
73      %minChain on a switch block (HOB in context of creation) iteration.
74      a = mat2cell([false(1,maxIdx);true(1,maxIdx)],2,ones(1,maxIdx));
75      iteration = setprodcell(a);
76
77      ptnlIte = size(iteration,1); %List of potential iterations
78      keepIte = false(ptnlIte,1); %Pre-allocation of valid iteration space
79
80      for i = 1:ptnlIte
81          ite = iteration(i,:);
82          iteLength = sum(RedGPLength(ite));
```

```
        if iteLength == radix
            keepIte(i) = true;
        else
            keepIte(i) = false;
        end
    end

    keptIte = iteration(keepIte,:);
    nkeptIte = length(keptIte);

    %Preallocating
    unfiltCLComp = cell(1,nkeptIte);
    for c = 1:nkeptIte
        CellBlocks = ReducedList(keptIte(c,:));
        CellBlocks = reshape(CellBlocks,[],1);
        unfiltCLComp{c} = cell2mat(CellBlocks);
    end

    if OverlapFil
        %number of unfiltered Composite Blocks
        nComp = length(unfiltCLComp);

        %number of topological configuration
        modes = size(minChains{1},2);
        overlapIdx = false(1,nComp);
        for f = 1:nComp
            for m = 2:modes
                ovlpcheck = unfiltCLComp{f}(:,[1 m]);

                %Finds indices where for proper IDing of Links, flip the
                %node designations
                flipIdx = ovlpcheck(:,1)>ovlpcheck(:,2);
                ovlpcheck(flipIdx,:) = fliplr(ovlpcheck(flipIdx,:));

                ovlptest = unique(ovlpcheck,'rows');
                if size(ovlptest,1) ~= radix
                    overlapIdx(f) = true;
                end
            end
        end
        CLComp = unfiltCLComp(~overlapIdx);
    else
```

```
125         CLComp = unfiltCLComp;
126     end
127 end
```

Published with MATLAB® R2019a

6.1.9 SelectionProcess.m

```
function [FinalSelection, SelectedIndex] = SelectionProcess(CL, Nec,
ForceTol)
%SelectionProcess Selects Node Groups based on their similarity
%   INPUT
%       CL - Chain Lists from which data will be extracted.
%       Nec - List of Necessary Connections
%       ForceTol - OPTIONAL A setting to force a certain tolerance setting.
%       Faster, but may not give complete solution.
%   OUTPUT
%       FinalSelection - List of the GPs selected
%       SelectedIndex - Index of the selected GPs in the CL list.

    if nargin < 3
        ForceTol = [];
    end

    %Get Group Relations
    [GR, RelData] = GetGroupRelation(CL{1});

    %Get Potential Tolerance Levels
    tolLvls = unique([0;GR{1}(:)]);

    %Start Acquiring Indices Selection based on tolerance
    if ~isempty(ForceTol)
        if ForceTol < 0
            %lower cap
            tolerance = 0;
        elseif ForceTol > length(tolLvls) - 2
            %upper cap
            tolerance = length(tolLvls) - 2;
            % -2 to prevent self-connection
        else
            tolerance = ForceTol;
        end

        SelectedIndex = GroupRelationSelection(GR{1},tolerance);
        FinalSelection = CL{1}(:,:,SelectedIndex);
        if ~isempty(FindMissing(Nec,FinalSelection))
            disp('SelectionProcess.m Warning: ')
            disp('Forced Tolerance Selection yielded incomplete solution.')
```

```
40          end
41      else
42          for tolerance = 0:length(tolLvls)-2
43              SelectedIndex = GroupRelationSelection(GR{1},tolerance);
44              Selected = CL{1}(:,:,SelectedIndex);
45
46              stillMissing = FindMissing(Nec,Selected);
47
48              if isempty(stillMissing)
49                  break;
50              elseif tolerance == 0
51                  disp('SelectionProcess.m: Zero tolerance failed.')
52              end
53          end
54
55          if ~isempty(stillMissing)
56              %Extracting Link Data
57              Links = RelData{1,1};
58              RelLength = length(GR);
59              for g = 2:RelLength
60                  Links(:,:,g) = RelData{g,g};
61              end
62
63              while ~isempty(stillMissing)
64                  %Preallocate contribution count memory,
65                  %clear out NewIndex selection in case of new iteration.
66                  contribution = zeros(RelLength,1);
67                  NewIndex = [];
68
69                  %Scans through for number of potential link contributions
70                  %in each GroupPairs.
71                  for c = 1:RelLength
72                      intersection =
intersect(stillMissing,Links(:,:,c),'rows');
73                      contribution(c) = size(intersection,1);
74                      if contribution(c) == size(stillMissing,1)
75                          %If a GroupPair has all links missing, just pick
76                          %this quick.
77                          NewIndex = c;
78                          break;
79                      end
80                  end
```

```
                if isempty(NewIndex)
                    %If no single GroupPair has all missing links, then
                    %pick the one that contributes the most, then ready for
                    %new iteration.
                    NewIndex = find(contribution == max(contribution));
                end
                NewSelectedIndex = [SelectedIndex NewIndex];

                %Updates
                Selected = CL{1}(:,:,NewSelectedIndex);
                stillMissing = FindMissing(Nec,Selected);
            end
            SelectedIndex = NewSelectedIndex;
        end
        FinalSelection = Selected;
    end


end
```

Published with MATLAB® R2019a

6.1.10 GetGroupRelation.m

```
function [ Count, Data ] = GetGroupRelation( NodeGroups )
%GraphNodeGroup Takes a list of NodeGroups and Graph their relation to each
%other
%   Simply takes of list of Node Group Pairs (probably generated by
%   ChainPairs.m) and creates 2 relations matrices.
%   Input
%       NodeGroups - List of Node Group Pairs
%   Output
%       Count - Cell containing
%           {1} - the intersection matrix (how alike two groups pairs are)
%           {2} - the difference matrix (how different two group pairs are)
%       Data - Cell containing
%           {1} - Cell matrix containing the actual intersecting links
%           {2} - Cell matrix containing the actual differing links
    %%
    %Initialization
```

```
17      sizeNG = [size(NodeGroups,1), size(NodeGroups,2), size(NodeGroups,3)];
18      LinkCount = sizeNG(1)*(sizeNG(2)-1);
19      NGCount = sizeNG(3);
20
21      %Preallocates Link Data Space
22      Links = zeros(LinkCount, 2, NGCount);
23
24      %Preallocates difference and intersection cell space
25      Diff = cell(NGCount);
26      Intersection = cell(NGCount);
27
28      %%
29      %Extract Link Data from Node Groups
30      for g = 1:NGCount
31          Links(:,:,g) = GetGroupLinks(NodeGroups(:,:,g));
32      end
33
34      %%
35      %Create the Difference and Intersection Table
36      for i = 1:NGCount
37          for j = i:NGCount
38              %Only compares upper triangle of matrix to save computation
39              %time
40              intersection = ...
41                  intersect(Links(:,:,i),Links(:,:,j),'rows');
42              diff = ...
43                  setdiff(Links(:,:,i),Links(:,:,j),'rows');
44
45              if size(diff,1) + size(intersection,1) ~= LinkCount
46                  disp('GetGroupRelation.m: Link Count Mismatch')
47                  disp([i j])
48                  disp(size(diff,1) + size(intersection,1))
49              end
50
51              if i == j
52                  Intersection{i,i} = Links(:,:,i);
53              else
54                  Intersection{i,j} = intersection;
55                  Intersection{j,i} = intersection;
56                  Diff{i,j} = diff;
57                  Diff{j,i} = diff;
58              end
```

```
59
60          end
61      end
62
63      IntersectCount = cellfun('size',Intersection,1);
64      DiffCount = cellfun('size',Diff,1);
65
66      %%
67      %Output compilation
68      Count = cell(2,1);
69      Count{1} = IntersectCount;
70      Count{2} = DiffCount;
71
72      Data = cell(2,1);
73      Data{1} = Intersection;
74      Data{2} = Diff;
75  end
```

Published with MATLAB® R2019a

6.1.11 GroupRelationSelection.m

```
1  function [ NodeGroupList, gGraph ] = GroupRelationSelection( GNGCount,
tolerance )
2  %GroupRelationSelection Selects Node Groups based on their relationship
3  %    INPUT
4  %    GNGCount - Matrix of similarity relationships between NodeGroups in a
5  %    NodeGroupList.
6  %    tolerance - Tolerance level. 0 for no similarity. 1 for next minimum
7  %    similarity.
8  %    OUTPUT
9  %    NodeGroupList - Output of the index for the Node Groups
10 %    gGraph - Similarity graph as per the tolerance value.
11
12     %Rather than having the user manually entering the exact similarity
13     %values that are tolerated, system finds the key tolerance values and
14     %user picks "first tolerance values" or so on.
15     tolLvls = unique([0;GNGCount(:)]);
16
17     if nargin < 2 || isempty(tolerance) || tolerance < 0
```

```
 18          tolerance = 0;
 19      elseif tolerance > length(tolLvls) - 2
 20          disp('GroupRelationSelection.m Warning: ')
 21          disp('Tolerance value exceeds levels available in system.')
 22          tolerance = length(tolLvls) - 2; %-2 to prevent self-connection
 23      end
 24
 25      %First, convert the relationship data to graph format based on the
 26      %tolerance value. The default state is tolerance 0, meaning that for
 27      %ANY common links between the Node Groups, there is a link.
 28      gGraph = graph(GNGCount>tolLvls(tolerance+1));
 29
 30      %Identifies isolated components (subgroup of nodes) in graph
 31      [bins,binsizes] = conncomp(gGraph,'OutputForm','cell');
 32      subList = cell(1,length(bins));
 33
 34      for i=1:length(bins)
 35          %Extracts subgraph
 36          subgGraph = subgraph(gGraph,bins{i});
 37          gdist = distances(subgGraph);
 38
 39
 40          %List of Node Groups to use. Logic Array format.
 41          shortlist = false(1, length(gdist));
 42          priorlist = shortlist; %stores prior shortlist state before any
changes
 43          %Selection of Initial Node Group
 44          potStartIdxList = find(
sum(rem(gdist,2)==0)==max(sum(rem(gdist,2)==0)) );
 45 %          potStartIdxList = find(
sum(rem(gdist,2)==0)==min(sum(rem(gdist,2)==0)) );
 46
 47          startIndex = potStartIdxList(ceil(rand*length(potStartIdxList)));
 48          %of the possible options, randomly selects one.
 49
 50          shortlist(startIndex) = 1;
 51          unchanged = isequal(shortlist, priorlist);
 52          while ~unchanged
 53              priorlist = shortlist;
 54              %tdist: test gdist that will get further and further reduced
 55              %Resets the reduced gdist matrix
 56              tdist = -ones(length(gdist));
```

```
 57              %Creates list of potential Next Indices
 58                  %The rem(gdist([shortlist,:),2)==0) part creates a matrix
 59                  %where every row represents one of the prospective indices.
We
 60                  %find every index that is a multiple of 2 steps away.
 61                  %The all is an "and" for every rows. the &~prospective
removes
 62                  %past indices from the potential Index list.
 63              ptnlIdx = rem(gdist(shortlist,:),2)==0;
 64
 65              if size(ptnlIdx,1)>1
 66                  ptnlIdx = all(ptnlIdx) & ~shortlist;
 67              else
 68                  ptnlIdx = ptnlIdx & ~shortlist;
 69              end
 70
 71              %Creates a reduced gdist matrix while maintaining size (and
index)
 72              tdist(ptnlIdx,:) = gdist(ptnlIdx,:);
 73              tdist(:,ptnlIdx) = gdist(:,ptnlIdx);
 74              %Selection of Next Node Group
 75              if any(ptnlIdx) && max(sum(rem(tdist,2)==0))>0
 76 %              if any(ptnlIdx) && min(sum(rem(tdist,2)==0))>0
 77                  potNextIdx=ptnlIdx;
 78                  potNextIdx(ptnlIdx) =
sum(rem(tdist(ptnlIdx,ptnlIdx),2)==0)==...
 79                      max(sum(rem(tdist(ptnlIdx,ptnlIdx),2)==0));
 80                  potNextIdxList = find(potNextIdx & ptnlIdx);
 81                  if ~isempty(potNextIdxList)
 82                      nextIndex =
potNextIdxList(ceil(rand*length(potNextIdxList)));
 83                      %Once selected, modifies the shortlist
 84                      shortlist(nextIndex) = 1;
 85                  end
 86              end
 87              unchanged = isequal(shortlist, priorlist);
 88          end
 89          subList{i} = find(shortlist) + sum(binsizes(1:i-1));
 90      end
 91      %Finalizes the shortlist as output
 92      NodeGroupList = cell2mat(subList);
 93 end
```

Published with MATLAB® R2019a

6.1.12 FindMissing.m

```
function [ Missing ] = FindMissing( Nec, SelectedGroups )
%FindMissing Finds any Links not provided by the so-far selected Groups.
%   Author: Edgar Pan, McGill University. edgar.pan@mail.mcgill.ca

    %%
    %Early catch for Empty SelectedGroup
    if isempty(SelectedGroups) | ~any(SelectedGroups)
        Missing = Nec;
        return;
    end

    %%
    %Extracts essential data from Groups
    Links = GetGroupLinks(SelectedGroups);

    %%
    %The actual list comparison
    Missing = setdiff(Nec,Links,'rows');

    if isempty(Missing)
        Missing = []; %turns any empty row matrix into simple empty value.
    end

end
```

Published with MATLAB® R2019a

6.1.13 GetGroupLinks.m

```
function [ Links ] = GetGroupLinks( GP )
%GetGroupLinks Extracts the information on what Links are provided by a set
%of Group Pairs
%   Output
%       Links - List of All Links provided by the Group Pairs inputed
%   Input
%       GP - Group Pairs
    %%
    %Extracts essential data from Groups
    [radix, topoLength, HOBs] = size(GP);

    List = permute(GP,[1 3 2]);
    List = reshape(List,[],topoLength,1);

    Links = [];
    for i = 2:topoLength
        %Creates a list of all links established by kept GPs
        Links = cat(1,Links,List(:,[1 i]));
    end
    %Re-orders links such that smaller number comes first.
    %i.e. [2 1] becomes [1 2]
    Links(Links(:,1)>Links(:,2),:) = fliplr(Links(Links(:,1)>Links(:,2),:));

    %%
    %Verifies the values of Links
    uLinks = unique(Links,'rows','stable');

    if (size(uLinks,1)~= radix*(topoLength-1)*HOBs)
        disp('GetGroupLinks WARNING: Link Overlap has occurred')
        disp('Recommend verify Group Pair List.')
    end
end
```

Published with MATLAB® R2019a

6.1.14 DrawHobSystem.m

(Extra programs to visualize the results. Minimal Comment because not important to function of main program)

```
function DrawHobSystem (N, Selected, ActiveTopology, HOBLabel, Common)
    if nargin < 5
        DrawHobNetwork(N,Selected,[])
    else
        DrawHobNetwork(N,Selected,Common)
    end
    for i = 1:size(Selected,3)
        figure('Position',[270 400 250 250])
        hold on
        axis([-5 5 -5 5])
        DrawHobSwitch(Selected(:,:,i),0,ActiveTopology)
        if HOBLabel
            title(['H' num2str(i)])
        end
        hold off
    end
end
```

Published with MATLAB® R2019a

6.1.15 DrawHobNetwork.m

(Extra Programs, a few parts based on code provided by Prof Odile Liboiron-Ladouceur)

```
function DrawHobNetwork( N, GroupPairs, inCommon, varargin )
%DrawHobNetwork Draws in a circle a network system interconnected via HOB
%switches.
%   Author: Edgar Pan (edgar.pan@mail.mcgill.ca)

    %%
    %CONSTANTS
    Radius_Outer = 10;

    %%
    %Graph Parameters
    params = struct();
    for var = 1:2:length(varargin)-1
        params.(varargin{var}) = varargin{var+1};
    end

    if isempty(inCommon)
        Common = zeros(N);
    else
        Common = inCommon;
    end

    %%
    %Reads the Numbers of Nodes present and draws out their coordinates

    V = 2*pi/N*(0:N-1);
    XY_N = Radius_Outer*[cos(V); sin(V)]';

    %Analyzes the Group Pairs data.
    sizeGP = [size(GroupPairs,1), size(GroupPairs,2), size(GroupPairs,3)];
    U_init = 2*pi/sizeGP(3)*(0:sizeGP(3)-1);

    %Breaks alignment of nodes between Servers and HOBs
    divisor = 1;
    limit = 100;
    U_shift = U_init;
    while any(ismember(U_shift,V)) && divisor < limit
        U_shift = U_init + pi/divisor;
        divisor = divisor + 1;
```

```
40      end
41      U = U_shift;
42
43      XY_H = Radius_Outer*1/2*[cos(U); sin(U)]';
44
45      %Unifies the coordinate lists of Servers and HOBs
46      W = cat(2,V,U); %list of angles
47      XY = cat(1,XY_N,XY_H); %list of XY coordinates
48
49  %     disp(W)
50
51      %%
52      %Generates the Matrix Data
53      A = zeros(N+sizeGP(3));
54          %The first N nodes represents the Server Nodes
55          %The additional sizeGP(3) nodes represents the HOB blocks.
56      A(1:N,1:N) = eye(N);
57          %Marks the Servers as Self Connecting.
58          %This is just a notation in order to ID and distinguish Servers
59          %from HOBs
60      for i = 1:sizeGP(3)
61          for r = 1:sizeGP(1)
62              A(N+i,GroupPairs(r,1,i)) = A(N+i,GroupPairs(r,1,i)) + 1;
63              A(GroupPairs(r,2,i),N+i) = A(GroupPairs(r,2,i),N+i) + 1;
64          end
65      end
66
67      %%
68      %Parsing the Matrix Data
69
70      %Self-connecting edges.
71      Serv = diag(diag(A));
72      HOBs = diag(~diag(A));
73
74      %Stores HOBs only links. Remove the self-connection.
75      hA = A - diag(diag(A));
76
77      %Stores the Adjacency Matrices for "left" and "right" side of HOB for
78      %infrastructure purposes. Not really used. Potential for future
79      %expansion.
80      HOB_In = tril(hA,-1);
81      HOB_Out = triu(hA,1);
```

```
 82
 83      %Permanent Connections
 84      %Creates a larger Adjacency matrix and inserts the Common Links in.
 85      Perm = zeros(N+sizeGP(3));
 86      Perm(1:N,1:N) = Common;
 87
 88      %Compile the full Adjacency matrix between the Server nodes and the
 89      %HOB nodes.
 90      Full = hA + Perm;
 91
 92      %%
 93      %Splitting the thicker connections into separate matrices
 94      hiBWs = hA + hA' > 1;
 95
 96
 97      %%
 98      %Convert to Plot form
 99      [hiBWX,hiBWY] = makeXY(hiBWs,XY);
100      [ServX,ServY] = makeXY(Serv,XY);
101      [HOBsX,HOBsY] = makeXY(HOBs,XY);
102      [PermX,PermY] = makeXY(tril(Perm,0),XY); %Permanent Connection coord
103      [HOBIX,HOBIY] = makeXY(HOB_In,XY);
104      [HOBOX,HOBOY] = makeXY(HOB_Out,XY);
105
106
107      %%
108      %Initialization of the figures
109      figure
110      hold on
111
112      %%
113      %With the Servers and HOBs marked, now it's just a matter of drawing
114      %the lines representing the connections.
115      %Note: The earlier line is plotted, the lower in layer it is.
116
117      plot(PermX,PermY,'-','Color',[0.8 0.8 0.8],params)
118
119      plot(hiBWX,hiBWY,'-','Color',[0.75 0.75 0.75],'Linewidth',2.5,params)
120
121  %     plot(HOBIX,HOBIY,'-','Color',[0.9 0.5 0.7],params)
122  %     plot(HOBOX,HOBOY,'-','Color',[0.4 0.8 0.8],params)
123      plot(HOBIX,HOBIY,'-','Color',[0.4 0.4 0.4],params)
```

```
124     plot(HOBOX,HOBOY,'-','Color',[0.4 0.4 0.4],params)
125 %     plot(HOBIX,HOBIY,'-','Color','K',params)
126 %     plot(HOBOX,HOBOY,'-','Color','K',params)
127
128     %%
129     %With the Coordinates set, now it's a matter of marking them on a map.
130     plot(ServX,ServY,'o','Color',[.3 0 0],params)
131     plot(HOBsX,HOBsY,'s','Color',[.3 0 0],params)
132
133
134     %%
135     %Labeling
136     for G = 1:N
137      text(XY_N(G,1),XY_N(G,2),['  S'
num2str(G)],'Color','G','FontSize',12,'FontWeight','b')
138     end
139     for G = 1:sizeGP(3)
140      text(XY_H(G,1),XY_H(G,2),['  H'
num2str(G)],'Color','C','FontSize',10,'FontWeight','b')
141     end
142     hold off
143
144     %%
145     function [x,y] = makeXY(A,xy)
146         if any(A(:))
147             [J,I] = find(A');
148             m = length(I);
149             xmat = [xy(I,1) xy(J,1) NaN(m,1)]';
150             ymat = [xy(I,2) xy(J,2) NaN(m,1)]';
151             x = xmat(:);
152             y = ymat(:);
153         else
154             x = NaN;
155             y = NaN;
156         end
157     end
158 end
```

Published with MATLAB® R2019a

6.1.16 DrawHobSwitch

```
function DrawHobSwitch( GroupPair, shift, topoSet )
%DrawHobSwitch Draws the internal configuration of the individual switches
%   Author: Edgar Pan (edgar.pan@mail.mcgill.ca)
    %%
    %Initialization
    sizeGP = size(GroupPair);
    radix = sizeGP(1);

    if nargin < 2 || isempty(shift)
        shift = [0 0];
        disp(topoSet)
    end
    %%
    %Drawing the Base Rectangle, representing the Physical Case.
    boxCorn = [-1.5 -2.5] + shift;
    boxSize = [3 5];
    rectangle('Position', [boxCorn boxSize])

    %%
    %Setting the Vertical Coordinates for the Ports
    portGap = boxSize(2)/(radix+1);
    portVert = boxCorn(2):portGap:(boxCorn(2)+boxSize(2)-portGap);
    portVert = flip(portVert(2:length(portVert)));

    portCoordL = [ones(radix,1)*boxCorn(1) portVert'];
    portCoordR = [ones(radix,1)*(boxCorn(1)+boxSize(1)) portVert'];

    %%
    %Draws out the ports and labels them.
    portSize = [min(portGap/6, 1) min(portGap/2,1)];
    portLabelL = GroupPair(:,1);
    portLabelR = GroupPair(:,2);
    %Left
    for L = 1:radix
        rectangle('Position', [portCoordL(L,1)-portSize(1) ...
            portCoordL(L,2)-portSize(2)/2 portSize])
        text(portCoordL(L,1)-portSize(1)-0.2,portCoordL(L,2),...
            ['S' num2str(portLabelL(L))],'Color','G',...
            'FontSize',12,'FontWeight','b','HorizontalAlignment','Right')
    end
```

```
    for R = 1:radix
        rectangle('Position', [portCoordR(R,1) ...
            portCoordR(R,2)-portSize(2)/2 portSize])
        text(portCoordR(R,1)+0.5,portCoordR(R,2),...
            ['S' num2str(portLabelR(R))],'Color','G',...
            'FontSize',12,'FontWeight','b')
    end

    %%
    %Goes through the GroupPair list and draws out each links
    RGB = [0 0 0];
    for j = 2:sizeGP(2)
        available = true(radix,1);
        if j == 2
            RGB = [0 0 1]; %Blue
        elseif j == 3
            RGB = [1 0 0]; %Red
        end
        if topoSet + 1 == j
            style = '-';
        else
            style = '--';
        end
        for i = 1:sizeGP(1)
            %Connecting GP(i,1) to GP(i,j)
            Source = portCoordL(i,:);
            DestPotentIndex = (portLabelR == GroupPair(i,j)) & available;
            selectedDest = find(DestPotentIndex,1);
            available(selectedDest) = 0;
            Destination = portCoordR(selectedDest,:);

            plot([Source(1) Destination(1)],...
                [Source(2) Destination(2)],style,'Color',RGB)
        end
    end
end
```

Published with MATLAB® R2019a

6.2 Tables

(Tables in next few pages)

6.2.1 Top 20 HPL and HPCG (November 2017 Results) – Part 1 Supercomputer Backgrounds

HPL	HPCG	Name	*Computer*	Company	Latest		Topology
					Year	Mo	Name
1	5	**Sunway TaihuLight**	*Sunway MPP*	NRCPC	2016	06	Sunway NRCPC
2	2	**Tianhe-2 (MilkyWay-2**	*TH-IVB-FEP Cluster*	NUDT	2013	06	TH Express-2
7	3	**Trinity**	*Cray XC40*	Cray Inc.	2017	11	Cray Aries
4	-	**Gyoukou**	*ZettaScaler-2.2 HPC sys*	ExaScaler	2017	11	InfiniBand EDR
8	7	**Cori**	*Cray XC40*	Cray Inc.	2016	11	Cray Aries
5	9	**Titan**	*Cray XK7*	Cray Inc.	2012	11	Cray Gemini
3	4	**Piz Daint**	*Cray XC50*	Cray Inc.	2017	06	Cray Aries
9	6	**Oakforest-PACS**	*PRIMERGY CX1640*	Fujistu	2016	11	Intel Omni-Path
6	8	**Sequoia**	*IBM BlueGene/Q*	IBM	2013	06	BG/Q
12	-	**Stampede2**	*PowerEdge C6320P/C6420*	Dell EMC	2017	11	Intel Omni-Path
14	28	**Marconi**	*CINECA Cluster, Lenovo SD5*	Lenovo	2017	11	Intel Omni-Path
13	10	**TSUBAME3.0**	*SGI ICE XA, IP139-SXM2*	HPE	2017	11	Intel Omni-Path
10	1	**K Computer**	*K Computer*	Fujistu	2011	11	Tofu Interconnect
16	15?	**MareNostrum**	*Lenovo SD530*	Lenovo	2017	11	Intel Omni-Path
11	12	**Mira**	*IBM BlueGene/Q*	IBM	2012	06	BG/Q
18	-	**Theta**	*Cray XC40*	Cray Inc.	2017	06	Cray Aries
-	18	**Stampede**	*PowerEdge C8220*				InfiniBand FDR
15	-	**-**	*Cray XC40*	Cray Inc.	2016	11	Cray Aries
19	14	**Hazel Hen**	*Cray XC40*	Cray Inc.	2015	11	Cray Aries
20	16	**Shaheen II**	*Cray XC40*	Cray Inc.	2015	06	Cray Aries
17	11	**Pleiades**	*SGI ICE X*	HPE	2016	11	InfiniBand FDR
21	13	**Pangea**	*SGI ICE X*	HPE	2016	06	InfiniBand FDR
22	19	**JUQUEEN**	*BlueGene/Q*	IBM	2013	06	BG/Q
24	20	**Cheyenne**	*ICE XA*	HPE	2016	11	InfiniBand EDR
38	17	**SORA-MA**	*Fujitsu PRIMEHPC FX100*	Fujistu	2016	06	Tofu Interconnect 2

6.2.2 Top 20 HPL and HPCG (November 2017 Results) – Part 2 Supercomputer Processor and Topologies

HPL	HPCG	Name	Processor			Power (kW)	Total Cores	Rpeak (Tflops/s)	Rpeak/Cores (Gflops/s)
			Name	Cores	Speed				
1	5	**Sunway TaihuLight**	Sunway SW26010	260	1.45GHz	15,371	10,649,600	125,435.9	11.778
2	2	**Tianhe-2**	Intel Xeon E5-2692	12	2.20GHz	17,808	3,120,000	54,902.4	17.597
7	3	**Trinity**	Intel Xeon Phi 7250	68	1.40GHz	3,844	979,968	43,902.6	44.800
4	-	**Gyoukou**	Xeon D-1571	16	1.30GHz	1,350	19,860,000	28,192.0	1.420
8	7	**Cori**	Intel Xeon Phi 7250	68	1.40GHz	3,939	622,336	27,880.7	44.800
5	9	**Titan**	Opteron 6274	16	2.20GHz	8,209	560,640	27,112.5	48.360
3	4	**Piz Daint**	Xeon E5-2690v3	12	2.60GHz	2,272	361,760	25,326.3	70.009
9	6	**Oakforest-PACS**	Intel Xeon Phi 7250	68	1.40GHz	2,719	556,104	24,913.5	44.800
6	8	**Sequoia**	Power BQC	16	1.60GHz	7,890	1,572,864	20,132.7	12.800
12	-	**Stampede2**	Intel Xeon Phi 7250	68	1.40GHz	-	368,928	18,215.8	49.375
14	28	**Marconi**	Intel Xeon Phi 7250	68	1.40GHz	-	314,384	15,372.0	48.896
13	10	**TSUBAME3.0**	Xeon E6-2680v4	14	2.40GHz	792	135,828	12,127.1	89.283
10	1	**K Computer**	SPARC64 VIIIfx	8	2.00GHz	12,660	786,432	11,280.4	14.344
16	15	**MareNostrum**	Xeon Platinum 8160	24	2.10GHz	1,632	153,216	10,296.1	67.200
11	12	**Mira**	Power BQC	16	1.60GHz	3,945	786,432	10,066.3	12.800
18	-	**Theta**	Intel Xeon Phi 7230	64	1.30GHz	1,087	231,424	9,627.2	41.600
-	18	**Stampede**				-	522,080	9,000.0	17.239
15	-	-	Xeon E5-2695v4	18	2.10GHz	-	241,920	8,128.5	33.600
19	14	**Hazel Hen**	Xeon E5-2680v3	12	2.50GHz	3,615	185,088	7,403.5	40.000
20	16	**Shaheen II**	Xeon E5-2698v3	16	2.30GHz	2,834	196,608	7,235.2	36.800
17	11	**Pleiades**	Intel Xeon E5-2670/E5-2680v2-4		2.4—2.8	4,407	241,108	7,107.1	29.477
21	13	**Pangea**	Xeon E5-2670/E5-2680v3	8&12	2.6/2.5	4,150	220,800	6,712.3	30.400
22	19	**JUQUEEN**	Power BQC	16	1.60GHz	2,301	458,752	5,872.0	12.800
24	20	**Cheyenne**	Xeon E5-2697v4	18	2.30GHz	1,727	144,900	5,332.3	36.800
38	17	**SORA-MA**	SPARC64 Xifx	32	1.98GHz	1,652	110,160	3,481.1	31.600

6.2.3 Top 20 HPL and HPCG (November 2017 Results) – Part 3 Topology and HPL and HPCG Results

HPL	HPCG	Name	Cores /CPU	Topology Type	Total Cores	CPU (Nodes)	Rpeak (Tflops/s)	HPL		HPCG	
								Rmax	Rpeak%	R	Rpeak%
1	5	**Sunway TaihuLight**	260	Sunway	10,649,600	40,960.0	125,435.9	93,014.6	74.2%	481	0.4%
2	2	**Tianhe-2**	12	Fat Tree	3,120,000	260,000.0	54,902.4	33,862.7	61.7%	580	1.1%
7	3	**Trinity**	68	Dragonfly	979,968	14,411.3	43,902.6	14,137.3	32.2%	546	1.2%
4	-	**Gyoukou**	16	Switched Fabric	19,860,000	1,241,250.0	28,192.0	19,135.8	67.9%	-	-
8	7	**Cori**	68	Dragonfly	622,336	9,152.0	27,880.7	14,014.7	50.3%	355	1.3%
5	9	**Titan**	16	Torus 3D	560,640	35,040.0	27,112.5	17,590.0	64.9%	322	1.2%
3	4	**Piz Daint**	12	Dragonfly	361,760	30,146.7	25,326.3	19,590.0	77.4%	486	1.9%
9	6	**Oakforest-PACS**	68	Switched Fabric	556,104	8,178.0	24,913.5	13,554.6	54.4%	385	1.5%
6	8	**Sequoia**	16	Torus 5D	1,572,864	98,304.0	20,132.7	17,173.2	85.3%	330	1.6%
12	-	**Stampede2**	68	Switched Fabric	368,928	5,425.4	18,215.8	8,317.7	45.7%	-	-
14	28	**Marconi**	68	Switched Fabric	314,384	4,623.3	15,372.0	7,471.1	48.6%	69	0.4%
13	10	**TSUBAME3.0**	14	Switched Fabric	135,828	9,702.0	12,127.1	8,125.0	67.0%	189	1.6%
10	1	**K Computer**	8	Torus 6D/Mesh	786,432	98,304.0	11,280.4	10,510.0	93.2%	603	5.3%
16	15?	**MareNostrum**	24	Switched Fabric	153,216	6,384.0	10,296.1	6,470.8	62.8%	122	1.2%
11	12	**Mira**	16	Torus 5D	786,432	49,152.0	10,066.3	8,586.6	85.3%	167	1.7%
18	-	**Theta**	64	Dragonfly	231,424	3,616.0	9,627.2	5,884.6	61.1%	-	-
-	18	**Stampede**		Switched Fabric	522,080		9,000.0	5,168.0	57.4%	97	1.1%
15	-	-	18	Dragonfly	241,920	13,440.0	8,128.5	7,038.9	86.6%	-	-
19	14	**Hazel Hen**	12	Dragonfly	185,088	15,424.0	7,403.5	5,640.2	76.2%	138	1.9%
20	16	**Shaheen II**	16	Dragonfly	196,608	12,288.0	7,235.2	5,537.0	76.5%	114	1.6%
17	11	**Pleiades**		Switched Fabric	241,108		7,107.1	5,951.6	83.7%	175	2.5%
21	13	**Pangea**	8&12	Switched Fabric	220,800		6,712.3	5,283.1	78.7%	163	2.4%
22	19	**JUQUEEN**	16	Torus 5D	458,752	28,672.0	5,872.0	5,008.9	85.3%	95	1.6%
24	20	**Cheyenne**	18	Switched Fabric	144,900	8,050.0	5,332.3	4,788.2	89.8%	87	1.6%
38	17	**SORA-MA**	32	Torus 6D/Mesh	110,160	3,442.5	3,481.1	3,157.0	90.7%	110	3.2%

7 Reference

[1] (October 12). *Top500 List - November 2019*. Available: https://www.top500.org/lists/top500/2019/11/

[2] S. Wang, D. Li, J. Geng, Y. Gu, and Y. Cheng, "Impact of Network Topology on the Performance of DML: Theoretical Analysis and Practical Factors," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, 2019, pp. 1729-1737.

[3] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 63-74, 2008.

[4] E. Pan. (2020). *NGS*. Available: https://github.com/EdgarPan/NGS

[5] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., 2003.

[6] L. Youyao, H. Jungang, and D. Huimin, "A Hypercube-based Scalable Interconnection Network for Massively Parallel Computing," *Journal of Computers*, vol. 3, no. 10, pp. 58-65, October 2008.

[7] Y. Saad and M. H. Schultz, "Topological properties of hypercubes," *IEEE Transactions on Computers*, vol. 37, no. 7, pp. 867-872, 1988.

[8] S. Cheng, W. Zhong, K. E. Isaacs, and K. Mueller, "Visualizing the Topology and Data Traffic of Multi-Dimensional Torus Interconnect Networks," *IEEE Access*, vol. 6, pp. 57191-57204, 2018.

[9] (January 15). *Top500 List - November 2017*. Available: https://www.top500.org/lists/top500/list/2017/11/

[10] C. E. Leiserson, "Fat-trees: Universal networks for hardware-efficient supercomputing," *IEEE Transactions on Computers*, vol. C-34, no. 10, pp. 892-901, October 1985.

[11] C. Clos, "A study of non-blocking switching networks," *The Bell System Technical Journal*, vol. 32, no. 2, pp. 406-424, 1953.

[12] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-Driven, Highly-Scalable Dragonfly Topology," in *2008 International Symposium on Computer Architecture*, Beijing, China, 2008, vol. 36, pp. 77-88.

[13] I. H. White, Q. Cheng, A. Wonfor, and R. V. Penty, "Large port count optical router using hybrid MZI-SOA switches," in *2014 16th International Conference on Transparent Optical Networks (ICTON)*, 2014, pp. 1-5.

[14] M. Y. Teh, J. J. Wilke, K. Bergman, and S. Rumley, "Design Space Exploration of the Dragonfly Topology," in *High Performance Computing*, Cham, 2017, pp. 57-74: Springer International Publishing.

[15] A. Shpiner, Z. Haramaty, S. Eliad, V. Zdornov, B. Gafni, and E. Zahavi, "Dragonfly+: Low Cost Topology for Scaling Datacenters," presented at the 2017 IEEE 3rd International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB), 2017, 2017. Available: https://ieeexplore.ieee.org/document/7885210/

[16] H.-W. Meuer, E. Strohmaier, and J. Dongarra. (1993). *Top500 Supercomputer Sites*. Available: http://www.netlib.org/benchmark/top500.html

[17] J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK Benchmark: past, present and future," *Concurrency and Computation: Practice and Experience,* vol. 15, no. 9, pp. 803-820, July 14 2003.

[18] J. J. Dongarra, "The LINPACK Benchmark: An explanation," Berlin, Heidelberg, 1988, pp. 456-474: Springer Berlin Heidelberg.

[19] J. Dongarra, M. Heroux, and P. Luszczek, "High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems," *International Journal of High Performance Computing Applications,* vol. 30, no. 1, pp. 3-10, August 17 2015.

[20] S. A. Jyothi, A. Singla, P. B. Godfrey, and A. Kolla, "Measuring and Understanding Throughput of Network Topologies," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Salt Lake City, UT, 2016, pp. 761-772.

[21] ReflexPhotonics, "Hybrid Optical Bridge - Hardware for the Software Defined Network," ed, 2016.

[22] O. Liboiron-Ladouceur, "Software-enabled energy-efficient hardware infrastructure for next-generation data centres," 2017.

[23] P. Samadi, K. Wen, J. Xu, Y. Shen, and K. Bergman, "Reconfigurable optical dragonfly architecture for high performance computing," in *2016 Optical Fiber Communications Conference and Exhibition (OFC)*, 2016, pp. 1-3.