ABSTRACT OF DISSERTATION

Timothy Ian Mattox

The Graduate School

University of Kentucky

2006

EXPLOITING SPARSENESS OF COMMUNICATION PATTERNS FOR THE DESIGN OF NETWORKS IN MASSIVELY PARALLEL SUPERCOMPUTERS

ABSTRACT OF DISSERTATION

A dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy in the
College of Engineering
at the University of Kentucky

By
Timothy Ian Mattox

Lexington, Kentucky

Director: Dr. Henry G. Dietz,
Professor of Electrical and Computer Engineering and James F. Hardymon Chair in Networking

Lexington, Kentucky

2006

ABSTRACT OF DISSERTATION

EXPLOITING SPARSENESS OF COMMUNICATION PATTERNS FOR THE DESIGN OF NETWORKS IN MASSIVELY PARALLEL SUPERCOMPUTERS

A limited set of Processing Element (PE) pairs in a parallel computer cover the internal communications of scalable parallel programs. We take advantage of this property using the concept of Sparse Flat Neighborhood Networks (Sparse FNNs). Sparse FNNs are network designs that provide single-switch latency and full wire bandwidth for each specified PE pair, despite using relatively few network interfaces per PE and switches that have far fewer ports than there are PEs. This dissertation discusses the design problem, runtime support, and working prototype (KASY0) for Sparse FNNs. KASY0 not only demonstrated the claimed properties, but also set world records for its price/performance and performance on a specific application.

Parallel supercomputers execute many portions of an application simultaneously. For scalable programs, the more PEs the system has, the greater the potential speedup. Portions executing on different PEs may be able to work independently for short periods, but the performance desired might not be achieved due to delays in communication between PEs. The set of PE pairs that will communicate often is both predictable and small relative to the number of possible PE pairings. This sparseness property can be exploited in the design and implementation of networks for massively parallel supercomputers.

The sparseness of communicating pairs is rooted in the fact that each of the human-designed communication patterns commonly used in parallel programs has the property that the number of communicating pairs grows relatively slowly as the number of PEs is increased. Additionally, the number of pairs in the union of all communication patterns used in a suite of parallel programs grows surprisingly slowly due to pair synergy: the same pair often appears in multiple communication patterns. Detailed analysis of communication patterns clearly shows that the number of PE pairs actually communicating is very sparse, although the structure of the sparseness can be complex.

KEYWORDS: Parallel Supercomputer, Communication Latency, Guaranteed Bandwidth, Scalable Interconnection Networks, Sparse Communication Patterns.

Timothy I. Mattox

June 29, 2006

EXPLOITING SPARSENESS OF COMMUNICATION PATTERNS FOR THE DESIGN OF NETWORKS IN MASSIVELY PARALLEL SUPERCOMPUTERS

By
Timothy Ian Mattox

Dr. Henry G. Dietz
Director of Dissertation

Dr. YuMing Zhang
Director of Graduate Studies

June 29, 2006

RULES FOR THE USE OF DISSERTATIONS

Unpublished dissertations submitted for the Doctor's degree and deposited in the University of Kentucky Library are as a rule open for inspection, but are to be used only with due regard to the rights of the authors. Bibliographical references may be noted, but quotations or summaries of parts may be published only with the permission of the author, and with the usual scholarly acknowledgments.

Extensive copying or publication of the dissertation in whole or in part also requires the consent of the Dean of the Graduate School of the University of Kentucky.

A library that borrows this dissertation for use by its patrons is expected to secure the signature of each user.

Name Date

DISSERTATION

Timothy Ian Mattox

The Graduate School

University of Kentucky

2006

EXPLOITING SPARSENESS OF COMMUNICATION PATTERNS FOR THE DESIGN OF NETWORKS IN MASSIVELY PARALLEL SUPERCOMPUTERS

DISSERTATION

A dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy in the
College of Engineering
at the University of Kentucky

By
Timothy Ian Mattox

Lexington, Kentucky

Director: Dr. Henry G. Dietz,
Professor of Electrical and Computer Engineering and James F. Hardymon Chair in Networking

Lexington, Kentucky

2006

For Kathleen and Samantha…

Acknowledgments

This dissertation would not have been possible without the financial, moral, and academic support of a surprisingly large number of individuals and organisations. My parallel computing research has been brewing for so many years that I fear I can not remember everyone by name that I should. Thus, I first give thanks to all the unnamed students and researchers, including the simply curious kids (and adults) on field trips, who peaked into our computing labs over the years, both at Purdue and the University of Kentucky, wanting to find out what I was doing with so many racks of computers. Their continued interest has motivated me to keep going, and to try to accomplish creative things to inspire the next generation of computer engineers.

I must thank my committee chair, Hank Dietz, for his long hours (over so many years that I dare not count them) dedicated to helping me make my research work farther reaching and more sound. His drive to find the better answer, the more elegant approach, and his ever present belief that I could just "make it work" has helped me achieve amazing things under his watch. Although I am moving on, I look forward to and will appreciate continued collaboration with Hank, and of course his friendship.

My friend and colleague Bill Dieter deserves special mention. Bill, thanks for sharing your white-board and the time to talk through various arcane details of FNN research. Thanks for introducing me to Yats that first week I was at UK, at a time when I hardly knew anyone. Bill, you made me feel at home here at UK much sooner than I dared hope. I'll miss our semi-regular game nights, and hope we can each find more people to play Illuminati, Settlers of Catan, StarCraft, Primordial Soup, etc.

I would like to thank the generous financial support of NASA, and the James F. Hardymon Chair endowment, that have funded my work at the University of Kentucky. I would also like to thank Steve Zelencik of Advanced Micro Devices, for supporting our research group at Purdue and for donating the Athlon processors used in KLAT2, the first machine with a FNN.

My colleagues on the NASA EPSCoR grant have been a great resource. I thank Thomas Hauser for his enthusiastic adoption of the new cluster technologies I worked on, and especially for his efforts on our Gordon Bell award submission. Thanks go out to Ray LeBeau for keeping me sane during some stressful times, and for his continued efforts in demonstrating that my cluster research is usable in a production setting. I thank Tim Dowling for sharing his unique perspectives with me, and for his motivational talks which got me started writing my dissertation. Seeing my work directly help in the pursuit of planetary atmospheric science is inspiring. I thank George Huang for

his efforts in pulling three distinct research groups together to leverage each of our own strengths towards a common goal of applying cluster computational power for real science and engineering results.

I must thank all the students and researchers in the Comparative Planetology Laboratory (CPL) at the University of Louisville, the Computational Fluid Dynamics (CFD) Group at the University of Kentucky, and the KAOS (Compilers, Hardware Architectures, and Operating Systems) lab for their efforts in constructing, testing, repairing, and making use of the various computer clusters I was responsible for while at UK. I must especially thank Lakshmi not only for her efforts in helping me keep KLAT2 and KASY0 operational, but also for her time spent reading and reviewing drafts and sections of this dissertation.

There is a broader community of people that I must also thank for their efforts which made this dissertation possible. I speak of the Free and Open Source Software (FOSS) developers which have created Linux and all the various tools and software that go with it. Specifically, I thank Jeff Squyres for his efforts on LAM/MPI and for his encouragement each time we saw each other at the Supercomputing conferences. I also thank Greg Kurtzer for developing the Warewulf cluster management toolkit, which allowed me to get KASY0 up and operational without having to create something like Warewulf on my own.

I wish to extend my thanks to my committee members and outside examiner for their constructive criticisms, supportive comments, and overall enthusiasm for my work. Prof. Henry (Hank) G. Dietz, Prof. J. Robert Heath, Prof. James E. Lumpp, Prof. Raphael A. Finkel, Prof. Kenneth L. Calvert, and Prof. Craig C. Douglas all deserver many thanks for serving on my committee, and giving so generously of their time. I especially thank Prof. Finkel and Prof. Calvert for their detailed help in making this dissertation a much better document. I have already thanked my advisor for his collaboration in general during the many years I have worked with him, but I must single Hank out for the extraordinary effort he put in so that I could defend this dissertation in May 2006. Thank you so much!

I would be remiss if I did not mention my family, especially my parents for helping me along this path towards a Ph.D. I thank my Mom and Dad for instilling in me a wonder and awe for science and engineering. I thank my sister Holly for the many understanding phone conversations about the tough parts of doing research and how to cope. I thank my sister Peggy for being so supportive of me throughout this long process.

I close my acknowledgements with a thank you for the two most important people in my life: my wife and my daughter; the two women who have motivated me finally to finish. Thank you Kathleen for being understanding and supportive, even when things didn't seem to progress. Thank you for making me chocolate chip cookies at just the time I needed them. And most importantly of all, thank you Kathleen for motivating me to finish so that we could move on with our lives. Finally, I must thank Samantha, my newborn daughter who has no knowledge of computers, of school, or even words, for setting an all important deadline that I must not miss. When I hold Samantha in my arms, I am so thankful that I am alive and able to help bring such joy into the world.

Contents

List of Tables

List of Algorithms

List of Figures

Chapter 1

Introduction

1.1 Scope of Work

This dissertation examines the feasibility of designing custom networks for parallel computers that efficiently support the communications of target suites of scalable parallel programs. Specifically it shows:

1. Individual communication patterns of scalable parallel programs are sparse.
2. The union of many communication patterns is sparse, in part due to pair synergy.
3. A Sparse Flat Neighborhood Network (Sparse FNN) efficiently and simultaneously supports these communication patterns.
4. The complex problem of designing Sparse FNNs is solved with a combination of Heuristic and Genetic Algorithm techniques.
5. Sparse FNNs scale to the largest size parallel machines built today, with over 65 thousand nodes.
6. The practical details in using Sparse FNNs, such as message routing, are either solved or solvable.
7. The cost and performance benefits of Sparse FNNs are directly demonstrated in a parallel computer, KASY0.

1.2 Background

In traditional parallel computers, the Processing Elements (PEs)[1] do the real work of a parallel program. Thus, when building or buying a new parallel computer one would like to maximize the

[1]This dissertation uses the term PE to interchangeably mean “processor,” “core,” “CPU,” or “uniprocessor node,” as distinguished from hardware units that may execute semi-independently despite being physically grouped together, such as “multi-core processor” or “multiprocessor node.”

aggregate computational power of the PEs. The network connecting those PEs greatly influences the achieved performance of a program based on how well the network satisfies the program's communication requirements. Yet, any money spent on the network means less money is spent on the PEs, so there is a cost vs. performance trade-off between the PEs and the network connecting them. An engineering approach to network design seems appropriate to resolve this cost vs. performance trade-off.

Network design is one of the fundamental problems in high-performance parallel computing today and has been so since the beginning of parallel computer design. The emphasis in network design traditionally has been on selecting a "universal" topology with good mathematical properties and then mapping program communications onto that network architecture[22, 42, 58, 69]. A given network design can be evaluated based on a variety of criteria including cost per PE, average communication latency between PE pairs, available bandwidth between PE pairs, and the number of PEs that can be effectively supported by the network (i.e. the network's scalability). A useful secondary property of a network, bisection bandwidth[22], is also traditionally used to compare parallel computer networks. Bisection bandwidth is defined as the link bandwidth times the minimum number of links that must be cut when dividing the network into two halves with equal numbers of PEs.

Many factors affect these network evaluation criteria, some of which are more easily controlled than others. Whether network links directly connect PEs, or if there are switches between the PEs, will affect the latency of individual messages; especially if they need to traverse multiple links. The topology of the links between components affects the scalability, the bisection bandwidth, and the average latency between PEs. The base technology implementing the links determines their minimum latency and maximum bandwidth. The broad use of network components (links, switches, etc.) outside the high performance computing market reduces their cost due to economies of scale. The method of routing messages inside the network, such as circuit switched, packet switched, wormhole switched[22], will affect the latency of messages, among other effects. The quality and design of the software interface to the network can greatly affect the latency and bandwidth between PEs. Clearly, there are many factors that contribute to the evaluation of a particular network design.

1.3 Traditional Network Architectures

Parallel computer networks can be broadly classified into two groups, direct and indirect networks[22]. Direct networks are those that employ point to point links between PEs; indirect networks are those where PEs connect through switches. The former usually requires routing through PEs that have multiple Network Interfaces (NIs), while the latter usually does all the routing of packets in the switches. Some of the literature[3, 69] calls these two categories of networks by different names, static and dynamic, because the apparent connectivity of a direct network is static, while the indirect networks have dynamic connectivity. Implementations do not always fit clearly into any of these categories. For example, some commercial networks utilize dedicated switches at each PE

so that message routing does not interfere with a PE's computations, but are physically wired as direct networks. In this paper we will distinguish between these categories based on whether a PE is associated with each node in the representative graph for a network: Direct networks have one or more PEs associated with each node in the graph, and indirect networks only have PEs associated with a proper subset of the graph's nodes.

A direct fully-connected network has the lowest latency of any network, and is a great design for small numbers of PEs. However it obviously does not scale to even medium-sized parallel computers, due to its $O(N^2)$ wiring complexity and the *N-1* NIs needed per PE. A multi-dimensional mesh is a common category of direct networks. They can be found in a variety of commercial machines such as the 2D mesh in the Intel Paragon XP/S[5], and the 3D tori in the Cray T3D[17] and IBM BlueGene/L[24] machines. A hypercube is another common direct network topology found in commercial machines such as the Thinking Machines CM-1[60] and CM-2[62], and the nCUBE[50] series of machines. Both the mesh and hypercube network topologies scale better than a direct fully-connected network but can suffer from high latency between many PE pairs, especially when scaled to large numbers of PEs. In graph theory terms, the worst-case latency between PEs in a direct network grows with the diameter of the network[2]. One alternative, yet to be seen in a commercial machine but widely discussed, is to use networks constructed from Cayley graphs[2, 69] with high-degree nodes to keep the graph diameter small. Another interesting approach to keeping the graph diameter small is to use a binary de Bruijn directed graph[57] which uses degree-4 nodes and has a diameter of only $\log_2 N$. Although a de Bruijn graph was used in JPL's Galileo project for an 8,192 PE signal processing computer called the Big Viterbi Decoder[14, 15], de Bruijn graphs have practical difficulties that have limited their use for general purpose parallel supercomputers.

The fastest indirect network would contain a single crossbar switch connecting all the PEs, but a crossbar does not scale to large numbers of PEs due to its $O(N^2)$ switch-point complexity. In order to meet the scaling criteria, one can use multiple switches to construct the network, sometimes called a Multistage Interconnection Network (MIN)[22]. A simple tree with the PEs at the leaves and switches at the interior nodes scales very cost effectively but does not have a high bisection bandwidth, which limits its effectiveness. To alleviate the bisection bandwidth problem, additional switches can be added to the tree topology to maintain a constant link count at each layer; this topology is a Fat-Tree[42], which has been used in commercial machines such as the Thinking Machines CM-5[43, 63]. Alternatively, the switches in a MIN can be arranged to form a non-blocking Clos[12] network. Yet another arrangement of switches in a MIN, called the Butterfly network, was used in machines from BBN Advanced Computers Inc.[4] There are many variants on MINs found in the literature[58] that have various routing schemes, offer differing levels of fault-tolerance, and support various sets of conflict-free permutations.

Common to all these traditional scalable-network architectures, which excludes the crossbar and

[2]The diameter of the graph for an indirect network is not strictly related to the worst-case latency between PEs, because not all nodes in the graph represent PEs. The distance between two nodes in the graph is only relevant to the worst-case latency if both nodes represent PEs. For example, the distance between a switch and a PE is unimportant, yet could be larger than the distance between any two PEs in the graph.

the direct fully-connected network, is the need for many messages to be routed through multiple intermediate network nodes. A message incurs a switching/routing delay at each intermediate node in its path (e.g. for each switch-hop). The amount of delay depends not only on the technology used to construct the network node, but also on the switching protocol used. A common switching protocol is called store-and-forward, which does as its name implies: A packet is fully stored in a buffer in the switch prior to it being forwarded. This buffering is done so that corrupted packets can be detected and discarded by the switch. The primary alternative is to do what is called cut-through routing, which begins forwarding a packet as soon as enough of the header has been observed. Although cut-through routing reduces latency, the switch is not able to discard corrupted packets. The IEEE standards for switched Ethernet require the use of the store-and-forward method for this reason. Therefore, commodity Ethernet switches employ the store-and-forward method that delays a packet by at least a full wire delay, which can be quite significant. Thus, most high end commercial parallel machines use non-commodity (i.e. costly) networking hardware that supports some form of cut-through switching/routing, usually some variant of wormhole routing[51]. Although cut-through/wormhole routing helps mitigate switch-hop delay, this delay can never be eliminated.

Another aspect common to all the traditional scalable-network architectures is that many of the interior links along the path between network nodes are shared between multiple PE to PE paths. This sharing can lead to either routing complexity and/or bandwidth bottlenecks and possibly increased latency. For scalable-networks with multiple paths between PE pairs, it can be difficult, if not impossible, for non-centralized routing algorithms to guarantee that none of these shared links is oversubscribed. If the network doesn't have multiple paths between PE pairs (a tree for example), multiple messages may have to contend for the bandwidth of a shared link. In either case, if multiple messages contend for a link, then either one of them blocks and experiences increased latency, or the effective bandwidth of the link is shared, causing each message to experience reduced bandwidth.

1.4 Non-topological Approaches to Improving Latency and Bandwidth

For high performance parallel computing, communication latency and bandwidth both are very important. While network topologies can influence how these performance criteria change as one scales a network design from tens to tens of thousands of PEs, the base performance of a network at any size is constrained by its implementation technologies, both hardware and communications support software. It generally is possible to tune the latency and bandwidth of a given network topology by selecting among the various implementation technologies that can support the routing required by that topology.

As of this writing, Fast Ethernet, Gigabit Ethernet, 10G Ethernet, InfiniBand[34], SCI[21], Myrinet[7], QsNet[56], and several other hardware implementation technologies are available at various performance[9, 44] and cost levels. Most of these network technologies have very similar conceptual properties, for example, all but SCI use bidirectional links. Due to the Spanning Tree Protocol (STP) of the various Ethernet technologies, topologies with cycles would require man-

aged Layer-3 routers, while cycle-free topologies could make use of unmanaged Layer-2 switches, which often are less expensive. In addition to standalone network products, the systems architect has the option to use custom link and/or switching technologies, though the cost of designing and implementing specialized new technologies is beyond the budget constraints of all but a few supercomputer vendors. In addition to fast signaling rates for high link bandwidth, many of the network technologies commonly used for supercomputing employ various latency reducing techniques such as cut-through routing and OS-bypass methods such as those associated with VIA (Virtual Interface Architecture)[59, 70]. The GAMMA[27] project has implemented Active Messages[66] for specific Ethernet Network Interface (NI) adapters, which greatly reduces the software overhead, and thus the communication latency, for networks using supported hardware. Fast Messages[55] is another support software model intended to reduce the latency of communications; it has been implemented for Myrinet, cLAN[59], and the custom network within the Cray T3D[17].

1.5 Overgeneralization: Five Trees Do Not A Forest Make

The primary aspect of network design that this dissertation addresses is the fact that making the network overly general has a high complexity, performance, and monetary cost. KASY0, discussed in Section 6.2, is so effective because it is specialized to handle the five communication patterns that matter – and not all the other possible patterns that do not appear in any of the intended applications.

A small number of special-purpose application-specific computing systems have used networks that are designed to provide precisely the performance needed – no more and no less. For example, GRAPE-6[45] is the latest in a series of designs that essentially hard-wire the data paths that implement the calculation of the gravitational interaction between particles: the name GRAPE actually stands for "GRAvity piPE." This extreme level of specialization has yielded a variety of performance records. GRAPE has been recognized by no fewer than seven Gordon Bell Awards, and did so at a very modest cost by supercomputer standards. On a smaller scale, Graphics Processing Units (GPUs) in modern video cards have profited from the same type of extreme specialization in interconnecting function units.

Network designers for relatively general-purpose supercomputers have been coping with the problem that, unlike the above examples, the hardware must support more than a single fixed communication pattern. Rather than trying to find a reasonably tight cover for the very complex set of communication patterns used by a large class of applications, designers of networks interconnecting PEs within a supercomputer revert to selecting among a small number of "standard" network designs that are known to give acceptable performance for nearly any pattern imaginable. In fact, before the work presented in this dissertation, it was not clear that a useful set of communication patterns could be supported with significantly less hardware than the standard designs require, nor was it clear that these customized designs would deliver markedly better performance.

Historically, Non-Recurring Engineering (NRE) cost for creation of a supercomputer has been notoriously larger than the market for any one supercomputing application could make profitable. It

is not just that NRE cost is high, but also that creating a design typically required building custom components and custom interfaces between them, which takes a long time – and time to market is a critical issue in a field where a six-month delay corresponds to a $1.4\times$ increase in the performance of the competition[64]. However, in 1994 a new approach began to emerge, most commonly known as "Beowulf" [61] and commodity-based cluster computing. This approach uses mostly standard components and interfaces to build a parallel supercomputer, thus dramatically reducing the NRE cost and development time. Additionally, the useful lifespan of a system is extended because components and subsystems can be interchanged with newly developed ones without scrapping the design or even most of the hardware and system software. More significantly for the purpose of this dissertation, use of interchangeable parts also means that it is cheap to support "mass customization" – the ability to individually tune the design for each system without incurring any major cost penalty. If the network had to be hand-designed for each system, the NRE cost and development time per system still would be prohibitive; fortunately, this dissertation proves that at least a fairly large class of these design problems can be fully automated, producing the full benefit with the only NRE costs being the formulation of the design problem and the execution time of the design software.

1.6 Dissertation Walk Through

Chapter 2 begins by introducing the Flat Neighborhood Network (FNN) concept and continues with a discussion of the surprisingly large size of the FNN solution space. The middle of the chapter presents a corpus of communication patterns as found in the literature. Each pattern is discussed as well as presented graphically. The concept of pair synergy is presented along with tables showing how a variety of communication patterns have significant overlap. The chapter concludes with a discussion of how FNNs can be split into Universal, Sparse and Fractional FNNs. The Sparse FNN concept is a core contribution of this dissertation.

Chapter 3 describes the tools for designing FNNs. The first FNN design tool is a Genetic Algorithm (GA) for design of Universal FNNs. Its discussion is followed by a presentation of a tool, created as part of this dissertation's work, that generates Sparse FNN design specifications based on selected communication patterns. The greedy heuristic that was developed to design Sparse FNNs is then presented in detail. That is followed by a discussion of the final Sparse FNN design tool that was developed to directly incorporate the heuristic within a GA. This Sparse FNN GA was parallelized to support more efficient exploration of the Sparse FNN design space using a parallel supercomputer to run the design searches. The chapter concludes with a discussion about how the tools explore different parameter sets that lead to alternative Sparse FNN design solutions.

Chapter 4 presents empirical information about scaling gleaned from solving a large number of different Sparse FNN design problems – considering billions of potential designs to create millions of "optimized" designs exploring scalability of solutions to approximately a thousand different parameter sets. This data reveals that a useful mixture of communication patterns can be simultaneously supported by Sparse FNNs using commodity network hardware to interconnect tens to tens of

thousands of PEs. The chapter concludes with a presentation of a Sparse FNN design for a machine with 65,536 PEs – essentially the same size as the network in the largest BlueGene/L, which is generally accepted as the fastest supercomputer ever built.

Chapter 5 discusses practical implementation details for actually deploying FNNs in real parallel systems. Specifically, several different approaches to solve the routing problem for FNNs are presented. That discussion is followed by a section detailing the specific implementation of FNN runtime support in the Linux OS. Following that section is a discussion about options for fault tolerance on FNNs. The chapter concludes with a discussion of possible FNN implementations using technologies such as InfiniBand.

Chapter 6 presents details about KLAT2 and KASY0, two record-breaking supercomputers that we constructed to be the first systems utilizing Universal and Sparse FNNs, respectively. The section on KLAT2 discusses its importance as the inspiration for this dissertation. The KASY0 machine is then presented in detail, with some application performance results. The chapter concludes with a discussion about KASY0's network and its role in this dissertation.

Chapter 7 begins by listing the various institutions that are known to be using our FNN technology. The chapter continues with a list of potential areas for future study relating to FNNs. Following this list is a philosophical discussion of the overall trends in network design.

Chapter 8 concludes the dissertation by summarizing the thesis and the results of this work.

Chapter 2

A New Network Design Solution

We suggest that network form should follow function: The best network for a parallel supercomputer is the design which, of all feasible networks, yields performance characteristics best matching the latency and bandwidth needs of the targeted parallel program(s) while simultaneously satisfying the relevant cost and scalability constraints. The primary problem in taking this approach is that the design space is surprisingly large and complex. There are vast numbers of possible network designs, and there are even more possible parallel programs, each with its own communication requirements. This chapter discusses our approach to solving this complex problem by simplifying the design space on both fronts.

In Section 2.1, we introduce Flat Neighborhood Networks (FNNs) which are a class of networks that are defined by set of latency and bandwidth properties. By restricting our search to FNNs, the overall design problem is simplified, though the solution space is still quite large as discussed in Section 2.2. In Section 2.3 we discuss the communication patterns of parallel programs which further refine the requirements for design solutions. Finally in Section 2.4 we close the chapter with a taxonomy of FNNs.

2.1 The Flat Neighborhood Network (FNN) Architecture

A Flat Neighborhood Network (FNN)[26] is a type of switching network that provides specific latency and bandwidth properties. The concept of FNNs was first demonstrated by H.G. Dietz and the author in early 2000 with the construction of the KLAT2 (Kentucky Linux Athlon Testbed 2) supercomputer[18, 19, 20, 30]. A FNN is a hybrid network that can be symmetric or asymmetric, with some properties seen in direct networks, yet it is truly an indirect network as described in the literature. Like a direct network, each PE in a FNN has multiple network links, yet these links generally do not connect directly to other PEs. Although FNNs are indirect networks, each message should incur the latency of passing through only a single switch to reach its destination (often called a single switch-hop), thus yielding low latency and guaranteed bandwidth between PE pairs.

Although FNNs can be built with a wide range of network implementation technologies, it is clear that commodity network technologies are a particularly good match for the technique, so the

discussion, examples, and prototypes favor Ethernet technology networks in a Beowulf/cluster context. Section 5.4 discusses the viability of some alternative implementation technologies. Due to the latency overhead of Ethernet's store-and-forward packet switching standard, it is especially important to maintain the FNN single switch-hop design constraint for any frequently communicating PE pair.

When executing a parallel program p on an N PE parallel computer, a PE i needs to communicate with a set L of other PEs. We call that set $L(p, i)$, the neighbor list of PE i for program p. To guarantee low latency and conflict free bandwidth, PE i must have a single switch-hop path to each neighbor PE in L. For small *N*, this goal is achievable using a network consisting of just one switch which is connected to every PE. For larger values of N, instead of using a hierarchy of switches, our solution is to use multiple NIs from PE i to connect to several switches, where each of the neighbors in L is connected to at least one of those switches. A FNN is a network which satisfies the single switch-hop property for each $L(p, i)$.

Finding a minimal graph that satisfies the FNN single switch-hop connectivity constraints turns out to be surprisingly difficult. The problem is actually a minor variation of the well-known graph/set theory problem called (v, k, t)-covering design[13, 28, 52]. A (v, k, t)-covering design is a family of k-element subsets, called blocks, whose members are chosen from the set $\{1, 2, \ldots, v\}$, such that each t-element subset is contained in at least one of the blocks. The number of PEs corresponds to v, the number of ports per switch corresponds to k, and pairwise grouping of PEs implies that t would be equal to 2. Finding a (v, k, t)-covering design with the minimum number of blocks is an open problem in mathematics. The only known general algorithm for finding a minimal covering is through exhaustive search, which is impractical for the numbers of PEs that are interesting to designers of parallel supercomputers. An excellent discussion of the standard covering problem, a summary of recent research on methods for constructing covering designs, and a database of the best known solutions and bounds on solutions are given at the La Jolla Covering Repository[38]. The FNN design problem differs from the standard covering problem primarily in that the number of network interfaces per PE is constrained; there is no corresponding constraint on (v, k, t)-covering design. Thus, a network that satisfies the FNN properties for all PE pairs is also a $(v, k, 2)$-covering design, but a $(v, k, 2)$-covering design is not necessarily a realizable FNN.

The design problem for FNNs also is similar to a very common design problem in statistics and scientific experiments called Balanced Incomplete Block Designs (BIBD) which are subsets of problems called t-designs and Partially Balanced Incomplete Block Designs (PBIBD)[13]. Although these statistics problems have similar properties to the FNN design problem, none of them is as close as the (v, k, t)-covering design problem. In particular, *Fisher's inequality*[13] states that a FNN that is a true BIBD would have at least as many switches as there are PEs, and each PE would have at least as many NIs as there are ports on each switch.

Until our work on KLAT2, the supercomputing networking literature does not seem to contain the FNN concept of connecting each PE to multiple switches in a flat topology to provide single-switch latency. One can speculate that it does not appear prior to that time due to the considerable

computational complexity of the graph problem in its general form, as found in the above related problems. The following section discusses how complex the FNN problem really is, which leads us to propose that a Genetic Algorithm (GA) is the key to finding FNN designs with a reasonable amount of effort. Use of a GA is not a standard approach in the literature for finding (v, k, t)-covering designs, but simulated annealing[53] is, and GAs often perform better than simulated annealing on related problems with unknown smoothness and complex metrics. At the same time we were building KLAT2, a group at Australia's National University designed a geometrically constructed symmetric FNN for the Bunyip Supercomputer[1]. Its network design was not generalized to other configurations. In an earlier work by R. Elbaum and M. Sidi[23] a GA was used for computer network design in 1995. However, the metrics used in that GA were not directly relevant for parallel computers, and the resulting designs did not have FNN properties.

There also is prior art involving use of search procedures to design switchless networks for parallel machines. Work by groups at the University of Bristol[11] and the University of Essex[40, 41, 67] in the 1990s applied various GA techniques to optimizing small irregular graphs as switchless networks for Transputers. The work published in Fall 2002 by Lakamraju et al.[39] is similar, but instead of using a GA, they used a filtering technique on randomly generated regular graphs. All these approaches have some cursory similarity to our FNN approach, but they restrict themselves to switchless designs in which PEs are directly connected to each other. Most importantly, none of the metrics that they used directly correspond to performance on a user-specified set of communication patterns.

2.2 The Size of the FNN Solution Space

The design problem for FNNs can be viewed as a search problem on a particular set of graphs. FNNs are members of the set of undirected bipartite graphs with N PEs on one side and S switches on the other, with each PE having at most η NIs and each switch having at most ρ ports. We will call this set of graphs $B_{N,S,\eta,\rho}$. All FNNs with the given parameters (N, S, η, ρ) are included in the set $B_{N,S,\eta,\rho}$, though this set might contain graphs which are not FNNs. Typically, the number of switches is determined by the switch width and the total number of NIs of all the PEs: $S = \left\lceil \frac{N \times \eta}{\rho} \right\rceil$. It is worthwhile to find the cardinality of the set of graphs $|B_{N,S,\eta,\rho}|$, to get a feel for how much effort is worth spending on designing better algorithms for finding FNNs. In this subsection, we explore various upper bounds on the size of the FNN solution space.

One approach towards estimating $|B_{N,S,\eta,\rho}|$ is to represent the bipartite graph as a matrix with two rows and W columns, where $W = \max(S \times \rho, N \times \eta)$, which is typically the number of wires in the network. The top row represents all the switch ports, and the bottom row represents all the NIs, and each column is a network wire connecting the given switch port and NI. By permuting the entries in one of the rows (either one works), all feasible networks with the given hardware can be generated, while maintaining the η and ρ constraints, which yields $|B_{N,S,\eta,\rho}| \leq W!$. Although this expression is an exact count for all possible ways of wiring the physical network given the

constraints, it vastly overestimates the number of networks that are functionally different (in GA terms, it counts genotypes rather than phenotypes). This approach counts graphs that differ only in the port and/or NI number used to connect a particular PE and switch. An improvement to this estimate is found by dividing out the permutations of the ports within each switch, yielding $|B_{N,S,\eta,\rho}| \leq \frac{(S\times\rho)!}{(\rho!)^S}$, or similarly by dividing out the permutations of the NIs for each PE, yielding $|B_{N,S,\eta,\rho}| \leq \frac{(N\times\eta)!}{(\eta!)^N}$. Unfortunately, one can not simultaneously divide out both sets of NI and port permutations, because in some cases in this representation, a PE may have multiple NIs connected to a particular switch, and for those cases, the permutations get divided out twice. Additionally, this permutation construction method will generate non-simple bipartite graphs, i.e. graphs with multiple connections between a particular PE and switch. Although it is possible to physically wire a network this way, and the FNN definition includes such patterns, this author believes such cases are not particularly beneficial, though some special situations[1] do warrant their use.

Another approach for estimating $|B_{N,S,\eta,\rho}|$ comes from representing the graphs using a $N \times S$ matrix, with each cell holding the number of connections between a particular PE and a particular switch. If we restrict our search to simple graphs (as discussed above), each cell of the matrix contains either zero or one. Thus, there are $2^{N\times S}$ possible matrices, which encompasses all possible simple undirected bipartite graphs with N PEs and S switches. By restricting our matrix to have only W ones, and thus simple bipartite graphs with W wires, we get a better bound $|B_{N,S,\eta,\rho}| \leq |B_{N,S,W}| = \left(\begin{array}{c} N \times S \\ W \end{array} \right)$. When restricting the search to networks with ρ ports per switch, the number of port restricted networks is $|B_{N,S,\rho}| = \left(\begin{array}{c} N \\ \rho \end{array} \right)^S$ from the fact that each of the S switches will connect to ρ PEs from the set of N PEs. Similarly, when restricting the search to networks with a maximum of η NIs per PE, the number of NI restricted networks is $|B_{N,S,\eta}| = \left(\begin{array}{c} S \\ \eta \end{array} \right)^N$ from the fact that each of the N PEs will connect to η switches from the set of S switches. Each of these latter two sets of graphs are subsets of the bipartite graphs $B_{N,S,W}$, and thus their cardinalities are even better upper bounds on $|B_{N,S,\eta,\rho}|$. It is rather difficult to find a closed form expression for the number of graphs with both the ρ and η restrictions. One can find a simple approximation by assuming the sets $B_{N,S,\rho}$ and $B_{N,S,\eta}$ are uncorrelated subsets of $B_{N,S,W}$, namely $|B_{N,S,\eta,\rho}| \approx \frac{|B_{N,S,\rho}|\times|B_{N,S,\eta}|}{|B_{N,S,W}|} = \frac{\left(\begin{array}{c} N \\ \rho \end{array} \right)^S \times \left(\begin{array}{c} S \\ \eta \end{array} \right)^N}{\left(\begin{array}{c} N \times S \\ W \end{array} \right)}$. Unfortunately, the two subsets are not independent. Fortunately, this approximation, $\frac{|B_{N,S,\rho}|\times|B_{N,S,\eta}|}{|B_{N,S,W}|}$, is quite good because it appears to be within a factor of two of the actual value for $|B_{N,S,\eta,\rho}|$; empirically, it is approximately a 50%

[1] A PE in the parallel machine that is acting as a manager or server may benefit from having multiple links to the same switch, thus giving it more bandwidth to the collection of worker PEs also connected to that switch. However, using multiple connections to the same switch only weakly improves fault tolerance because the switch becomes a common failure point.

N	S	η	ρ	W	$\lvert B_{N,S,\eta,\rho}\rvert$	$\frac{\lvert B_{N,S,\rho}\rvert \times \lvert B_{N,S,\eta}\rvert}{\lvert B_{N,S,W}\rvert}$	Ratio	Number of FNNs
3	3	2	2	6	6	9	1.446	6
6	3	2	4	12	90	133	1.473	90
9	3	2	6	18	1,680	2,489	1.482	1,680
4	6	3	2	12	1,860	2,761	1.484	720
8	4	2	4	16	44,730	67,092	1.500	0
6	6	3	3	18	297,200	451,343	1.519	21,600
5	10	4	2	20	56,586,600	86,657,527	1.531	3,628,800
8	6	3	4	24	60,871,300	93,396,534	1.534	3,402,000
10	6	3	5	30	14,367,744,720	22,174,227,646	1.543	152,409,600
12	6	4	8	48	154,700,988,750	240,073,862,451	1.552	154,700,988,750
12	6	3	6	36	unknown	5,760,579,740,420	unknown	19,196,100,000

Table 2.1: FNN Solution Space sizes

overestimate of $|B_{N,S,\eta,\rho}|$ for the small values of N and S for which the author has run exhaustive searches, as shown in Table 2.1.

A difficulty with all these expressions is that they count isomorphic graphs differing only in the numbering of switches as distinct. Unfortunately, dividing by $S!$ to remove these duplicate networks is inaccurate, because it may not be feasible to eliminate these isomorphic duplicates from the space actually explored by our search.

In addition to the $|B_{N,S,\eta,\rho}|$ and $\frac{|B_{N,S,\rho}|\times|B_{N,S,\eta}|}{|B_{N,S,W}|}$ values for a few small cases, Table 2.1 shows the exact number of FNNs for each case; these FNNs were found through a pruned exhaustive search for FNNs that supplied single-switch communication paths for all PE pairs.[2] As one can see, the growth rate of $|B_{N,S,\eta,\rho}|$ is astronomical, and for most real-world design cases, the FNN design space is too large to be exhaustively searched using supercomputers currently available or expected in the near future. One also can see that for a particular (N, S, η, ρ) parameter set, if there are any FNN solutions, there tend to be many to choose from. It is interesting to note that for a given N, there are a variety of (S, η, ρ) triples that might have a solution. Thus, when one designs a FNN, if there are several economically or otherwise viable choices for ρ and/or η, the total search space would be the sum of the various viable $|B_{N,S,\eta,\rho}|$ values. It is obvious, but perhaps disturbing, to further note that very few of the designs in the search space have any simple type of symmetry; considering only symmetric designs may miss the only viable solutions.

Now that the reader has seen the definition of FNNs and a rough estimate of the complexity of the FNN solution space, it is appropriate to discuss determination of the communication patterns which might be important to a suite of parallel programs. The next section discusses types of communication patterns and how they affect the number of neighbors each PE within a FNN must have.

[2] Later, this dissertation will distinguish FNNs as listed in this table from the Sparse FNNs which are the primary focus of this work. FNNs as discussed here are a subclass of Sparse FNNs called Universal FNNs.

2.3 Communication Patterns

What do applications need from a particular network design? The applications need the messages between communicating PEs to be delivered reliably and in a timely manner. What resources the network needs to accomplish that depends on the message sizes, quantity of messages, and the pattern of message source and destination pairs. A communication pattern defines a set of communicating pairs in a parallel computer. Many basic communication patterns are *1:1* and *onto* mappings of PEs onto PEs (*permutations*). For a given parallel program to perform well, its communication patterns must be efficiently supported by the network, giving low latency and high bandwidth for the messages within each pattern.

It is a simple fact of arithmetic that the number of neighbors each PE might need to talk to grows linearly with the number of PEs in a machine. More precisely, in a system with N PEs, the number of possible communication pairs involving a particular PE is $2(N-1)$ for unidirectional (ordered) pairs or $N-1$ for bidirectional (unordered) pairs. These formulas also correspond to the well-known fact that a direct fully-connected network would require $N(N-1)$ unidirectional wires or $\frac{N(N-1)}{2}$ bidirectional links. Thus, network complexity seems to scale as *O(N^2)*. However, when one reviews the various communication patterns commonly discussed in the parallel processing literature, it can be argued that most parallel programs require high performance on only a small fraction of the possible pairs for each PE. As discussed in the following sections, the fraction actually used in typical applications sharply decreases as the number of PEs is increased!

2.3.1 $O(1)$ Scaling Patterns

It is ironic that, despite the parallel processing community's concerns about the complexity of large scale networks, many commonly used communication patterns in parallel programs are patterns that have a constant number of pairs per PE independent of the number of PEs. This observation is supported by the fact that many of the largest systems built, such as the older Intel Paragon and Cray T3D machines and the current day IBM BlueGene machines, have successfully used networks with a simple mesh topology.

The number of bidirectional communication pairs in which each PE is involved when communicating with adjacent PEs within a 1D mesh is either one or two. For a toroidal 1D mesh (i.e., a ring), each PE participates in two bidirectional communication pairs; a non-toroidal 1D mesh differs only in that the two end PEs participate in just one pair each. The per-PE pair count is unaffected by the total number of PEs. It is useful to consider representing this pattern (and others) as a $N \times N$ connectivity matrix, with a column for each PE and a row for each PE. The cells of the matrix contain information about the connection between the cell's row PE and its column PE. In the simplest case, each cell contains either a one or a zero to indicate connected or not connected, respectively. If bidirectional links are used in constructing a network, there is a diagonal symmetry introduced to that network's connectivity matrix, because a connection from PE x to PE y would also support communication from PE y to PE x.

Figure 2.1 shows a representation of the simple bidirectional ring pattern for machines with 16, 32, 64, 128, and 256 PEs. Each box in the figure shows the upper right triangle of the connectivity matrix for the PEs, turned into a pixelated image. For a PE pair (x, y) where $x = y$ the PE is talking to itself; most networking technologies allow such a communication to be accomplished without actually touching the transport layer of the network, so such a communication may trivially be ignored because it imposes no requirements on the network transport hardware. For a PE pair (x, y) where $x > y$ that is in the communication pattern, there is a black pixel at the coordinates (x, y) with the origin in the upper left of the box, with increasing PE numbers going from left to right and from top to bottom. Thus for this ring pattern, there is a diagonal line that is just offset by one pixel from the center diagonal. The lower left triangle of the connectivity matrix if shown, would simply be a mirror image of the upper right triangle.

The conceptually most complex communication patterns commonly used in parallel programs are typically patterns consisting of a single pair per PE. For example, the *bit-reversal*[29] communication pattern, shown in Figure 2.2, may have a reasonably complex formula for which PEs communicate with each other, but each individual PE only is involved in a single pair. The same is true of *perfect-shuffle*[36], which is shown in Figure 2.3; it is significant that, given bidirectional links, *inverse-shuffle* is implemented by the exact same pairing that *perfect-shuffle* uses. For a machine with $N = k^2$ PEs, one can define a *matrix-transpose* communication where each PE containing a single element of a 2D matrix would exchange its element with one other PE to form the transposed matrix. Figure 2.4 shows this *matrix-transpose* pattern for 16, 64, and 256 PEs. Notice that all these patterns are constructed from permutations, so, with an appropriate network, each permutation can be implemented in a single message time-step. Perhaps surprisingly, for many traditional network topologies, each permutation would take multiple time-steps.

The design space becomes significantly larger when 2D meshes are considered, because there may be multiple ways to factor the PEs into a 2D mesh. For example, a 32-PE system could be viewed as the 2D ordered factorization 2×16, 4×8, 8×4, or 16×2 or, more commonly, as an unordered factorization listing dimensions in a normalized order, such as largest dimension first: 8×4 and 16×2. Where this dissertation refers to factorization without specifying the type, it refers to the normalized unordered factorization.

Once a factorization is selected, the pair count per PE is independent of the total number of PEs. Communicating with PEs that are adjacent by row or column yields between two and four pairs per PE, with edge PEs in non-toroidal meshes having the lower pair counts. Figure 2.5 shows the 2D torus patterns with four neighbors per PE on a single factorization of each of 16, 32, 64, 128, and 256 PE patterns. When including multiple factorizations of the same 2D torus pattern, as shown in Figure 2.6, the number of neighbors per PE is no longer a constant, because as the number of PEs increases, the number of possible 2D factorizations increases. However, for each individual factorization, the number of neighbors per PE is a constant.

A 3D torus has six rectilinear neighbors per PE, which are at ± 1 offsets along each of the three dimensions. The connectivity matrix for a single 3D torus factorization for each of 16, 32, 64, 128,

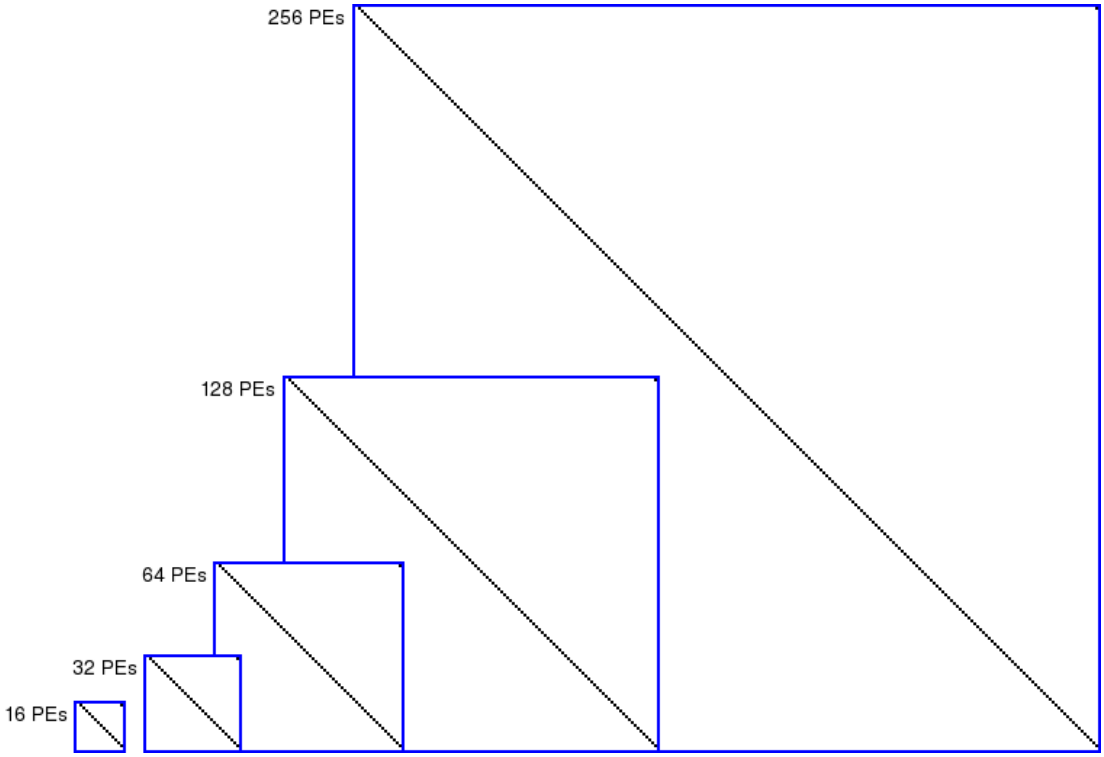

Figure 2.1: 1D Torus with ± 1 offsets (a Ring)

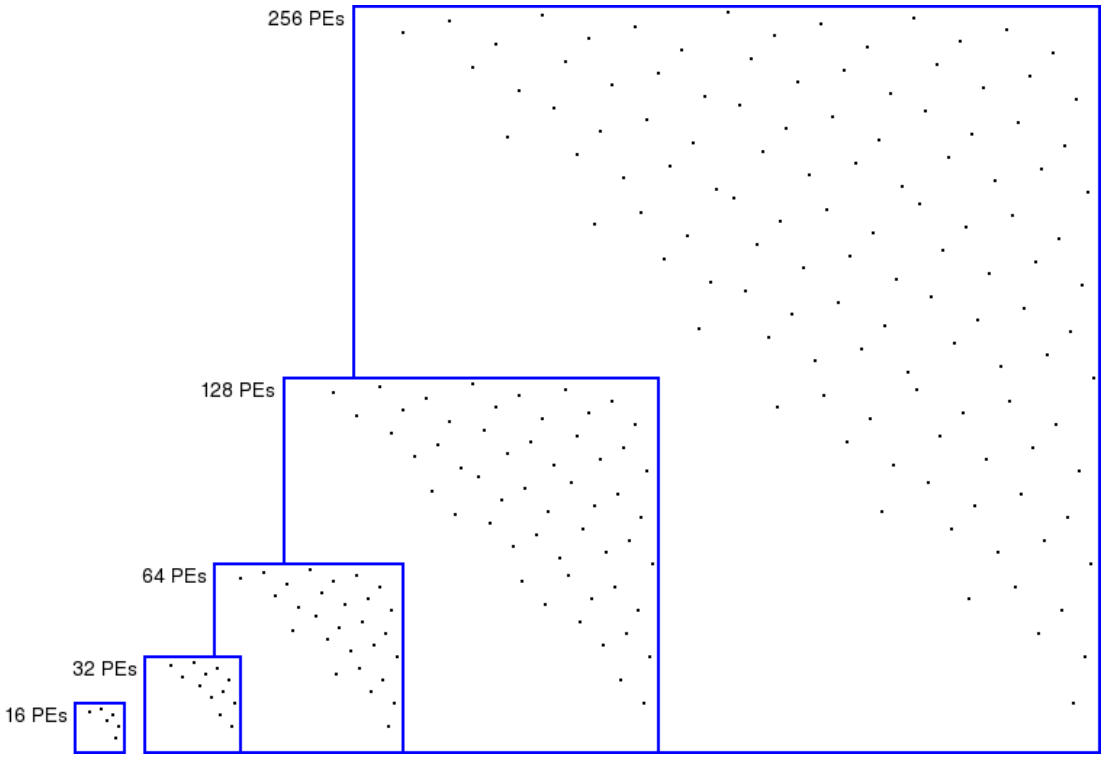

Figure 2.2: Bit-Reversal communication patterns

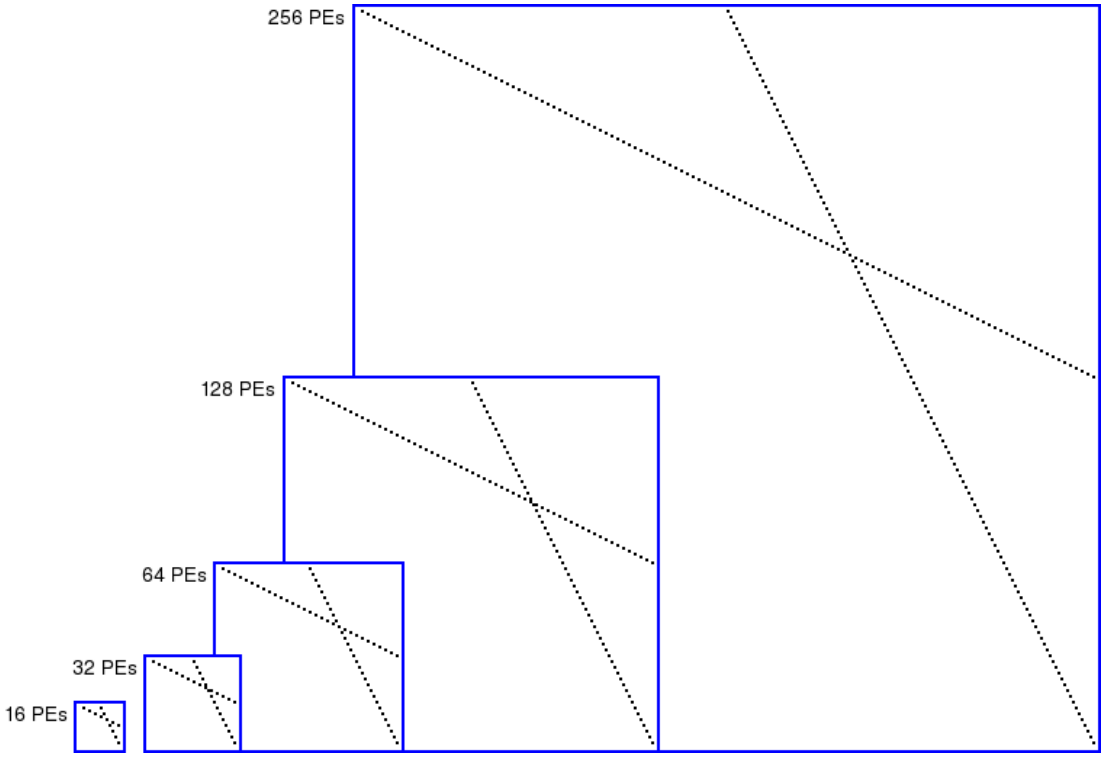

Figure 2.3: Perfect-Shuffle communication patterns

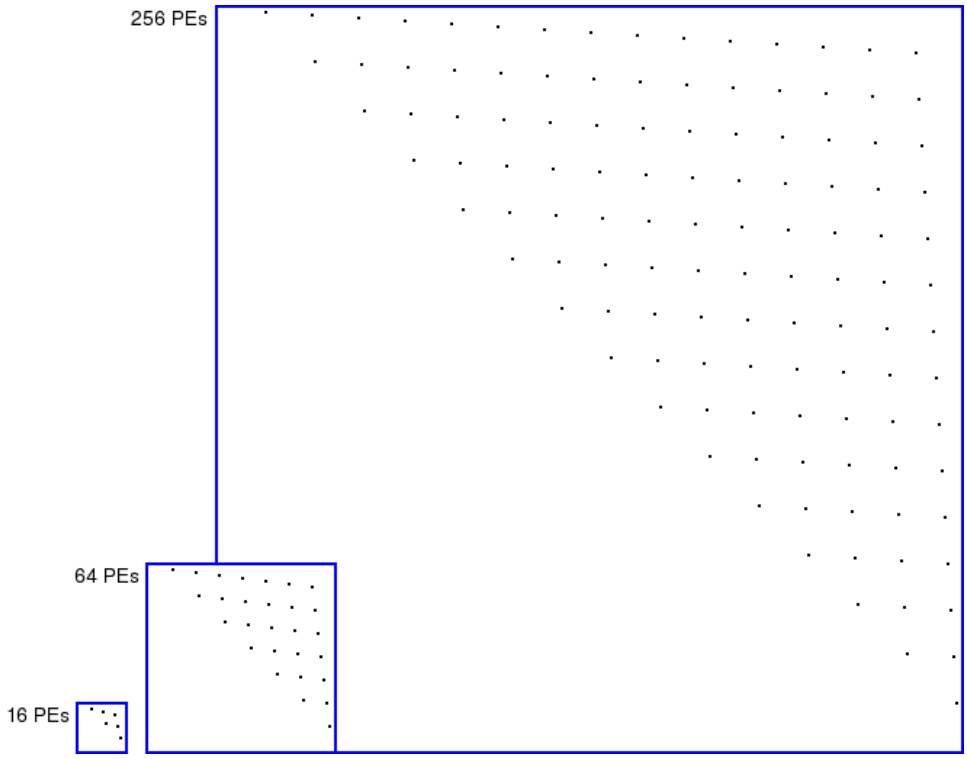

Figure 2.4: 2D Matrix Transpose of a single element per PE

and 256 PEs is shown in Figure 2.7, while Figure 2.8 shows the union of multiple factorizations for the same cases. Similarly, the 4D torus has eight rectilinear neighbors per PE, and several sample connectivity matrices for a single factorization are shown in Figure 2.9, while the union of multiple factorizations are shown in Figure 2.10.

Including diagonally adjacent PEs for a multi-dimensional grid or torus simply increases the constant number of neighbors per PE. For a 2D torus with diagonals, there are eight neighbors per PE and the corresponding connectivity matrices are shown in Figure 2.11 for a single factorization, and in Figure 2.12 for the union of multiple factorizations. Eight pairs may be a large fraction of all possible pairs in a small parallel computer, but it becomes a vanishingly small fraction of all possible pairs as the system design is scaled to thousands of PEs. For a 3D torus, the diagonal neighbors for a PE include both the twelve edge neighbors and eight corner neighbors, bringing the total to 26 neighbors per PE. Figure 2.13 shows the single factorizations for 3D tori which include the diagonal neighbors. Although 26 is still a constant relative to the number of PEs in the machine, it takes a fairly large machine before 26 would be considered a small number of neighbors per PE. When you look at the connectivity matrices for the union of multiple 3D factorizations shown in Figure 2.14, it is clear that including diagonal neighbors along with multiple factorizations yields a rather dense matrix for smaller numbers of PEs.

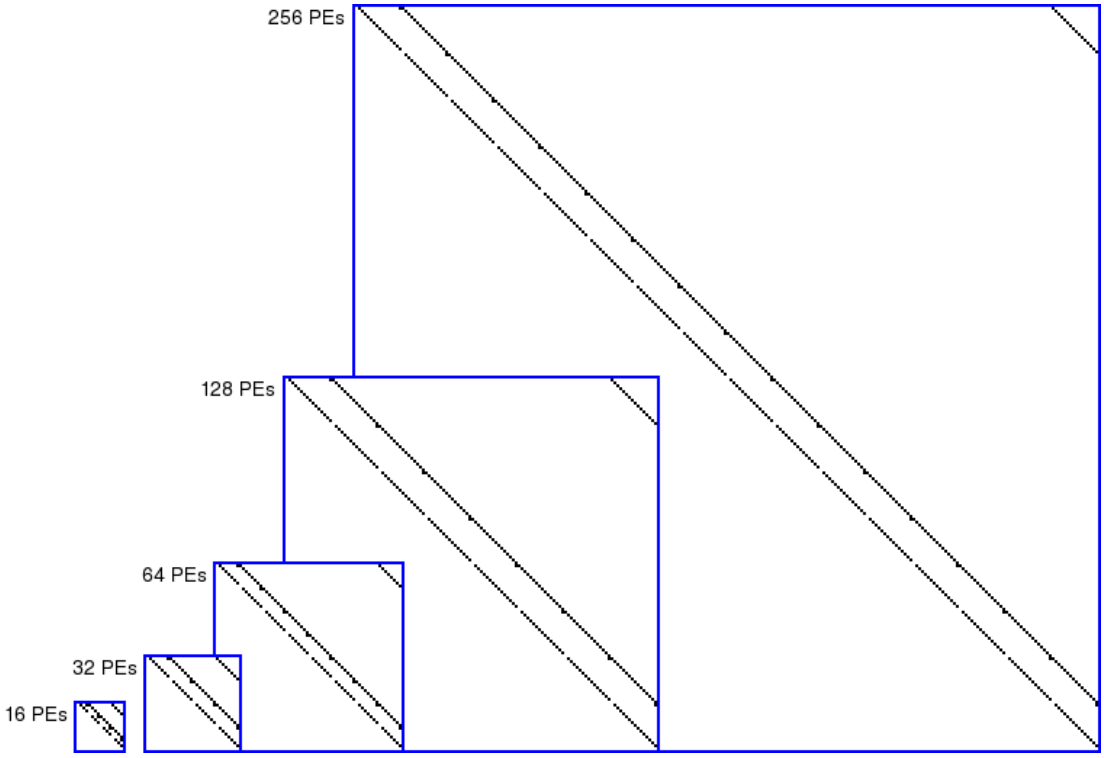

Figure 2.5: Single 2D Torus with ± 1 offsets

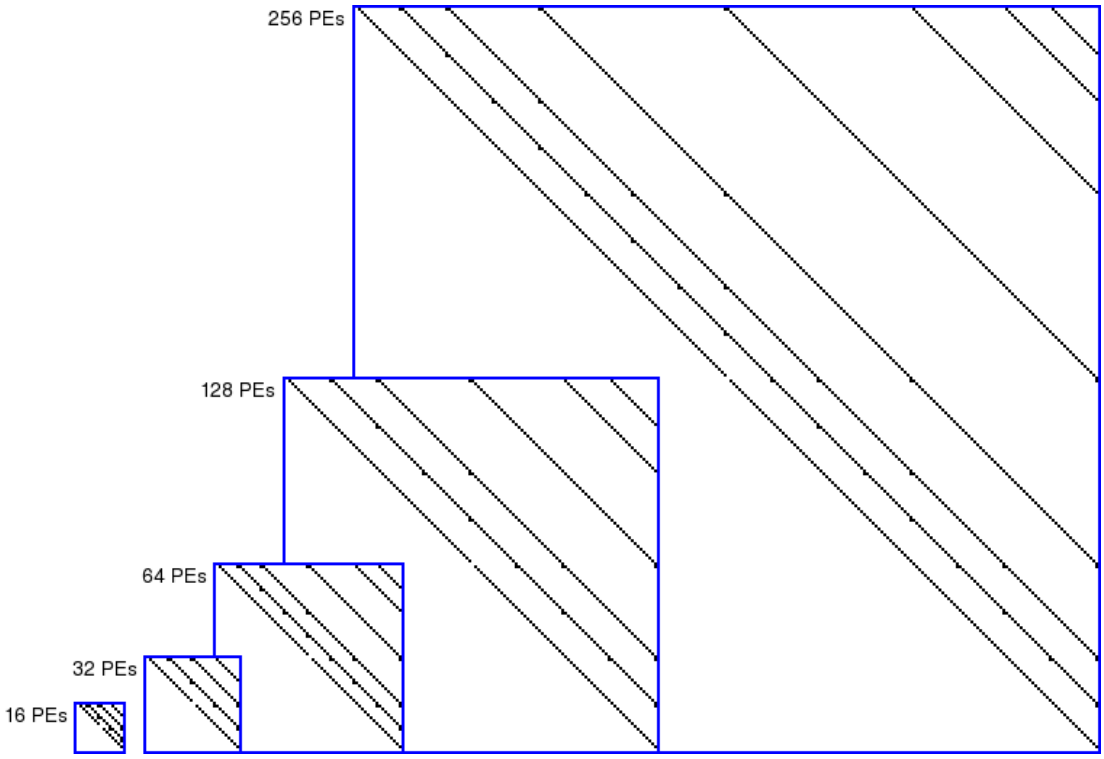

Figure 2.6: Multiple 2D Tori with ± 1 offsets

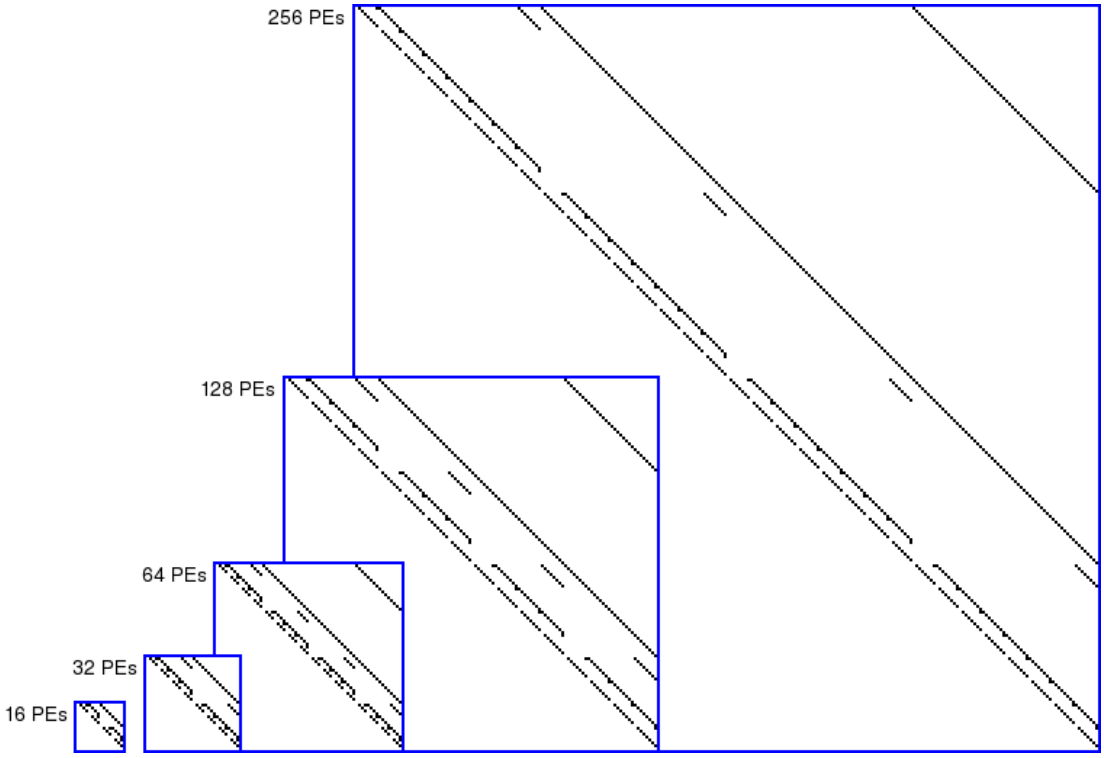

Figure 2.7: Single 3D Torus with ± 1 offsets

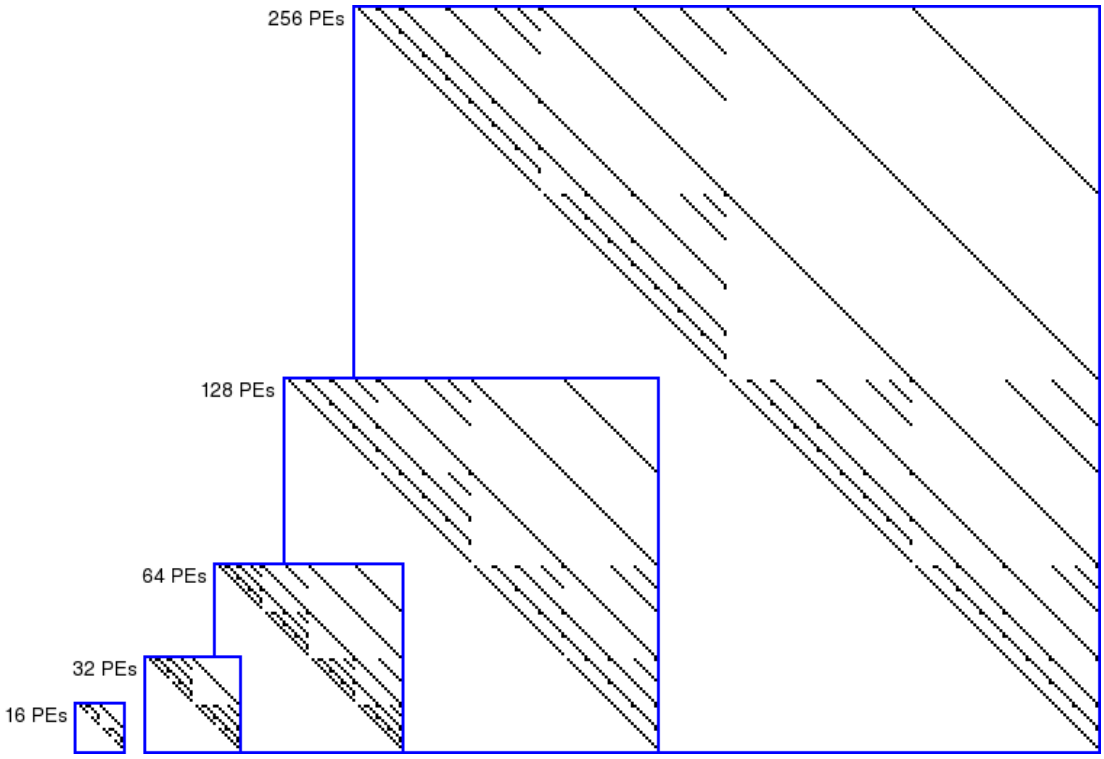

Figure 2.8: Multiple 3D Tori with ± 1 offsets

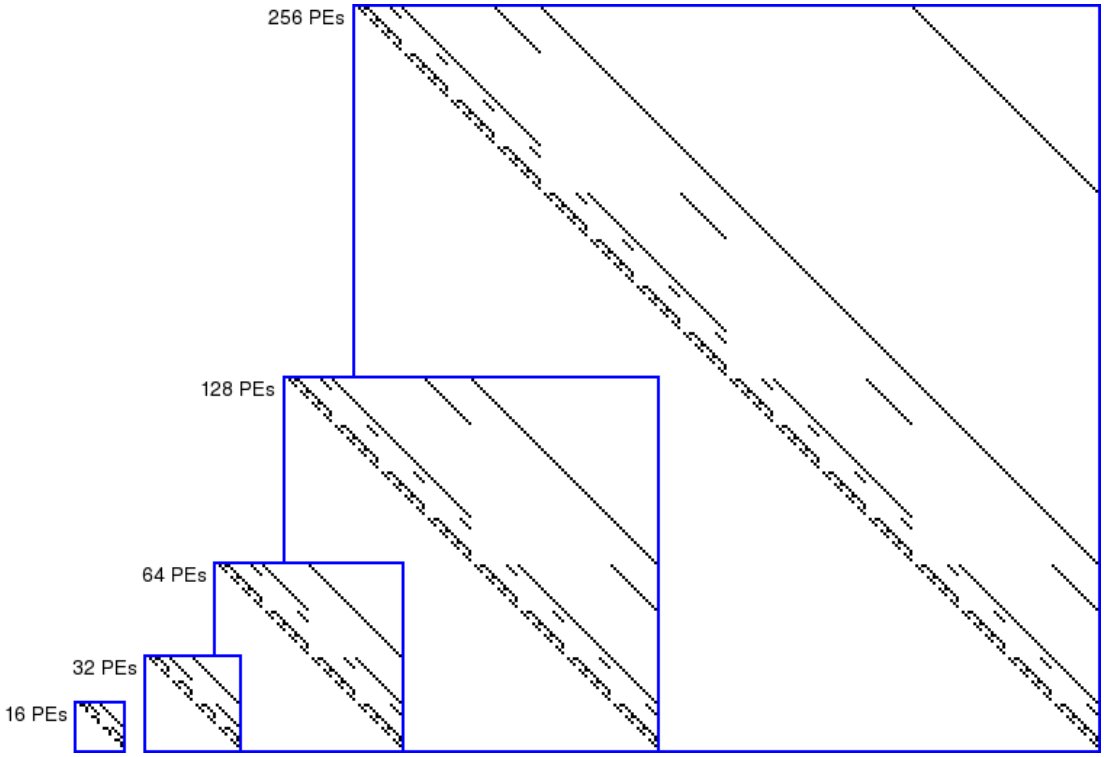

Figure 2.9: Single 4D Torus with ± 1 offsets

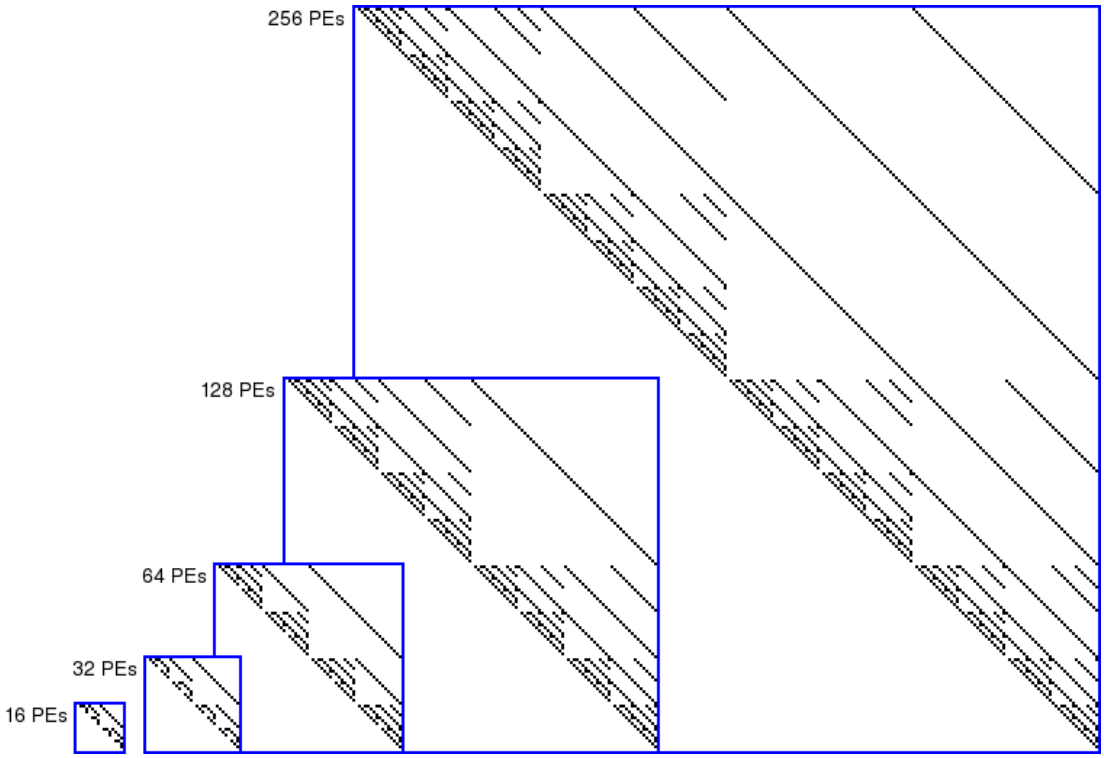

Figure 2.10: Multiple 4D Tori with ± 1 offsets

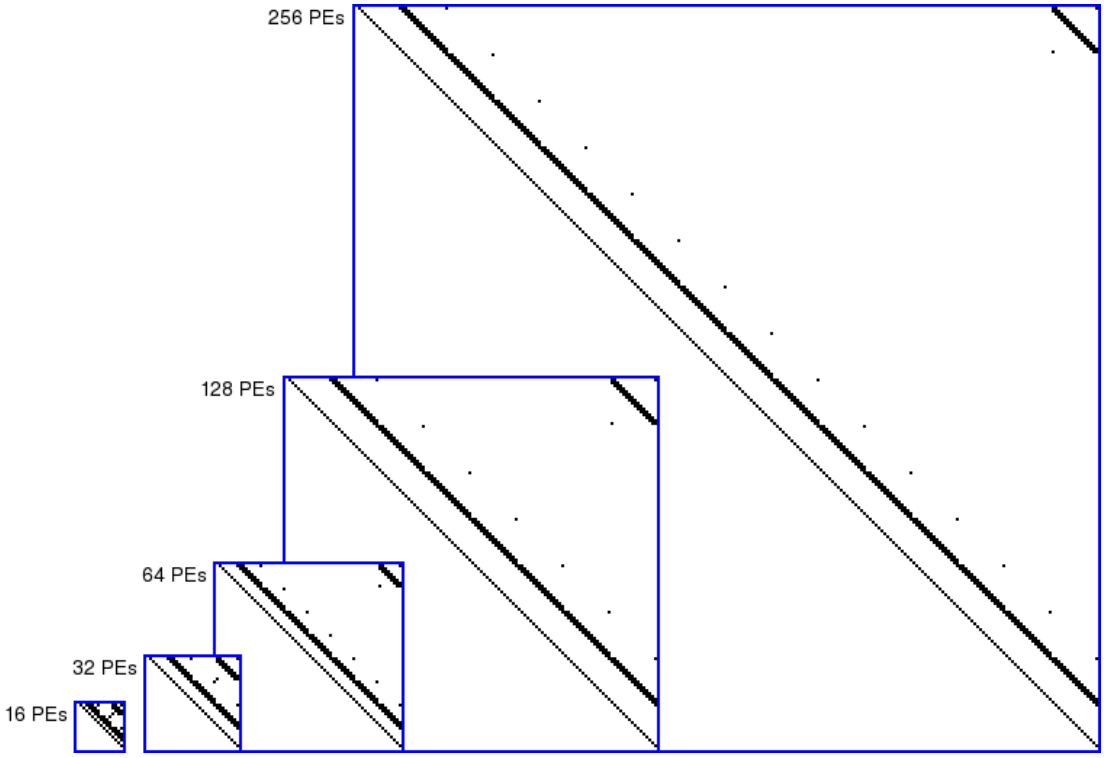

Figure 2.11: Single 2D Torus with ± 1 offsets including diagonals

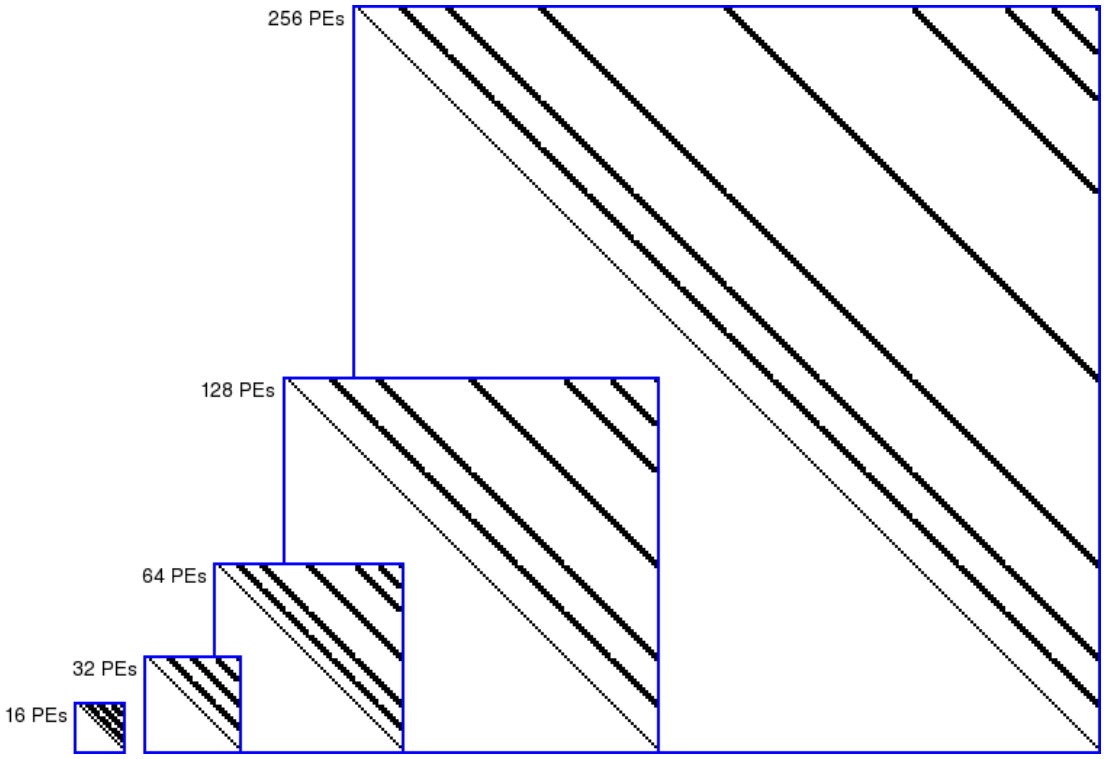

Figure 2.12: Multiple 2D Tori with ± 1 offsets including diagonals

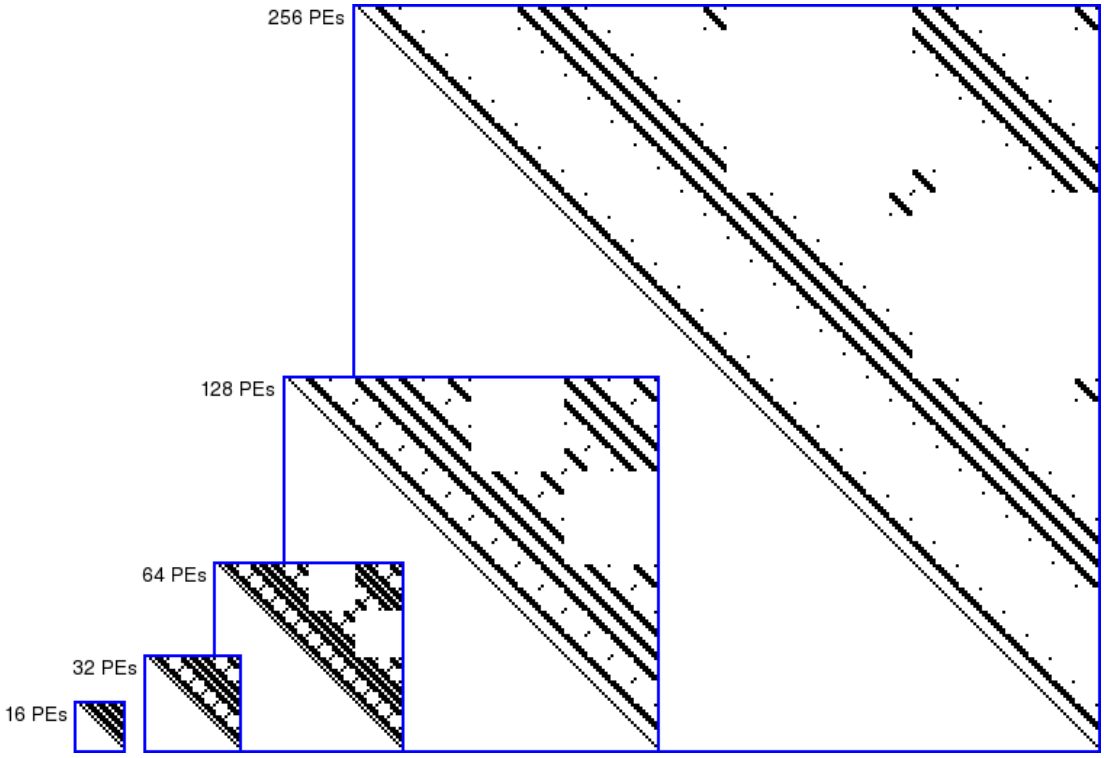

Figure 2.13: Single 3D Torus with ± 1 offsets including diagonals

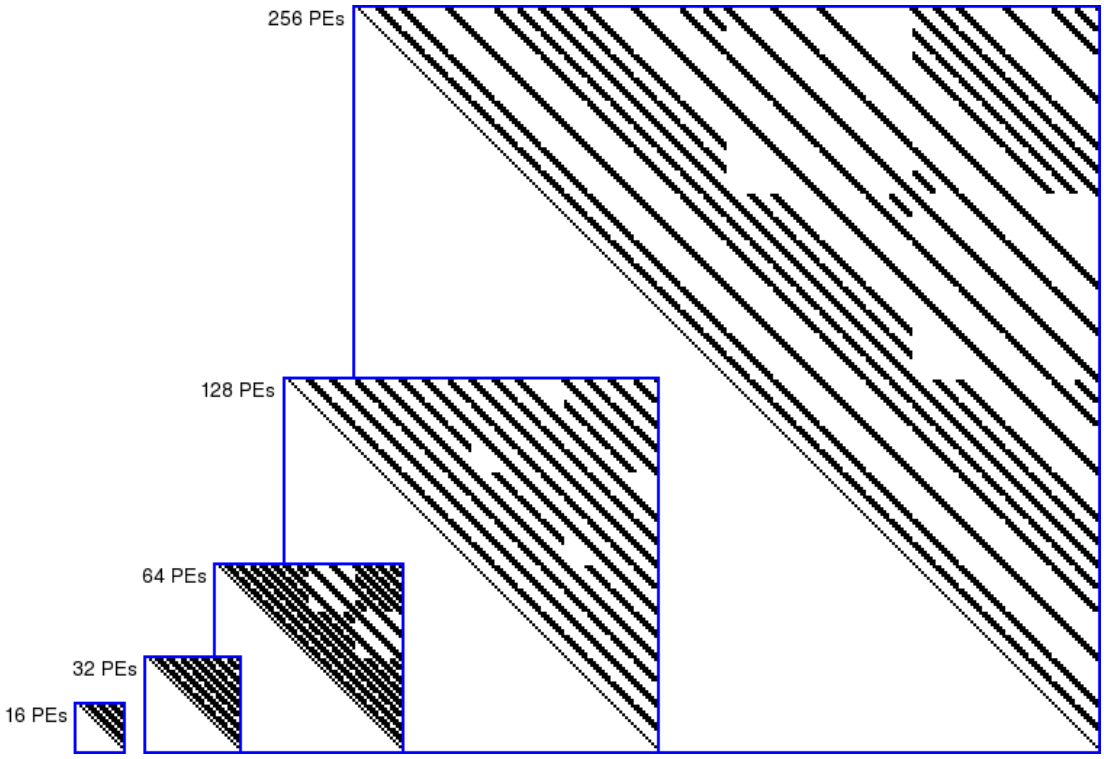

Figure 2.14: Multiple 3D Tori with ± 1 offsets including diagonals

2.3.2 $O(\log N)$ Scaling Patterns

In addition to $O(1)$ scaling patterns, nearly all programs contain some communication patterns that scale as $O(\log N)$. Fundamentally, $O(\log N)$ scaling patterns are most often an artifact of using a network that is incapable of performing computation. For example, *collective communications* including *reductions*, *parallel-prefix scans*, *broadcast*/*multicast*, and *barrier synchronization* are really operations sampling the global state of the parallel system. Aggregate Function Network (AFN) hardware implements them directly within the network[33], but efficient message-passing implementations typically involve a sequence of tree-structured communications.

Binary tree-structured communications follow adjacency in the familiar *hypercube* topology. Thus, each PE communicates with the PEs whose numbers differ from the source PE number by only a single bit position's value in the binary representation. For example, in a 32-PE system, PE 5 (binary 00101) would be paired with 1 (00001), 4 (00100), 7 (00111), 13 (01101), and 21 (10101). There are no more than $\lceil \log_2 N \rceil$ bit positions, so there are at most that many PEs differing from any given PE's number by precisely one bit position, and the number of pairs per PE grows as $O(\log N)$. Shown in Figure 2.15 are the connectivity matrices for the hypercube pattern on 16, 32, 64, 128, and 256 PEs.

Many message-passing libraries differentiate between what MPI[49] calls *all-reduce* and *reduce*. Using *all-reduce* suggests that all PEs should have their complete tree, whereas a *reduce* requires only the tree rooted at a specific point (typically, PE 0). Thus, *reduce* can be implemented using about half as many pairs per PE as suggested by *all-reduce*, although the root PE still requires the full set of pairs. If an *all-reduce* is implemented using a reduce followed by a *broadcast* from the root PE, rather than by directly performing N *reduces* simultaneously, then the complete tree of pairs is only needed for the root PE. This difference is significant in that *reduce* is far more common than *all-reduce* in parallel programs. However, to support *all-reduce* or *reduce* rooted at any PE, we define a pattern that has a neighbor list for each PE consisting of all PEs at an offset of $\pm 2^k$ for $0 \leq k < \log_2 N$, which we call a "1D torus with $\pm 2^k$ offsets". Several connectivity matrices for this pattern are shown in Figure 2.16. This pattern also supports implementations of *barrier synchronization*, such as the dissemination and tournament algorithms by Hensgen, Finkel, and Manbur[32].

MPI has a feature called a "communicator" which allows the programmer to restrict communications to a partition or subset of the machine. In practice, these subsets tend to follow regular patterns, such as the rows or columns of a 2D grid. To support reductions and barriers on these subsets, we extend the "$\pm 2^k$ offsets" patterns to multidimensional grids, where the offset is rectilinearly measured along a dimension of the grid. The connectivity matrix for the "$\pm 2^k$ offsets" pattern on a single 2D torus factorization for each of 16, 32, 64, 128, and 256 PEs is shown in Figure 2.17. Figure 2.18 shows the union of multiple factorizations for the same pattern. Similarly, the same patterns on single factorizations of a 3D torus are shown in Figure2.19, while the patterns on a union of multiple 3D factorizations are shown in Figure 2.20. Sample connectivity matrices for this same pattern on 4D tori are shown in Figure 2.21 and Figure 2.22.

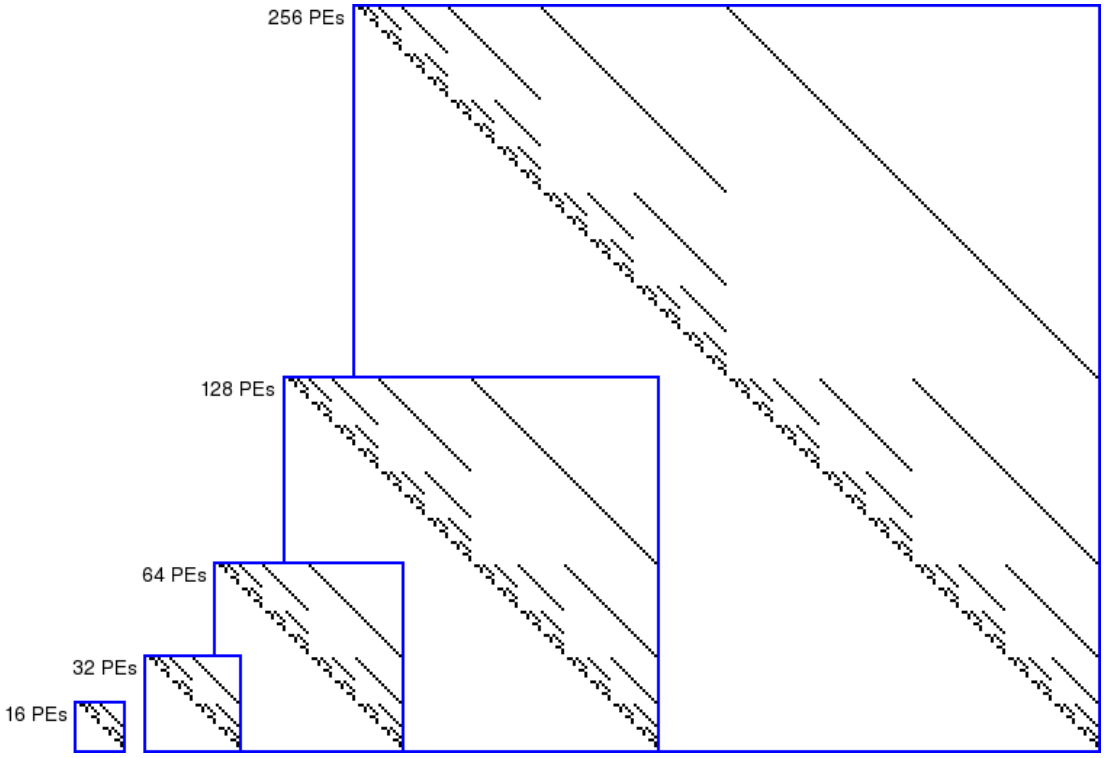

Figure 2.15: Hypercube communication patterns

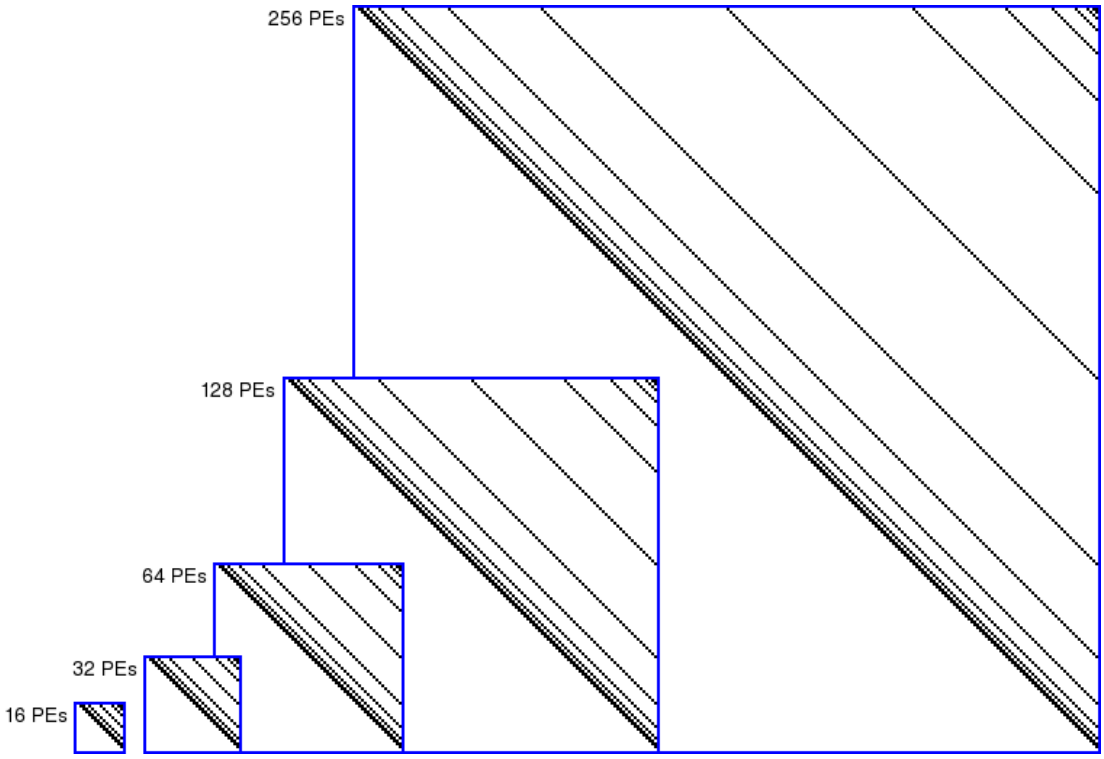

Figure 2.16: 1D Torus with $\pm 2^k$ offsets

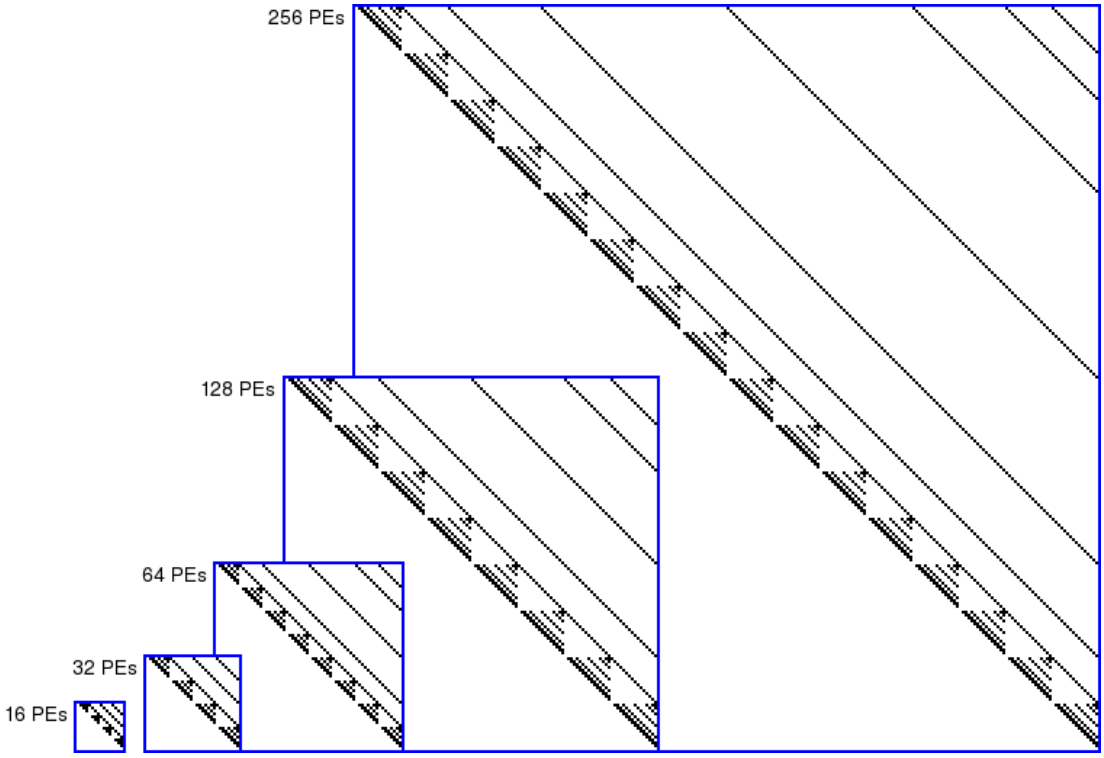

Figure 2.17: Single 2D Torus with $\pm 2^k$ offsets

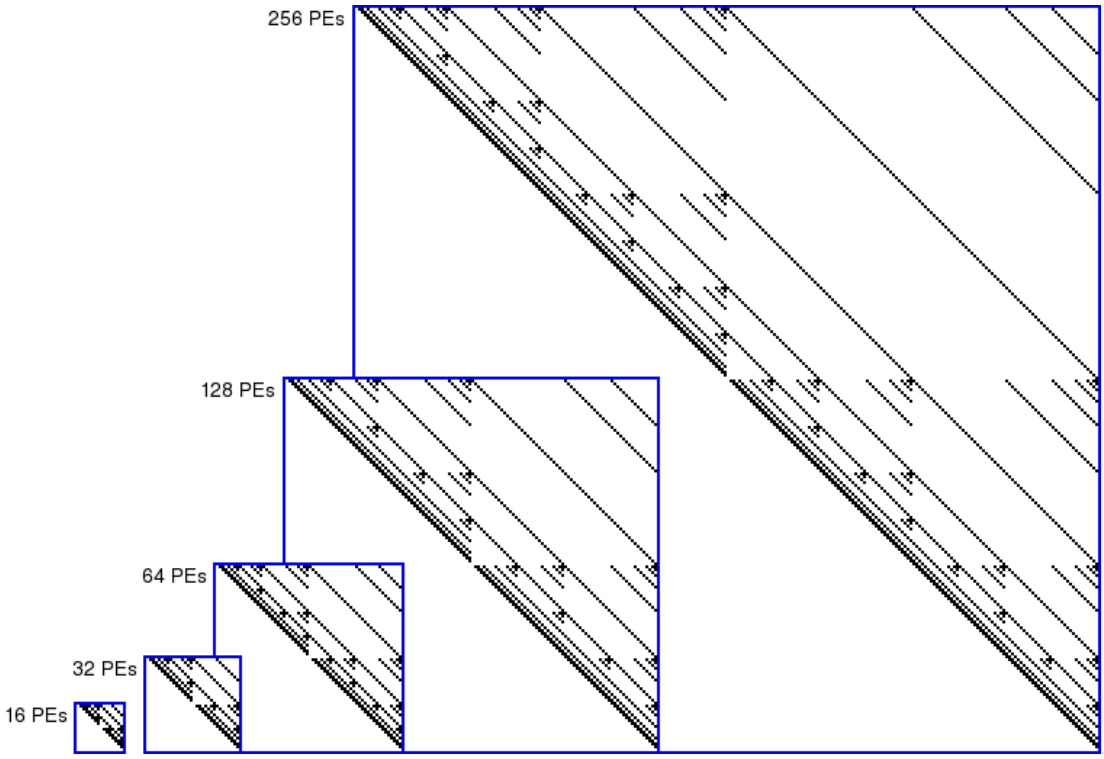

Figure 2.18: Multiple 2D Tori with $\pm 2^k$ offsets

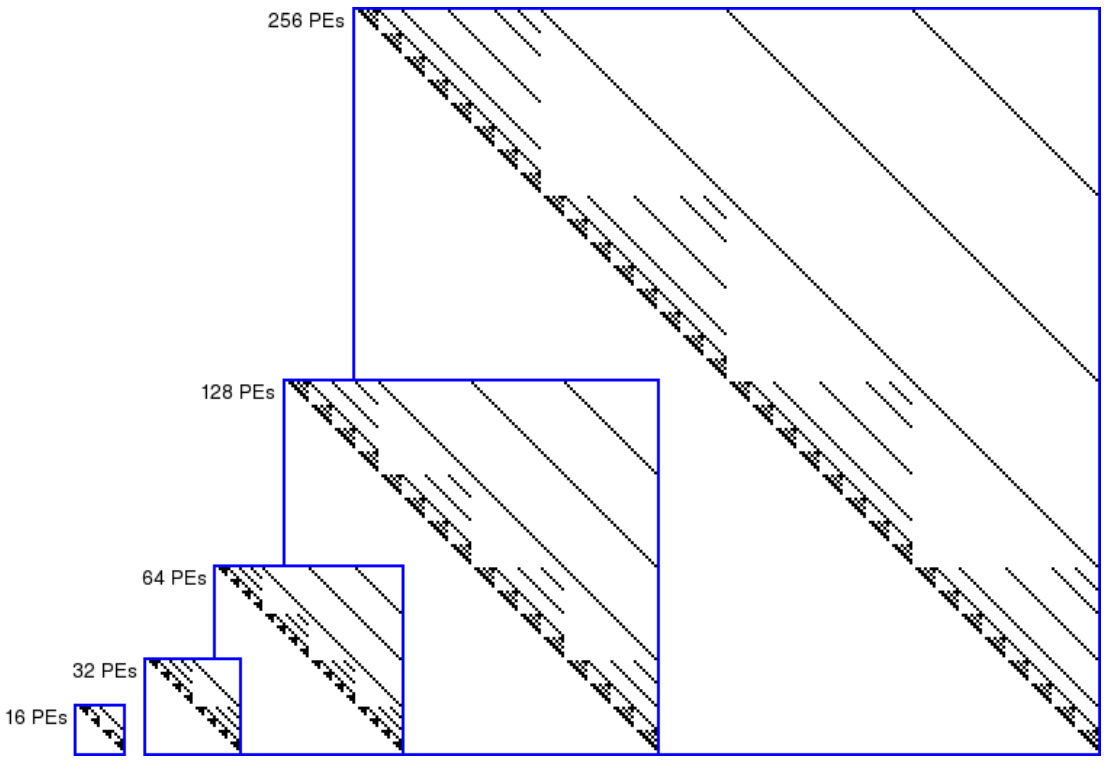

Figure 2.19: Single 3D Torus with $\pm 2^k$ offsets

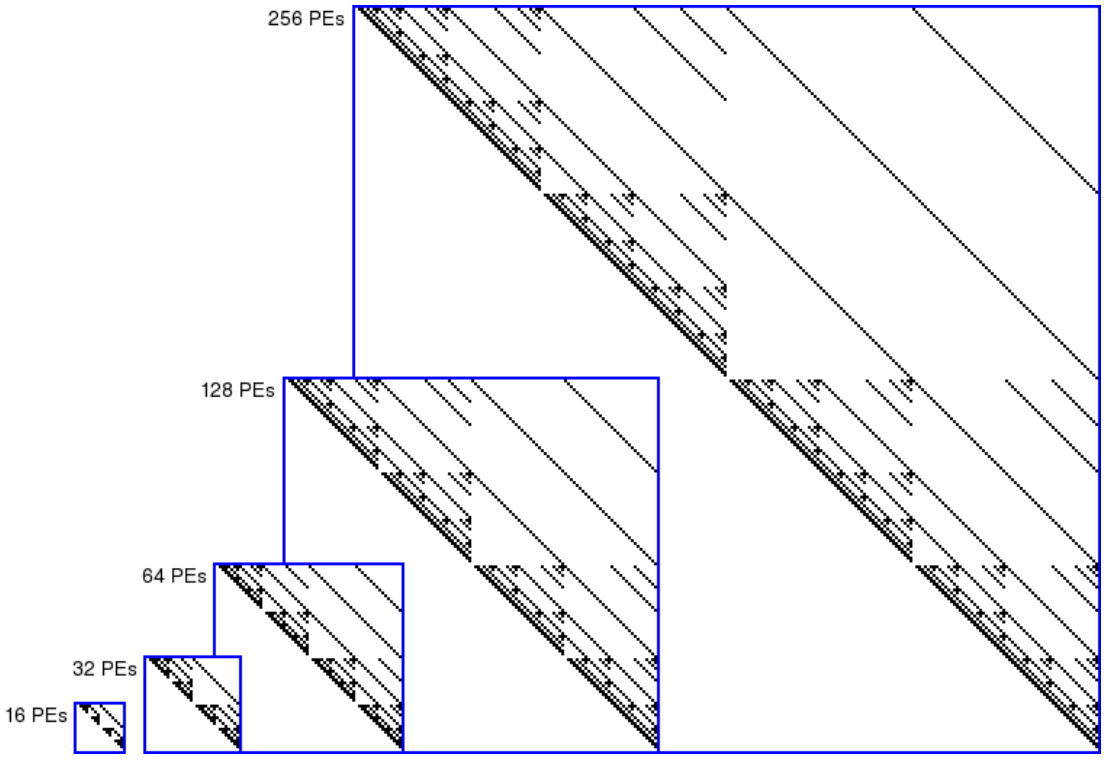

Figure 2.20: Multiple 3D Tori with $\pm 2^k$ offsets

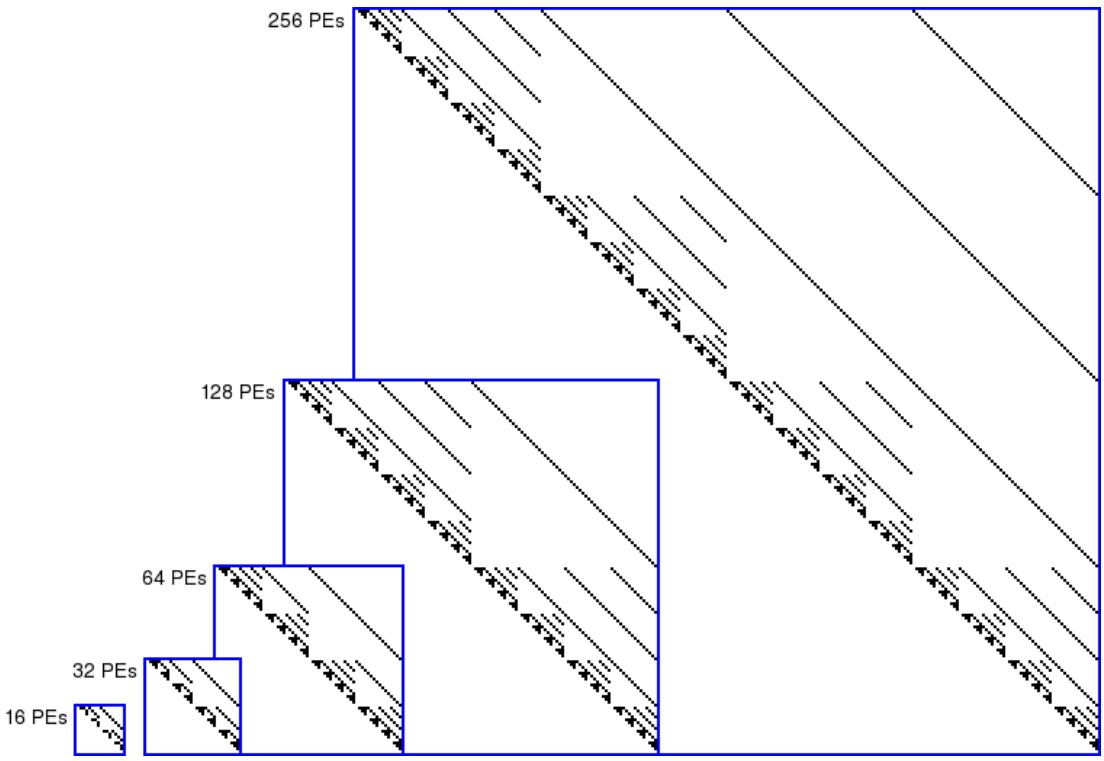

Figure 2.21: Single 4D Torus with $\pm 2^k$ offsets

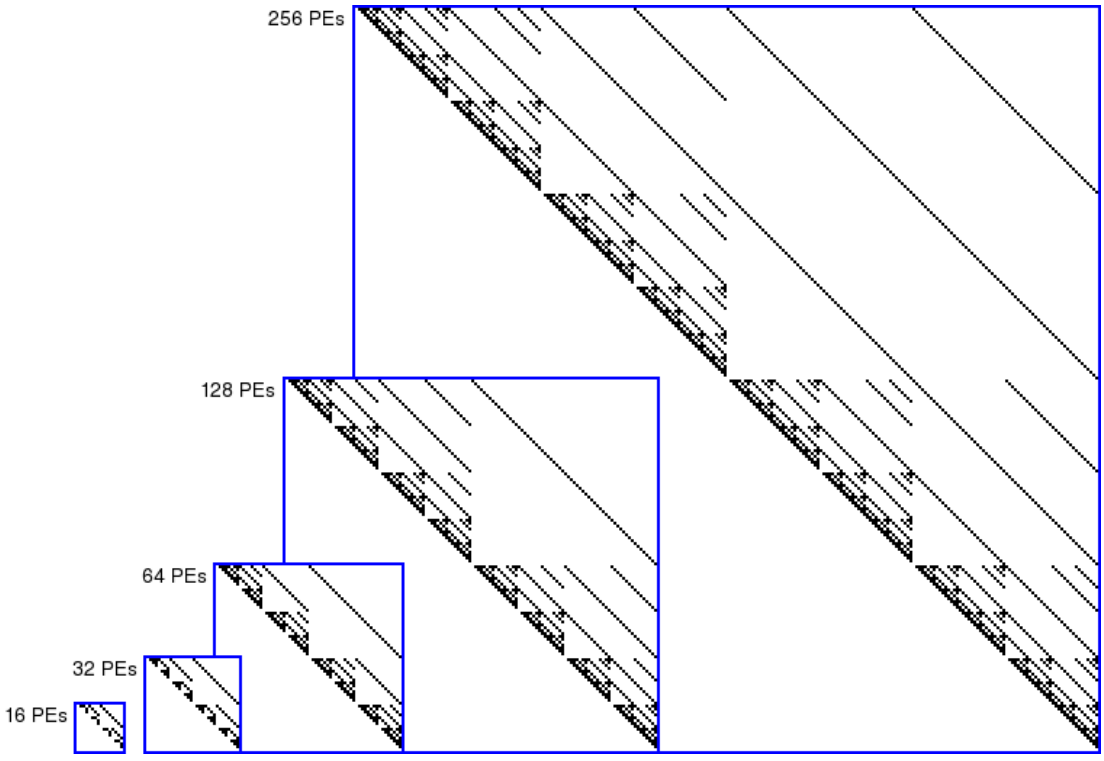

Figure 2.22: Multiple 4D Tori with $\pm 2^k$ offsets

2.3.3 $O(\sqrt[D]{N})$ Scaling Patterns

D-dimensional *scatter*, *gather*, and *personalized all-to-all* communications involve each PE interacting with every other PE in its dimension. In a 2D space, each PE needs to be paired with every PE in the same row or column, yielding $O(\sqrt{N})$ pairs per PE. The solid black triangles along the diagonal of the connectivity matrix in Figure 2.23 show the full row connectivity of this pattern, and the diagonal lines show the full column connectivity. As shown in Figure 2.24, the union of multiple factorizations of 2D grids with full row and column connectivity does not result in a sparse connectivity matrix. The 3D case scales pairs per PE as $O\left(\sqrt[3]{N}\right)$, which is shown in Figure 2.25 and Figure 2.26. Figure 2.27 and Figure 2.28 presents sample connectivity matrices for the 4D case. The 1D case is the worst; all PEs are in the same dimension, resulting in a completely solid connectivity matrix.

Superficially, it seems that all $N-1$ pairs are needed for each PE in order to support 1D *personalized all-to-all*. However, such a communication pattern can not be accomplished in a single time step unless $N-1$ messages can be simultaneously output by each PE. With fewer than $N-1$ NIs per PE (i.e. for $\eta < N-1$), this simultaneity is impossible. Further, the overhead associated with sending a message is significant; thus, unless messages are quite long, there is a significant penalty in sending $N-1$ messages rather than sending fewer, larger, messages that are repackaged and retransmitted until each PE had seen the data destined for it. The result is that *personalized all-to-all* is nearly always best implemented as a compound communication, often following a broadcast-like tree pattern[10]. Thus, it does not make sense to specify a design constraint for an abstract operation like *personalized all-to-all*, but rather to specify the design constraint that corresponds to the most efficient implementation that could be used by the specific MPI library used by applications. The pairs in such a pattern can be determined by examining the MPI library documentation or source code, or by accumulating statistics on pairs communicating in test runs using a particular MPI implementation.

The result is that these compound patterns are usually able to be efficiently implemented using primitive patterns that scale as $O(1)$ or $O(\log N)$. Thus, for implementing many practical communication patterns, the number of communicating pairs per PE scales approximately as $O(\log N)$, not as $O(N)$.

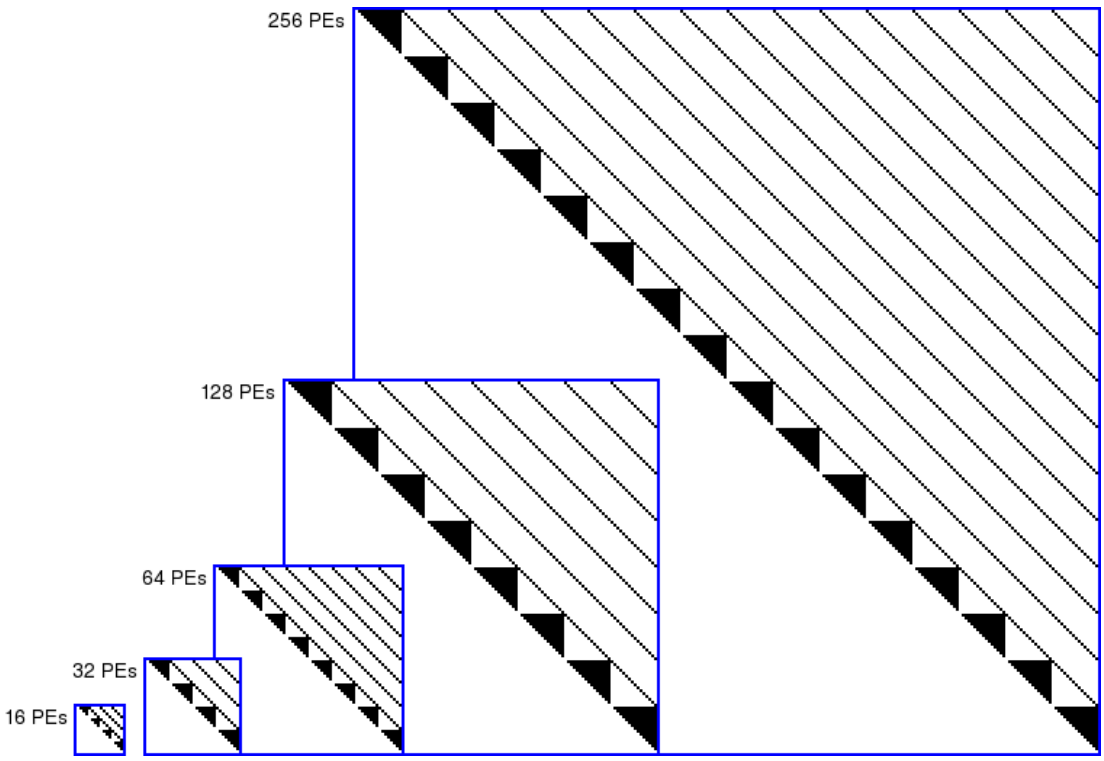

Figure 2.23: Single 2D Torus with connections between all PEs in the same row or column

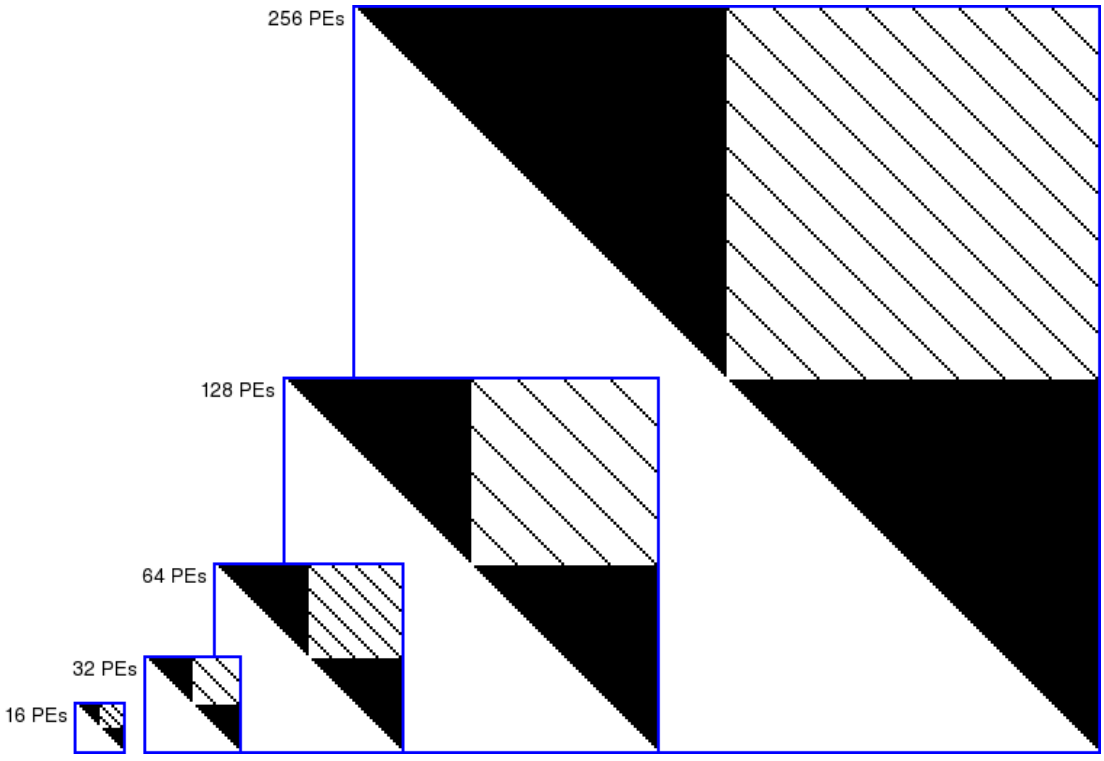

Figure 2.24: Multiple 2D Tori with connections between all PEs in the same row or column

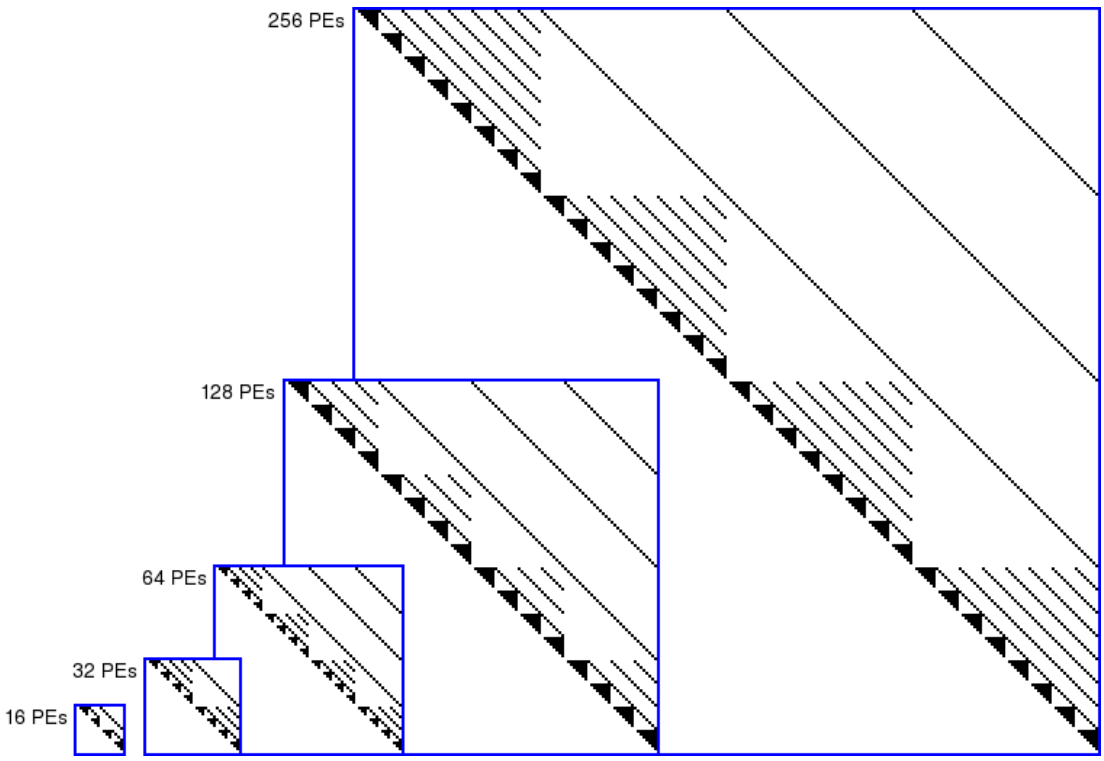

Figure 2.25: Single 3D Grid with connections between all PEs that differ in only one dimension

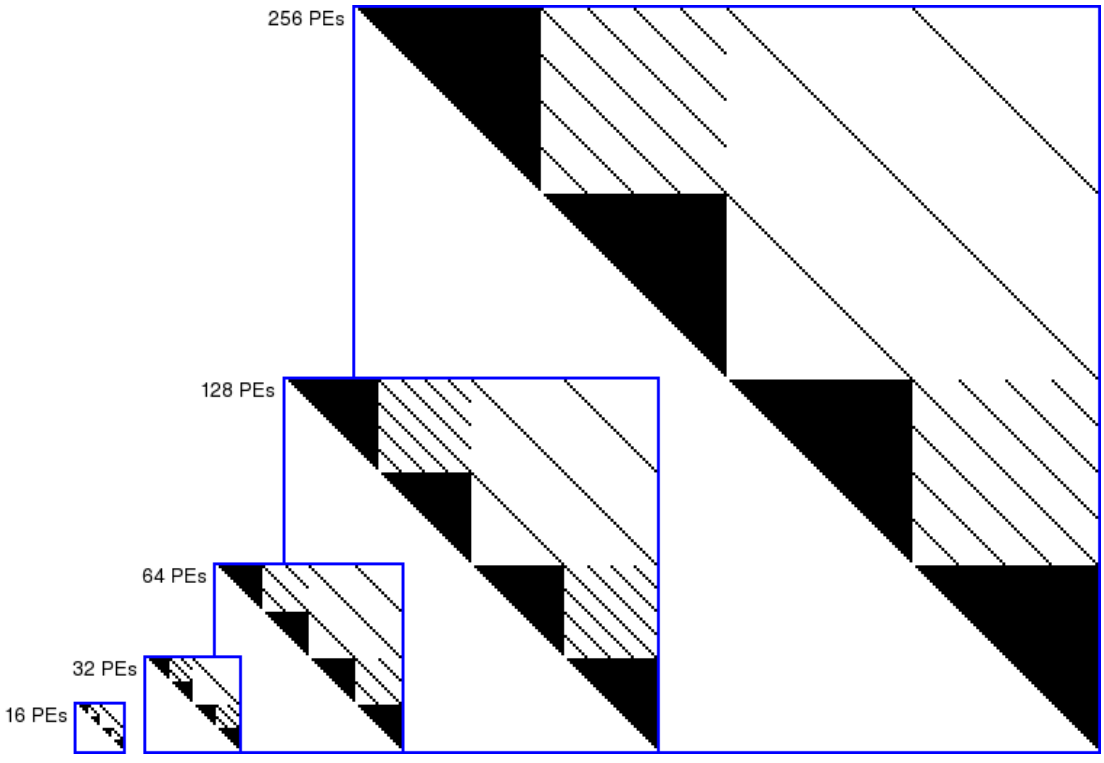

Figure 2.26: Multiple 3D Grids with connections between all PEs that differ in only one dimension

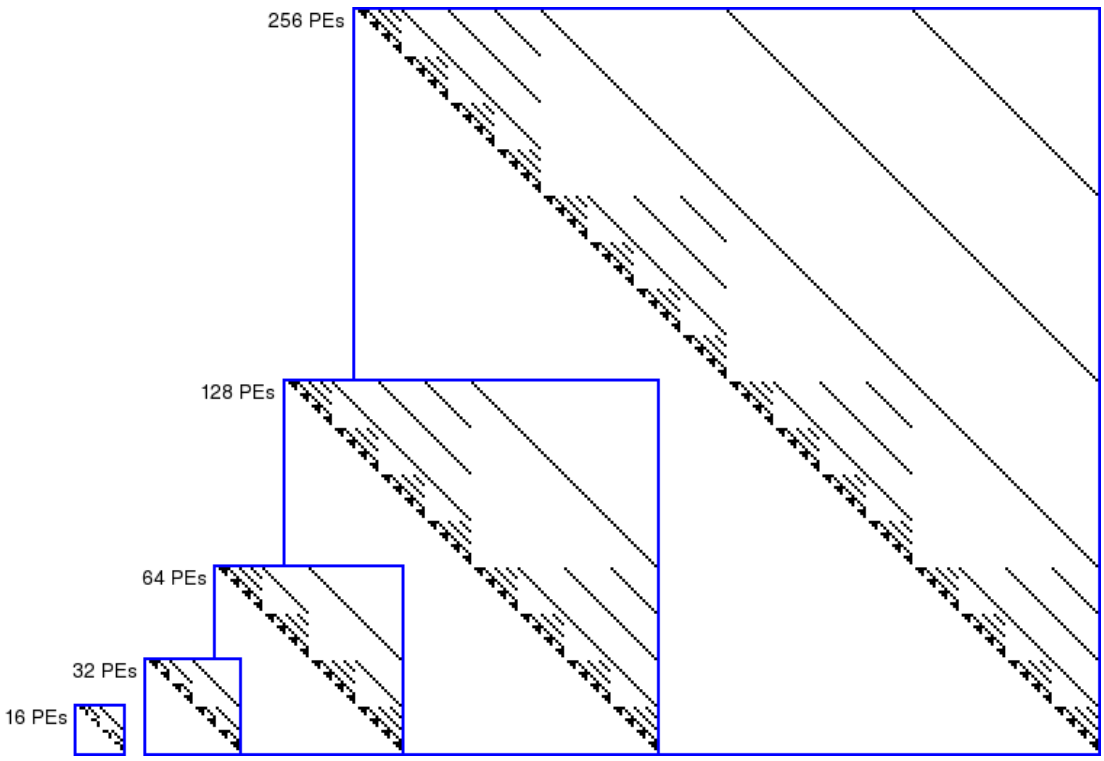

Figure 2.27: Single 4D Grid with connections between all PEs that differ in only one dimension

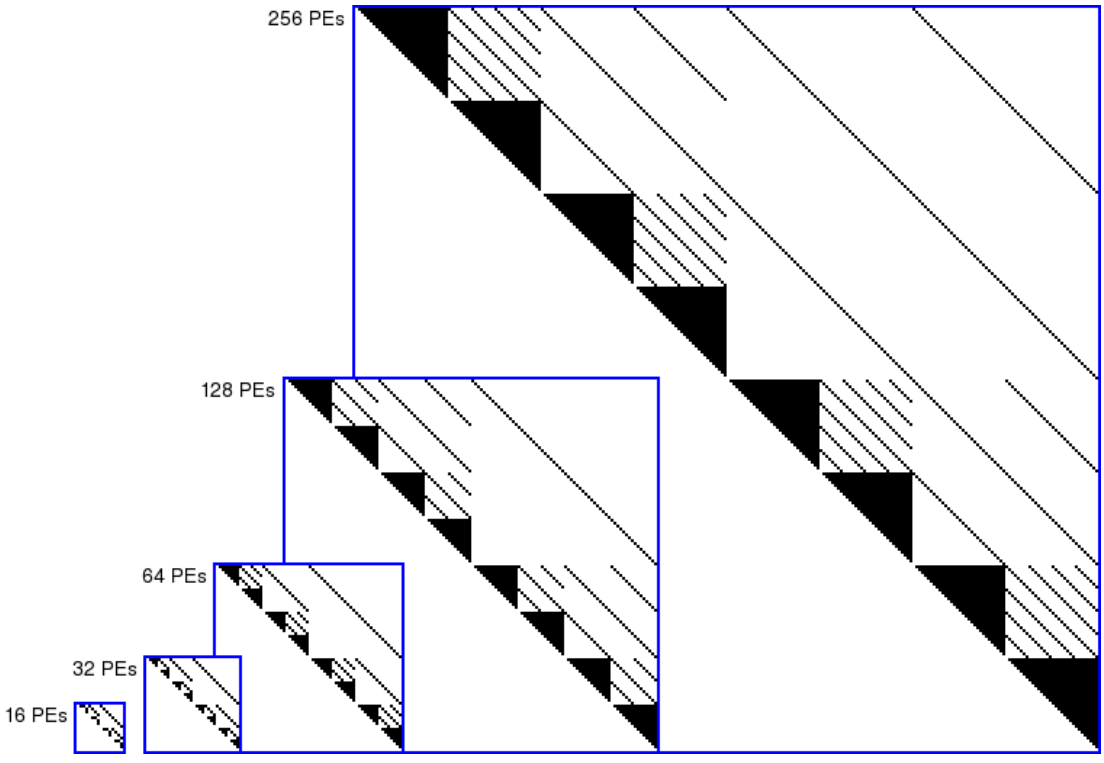

Figure 2.28: Multiple 4D Grids with connections between all PEs that differ in only one dimension

2.3.4 Pair Synergy

A suite of parallel programs generally requires not just support for one communication pattern, but for the union of all communication patterns used in any of the programs. In the worst case, one would expect that the number of pairs for each PE would be the sum of the pairs needed to support each pattern, and that both the number of pairs per pattern and the number of distinct patterns that need to be covered would grow as the number of PEs is increased. Fortunately, a pair required by one pattern very often is also required by another pattern. We call this property *pair synergy*.

For a suite of parallel programs, there are two ways in which the number of communicating pairs can increase as the number of PEs increases:

- The individual communication patterns may have pair counts that grow with the number of PEs, as shown in Sections 2.3.2 and 2.3.3.

- The number of communication patterns may increase with the number of PEs. This second mechanism is less common, in that it is rare that an individual program invents additional communication patterns as the number of PEs is increased. Typically, the number of patterns increases because of data sets that best match grids with particular aspect ratios. Generally, the new patterns created are variants of grids, with the increase in the number of patterns deriving from the fact that the number of ordered factorizations of the PEs generally is larger for systems with more PEs. For example, a 32 PE system can be factored into four 2D grids dimensioned 2×16, 4×8, 8×4, and 16×2; a 256 PE system can be factored into seven 2D grids dimensioned 2×128, 4×64, 8×32, 16×16, 32×8, 64×4, and 128×2. Although we have shown all ordered factorizations in these two examples, it is commonly sufficient to limit the number of factorizations to the number of unordered factorizations by simply picking an arbitrary order for listing the dimensions; for example, 2×16 and 16×2 might be normalized to 16×2 (i.e., the largest dimension first). The increase in unordered factorizations depends on the specific numbers of PEs, but approximately $O(\log N)$ additional patterns is typical, in which case the complexity of the communication pattern set is effectively multiplied by $\log N$. A similar source of additional patterns can come from runtime partitioning of the system to reflect variations in machine load and problem mix.

Without pair synergy, these two mechanisms would yield significant complexity increases – especially where the number of PEs happens to have many factorizations. The question thus becomes how many potential pairs are removed from the complexity formula because they are covered by pair synergy?

For simple permutation communication patterns, it is possible to estimate the amount of pair synergy that should occur due to random chance. A simple unidirectional permutation communication pattern is one where each PE in the machine sends a single message to one other PE, or possibly to itself, and each PE receives at most one message. These permutation communication patterns can be represented as a set of source and destination ordered pairs. A random unidirectional permutation

communication on N PEs has no internal pair synergy, however, it has an expected value of one PE that will be talking to itself without needing the network to support that pair. Given two randomly selected unidirectional permutation communications on N PEs, the expected number of ordered PE pairs that would be common to both patterns is one – for each source PE in the first permutation, intuitively there is a one out of N chance of selecting the same destination in the second permutation, thus the expected value for PEs selecting the same destination is $N \times \frac{1}{N} = 1$. However, the reduction in pair count is slightly less, because there is a one out of N chance that the common case was a PE talking to itself, which inherently does not require a pair to be covered by the network hardware.

Given bidirectional network hardware, a permutation communication pattern has twice as many ordered pairs, one for each forward path as in the unidirectional case, and one for each reverse path. However, for each PE that has itself as a destination, there is only a single ordered pair in the set. Thus, the expected value for the number of synergistic pairs between two randomly selected bidirectional permutation patterns is slightly less than two. Monte Carlo simulations of the bidirectional case for $N > 64$ converge on the value two.

Randomly occurring pair synergies among more than two randomly selected permutations benefit from the same binning effects demonstrated in the well-known "Birthday Paradox" [48]. Although the probability of at least one pair being a duplicate is quite high (corresponding roughly to probability of at least one shared birthday), the expected number of shared pairs remains quite low for reasonable numbers of permutations.

Fortunately, many sets of useful communication patterns exhibit much higher pair synergy than one would expect given the above discussion of random independent permutations. The number of shared pairs is particularly high among various mesh patterns. For example, nearly all pairs required to support 1D adjacency also are required to support 2D adjacency; only 1D pairs involving PEs in edge positions in the 2D pattern are not in the 2D pattern. Similarly, *hypercube* adjacency has many pairs that overlap those of meshes. The result is a significant reduction in the total number of pairs required for the union of multiple communication patterns, although the precise amount of reduction is highly dependent on the set of patterns specified.

Table 2.2 presents a numerical representation of the pair synergy between various $O(1)$ communication patterns and the $O(\log N)$ hypercube pattern. Specifically, each cell represents the percentage of PE pairs for the row's pattern that are covered by the column's pattern. For example, the cell in the upper right corner indicates that for a 256 PE machine, the hypercube pattern covers 50% of the PE pairs in the bidirectional Ring (1D torus with ± 1 offsets). In the lower left corner, the table shows that the bidirectional Ring covers only 12.5% of the pairs in the hypercube pattern. Table 2.3 presents the same data calculated for the 1,024 PE case, while Table 2.4 shows the data for the 4,096 PE case.

Table 2.5 shows the same synergy data for the 256 PE case, with the tori with ± 1 offsets patterns replaced by their $\pm 2^k$ offset versions. Similarly, Table 2.6 is for the 1,024 PE case, and Table 2.7 is for the 4,096 PE case. It is worth noting that any of the tori with $\pm 2^k$ offsets always cover the entire

hypercube pattern.

There also is significant pair synergy between different factorizations of the same grid/tori patterns, which is of special importance in that it mitigates the growth in PE pairs due to the growth in the number of possible grid/tori factorizations. For the 2D tori patterns in the 256 PE example shown in Figure 2.6, there are four normalized unordered factorizations, 128×2, 64×4, 32×8, and 16×16, each with 512 PE pairs. If there was no pair synergy, the combined pattern would have 2,048 PE pairs. Fortunately, pair synergy has reduced the total to only 1,180 PE pairs. For the similar 1024 PE case, with five 2D factorizations, the combined pattern without pair synergy would have 10,240 pairs, yet the real value is only 5,692 pairs. Finally, for the 4096 PE case, with six 2D factorizations, the number of PE pairs would have been 49,152 without pair synergy, yet the total is only 26,748. This savings is significant, but higher dimensionality factorizations yield even greater savings by pair synergy.

For the 3D tori patterns in the 256 PE example shown in Figure 2.8, there are five normalized unordered factorizations, $64 \times 2 \times 2$, $32 \times 4 \times 2$, $16 \times 8 \times 2$, $16 \times 4 \times 4$, and $8 \times 8 \times 4$, each with 768 PE pairs. If there was no pair synergy, the combined pattern would have 3,840 PE pairs. Fortunately, pair synergy has reduced the total to only 1,528 PE pairs, which is less than 40% of the simplistic prediction. For the similar 1024 PE case, with eight 3D factorizations, the combined pattern without pair synergy would have 24,576 pairs, yet the real value is only 7,544 pairs, which is under 31% of the prediction. Finally, for the 4096 PE case, with twelve 3D factorizations, the number of PE pairs would have been 147,456 without pair synergy, yet the total is only 39,416, less than 27% of the prediction.

For the 4D tori patterns in the 256 PE example shown in Figure 2.10, there are five normalized unordered factorizations, $32 \times 2 \times 2 \times 2$, $16 \times 4 \times 2 \times 2$, $8 \times 8 \times 2 \times 2$, $8 \times 4 \times 4 \times 2$, and $4 \times 4 \times 4 \times 4$, each with 1024 PE pairs. If there was no pair synergy, the combined pattern would have 5,120 PE pairs. The actual total is only 1,840 PE pairs, which is less than 36% of the no-synergy estimate. For the similar 1024 PE case, with nine 4D factorizations, the combined pattern without pair synergy would have 36,864 pairs, yet the real value is only 8,816 pairs, which is under 24% of the prediction. Finally, for the 4096 PE case, with fifteen 4D factorizations, the number of PE pairs would have been 245,760 without pair synergy, yet the total is only 45,808, less than 19% of the prediction.

Throughout Tables 2.2-2.7, only Transpose is regularly comparable to a randomly-selected permutation's level of pair synergy with the other patterns. All the other patterns fare markedly better, with meshes and hypercubes consistently having very high levels of pair synergy. Note that the high degree of overlap, for example between 2D and 3D patterns, does not mean that a computer whose network only implements one of those patterns would yield high performance on the other pattern – even delivering lower performance for a single PE pair typically will drop performance of a parallel algorithm down to that level (and sometimes lower due to interference between fast and slow paths). It does mean that relatively little additional network hardware might be needed to support multiple patterns instead of just one.

Pattern	1D ±1 offsets	2D ±1 offsets	3D ±1 offsets	4D ±1 offsets	Perfect Shuffle	Bit-Reversal	Transpose	Hypercube
1D ±1 offsets	100.0%	93.8%	87.5%	75.0%	0.8%	0.0%	0.0%	50.0%
2D ±1 offsets	46.9%	100.0%	43.8%	75.0%	0.8%	0.0%	0.0%	50.0%
3D ±1 offsets	29.2%	29.2%	100.0%	58.3%	0.8%	1.0%	0.0%	50.0%
4D ±1 offsets	18.8%	37.5%	43.8%	100.0%	0.8%	0.0%	0.0%	50.0%
Perfect Shuffle	0.8%	1.6%	2.4%	3.2%	100.0%	11.5%	0.0%	0.0%
Bit-Reversal	0.0%	0.0%	6.7%	0.0%	24.2%	100.0%	5.0%	0.0%
Transpose	0.0%	0.0%	0.0%	0.0%	0.0%	5.0%	100.0%	0.0%
Hypercube	12.5%	25.0%	37.5%	50.0%	0.0%	0.0%	0.0%	100.0%

Table 2.2: Pair Synergy for 256 PE Tori with ±1 offsets and other patterns

Pattern	1D ±1 offsets	2D ±1 offsets	3D ±1 offsets	4D ±1 offsets	Perfect Shuffle	Bit-Reversal	Transpose	Hypercube
1D ±1 offsets	100.0%	96.9%	93.8%	87.5%	0.2%	0.0%	0.0%	50.0%
2D ±1 offsets	48.4%	100.0%	46.9%	43.8%	0.2%	0.0%	0.0%	50.0%
3D ±1 offsets	31.3%	31.3%	100.0%	29.2%	0.2%	0.5%	0.0%	50.0%
4D ±1 offsets	21.9%	21.9%	21.9%	100.0%	0.2%	0.0%	0.0%	50.0%
Perfect Shuffle	0.2%	0.4%	0.6%	0.8%	100.0%	6.0%	0.1%	0.0%
Bit-Reversal	0.0%	0.0%	3.2%	0.0%	12.3%	100.0%	5.6%	0.0%
Transpose	0.0%	0.0%	0.0%	0.0%	0.2%	5.6%	100.0%	0.0%
Hypercube	10.0%	20.0%	30.0%	40.0%	0.0%	0.0%	0.0%	100.0%

Table 2.3: Pair Synergy for 1024 PE Tori with ±1 offsets and other patterns

Pattern	1D ±1 offsets	2D ±1 offsets	3D ±1 offsets	4D ±1 offsets	Perfect Shuffle	Bit-Reversal	Transpose	Hypercube
1D ±1 offsets	100.0%	98.4%	93.8%	87.5%	0.0%	0.0%	0.0%	50.0%
2D ±1 offsets	49.2%	100.0%	46.9%	87.5%	0.0%	0.0%	0.0%	50.0%
3D ±1 offsets	31.3%	31.3%	100.0%	29.2%	0.0%	0.0%	0.0%	50.0%
4D ±1 offsets	21.9%	43.8%	21.9%	100.0%	0.0%	0.0%	0.0%	50.0%
Perfect Shuffle	0.0%	0.1%	0.1%	0.2%	100.0%	3.1%	0.0%	0.0%
Bit-Reversal	0.0%	0.0%	0.0%	0.0%	6.2%	100.0%	1.4%	0.0%
Transpose	0.0%	0.0%	0.0%	0.0%	0.0%	1.4%	100.0%	0.0%
Hypercube	8.3%	16.7%	25.0%	33.3%	0.0%	0.0%	0.0%	100.0%

Table 2.4: Pair Synergy for 4096 PE Tori with ±1 offsets and other patterns

Pattern	1D $\pm 2^k$ offsets	2D $\pm 2^k$ offsets	3D $\pm 2^k$ offsets	4D $\pm 2^k$ offsets	Perfect Shuffle	Bit-Reversal	Transpose	Hypercube
1D $\pm 2^k$ offsets	100.0%	87.5%	76.7%	70.0%	0.7%	0.4%	0.0%	53.3%
2D $\pm 2^k$ offsets	93.8%	100.0%	76.8%	78.6%	0.7%	0.0%	0.0%	57.1%
3D $\pm 2^k$ offsets	88.5%	82.7%	100.0%	80.8%	0.6%	0.5%	0.0%	61.5%
4D $\pm 2^k$ offsets	87.5%	91.7%	87.5%	100.0%	0.5%	0.0%	0.0%	66.7%
Perfect Shuffle	5.5%	4.7%	4.0%	3.2%	100.0%	11.5%	0.0%	0.0%
Bit-Reversal	6.7%	0.0%	6.7%	0.0%	24.2%	100.0%	5.0%	0.0%
Transpose	0.0%	0.0%	0.0%	0.0%	0.0%	5.0%	100.0%	0.0%
Hypercube	100.0%	100.0%	100.0%	100.0%	0.0%	0.0%	0.0%	100.0%

Table 2.5: Pair Synergy for 256 PE Tori with $\pm 2^k$ offsets and other patterns

Pattern	1D $\pm 2^k$ offsets	2D $\pm 2^k$ offsets	3D $\pm 2^k$ offsets	4D $\pm 2^k$ offsets	Perfect Shuffle	Bit-Reversal	Transpose	Hypercube
1D $\pm 2^k$ offsets	100.0%	89.8%	80.9%	73.7%	0.2%	0.2%	0.0%	52.6%
2D $\pm 2^k$ offsets	94.8%	100.0%	81.3%	73.6%	0.2%	0.0%	0.0%	55.6%
3D $\pm 2^k$ offsets	90.4%	86.0%	100.0%	75.0%	0.2%	0.2%	0.0%	58.8%
4D $\pm 2^k$ offsets	87.5%	82.8%	79.7%	100.0%	0.1%	0.2%	0.0%	62.5%
Perfect Shuffle	1.8%	1.6%	1.4%	1.2%	100.0%	6.0%	0.1%	0.0%
Bit-Reversal	3.2%	0.0%	3.2%	3.2%	12.3%	100.0%	5.6%	0.0%
Transpose	0.0%	0.0%	0.0%	0.0%	0.2%	5.6%	100.0%	0.0%
Hypercube	100.0%	100.0%	100.0%	100.0%	0.0%	0.0%	0.0%	100.0%

Table 2.6: Pair Synergy for 1024 PE Tori with $\pm 2^k$ offsets and other patterns

Pattern	1D $\pm 2^k$ offsets	2D $\pm 2^k$ offsets	3D $\pm 2^k$ offsets	4D $\pm 2^k$ offsets	Perfect Shuffle	Bit-Reversal	Transpose	Hypercube
1D $\pm 2^k$ offsets	100.0%	91.4%	83.7%	77.2%	0.0%	0.1%	0.0%	52.2%
2D $\pm 2^k$ offsets	95.6%	100.0%	82.4%	84.1%	0.0%	0.0%	0.0%	54.5%
3D $\pm 2^k$ offsets	91.7%	86.3%	100.0%	77.4%	0.0%	0.1%	0.0%	57.1%
4D $\pm 2^k$ offsets	88.8%	92.5%	81.3%	100.0%	0.0%	0.0%	0.0%	60.0%
Perfect Shuffle	0.5%	0.5%	0.4%	0.4%	100.0%	3.1%	0.0%	0.0%
Bit-Reversal	1.6%	0.0%	1.6%	0.0%	6.2%	100.0%	1.4%	0.0%
Transpose	0.0%	0.0%	0.0%	0.0%	0.0%	1.4%	100.0%	0.0%
Hypercube	100.0%	100.0%	100.0%	100.0%	0.0%	0.0%	0.0%	100.0%

Table 2.7: Pair Synergy for 4096 PE Tori with $\pm 2^k$ offsets and other patterns

2.4 FNN Taxonomy: Universal, Sparse, and Fractional FNNs

The basic FNN properties described in Section 2.1 were described without reference to a specific set of communication patterns. At the time we invented the FNN concept, it was oblivious to knowledge about specific communication patterns or PE pairs. In this section, we will introduce a simple taxonomy extending the original FNN concept to explicitly use information about the communications needed by a set of programs. These new variants of FNN can therefore take advantage of sparseness to produce lower part-count, cheaper, higher-performance network designs.

FNNs can be divided into three categories, Universal FNNs, Sparse FNNs and Fractional FNNs. A **Universal FNN** – the original concept – guarantees the FNN properties (single-switch latency and dedicated bandwidth) for *all possible* PE pairs in the machine, not just the ones that are expected to be used. A **Sparse FNN** guarantees the FNN properties for only a selected set of PE pairs in the machine, based on supporting a selected set of communication patterns. More precisely, a Sparse FNN guarantees that *all requested* PE pairs will have the FNN properties. For **Fractional FNNs**, each PE pair has a weighted importance value, and only a fraction of these PE pairs will have the FNN properties. A specific Fractional FNN attempts to *maximize the sum of the importance values* for the set of covered PE pairs. It is worth noting that unlike Universal FNNs, both Sparse FNNs and Fractional FNNs are not $(v, k, 2)$-covering designs as discussed in Section 2.1 because not all PE pairs have single-switch latency. Which FNN type is best for a parallel computer design depends on how much is known about the expected communications, the nature of these communication patterns, and the desired generality vs. cost trade-off of the resulting design.

Both Sparse FNNs and Fractional FNNs take advantage of a priori knowledge about the communication patterns that are likely to be used; in the (relatively rare) cases where no such information is available, a Universal FNN is most appropriate. The original concept and implementations of FNNs from 2000 guarantee single-switch latency for all pairs of PEs in the machine, and they were described by that name in a number of publications [16, 18, 19, 20, 30, 37], but as we realized that other variants could be important, we came to distinguish this type as Universal FNNs. A Universal FNN has the property that *any* communication pattern that is a permutation will pass through the network conflict free[3]. However, Universal FNNs have scaling constraints that limit their cost effective applicability. For instance, it is clear that for an N PE system, with ρ-port switches, that each PE will need to connect to at least $\eta \geq \frac{N-1}{\rho-1}$ switches. With a fixed size ρ, as N grows, the number of NIs per PE, η, must grow as well. As η increases, the cost per PE of a Universal FNN increases, and at some point it exceeds the cost of other more traditional network designs such as Fat-trees or Clos networks. In practice, the cost trade-off is often driven by the relative cost of more NIs (for an FNN) vs. using routers instead of switches (for Fat-trees or Clos networks). In situations where a Universal FNN does not offer cost savings, it is up to the designer to determine if the lower latency of the Universal FNN is worthwhile. Universal FNNs also can sometimes deliver more bandwidth

[3] Although the traffic presented to each individual switch in the FNN would be free of input and output port conflicts, real switches might have internal constraints that can cause some packets to be delayed more than others depending on the specifics of the traffic.

by simultaneously using multiple NIs per PE; which is a technique similar to the concept called "trunking" or "link aggregation." With current commodity PEs, switches, and NIs, Universal FNNs are both cheaper and higher performance than more conventional designs for up to a few hundred PEs.[4]

For program suites that have known primary communication patterns, a Sparse FNN will produce a cheaper and more scalable solution than a Universal FNN. A Sparse FNN guarantees the single-switch latency property for only a selected set of PE pairs, rather than all possible PE pairs. The PE pairs are selected based on the communication patterns that are expected to be used. The Sparse FNN has the property that any permutation pattern within the selected communication patterns will pass through the network conflict free[5]. Communication patterns that include PE pairs not covered by a Sparse FNN design are still able to be executed using it, but with performance approximating that of a more conventional network. When utilizing the same network technology, Sparse FNNs yield comparable performance on the selected communication patterns and scale to much larger numbers of PEs than Universal FNNs; tens of thousands of PEs can be supported with commodity hardware that a Universal FNN could not use for much more than a hundred PEs.

Fractional FNNs can be most appropriate for either of two very different reasons. Sometimes, the communication patterns used by a program are in theory knowable, but not directly available – for example, because the author of the application has kept the algorithms and code proprietary. In this case, experimentally determining statistical properties of the code's communication patterns can produce weightings that can guide a Fractional FNN design, but it may be difficult to determine a fixed threshold by which the "negligible" pairs could be removed to create a specification for a Sparse FNN. The other reason a Fractional FNN can be most appropriate is that sometimes the hardware budget simply is not sufficient to produce even a Sparse FNN, in which case, a Fractional FNN can more intelligently select how to subset the communications that are given optimal performance. This best-effort approach gives Fractional FNNs additional scalability beyond Sparse FNNs, although supporting up to tens of thousands of PEs is not a serious limitation of Sparse FNNs at this time.

Relative to this dissertation, we view Universal FNNs as the foundation and inspiration, Sparse FNNs as the primary contribution, and Fractional FNNs as a potentially important direction for the future.

[4]There are more than a few public recognitions of this fact, including prestigious awards[16, 30]. The primary impediments to Universal FNNs being widely adopted seem to be rooted in the asymmetry of the designs and the entire concept of a network design being too complex for a human to create. We hope that these issues will slowly fade as the design tools and runtime software support improve, and the current work involving Sparse FNNs certainly has helped to make FNNs in general appear less risky.

[5]Although the traffic presented to each individual switch in the FNN would be free of input and output port conflicts, real switches might have internal constraints that can cause some packets to be delayed more than others depending on the specifics of the traffic.

Chapter 3

Techniques for Designing Universal and Sparse FNNs

There is a direct way to construct Universal FNNs using small fully connected graphs, otherwise known as complete graphs, $K_{\eta+1}$, where $\eta + 1$ is the number of nodes in the graph. To do so, simply replace each edge in $K_{\eta+1}$ with a ρ-port switch, and each node of the graph with $\frac{\rho}{2}$ PEs. This construction method results in symmetric Universal FNNs with $\frac{\rho\times(\eta+1)}{2}$ PEs and η NIs per PE. These FNNs have an obvious geometric shape, such as a triangle ($\eta = 2$) or tetrahedron ($\eta = 3$) as shown in Figure 3.1. Figure 3.1 is based on a similar figure in [1], which describes the network for the Bunyip Supercomputer. For Bunyip, each lettered circle represents a 24 node sub-cluster and each numbered square is a 48-port switch. The smallest case of $\eta = 1$ results in the single ρ-port switch design with ρ PEs. This geometric construction technique is useful for finding Universal FNN designs by hand, but does not result in designs for large systems that are as cost effective as the designs found by the techniques discussed below. Also, the geometric construction technique does not yield answers for arbitrary numbers of PEs – in general, asymmetric designs are needed. However, relative to this dissertation, the primary limitation to the geometric construction technique is that it cannot be directly applied to Sparse FNN design.

3.1 A Genetic Algorithm (GA) for Finding Universal FNN Designs

As H.G. Dietz and I were working-out the basic concept of FNNs in April 2000, the original FNN design tool was created by H.G. Dietz; with minor changes, it is still the primary tool used for designing Universal FNNs. This design tool uses GA technology, but is highly specialized to the problem of designing a FNN. The genetic material for an individual is a direct representation of a network wiring pattern. The primary data structure is a table of bitmasks for each PE; each PE's bitmask has a one only in positions corresponding to each neighborhood (switch) to which that PE has a NI connected. This data structure does not allow a PE to have multiple NIs connected to the same switch, thus eliminating the non-simple bipartite graphs from the search. Enforcing this

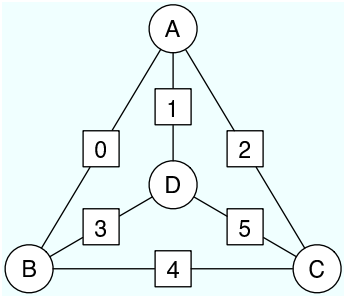

Figure 3.1: A Tetrahedral Universal FNN like the Bunyip supercomputer's network

constraint and limiting the number of "one" bits in the bitmask to η (NIs per PE) greatly narrows the search space, as described in Section 2.2.

To quickly converge on a good solution, the GA is applied in two distinct phases. Large network design problems with complex evaluation functions[1] are first converted into smaller problems to be solved for a simplified evaluation function. This rephrased problem often can be solved very quickly and then scaled up, yielding a set of initial configurations that make the full search converge faster.

The simplified evaluation function only values basic FNN connectivity, making each PE directly reachable from every other. The problem is made smaller by dividing both N (PE count) and ρ (ports per switch) by the same number while keeping η (NIs per PE) unchanged. For example, a design problem using 24-port switches and 48 PEs is first scaled to 2-port switches and 4 PEs; if no solution is found within the alloted time, then 3-port switches and 6 PEs are tried, then 4-port switches and 8 PEs, etc. This scaling technique is an extension of the geometric construction technique discussed at the beginning of this chapter. A number of generations after finding a solution to one of the simplified network design problems, the population of network designs is scaled back to the original problem size, and the GA resumes using the designer-specified evaluation function.

The initial population for the GA is constructed for the scaled-down problem using a very straightforward process in which each PE's NIs are connected to the lowest-numbered switch that still has ports available and is not connected to the same PE via another NI. Additional dummy switches are created if the process runs out of switch ports; similarly, dummy NIs are assigned to virtual PEs to absorb any unused real switch ports. The resulting scaled down initial FNN design satisfies all the constraints except PE-to-PE connectivity. Because the full-size GA search typically begins with a population created from a scaled-down population, it also satisfies all the basic design constraints except connectivity. By making all the GA transformations preserve these properties, the evaluation process needs to check only connectivity, not switch port usage, NI usage, etc.

The GA's generation loop begins by evaluating all new members of a population of potential FNN designs. Determining which switches are shared by two PEs is a simple matter of bitwise AND of the two bitmasks; counting the ones in that result measures the available bandwidth between the

[1]The complex evaluation function for a network design might incorporate additional criteria beyond measuring how well the basic FNN properties have been satisfied. For example, the number of single-switch-hop paths between PE pairs that form a 2D torus might be important.

pair of PEs. Which evaluation function is used depends on whether the problem has been scaled down. The complete population is then sorted in order of decreasing fitness, so that the top KEEP entries will be used to build the next generation's population. In order to ensure some genetic variety, the last FUDGE FNN designs that would be kept intact are randomly exchanged with others that would not have been kept. If a new FNN design is the best fit, it is reported.

Aside from the GA using different evaluation functions for the full size and scaled-down problems, there are also different stopping conditions applied at this point in the GA. Because we cannot know what the evaluation value would be for the optimum design for the full-size search, it terminates only when the maximum number of generations has elapsed. In contrast, the scaled-down search will terminate in fewer generations if a FNN design with the basic connectivity is found earlier in the search.

Crossover is then used to synthesize CROSS new FNN designs by combining aspects of pairs of parent FNN designs that were marked to be kept. The procedure used begins by randomly selecting two different parent FNN designs, one of which is copied as the starting design for the child. This child then has a random number of substitutions made, one at a time, by randomly picking a PE and making its set of NI connections match those for that PE in the other parent. This forced match process works by exchanging NI connections with other PEs (which may be real or dummy PEs) in the child that had the desired NI connections. Thus, the resulting child has properties taken from both parents, yet always is a complete specification of the NI to switch mapping. In other words, crossover is based on exchange of closed sets of connections, so the new configuration always satisfies the designer-specified constraints on η and ρ.

Mutation is used to create the remainder of the new population from the kept and crossover designs. Two different types of mutation operation are used, both applied a random number of times to create each mutated FNN design:

1. The first mutation technique swaps individual NI-to-switch connections between PEs selected at random.

2. The second mutation technique simply swaps the connections of one PE with those of another PE, essentially exchanging PE numbers.

Thus, the mutation operators are also closed and preserve the basic η and ρ design constraints. The generation process is then repeated with a population consisting of the kept designs from the previous generation, crossover products, and mutated designs.

The output of the FNN GA is simply a table that represents the FNN wiring pattern found. Each line begins with a switch number followed by a colon, which is then followed by the list of PE numbers connected to that switch. This list is given in sorted order, but for ideal switches, it makes no difference which PEs are connected to which ports, provided that the ports are on the correct switch. It also makes very little difference which NIs within a PE are connected to which switch. However, to construct routing tables, it is necessary to know which NIs are connected to

each switch, so we find it convenient to also order the NIs such that, within each PE, the lowest-numbered NI is connected to the lowest-numbered switch, etc. We use this simple text table as the input to the other FNN tools for generating colored wiring labels, routing tables, etc.

Although the original Universal FNN design tool did not need to consider specific communication patterns, we found that it was possible to allocate additional unit-latency bandwidth to specific communication patterns within a Universal FNN at virtually no additional cost in complexity. Thus, we began exploring communication patterns. This transition also became the point at which the work of this dissertation began to diverge from the tool originally created by H.G. Dietz.

3.2 Specification of Communication Patterns

To design Sparse FNNs, it is necessary to first specify the set of communication patterns that we wish the Sparse FNN to support with guaranteed pairwise bandwidth and unit latency. This set can be represented by an $N \times N$ weighted connectivity matrix for N PEs, with the value at matrix element (x, y) being the relative importance of the communications between PE x and PE y. If the weights are restricted to zero and one, the matrix is in the same form as an adjacency/connectivity matrix; two PEs are considered adjacent if they are connected to a common switch. There are three primary approaches for determining these connectivity matrices.

1. Literature search within the target applications' domain(s)
2. Examination of the source code for the target applications
3. Analysis of instrumented test runs of the target applications

The first approach tends to yield a higher level representation of the communication patterns, which has to be converted into the connectivity matrix. For example, some frequency-domain transformation algorithms (such as various codings of FFT) communicate using a bit-reversal pattern in which PE x communicates with PE y where y's binary value is equal to the bits of the binary value of x listed in reverse order. Another example application domain is Computational Fluid Dynamics (CFD). CFD communication patterns commonly include the nearest neighbors on a rectilinear grid usually of two or three dimensions. Yet another example is quantum chromodynamics (QCD)[8] code which favors four dimensional nearest neighbor communications. Using such a literature search, one can construct the connectivity matrix for a desired Sparse FNN based on the genre of applications and algorithms that are expected to run on the machine. Of course, issues such as the particular mesh factorization used in a code often are not specified in the papers discussing its algorithms, so a significant degree of uncertainty remains after even the most careful reading of research publications.

The second approach constructs the connectivity matrix by directly examining the source code for the applications that will be run on the machine or consulting the code's author or documentation. Care must be taken to distinguish between how the author or application code views a

communication and how it really is implemented. Libraries like MPICH or LAM-MPI do not always implement high-level communications, such as MPI broadcast[49], in the way that one might expect. For example, broadcast could be done using a real hardware broadcast (which few systems support), a sequence of N messages, a K-ary tree of messages, etc. Perhaps the most seriously misleading operation is the personalized all-to-all, which seems to imply all PE pairs communicate, but is almost never implemented using a technique like that. Like the literature search, this approach also results in high level descriptions of the communication patterns that need to be converted into a connectivity matrix.

The third approach involves automated determination of communication patterns using instrumented runs of target applications. To do this task requires an already existing computer sufficiently powerful to run the target application on a representative data set, and for that computer system to have the ability to collect and count the communication events. This raw communication event trace would then need to be converted into a desired connectivity matrix. Some thresholding of the data would be required to prevent code startup or other rare communication events from overly influencing the contents of the connectivity matrix.

To assist in the third approach of using an instrumented test run of the target applications, the runtime support for FNNs discussed in Section 5.1.3 includes a data collection and reporting module that directly counts the packets sent from each PE to every other PE. This data can be processed by a script to generate a connectivity matrix suitable for the Sparse FNN design tools.

To assist in creating the connectivity matrices for the first two approaches, a `compattern` tool was written by the author. The `compattern` software allows the communication matrix to be specified as the union of any of the patterns we found to be common in a literature search. Each communicating pair yields a one entry in the matrix, every other pair yields a zero. The patterns available include:

- Hypercube, single bit difference in PE ID number (N must be a power of 2)
- Bit-Reversal of the PE ID number (N must be a power of 2)
- Perfect-Shuffle (N must be even)
- 2D Matrix Transpose of a single element per PE (N must be a square)
- 1D, 2D, 3D, 4D Grids or Tori with various sub-patterns independently selectable:
  - Distance 1 offsets in W, X, Y, or Z
  - Distance 1 diagonals in 2D and 3D
  - Power of 2 offsets in W, X, Y, or Z
  - All PEs that differ in only one dimension (e.g. every PE in same row, column, etc.)

Graphical representations of a variety of these patterns are shown in Figures 2.1-2.28 in Section 2.3. Each of those figures shows the upper right triangle of the connectivity matrix for 16, 32, 64, 128, and 256 PEs. For 2D, 3D, and 4D grids/tori, one can select to use just one balanced

factorization of N, or a set of normalized unordered factorizations. For example a 512 PE cluster can be specified as having any of the following 3D grids: 128x2x2, 64x4x2, 32x8x2, 32x4x4, 16x16x2, 16x8x4, or 8x8x8; the default would be 8x8x8. Once the PE count, communication patterns, and grid factorization are selected, the `compattern` tool creates a connectivity matrix that can be used by the Sparse FNN design tools discussed in the next two sections.

3.3 A Greedy Heuristic for Finding Sparse FNN Designs

The problem of finding a wiring pattern that satisfies a specific set of communication patterns is different from finding a wiring pattern that satisfies all possible communication patterns. Initially, we encoded the communication patterns into the evaluation function used in the Universal FNN GA, but the geometric scaling trick is completely ineffective for Sparse FNNs, and the GA trajectory toward a solution was unusably slow and unsteady. We needed a way to force the GA to make changes that had more direct relevance to the problem areas within the potential network designs being considered. In effect, the system uses a greedy heuristic to incorporate memetic information to help direct the search.

One can represent the communication patterns by assigning each PE a list of desired neighboring PEs, which we call a buddy list. For a Universal FNN, each PE's buddy list would consist of every PE except itself. For a Sparse FNN, an individual PE's buddy list is specific to the PE's position in the various communication patterns selected for the Sparse FNN. The important difference is that the buddy list of an individual PE in a Sparse FNN is highly dependent on its PE number. This difference dramatically reduces the effectiveness of the FNN GA described in Section 3.1 when applied to Sparse FNN designs.

First, the two mutation operations in the Universal FNN GA are much less likely to improve an individual PE's connectivity, relative to its buddy list, because two randomly selected PEs are not likely to have much similarity in their lists. Second, because Universal FNNs require that all pairwise communications are covered, it is relatively straightforward to scale a solution up by multiplying both the number of ports per switch, ρ, and the number of PEs, N, by the same factor; thus, the FNN GA starts by searching the much smaller spaces of scaled-down designs for a design which it could scale up to solve the specified problem. This scaling heuristic can be quite effective for Universal FNNs, however, it rarely helps for Sparse FNNs. Sparse FNN design needs a heuristic that respects the sparse nature of the buddy lists. To that end, I developed a greedy allocation heuristic to design Sparse FNNs, as described below.

3.3.1 The Basic Heuristic Sparse FNN Design Algorithm

The basic premise used by the heuristic Sparse FNN design algorithm is that, at each step in constructing a wiring pattern, the number of remaining unconnected buddies is maximally reduced. These steps are demarcated by deciding to connect a particular PE to a particular switch. We call each possible decision point a crosspoint, because when selected, a crosspoint connects one PE to

one switch. Here are the four basic phases that the heuristic algorithm follows:

1. Find the list of crosspoints that would maximally reduce the global number of unconnected buddies
2. Check for and eliminate crosspoints from the list that would directly lead to a failed design
3. Select one crosspoint from the remaining candidates list
4. Connect that crosspoint and update all affected data structures, then if not finished go back to phase 1

Each of those phases has various possible implementations, with some of the tested variations discussed in Section 3.3.3. First, however, it is appropriate to discuss the primary data structures used by the heuristic.

3.3.2 The Heuristic's Primary Data Structures

The current state of the Sparse FNN design problem is represented by a crosspoint matrix, with a column for each of S switches and a row for each of N PEs as shown in Figure 3.2. Each entry of this crosspoint matrix represents a potential connection between a PE and a switch. Initially, all of the crosspoints are marked as unconnected. Each crosspoint entry holds two integer reference counts and two single bit flags, all stored together as bit-fields in a signed 32-bit integer. The least significant 16 bits represents the amount that the global unconnected buddy count would be reduced if this crosspoint is connected next. In other words, this field holds the number of currently unconnected buddies that the given PE would now be connected to if this crosspoint was connected. The next 14 bits, the full reference count, indicates how many of those unconnected buddies have already used all their NIs. If the full count is nonzero, no new switches can be used to satisfy that buddy pair. Finally, the connected flag is the sign bit, and the available flag is in the next most significant bit. By using a signed integer, all connected crosspoints have a negative value regardless of the values in other fields, which simplifies the search in phase one of the heuristic. This particular bit-level encoding is not a fundamental requirement of the heuristic, but it dramatically reduces the execution time on the systems we have used to execute the design searches relative to other encoding schemes that we tested.

When the heuristic compares the importance of two crosspoints, a signed 32-bit integer compare is all that is required. In this scheme, connected or otherwise unavailable crosspoints rank below any crosspoint that is still available to be connected. Also, an available crosspoint with any full references will outrank a crosspoint with none. This latter feature allows the heuristic to quickly select crosspoints that must be connected to satisfy buddy pairs that have run out of NIs, although the particular crosspoint may not maximally reduce the global unconnected buddy count.

There also is an array that holds the number of remaining available ports on each switch, initialized to their port counts. Each PE has a list of which switches it is connected to, as well as a running

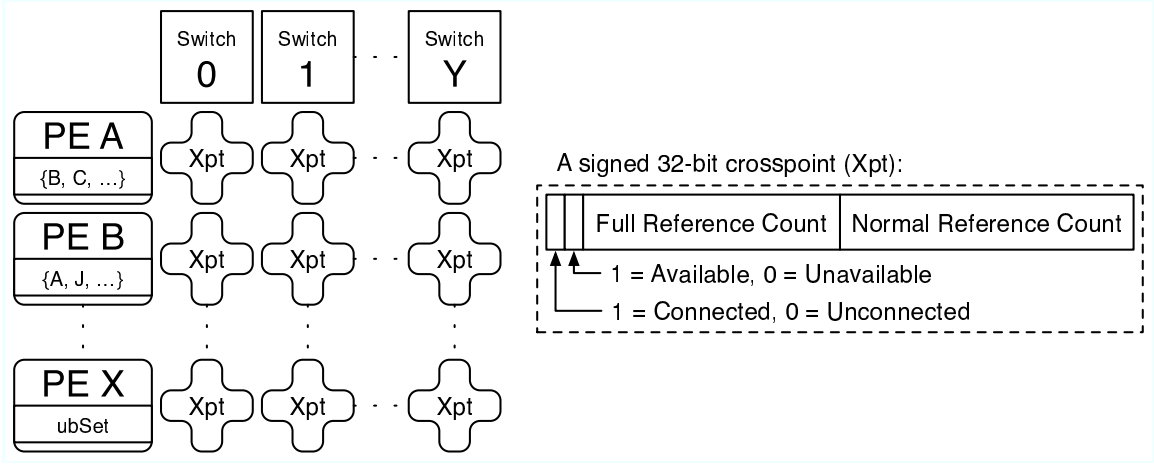

Figure 3.2: Heuristic's Data Structure

Algorithm 1 Initialize Heuristic's Data Structures

```
    procedure INITHEURISTIC
        {active switches} ← ∅
        {active PEs} ← ∅
        for all p ∈ {PEs in the design specification} do
5:          {active PEs} ← {active PEs} ∪ {p}
            peAvail[p] ← 0
            niAvail[p] ← η                          ▷ maximum allowed NIs per PE
            {switches that p is connected to} ← ∅
            {unconnected buddies of p} ← list from design specification   ▷ a.k.a. the ubSet
10:         unconnectedBuddyCt[p] ← value from design specification
        end for
    end procedure
```

total of available ports across that set of switches. Each PE has a list of currently unconnected buddies, called `ubSet` in Figure 3.2, initialized to its entire buddy list. For space reasons, the columns of the crosspoint matrix are only instantiated on an as-needed basis. Thus, the crosspoint columns for switches that are full, or not yet in use, are not actually stored. The initialization of these various data structures is shown in Algorithm 1.

3.3.3 Variations and Details of the Heuristic Algorithm's Four Phases

Shown in Algorithm 2 is the first phase of the heuristic – which is conceptually simple, although time consuming. It scans through each crosspoint on all the active switches collecting a list with the maximum rank. Various priority queue data structures were considered to reduce the time complexity of this phase, such as a Fibonacci heap. However, during phase four, such a priority queue needs to support both increasing and decreasing the key fields for many crosspoints. Thus, the time complexity of phase four would be greatly increased by the use of a priority queue for the cross-

Algorithm 2 Find Max Crosspoints

```
    function FINDMAXXPTS
        {candidate Xpts} ← ∅
        max ← AVAILABLE - 1
        for all s ∈ {active switches} do
 5:         for all p ∈ {active PEs} do
                v ← Xpt[s, p]
                if v ≥ max then
                    need ← unconnectedBuddyCt[p] - referenceCt(v)        ▷ needed ports
                    avail ← swAvail[s] - 1 + peAvail[p]                  ▷ available ports
10:                 if (niAvail[p] = 1) ∧ (need > avail) then
                        Xpt[s, p] ← v - AVAILABLE                        ▷ mark it as unavailable
                    else if v > max then
                        max ← v
                        {candidate Xpts} ← {(s, p)}
15:                 else
                        {candidate Xpts} ← {candidate Xpts} ∪ {(s, p)}
                    end if
                end if
            end for
20:     end for
        return {candidate Xpts}
    end function
```

points relative to using a linear array where crosspoints can be updated independently in constant time. The slowness of using a search through an array in the first phase instead of a priority queue is mitigated by the search using a linear access pattern through memory[2].

At the end of phase one, if the list is empty, a new switch is activated and a candidate list of crosspoints is selected to be the first connected to the new switch, as shown in Algorithm 3. These first crosspoints are selected based on the PEs requiring the most additional ports. In other words, each selected crosspoint corresponds to a PE with the most unconnected buddies that cannot fit onto any of the switches to which the PE is already connected. If there are any PEs that have only one NI left, the candidate crosspoints are only selected from PEs with only one NI remaining. This last rule helps the heuristic abort early if there is already a PE that can not be satisfied.

The second phase of the heuristic goes through the candidate list checking for crosspoints that would directly lead to a failed design. It is possible to quickly determine when connecting a particular PE's last NI to a switch, if its remaining unconnected buddies cannot be satisfied. For example, if a PE would have five unconnected buddies after connecting its last NI to a particular switch, that switch must have at least five additional ports available for those buddies to join the PE there. This test is quick enough that it is actually performed inside of phase one before a crosspoint is added to the candidate list. This quick check is not sufficient to detect all obviously bad crosspoint con-

[2]Linear access patterns on modern commodity processors and memory systems are much faster then random access patterns.

Algorithm 3 Find First Crosspoints

```
    function FINDFIRSTXPTS
        s ←a new unused switch
        {members of switch s} ← ∅
        swAvail[s] ← ρ                                  ▷ maximum allowed ports per switch
 5:     {active switches} ← {active switches} ∪ {s}
        last ← false
        for all p ∈ {active PEs} do
            if niAvail[p] = 1 then
                last ← true
10:         end if
            Xpt[s, p] ← AVAILABLE
        end for
        {candidate Xpts} ← ∅
        max ← −∞
15:     for all p ∈ {active PEs} do
            if last = (niAvail[p] = 1) then
                need ← unconnectedBuddyCt[p] - peAvail[p]
                if need ≥ max then
                    if need > max then
20:                     max ← need
                        {candidate Xpts} ← {(s, p)}
                    else
                        {candidate Xpts} ← {candidate Xpts} ∪ {(s, p)}
                    end if
25:             end if
            end if
        end for
        return {candidate Xpts}
    end function
```

ditions. The second phase really looks for crosspoints that would overflow a switch with required connections from friend of a friend constraints, which is shown in Algorithm 4. Specifically, when connecting the last NI of a PE to a switch, not only do its remaining unconnected buddies need to join it there, but for each of those buddies that would be using their last NI to do so, their unconnected buddies would also have to connect to this switch. This test recurses until either the switch would overflow, or no more full buddies are created. Any crosspoint that fails this test is removed from the candidate list, and the crosspoint's available flag is cleared so that it would not be picked again in the future. By skipping these crosspoints, the last NI of a PE could instead be connected sometime later to a switch with enough open ports to satisfy its unconnected buddies.

The third phase must select one crosspoint from the candidate list to be sent on to the fourth phase. If the list has only one member, selecting one is easy. Otherwise, some mechanism must differentiate the equally ranked crosspoints. One approach would be to try them all in sequence, and backtrack from failed designs. Unfortunately, that approach does not yield answers in a timely fashion, spending an exorbitant amount of time trying different connections on the last few switches, when the design needs to have connections changed on one of the early switches. Another alternative would be to select the candidates based on the maximum rank that would be subsequently found during the next first phase. This lookahead approach was found to be extremely costly, and made the heuristic take a very long time to find solutions. The best approach for the third phase of the heuristic seemed to be to just select one candidate crosspoint randomly from the list and move on. If the resulting design failed, try again from the beginning with a different random seed. With this approach for the third phase, the heuristic was able to find solutions for some Sparse FNN design problems very quickly (a fraction of a second runtime on a laptop for designs with hundreds of PEs). This random selection approach turned out to also be the key for combining the heuristic with a GA, which is discussed in the next section.

During the fourth phase of the heuristic, the selected crosspoint is marked as connected, and the various data structures are incrementally updated as appropriate, as shown in Algorithm 5. The newly connected buddies are removed from the appropriate unconnected buddy lists and some reference counts affected by this new connection are increased while some others are decreased. For formerly unconnected buddies that were already on this switch, their crosspoints on other switches are decremented, since they are now connected to this PE on this switch, as shown in Algorithm 6. For unconnected buddies that are not on this switch, their crosspoints on this switch are incremented. Also, the port availability counts are updated for each PE connected to this switch and for the switch itself. If this connection uses the last NI for a PE, the available flags for crosspoints in its row are cleared, to prevent the PE from being connected to more switches than it has NIs, thus satisfying the η constraint. Also, for each switch this PE is on that has ports available, the full reference counts for each of its unconnected buddies are incremented. As mentioned in Section 3.3.2, this step has the effect of promoting those crosspoints so that these critical connections will be preferentially selected during subsequent phase one passes. When the last port on a switch is used, the crosspoint column for that switch is marked as unavailable, and it's storage is freed, which satisfies

Algorithm 4 Crosspoint Closure Test

```
    function CLOSURESKIP(s, p)
        room ← swAvail[s] − 1
        P ← {members of switch s} ∪ {p}
        Q ← {p}
 5:     for all p ∈ Q do
            Q ← Q − {p}
            other ← peAvail[p]
            for all b ∈ ({unconnected buddies of p} − P) do
                must ← true
10:             if other > 0 then
                    for all c ∈ {switches that p is connected to} do
                        if Xpt[c, b] ≥ AVAILABLE then
                            other ← other − 1
                            must ← false
15:                         break
                        end if
                    end for
                end if
                if must then                          ▷ buddy must join this switch
20:                 P ← P ∪ {b}
                    room ← room − 1
                    if room < 0 then
                        return true                   ▷ switch would run out of room
                    else if niAvail[b] = 1 then
25:                     Q ← Q ∪ {b}                   ▷ must check his friends too
                    end if
                end if
            end for
        end for
30:     return false
    end function

    function XPTCLOSURETEST({candidate Xpts})
        for all (s, p) ∈ {candidate Xpts} do
            if CLOSURESKIP(s, p) then
35:             {candidate Xpts} ← {candidate Xpts} − {(s, p)}
                Xpt[s, p] ← Xpt[s, p] - AVAILABLE     ▷ mark it as unavailable
            end if
        end for
        return {candidate Xpts}
40: end function
```

Algorithm 5 Connect Crosspoint

```
    procedure CONNECTXPT(s, p)
        Xpt[s, p] ← Xpt[s, p] - AVAILABLE
        full ← (niAvail[p] = 1)
        for all b ∈ {unconnected buddies of p} do
 5:         v ← Xpt[s, b]
            if connected(v) then
                RECORDBUDDYCONNECTION(s, p, b)
            else if full then
                Xpt[s, b] ← v + 1 + FULLREF
10:             for all c ∈ {switches that p is connected to} do
                    Xpt[c, b] ← Xpt[c, b] + FULLREF
                end for
            else
                Xpt[s, b] ← v + 1
15:         end if
        end for
        for all m ∈ {members of switch s} do
            peAvail[m] ← peAvail[m] − 1
        end for
20:     Xpt[s, p] ← CONNECTED
        {switches that p is connected to} ← {switches that p is connected to} ∪ {s}
        niAvail[p] ← niAvail[p] − 1
        {members of switch s} ← {members of switch s} ∪ {p}
        swAvail[s] ← swAvail[s] − 1
25:     if swAvail[s] = 0 then
            {active switches} ← {active switches} - {s}
        end if
        if full then
            {active PEs} ← {active PEs} − {p}
30:     end if
    end procedure
```

Algorithm 6 Record Buddy Connection

```
    procedure RECORDBUDDYCONNECTION(s, p, b)
        {unconnected buddies of b} ← {unconnected buddies of b} − {p}
        unconnectedBuddyCt[b] ← unconnectedBuddyCt[b] − 1
        {unconnected buddies of p} ← {unconnected buddies of p} − {b}
 5:     unconnectedBuddyCt[p] ← unconnectedBuddyCt[p] − 1
        δ ← 1
        for all c ∈ {switches that p is connected to} do
            Xpt[c, b] ← Xpt[c, b] − δ
        end for
10:     if niAvail[b] ≤ 0 then
            δ ← 1 + FULLREF
        end if
        for all c ∈ ({switches that b is connected to} − {s}) do
            Xpt[c, p] ← Xpt[c, p] − δ
15:     end for
    end procedure
```

the ρ constraint. When updating the various data structures during this phase, a variety of simple tests are performed to check for a failed design due to this new connection. Unless backtracking is used in phase three, this failed design state forces the heuristic to halt early. Otherwise, the heuristic halts when there are no more NIs or switch ports left unused, or when the unconnected buddy lists are all empty.

3.4 Sparse FNN GA

As discussed in Section 3.3, the Universal FNN GA is not particularly suited to finding Sparse FNN designs. I developed a new steady state GA specifically for the Sparse FNN design problem which leveraged the approach taken by the Sparse FNN heuristic. In contrast to the Universal FNN GA, the Sparse FNN GA uses genetic material (DNA) that is not a direct representation of the network wiring pattern. Instead, the Sparse FNN DNA is used to influence the running of the Sparse FNN heuristic. In brief, the Sparse FNN GA uses DNA to select one of the candidate crosspoints during phase three of the heuristic described in Section 3.3.3. The steady state Sparse FNN GA has these five basic algorithmic steps, with further discussion in the subsections following:

1. Randomly select DNA from one or two parents in the current population
2. Generate new DNA using crossover or point mutations upon the parental DNA
3. Evaluate the network design that results from the newly generated DNA
4. Attempt to add the new individual to the current population, possibly removing a less fit individual to make room
5. If the selected end condition has not been reached, repeat from step one

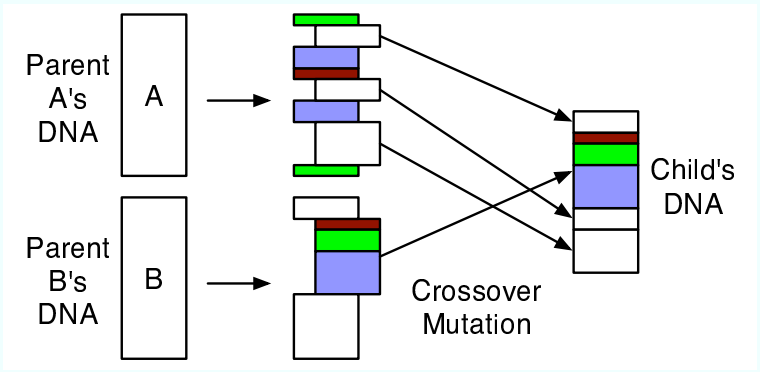

Figure 3.3: Crossover Mutation

3.4.1 What is the DNA used in the Sparse FNN GA?

To design a Sparse FNN, a target set of communication patterns that are to be efficiently supported by the network must be specified. This specification can be represented as a list of PE pairs that require single switch-hop communication path(s). This list can be directly generated in a canonical order from a connectivity matrix described in Section 3.2. The DNA for an individual in the Sparse FNN GA is a particular permutation of that list, assigning each desired PE pair a unique rank within the list. The third phase of the heuristic uses this ranking to break ties when selecting a crosspoint, which is described in Section 3.4.3. The permutation is stored as an array of integers, each an index into the master PE pair list. There is also an RNA representation that is a copy of the master list in the order specified by the DNA.

3.4.2 Sparse FNN GA Mutation Operations

A new individual in the Sparse FNN GA can be created either by cloning or by sexual reproduction. For clones, a random number of point mutations are applied that exchange the ranking of two randomly selected PE pairs in the DNA. At most one half of the DNA will be mutated in this way. If the parent's DNA evaluated to a failed design, the first point mutation exchange is biased to have a 25% chance of modifying the rank of the last PE pair used in the failed design. This targeted mutation greatly increased the speed at which the GA found solutions in a few simple test cases. With a bias of under 5% this improvement effect was not noticeable, and when the bias was over 33% there was no apparent further improvement in convergence speed.

For sexual reproduction, a crossover mutation operation is used, which is shown in Figure 3.3. The DNA from parent A is copied into the child's DNA. A random contiguous range of PE pairs is selected from parent B that is at most half the DNA. For each PE pair listed in parent B's selected DNA, those PE pairs are removed from the child's DNA. The holes in the child's DNA are then coalesced together to form one empty block in the same position as the parent B's selected DNA range. Finally, the parent B's selected DNA is copied into the empty block in the child's DNA.

Algorithm 7 A revised Heuristic Initialization sequence

```
    procedure NEWINITHEURISTIC
        {active switches} ← ∅
        {active PEs} ← ∅
        for all p ∈ {PEs in the design specification} do
 5:         {active PEs} ← {active PEs} ∪ {p}
            peAvail[p] ← 0
            niAvail[p] ← maximum allowed NIs per PE
            {switches that p is connected to} ← ∅
            unconnectedBuddyCt[p] ← 0
10:     end for
        n ← number of DNA elements
        for i ← 0, n − 1 do
            RNA[i] ← (a, b) ← pairList[DNA[i]]
            RNAndx[a,unconnectedBuddyCt[a]] ← i
15:         RNAndx[b,unconnectedBuddyCt[b]] ← i
            unconnectedBuddyCt[a] ← unconnectedBuddyCt[a] + 1
            unconnectedBuddyCt[b] ← unconnectedBuddyCt[b] + 1
        end for
    end procedure
```

This crossover mutation operation preserves the relative rankings within the set of DNA taken from parent A, and it preserves the absolute rankings of the DNA taken from parent B.

3.4.3 Evaluation Steps in the Sparse FNN GA

The cost of evaluating an individual in the Sparse FNN GA is considerably higher than for the Universal FNN GA due to the more abstract DNA representation. However, this more abstract DNA representation allows the GA to search the Sparse FNN design space much more efficiently than the Universal FNN GA. To evaluate an individual in the Sparse FNN GA, the DNA is converted into two primary data structures that will influence the running of the Sparse FNN heuristic. The initialization of the heuristic as discussed in Section 3.3.2 is modified to set up these two data structures, as shown in Algorithm 7. Specifically, a custom desired PE pairs list, the RNA, is generated in the order selected by the DNA. Each PE pair entry in the RNA is initially marked as unsatisfied. A second data structure is created that is the unconnected buddy list for each individual PE. These individual buddy lists are also in the order specified by the DNA, and each entry is simply an index into the RNA. Then the Sparse FNN heuristic is run, with a few changes to phases three and four.

In phase three, if the set of candidate crosspoints is for a new switch, select the crosspoint involving the earliest unsatisfied PE pair in the RNA. This selection step is simply a matter of finding the minimum RNA index value that any of the PEs from the candidate list has, as shown in Algorithm 8. In other words, of the given PEs, the PE is selected that has an unsatisfied buddy pair that is earliest in the RNA. If the candidate list is not for an empty switch, the crosspoint is

Algorithm 8 The Select First Crosspoint by RNA routine

```
    function SELECTFIRSTXPT({candidate Xpts})
        min ← number of DNA elements
        selected ← NULL
        for all (s, p) ∈ {candidate Xpts} do
 5:         i ← RNAndx[p, 0]
            if i < min then
                min ← i
                selected ← (s, p)
            end if
10:     end for
        return selected
    end function
```

selected that would satisfy the buddy pair that is earliest in the RNA, as shown in Algorithm 9. In either situation, the RNA selects which PE pairs get satisfied earlier than others, if the basic heuristic would have just picked a winner randomly. Also, as a hint to the mutation operation, the last RNA index that was used is recorded, so that if and when a design fails, the DNA that was most likely responsible can be preferentially mutated.

In phase four, as PE pairs are satisfied, their entry in the RNA is marked as such, and the corresponding entries in each PE's unconnected buddy lists are removed, as shown in Algorithm 10. This incremental update to the RNA and buddy lists makes the time complexity of the third phase be only $O(k)$ for a candidate list of length k.

3.4.4 The Parallel Sparse FNN GA

The parallel Sparse FNN GA uses a manager/worker computational scheme. Both the manager and the workers run the same basic serial Sparse FNN GA code with a few modifications. On the manager, the evaluation function does not directly call the Sparse FNN heuristic. Instead, the manager sends the DNA of the new individual to an idle worker, who will seed its population with that new DNA. If there are no idle workers, the manager will first wait for a worker to finish and collect its results, prior to sending a new individual to the worker. Until the manager gets results back from the workers, it defers adding individuals to its own population. Thus, the manager's GA runs in three phases: during one phase it generates new individuals without waiting for their evaluation results. In another phase it is both generating new individuals and incorporating evaluation results from the workers. The last phase is when the manager ceases to send out new individuals to the workers and simply collects the results from previously given work.

After receiving a seed individual from the manager, the worker runs the evaluation function on the new individual, then adds it to its local population. Then the worker runs the full serial Sparse FNN GA, and after a specified time limit, returns the best individual found so far to the manager. The worker then waits for a new seed individual from the manager. Before adding each seed individual, the worker removes the lower half of its population to make room for the new individual and its

Algorithm 9 The Select A Crosspoint by RNA routine

```
    function SELECTXPT({candidate Xpts})
        min ← number of DNA elements
        selected ← NULL
        for all (s, p) ∈ {candidate Xpts} do
 5:         j ← 0
            while (j < unconnectedBuddyCt[p]) ∧ (RNAndx[p, j] < min) do
                (a, b) ← RNA[RNAndx[p, j]]
                if b = p then                                  ▷ a is the buddy for p
                    b ← a
10:             end if
                if Xpt[s, b] = CONNECTED then
                    min ←RNAndx[p, j]
                    selected ← (s, p)
                end if
15:         end while
        end for
        return selected
    end function
```

Algorithm 10 Revised Record Buddy Connection

```
    procedure NEWRECORDBUDDYCONNECTION(s, p, b)
                                ▷ remove the entry corresponding to p from the list RNAndx[b, ?]
        {unconnected buddies of b} ← {unconnected buddies of b} − {p}
        unconnectedBuddyCt[b] ← unconnectedBuddyCt[b] − 1
                                ▷ remove the entry corresponding to b from the list RNAndx[p, ?]
        {unconnected buddies of p} ← {unconnected buddies of p} − {b}
 5:     unconnectedBuddyCt[p] ← unconnectedBuddyCt[p] − 1
        δ ← 1
        for all c ∈ {switches that p is connected to} do
            Xpt[c, b] ← Xpt[c, b] − δ
        end for
10:     if niAvail[b] ≤ 0 then
            δ ← 1 + FULLREF
        end if
        for all c ∈ ({switches that b is connected to} − {s}) do
            Xpt[c, p] ← Xpt[c, p] − δ
15:     end for
    end procedure
```

early offspring. If the design parameters of a new seed individual are different from the previous one, the worker invalidates the evaluation results of its already existing population, but keeps their DNA around in case it is already close to finding a solution. The design parameters that are allowed to change are the number of ports per switch (ρ), the number of NIs per PE (η), and the maximum allowed number of switches (S). None of those parameter changes would require a new canonical PE pair list, and thus all the worker's existing DNA would still be usable as input to the heuristic.

3.4.5 Sparse FNN Meta Search Problem

The manager is able to change some of the basic network design parameters on the fly, allowing one invocation of the program to search for solutions for the given communication pattern set with varying values for η, ρ, and S. Specifically, the manager is given a range of viable NIs per PE (usually from 2 to 8), and a set of switch widths, such as 48, 24, 16, and 8-ports. The search proceeds from widest switch to narrowest switch, attempting to find a solution for a given switch size (ρ) that uses the smallest number of NIs per PE (η). Once the minimum value of η for a particular ρ is reasonably known, narrower switches should need η to be the same or larger[3]. Thus, searches of narrower switch sizes with fewer NIs/PE are skipped.

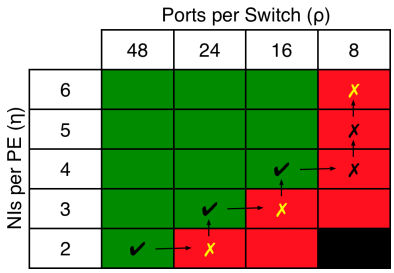

Figure 3.4: Meta Search example

Also, it is clear that for a particular switch width ρ, once a solution with $\eta = j$ is found, it is trivial to find a solution with $\eta = j + 1$. Thus, once a solution for a given ρ is found, it is not necessary to try with a larger η on that switch width. For each switch width, η is first tried at the same point as the next wider switch, increasing η until a solution is found in a reasonable amount of time. A hypothetical example is shown in Figure 3.4, where the dark green boxes in the upper left are known to have solutions, and the pink boxes in the lower right are suspected to not have solutions. The black box in the lower right corner is known to not be viable because the number of buddies of at least one PE is greater than 14. In this case the maximum possible neighbors for a PE connected to two 8-port switches is $\eta \times (\rho - 1) = 2 \times (8 - 1) = 14$. The search proceeds from the lower left to the upper right following the path indicated by the arrows. This search approach

[3]This relationship is not strictly true, because there is a packing problem involved. In some cases, a specific switch width may more easily cover a given communication pattern set than any other arbitrary width, even wider switches. However, if there is a solution with switches of width ρ, it is trivial to construct a solution using switches of width $k \times \rho$, for any positive integer k, by concatenating groups of k switches together to form the larger switches.

attempts to follow the dividing line between viable designs and non-viable designs. For any given switch width (ρ), it is possible that the GA gave up too soon for the $\eta = j - 1$ case, so after a first pass over all viable switch widths, a second pass is made attempting to find a solution using $\eta = j - 1$ for the widest switch that succeeded with $\eta = j$, as shown by the boxes with a yellow X in the figure.

It is also be feasible to follow the dividing line in the opposite direction, starting from the narrowest switches using the most NIs/PE. In the limit, the narrowest switch has two ports and is equivalent to a wire connecting two PEs. If η is not constrained, the solution to this degenerate case can be directly derived from the desired PE pair list. We did not choose this direction along the dividing line between viable and non-viable designs because the time and space complexity of the Sparse FNN heuristic, as it is currently written, is tuned for the number of switches to be no more than the number of PEs.

Chapter 4

How Well do Sparse FNNs Scale?

Now that we have a way to design Sparse FNNs, one wonders what size parallel machines can effectively use Sparse FNNs. There is no closed form scaling equation for Sparse FNNs, primarily because the design of an individual Sparse FNN is based on an arbitrary combination of communication patterns selected specifically for that individual design. The amount of networking equipment required for a given number of PEs can be dramatically different depending on the overlap and density of the selected communication patterns. The scaling of Sparse FNNs also is dependent on the number of NIs per PE (η) that can be used, and on the width (ρ) of the switches used. However, it is possible to explore a variety of Sparse FNN designs to observe trends in scalability. The following sections of this chapter present scaling data for over a thousand different Sparse FNN designs, spanning many combinations of input parameters. The chapter concludes with a presentation of a Sparse FNN design for a machine with 65,536 PEs.

4.1 Sparse FNN Scaling for Individual Patterns

In this and subsequent sections, a series of figure pairs is presented that represent the scaling of Sparse FNN designs for various sets of communication patterns for machines from only 8 PEs in size to machines with 16,384 PEs. The first figure in each pair presents the connectivity matrices for a few sample solutions for a given communication pattern set, typically for 16, 32, 64, 128, and 256 PEs. Also for this specific pattern set, many design parameter combinations for a wider range of machine sizes are condensed and presented in the second figure of the pair. The second figure contains information about the underlying communication pattern, as well as a summary of the results of many Sparse FNN designs that cover that communication pattern. The first pair of these figures will be discussed in greater detail to elucidate the meaning of the various elements in each figure.

4.1.1 The Hypercube Communication Pattern

As discussed in Section 2.3.2, the neighbors for a PE in the hypercube communication pattern include all PEs that differ by a single binary digit in their PE numbers. In other words, a hypercube has direct connectivity between all PE pairs that have a binary Hamming distance of one. In the case of a Sparse FNN that covers the hypercube communication pattern, each PE pair with a binary Hamming distance of one can communicate with the latency of a single switch hop. As the number of PEs in a hypercube increases, the number of requested neighbors for any given PE grows as $O(\log_2 N)$, while the number of possible PE pairs grows much faster as $O(N^2)$.

Sample Sparse FNN solutions using 3 NIs/PE ($\eta = 3$) are shown in Figure 4.1 for the hypercube pattern. Like the figures in Section 2.3 that showed the upper right triangle of the connectivity matrix for specific communication patterns, this figure shows the upper right triangle of the single-switch connectivity matrix for Sparse FNN solutions on a variety of machine sizes. The black pixels are the requested coverage, and the green pixels are extra PE pairs that also are covered with single-switch connectivity. If one looks closely at the figure, one can notice that not all the pixels have the same intensity. Darker pixels indicate that multiple single-switch paths are available between the corresponding PE pair. Because all these images represent Sparse FNN solutions, it is guaranteed that *every* requested PE pair has at least one single-switch path connecting the pair. While this figure gives some detailed information about a few Sparse FNN solutions to the hypercube pattern, it does not convey much information about how well the solutions scale. Shown in Figure 4.2 is the Sparse FNN scaling results for the hypercube communication pattern on a much larger set of solutions.

The colored lines generally going from the lower left corner to the upper right of Figure 4.2 represent Sparse FNN solutions using the narrowest switches (smallest ρ) of all the solutions found using a fixed number (η) of NIs per PE. For example, the solid red line represents solutions using only two NIs per PE, where the vertical coordinate of the line is the minimum width of the switches used by those solutions. Eight NIs per PE was the maximum number used in any attempted solution during the Sparse FNN design searches for all the figures. Although the data in these figures come from extensive runs of the Sparse FNN GA on the KASY0 supercomputer[35, 47], there is no guarantee that these solutions are the best possible Sparse FNN designs. Thus, the colored lines in the figure represent upper bounds on the minimum ρ needed for a given η; there may exist not-yet-found solutions using narrower switches.

The shaded gray region in Figure 4.2 shows the percentage of all possible PE pairings that are actually covered by the Sparse FNN solutions represented by the colored lines. The black line just below the gray region represents the percentage of all possible PE pairings that need to be covered to satisfy the hypercube communication pattern. Clearly, for any successful Sparse FNN design, its coverage must not be below the black line. For comparison, in a Universal FNN design, the black line would be across the top of the figure at a constant 100% coverage. Yet, for this Sparse FNN problem of covering the hypercube communication pattern, as the number of PEs increases, the requested fraction of possible PE pairings decreases dramatically. It is this downward sloped black line that allows Sparse FNNs to scale to much larger parallel machines than Universal FNNs.

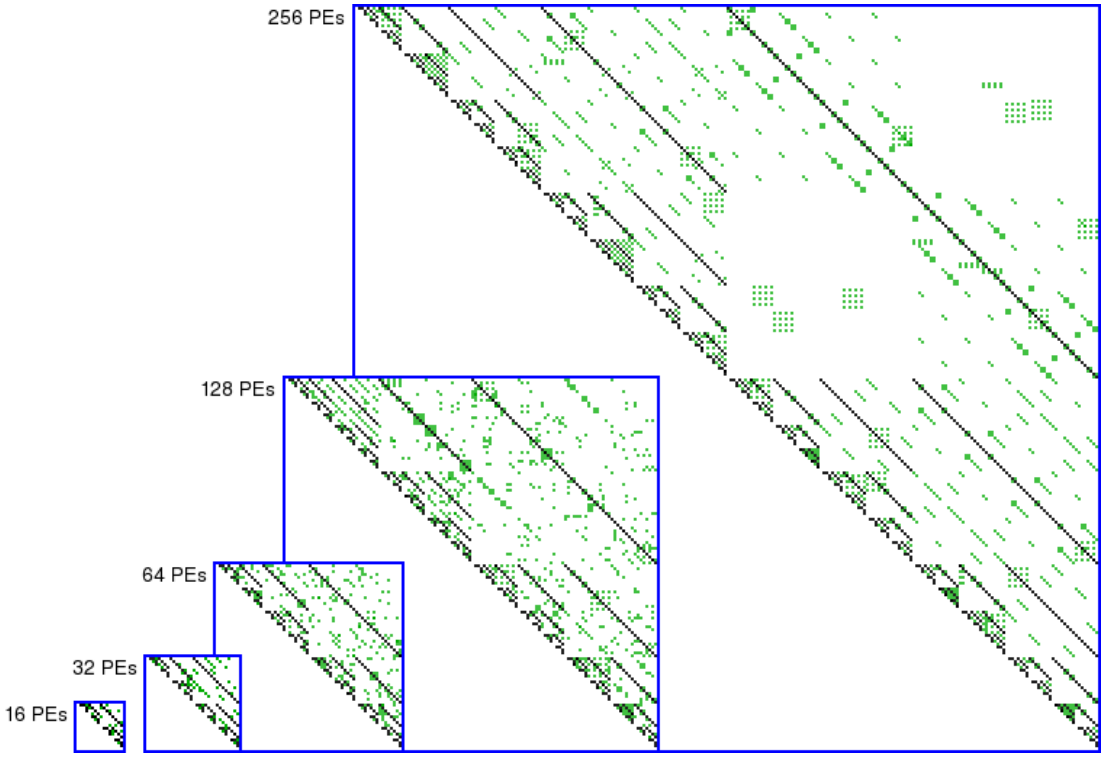

Figure 4.1: Solutions for the Hypercube, $\eta = 3$

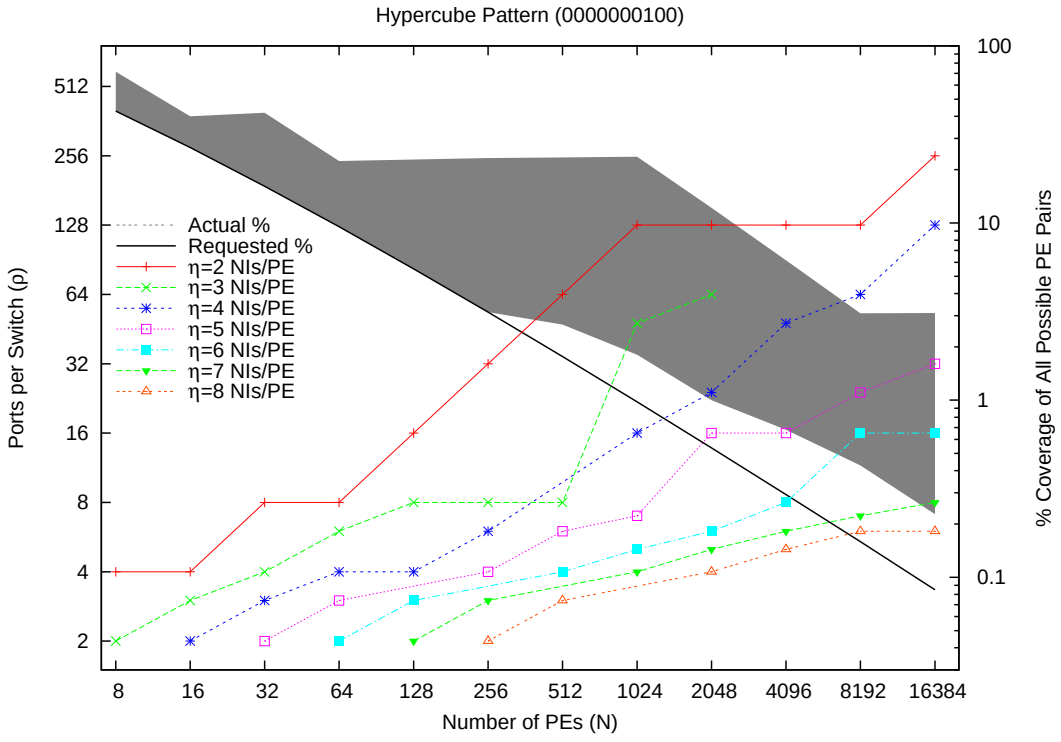

Figure 4.2: Hypercube scaling results

In most cases, arbitrarily wider switches can be used in a Sparse FNN design for a given η, if desired. Because there is a packing problem involved, a specific switch width may more easily cover a given communication pattern set than any other arbitrary width, even wider switches. However, if there is a solution with switches of width ρ, it is trivial to construct a solution using switches of width $k \times \rho$, for any positive integer k, by concatenating groups of k switches together to form the wider switches. These alternative Sparse FNN designs with wider switches are not shown in the figures to reduce the visual clutter. Also not shown in the figures are the number of switches (S) used for any particular solution. In general, the number of switches used is as many as are needed to supply enough total ports to connect all the NIs of all the PEs: $S = \left\lceil \frac{N \times \eta}{\rho} \right\rceil$. Thus Sparse FNN designs with wider switches tend to have fewer of them, while designs with more NIs per PE tend to have a larger number of switches. A continuous range of switch widths was not practical to explore due to the already large search space. Thus, the primary switches that were used when finding these solutions had the following number of ports per switch: 8, 16, 24, 32, 48, 64, 80, 96, 128, 256, and 512 ports. For comparison purposes, the figures include very narrow switches from seven ports all the way down to two ports – sizes that are not commercially viable, often costing more per port than wider switches.

A switch that has just two ports is equivalent to a directly-connected cable between two PEs. So, for the hypercube pattern, the data points along the bottom of Figure 4.2 represent traditional switchless implementations of the hypercube. In these directly connected cases, the requested communication pattern is covered precisely by the Sparse FNN with no extraneous PE pairs. These cases can be seen where the shaded gray region of the figure touches the black line for 256 or fewer PEs in a hypercube. Thus, for these few special cases with $\rho = 2$, there is a guarantee that no tighter cover exists, because the wiring pattern is a one-for-one match to the requested communication pattern. In most other cases with wider switches, extra PE pairs are covered by the Sparse FNN design that were not requested by the set of communication patterns. As we shall see, these extra pairs tend to dominate the coverage by Sparse FNN designs with wider switches.

This trend easily can be explained by casting the Sparse FNN design problem as a graph matching problem. The set of communication patterns specified for a Sparse FNN design with N PEs can be represented as a request graph with N vertices with edges between PEs for each desired communication path. Each ρ-port switch in a Sparse FNN can be modeled as a fully connected subgraph, K_ρ, with ρ vertices. The problem of designing a Sparse FNN is the same as repeatedly replacing ρ-vertex subgraphs of the request graph with K_ρ subgraphs, until there are no more original edges in the graph, with the constraint that each PE can be a member of at most η subgraphs. Clearly, each time a K_ρ subgraph is substituted for an original subgraph that was not fully connected, the new graph will have extra connectivity compared to the original request graph. For wider switches, it becomes more and more likely that the original subgraphs that are being replaced had fewer edges than the K_ρ subgraph that replaces them. In other words, attempting to cover a requested communications graph with large K_ρ subgraphs is not likely to be as precise a cover as one done using smaller K_ρ subgraphs, with K_2 yielding the tightest covers.

4.1.2 2D Communication Patterns

The first solutions for 2D patterns that we examine are for a 2D torus where the neighbors of a PE are the 8 nearest PEs, which includes the four diagonals. Sample Sparse FNN solutions with $\eta = 3$ are shown in Figure 4.3 for this pattern. Figure 4.4 shows the scaling results for the pattern. The number of neighbors per PE is a constant, and as can be seen in the figure, the directly connected solution with $\eta = 8$ is valid for all sizes of machines. Also, as a historical note, the case with $\eta = 4$ and $\rho = 4$ corresponds to the wiring pattern of the X-net on the BLITZEN[6] and MasPar MP-1[46] SIMD machines. A Sparse FNN that supports a single 2D torus for any particular number of PEs is not very interesting in itself, but leads us to the next set of 2D communications. Shown in Figure 4.5 are several solutions for the same 2D torus pattern as just described, but repeated for multiple factorizations of each machine size. For example, for the 256 PE case, not only is the 16×16 factorization included, but also the 128×2, 64×4, and 32×8 factorizations. These sample solutions have only 2 NIs/PE, because for this combination of patterns and machine sizes, there appear to be few cases that required just 3 NIs/PE, but rather either used 2 or 4 NIs/PE. Shown in Figure 4.6 are the scaling results for this pattern. The black line in this figure is higher and does not fall as rapidly as before with increasing machine size. Thus, as can be seen by the slopes of the colored lines, Sparse FNN designs for this combination pattern require more networking resources to be covered compared to the single 2D torus pattern. For a 1024 PE machine using 4 NIs/PE, the minimum switch width for a found solution was 48 ports. In contrast the previous single 2D torus example needed switches with only 4 ports each for the same problem.

Shown in Figure 4.7 are several solutions for the same multiple 2D tori pattern as just described, but without the diagonal neighbors. By removing the diagonal neighbors from the underlying 2D torus patterns, one can greatly improve the scaling costs for supporting multiple 2D factorizations, as shown in Figure 4.8. In this case, for the same 1024 PE machine with $\eta = 4$ NIs/PE, a solution was found that used switches with only 8 ports. Clearly, if the target program's performance is not critically affected by direct diagonal communications on a 2D grid, this design would be a much cheaper than the previous one using 48 port switches. Programs for 2D problems commonly piggyback diagonal neighbor communications with communications to either of the two common rectilinear neighbors. For example, when sending data to the Northern neighbor, also include the data that is relevant to the North-East neighbor in the same communication. Thus, in a subsequent communication step, the North neighbor can send the data to its Eastern neighbor.

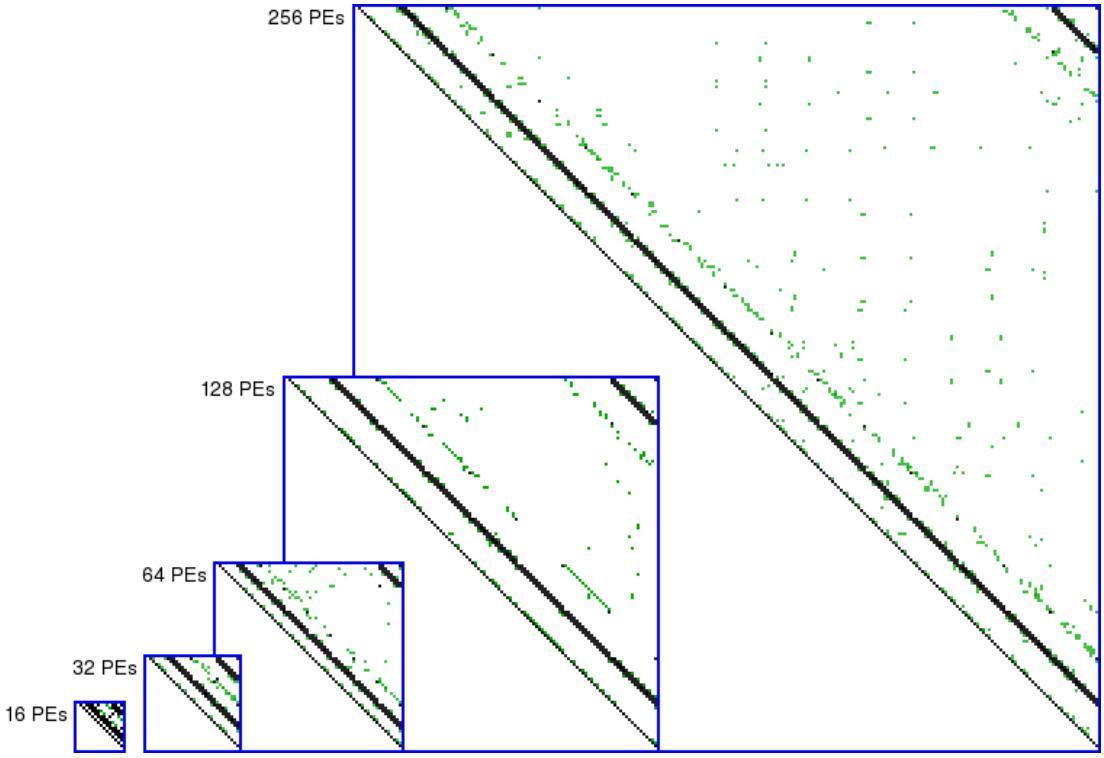

Figure 4.3: Solutions for Single 2D Torus with ± 1 offsets including diagonals, $\eta = 3$

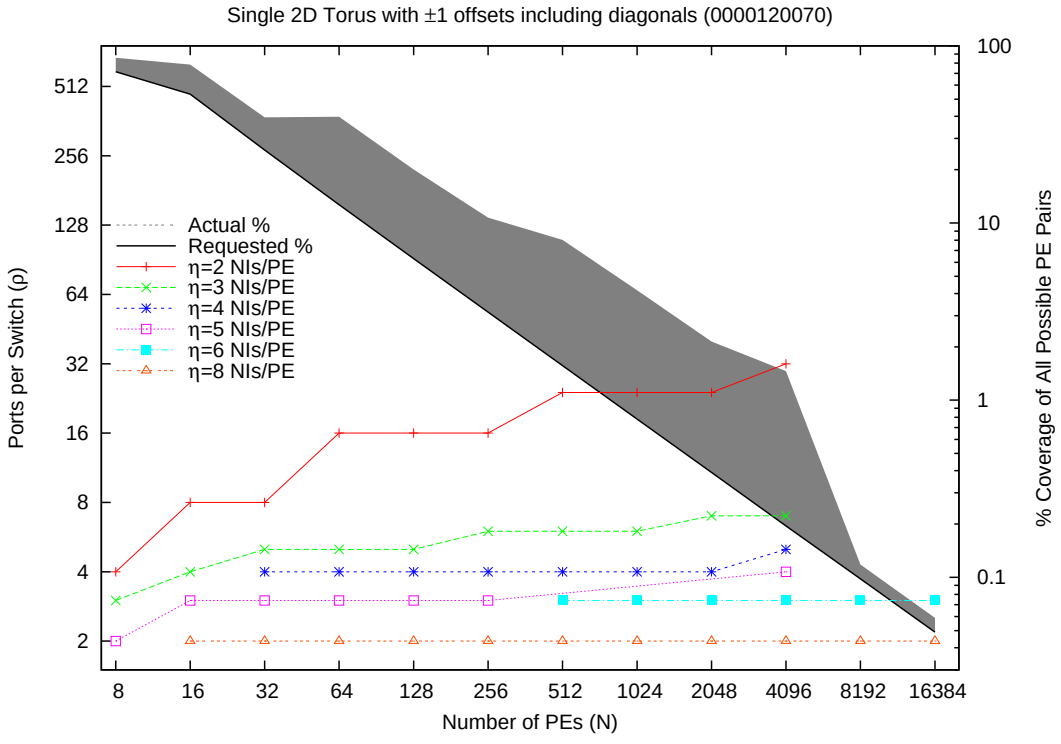

Figure 4.4: Scaling of Single 2D Torus with ± 1 offsets including diagonals

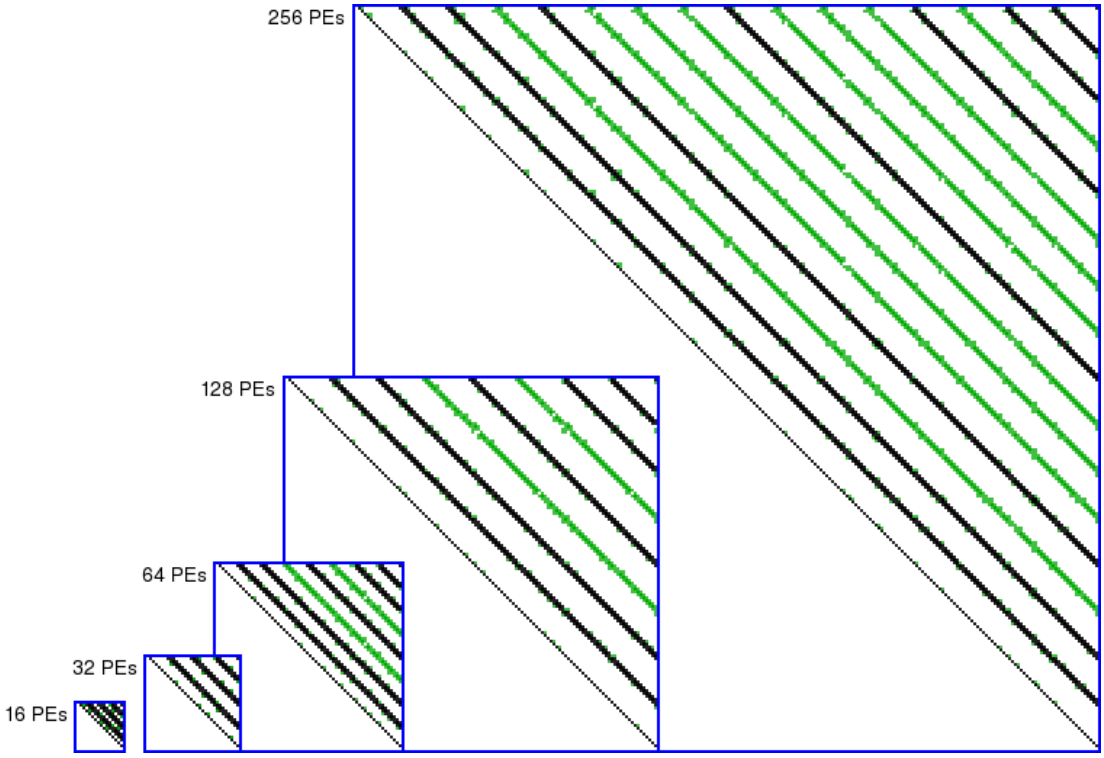

Figure 4.5: Solutions for Multiple 2D Tori with ± 1 offsets including diagonals, $\eta = 2$

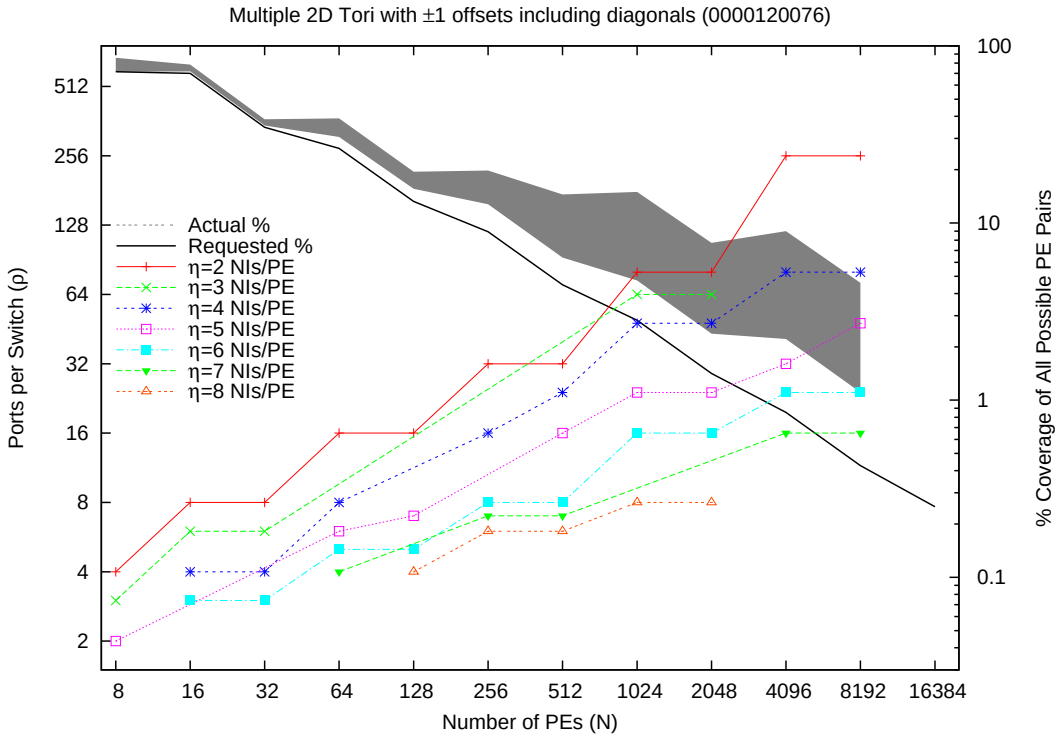

Figure 4.6: Scaling of Multiple 2D Tori with ± 1 offsets including diagonals

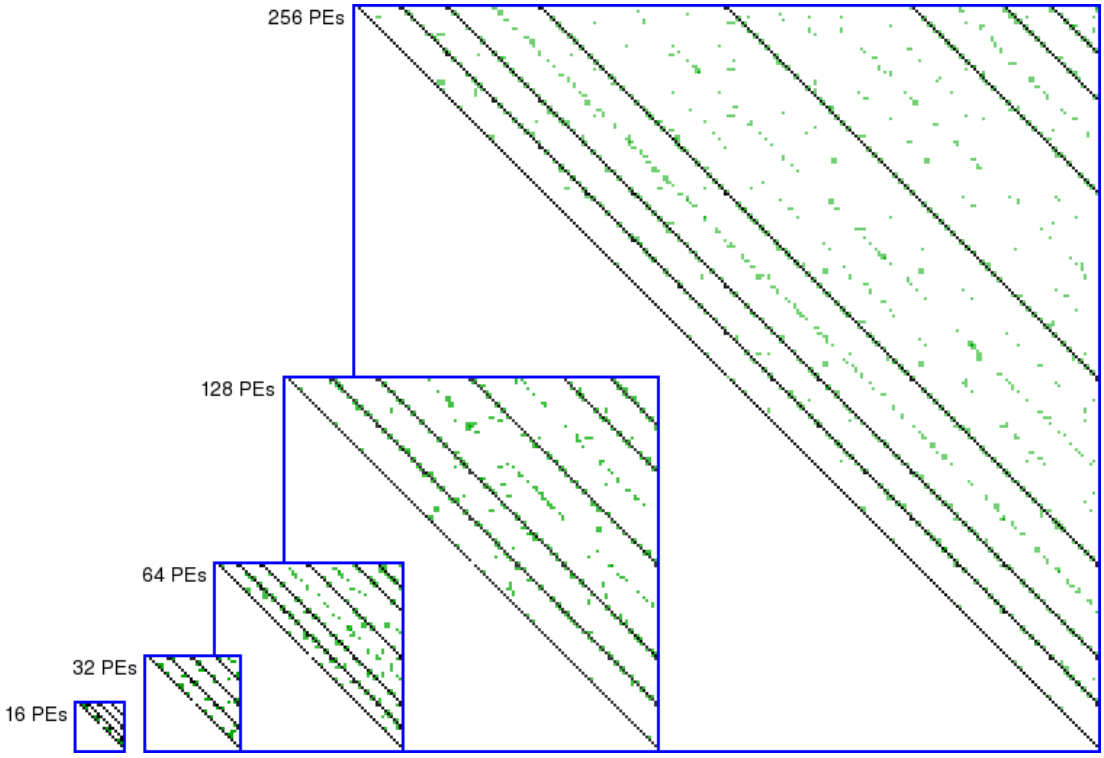

Figure 4.7: Solutions for Multiple 2D Tori with ± 1 offsets, $\eta = 3$

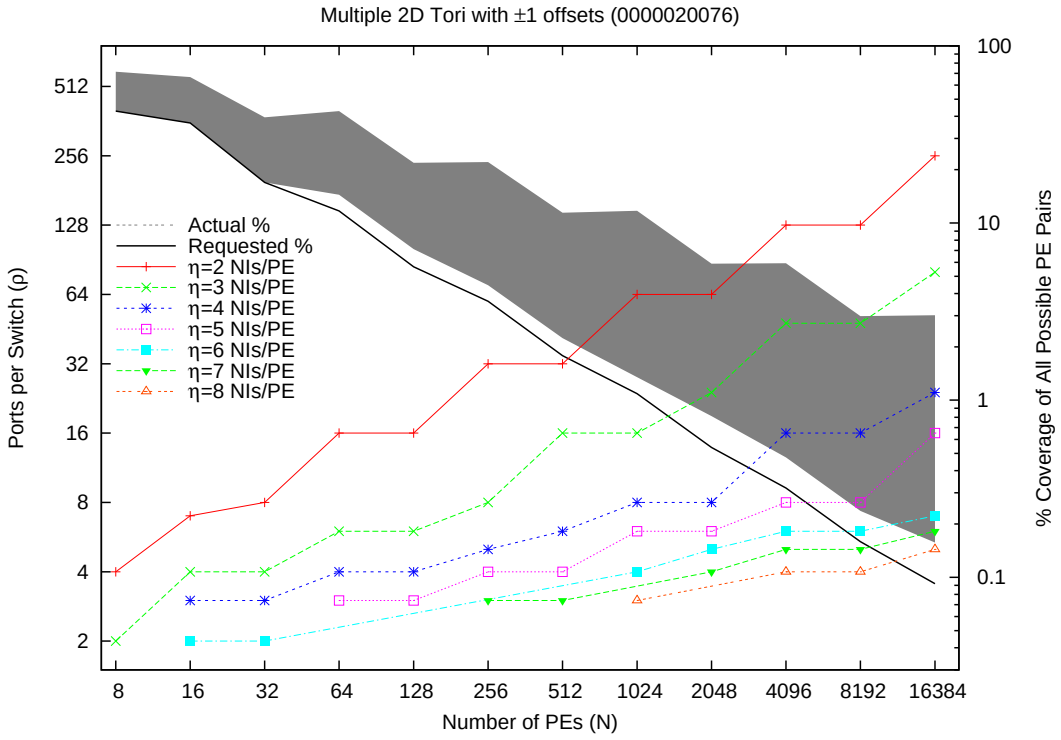

Figure 4.8: Scaling of Multiple 2D Tori with ± 1 offsets

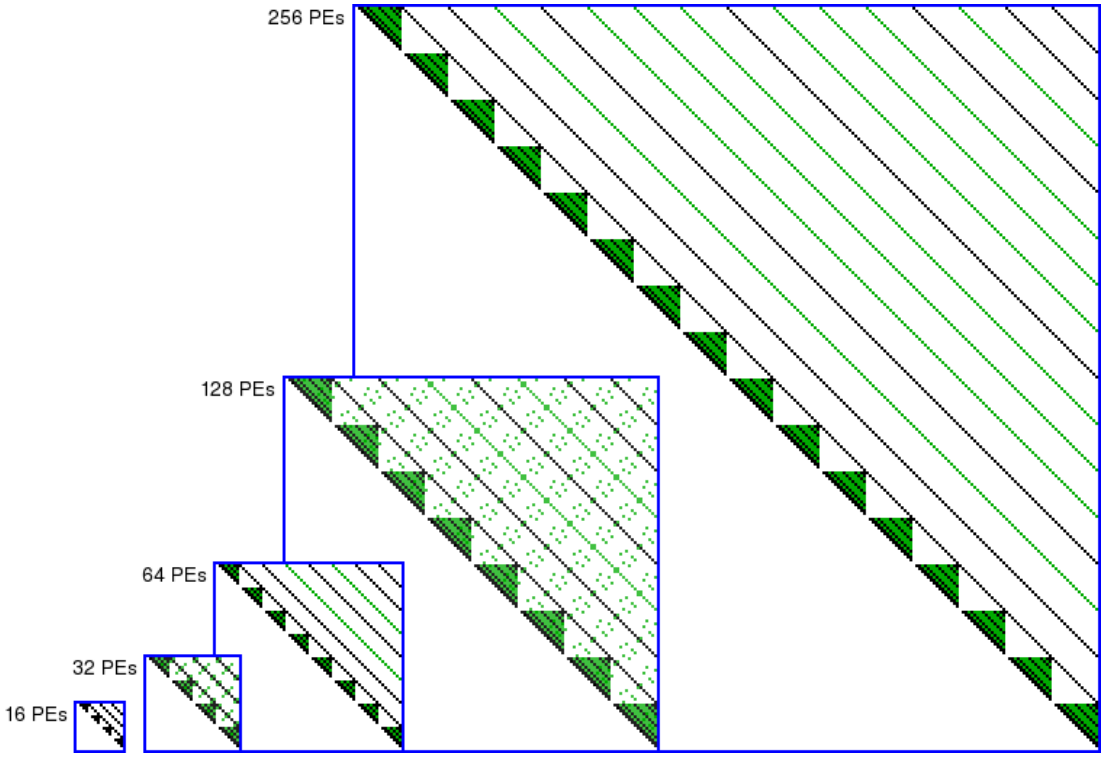

Figure 4.9: Solutions for Single 2D Torus with $\pm 2^k$ offsets, $\eta = 2$

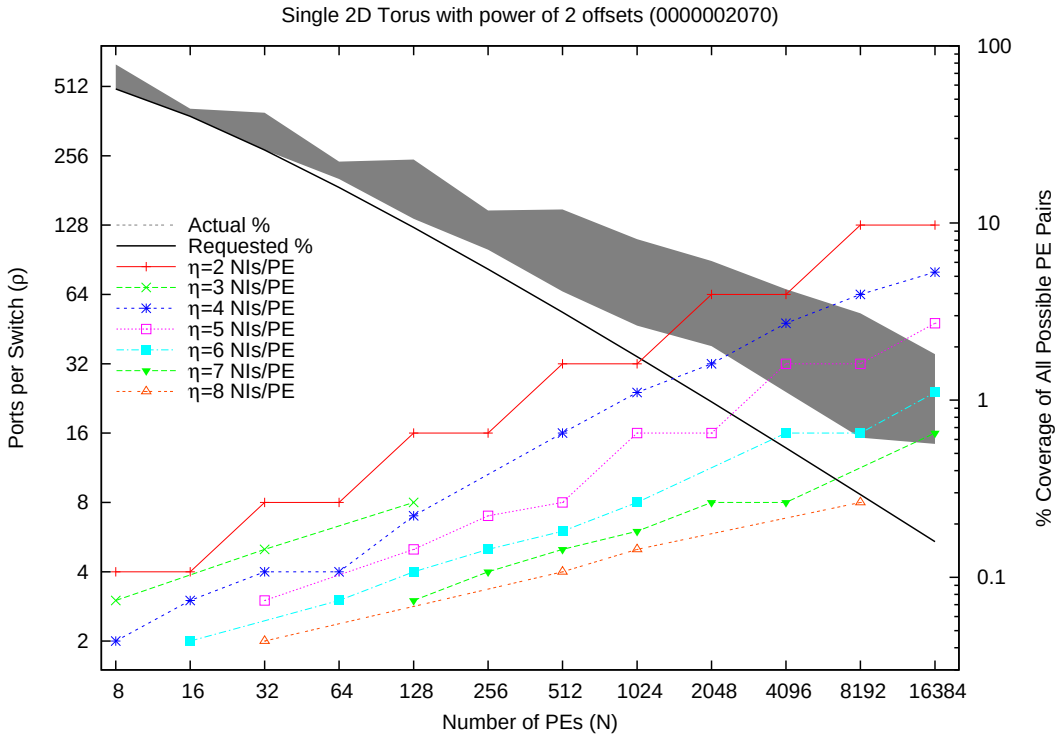

Figure 4.10: Scaling of Single 2D Torus with $\pm 2^k$ offsets

If the target applications need to perform reductions within a row or column, it would be beneficial for each PE to have connectivity with more than its two closest neighbors within the row (or column). Figure 4.9 shows several sample solutions using 2 NIs/PE for a single 2D torus factorization, but instead of the 4 (or 8) nearest neighbors per PE, it includes all PEs in the row (or column) that have a power of 2 distance along the row (or column). The scaling results for this pattern are shown in Figure 4.10. The minimum solution found for this pattern on 1024 PEs with $\eta = 4$ NIs/PE needed 24-port switches. Also of note are the solutions with $N = \rho^2$ PEs that used $\eta = 2$ NIs/PE. These particular solutions also cover the pattern shown in Figure 2.23 in Section 2.3.3. These $\eta = 2$ solutions are a rare symmetric Sparse FNN where one NI connects to a switch to cover the PE's row, and the other NI connects to a column switch, where each switch has $\rho = \sqrt{N}$ ports.

4.1.3 3D Communication Patterns

The first solutions for 3D patterns that we examine are for a 3D torus where the neighbors of a PE are the the six rectilinear nearest PEs that correspond to the six faces of a cube. Sample Sparse FNN solutions with $\eta = 3$ NIs/PE are shown in Figure 4.11 for this pattern, while Figure 4.12 shows the scaling results. As in the first 2D torus example in this chapter, the number of neighbors per PE is a constant, and thus not particularly interesting as a stand alone Sparse FNN design problem. However, in that first 2D torus example, we included the diagonal neighbors. If we do the equivalent for the 3D case, we need to include many more neighbors. Viewing the PEs as a tightly packed stack of cubes, there are twelve neighbors that touch a central PE on it's edges, and eight PEs that touch it at the corners, yielding twenty additional neighbors for that central PE. Sample solutions to this 26-neighbors-per-PE pattern are shown in Figure 4.13, while Figure 4.14 shows the scaling results. The discussion in the previous section about a programming optimization for eliminating direct diagonal communications can also be applied to the 3D case. Thus, it seems the added cost of supporting direct 3D diagonal communications is generally excessive.

The next pattern of interest is the 3D torus with six neighbors per PE, but repeated for multiple factorizations of each machine size. For example, for the 64 PE case, not only is the $4 \times 4 \times 4$ factorization included, but also the $16 \times 2 \times 2$, and $8 \times 4 \times 2$ factorizations. Sample solutions are shown in Figure 4.15 with $\eta = 3$, while Figure 4.16 presents the scaling results for this pattern. For comparison purposes, a 1024 PE solution found for this pattern with $\eta = 4$ used a minimum of 16 ports per switch, which is not too different from the similar 2D case which used 8-port switches.

As before in the 2D case, if instead of needing different grid factorizations, the target applications need to perform reductions along one or more dimensions, it would be beneficial for each PE to have connectivity with more than its two closest neighbors within a dimension. A few sample solutions with $\eta = 3$ are shown in Figure 4.17 for a single 3D torus factorization, but instead of the six nearest neighbors per PE, it includes all PEs in the row (or column, etc.) that have a power of 2 offset along one dimension. The scaling results for this pattern are shown in Figure 4.18. For this pattern, the 1024 PE with $\eta = 4$ case needed 16-port switches for the minimum solution found.

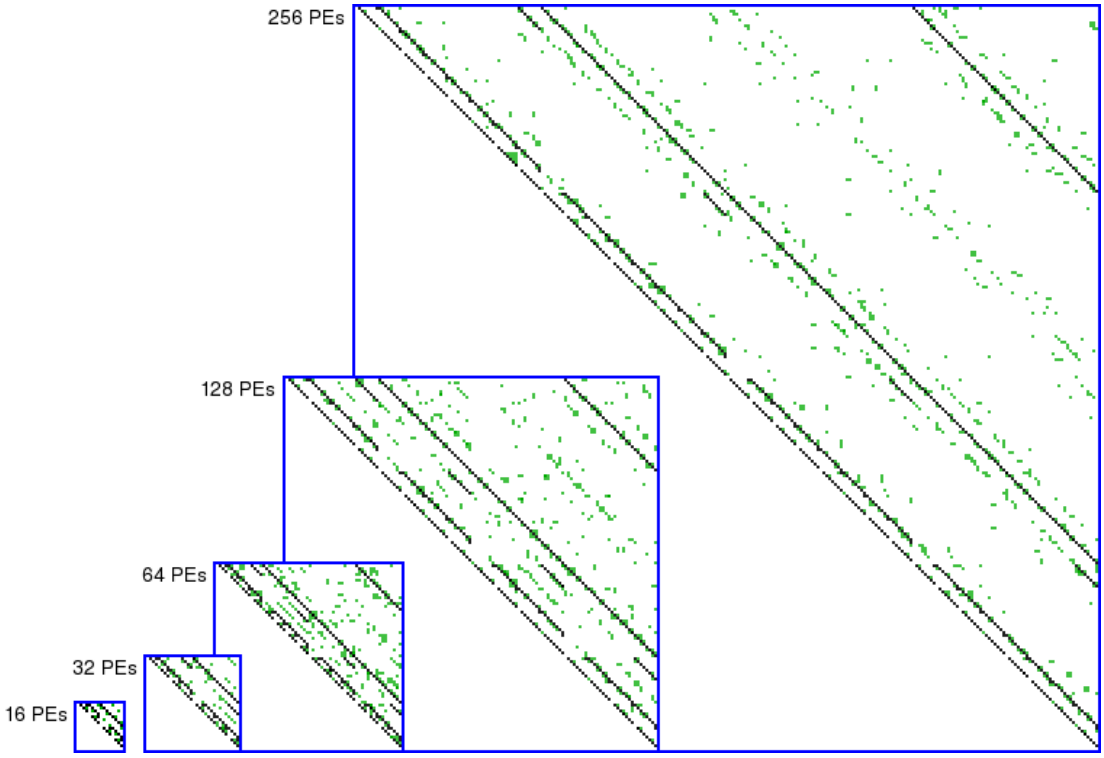

Figure 4.11: Solutions for Single 3D Torus with ± 1 offsets, $\eta = 3$

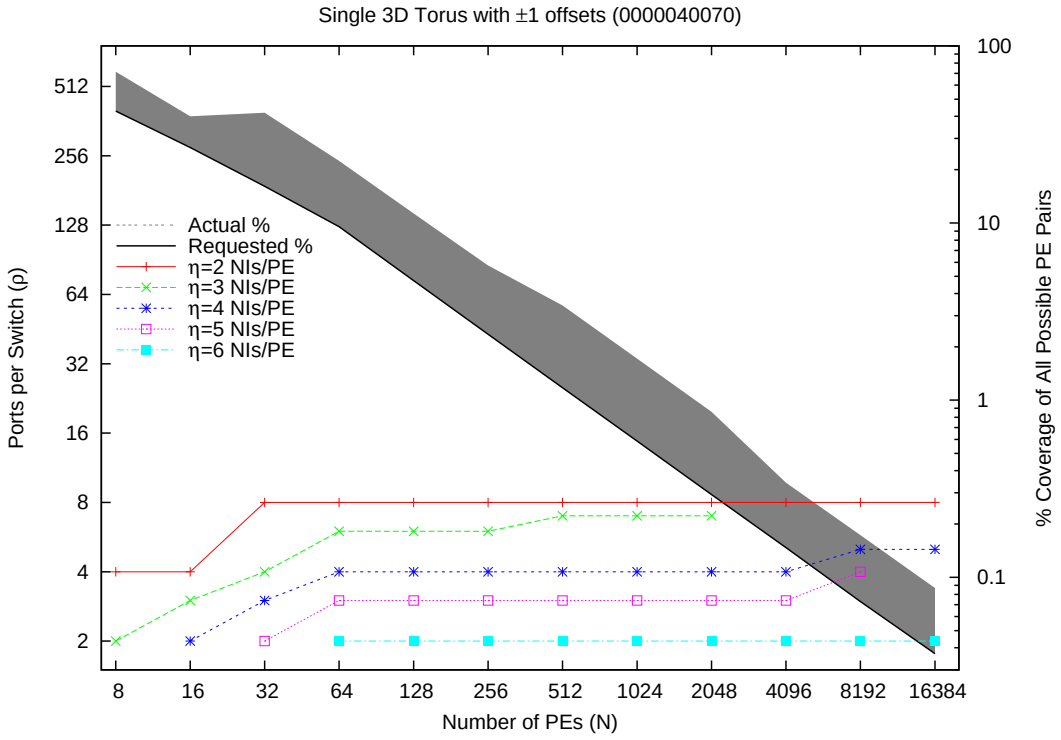

Figure 4.12: Scaling of Single 3D Torus with ± 1 offsets

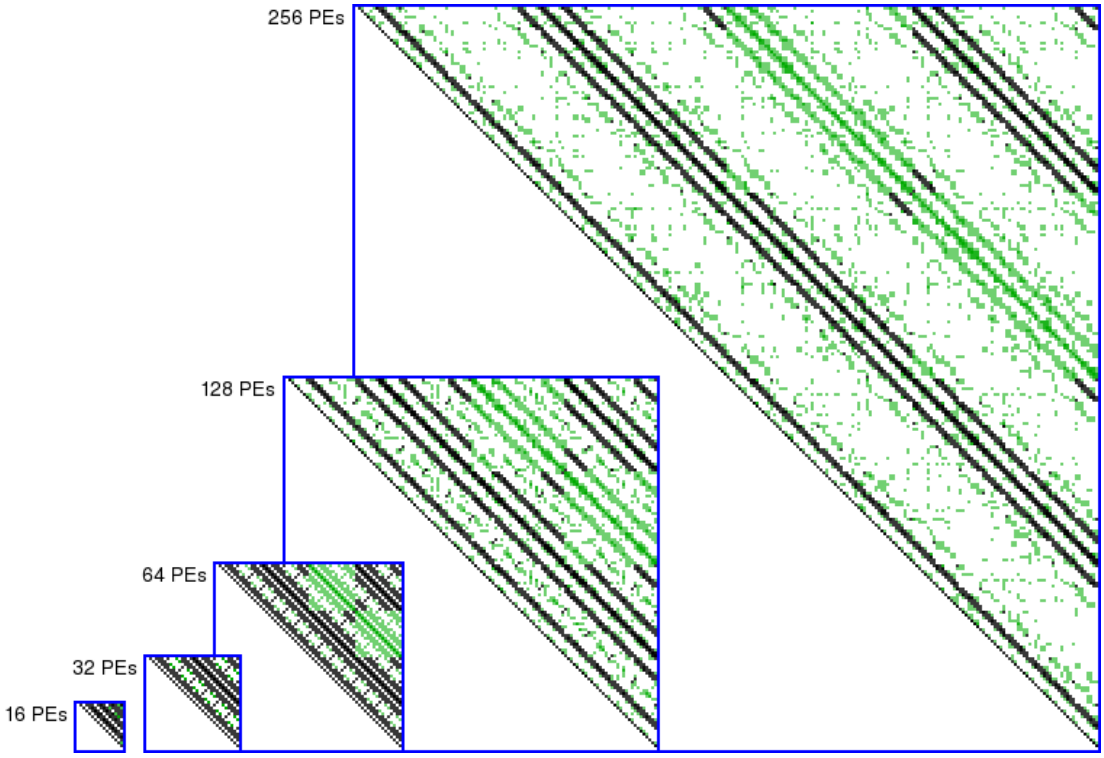

Figure 4.13: Solutions for Single 3D Torus with ± 1 offsets including diagonals, $\eta = 3$

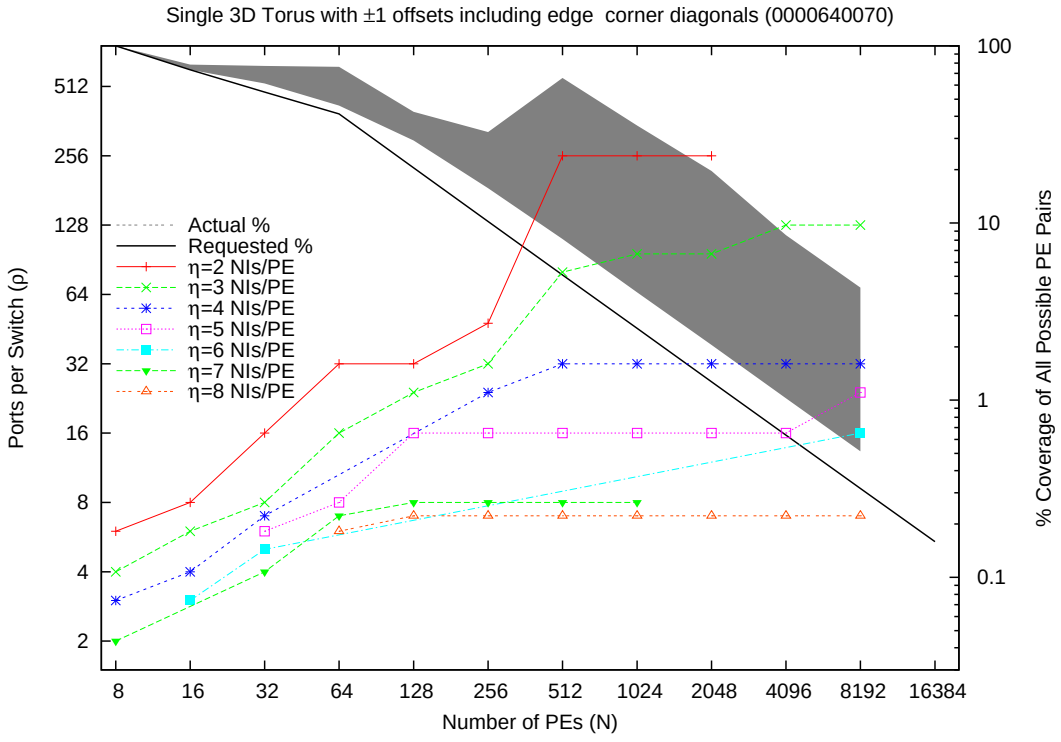

Figure 4.14: Scaling of Single 3D Torus with ± 1 offsets including diagonals

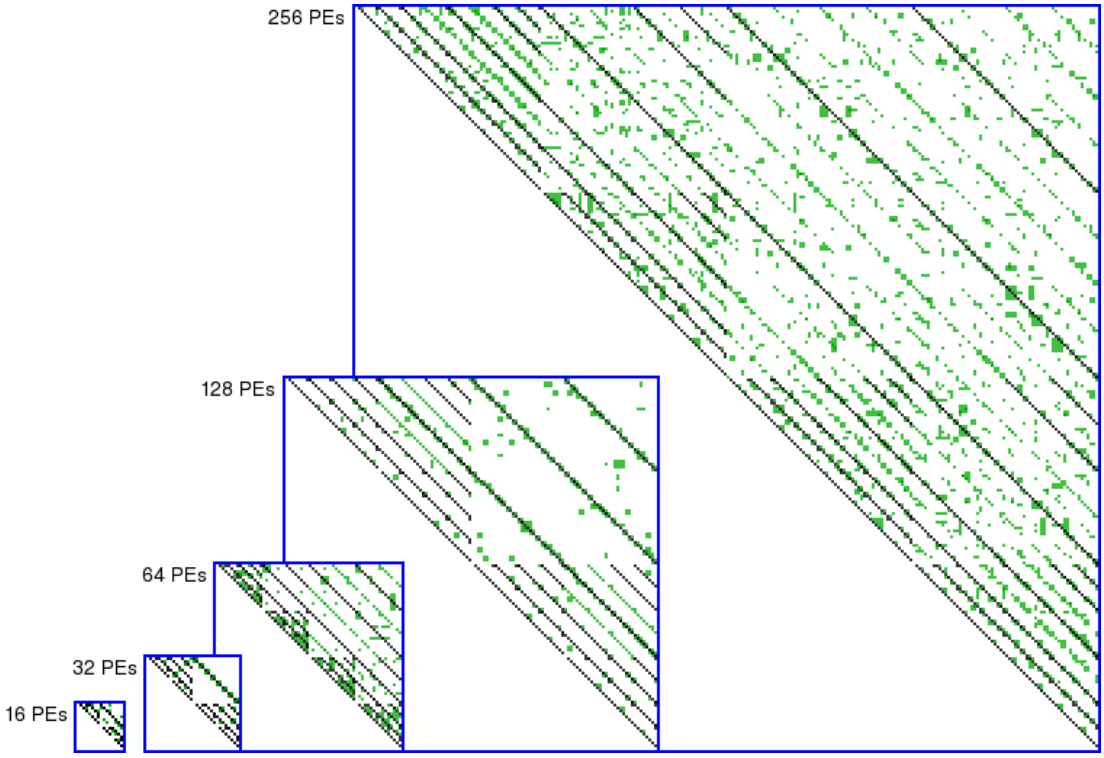

Figure 4.15: Solutions for Multiple 3D Tori with ± 1 offsets, $\eta = 3$

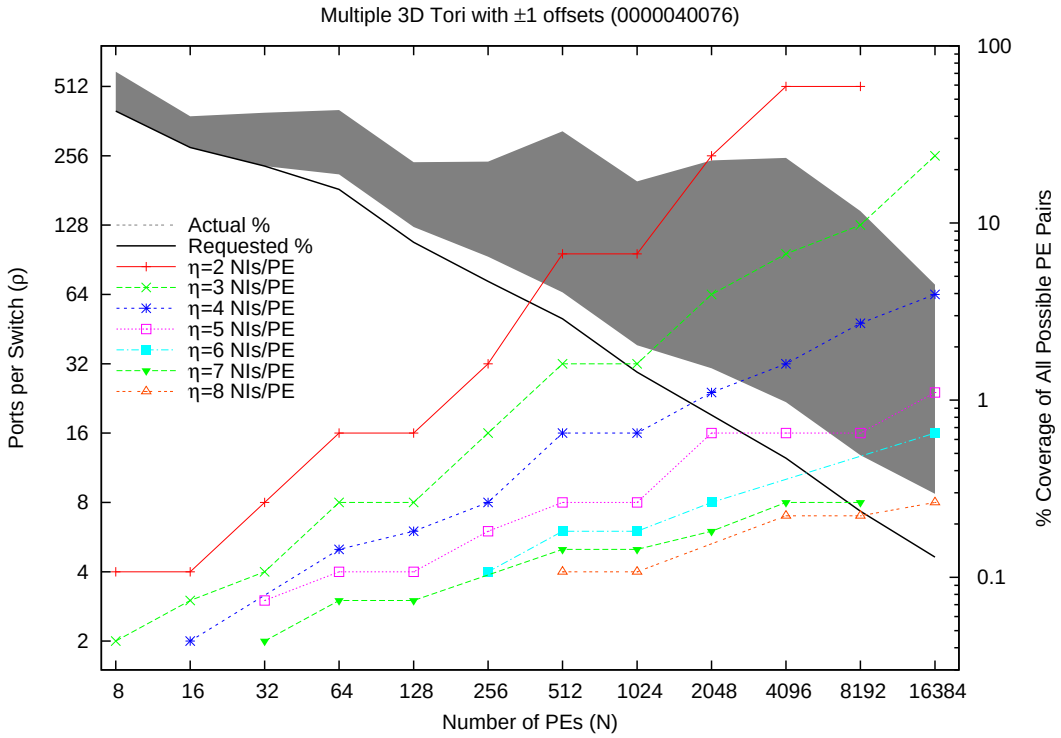

Figure 4.16: Scaling of Multiple 3D Tori with ± 1 offsets

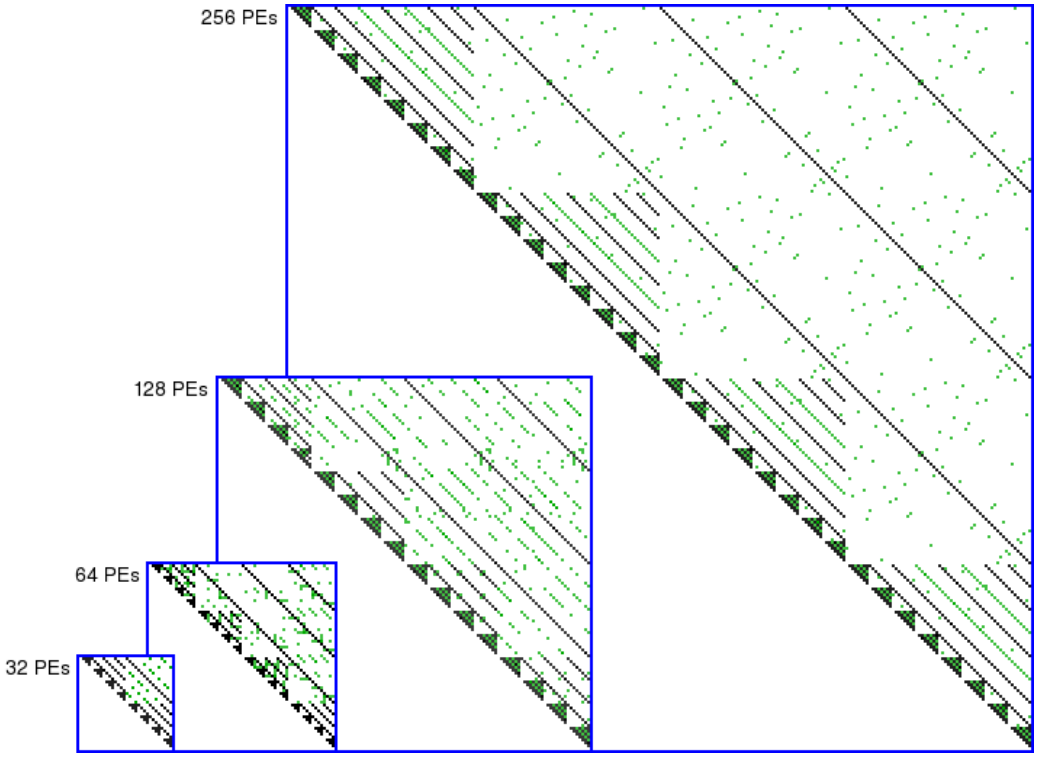

Figure 4.17: Solutions for Single 3D Torus with $\pm 2^k$ offsets, $\eta = 3$

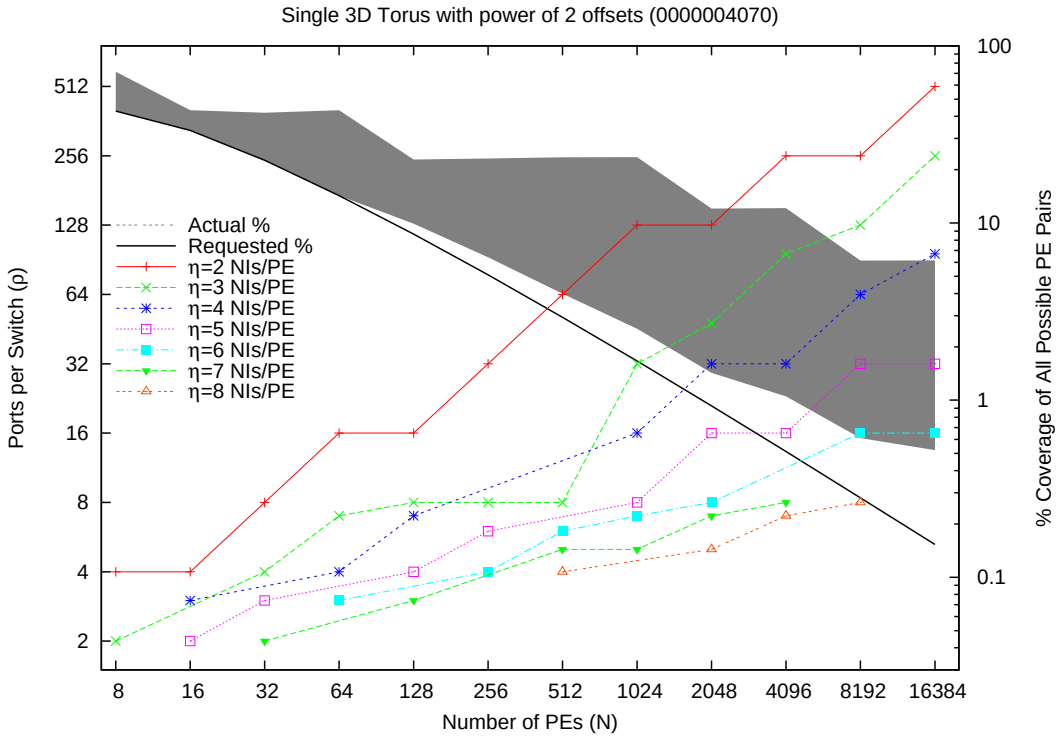

Figure 4.18: Scaling of Single 3D Torus with $\pm 2^k$ offsets

4.2 Sparse FNN Scaling for Combinations of Patterns

Although the scaling results for individual communication patterns presented in the previous section are informative, they do not encompass designs that are particularly advantageous for Sparse FNNs. This section presents scaling results for combinations of distinct communication patterns that together would not be covered efficiently by a traditional network topology. In contrast, as we shall see, a Sparse FNN is able to simultaneously support a variety of communication patterns efficiently.

4.2.1 Hypercube plus Tori with Single Factorizations

Figure 4.19 shows sample solutions with $\eta = 3$, while Figure 4.20 shows the scaling results for the following combination of communication patterns that include a single factorization for each torus sub-pattern:

- Hypercube
- Ring with distance 1 offsets in X
- Single 2D torus with distance 1 offsets in X, or Y
- Single 3D torus with distance 1 offsets in X, Y, or Z
- Single 4D torus with distance 1 offsets in W, X, Y, or Z

For this combination of patterns, the 1024 PE with $\eta = 4$ case needed 24-port switches for the minimum solution found. If the hypercube pattern is not sufficient for the expected global communications (such as for reductions on subsets of the PEs), one can include the power of 2 offset neighbors for the 2D, 3D, and 4D torus sub-patterns. Sample solutions for this expanded pattern are shown in Figure 4.21, while scaling results are shown in Figure 4.22. With this change in patterns, the 1024 PE with $\eta = 4$ case needed 32-port switches for the minimum solution found.

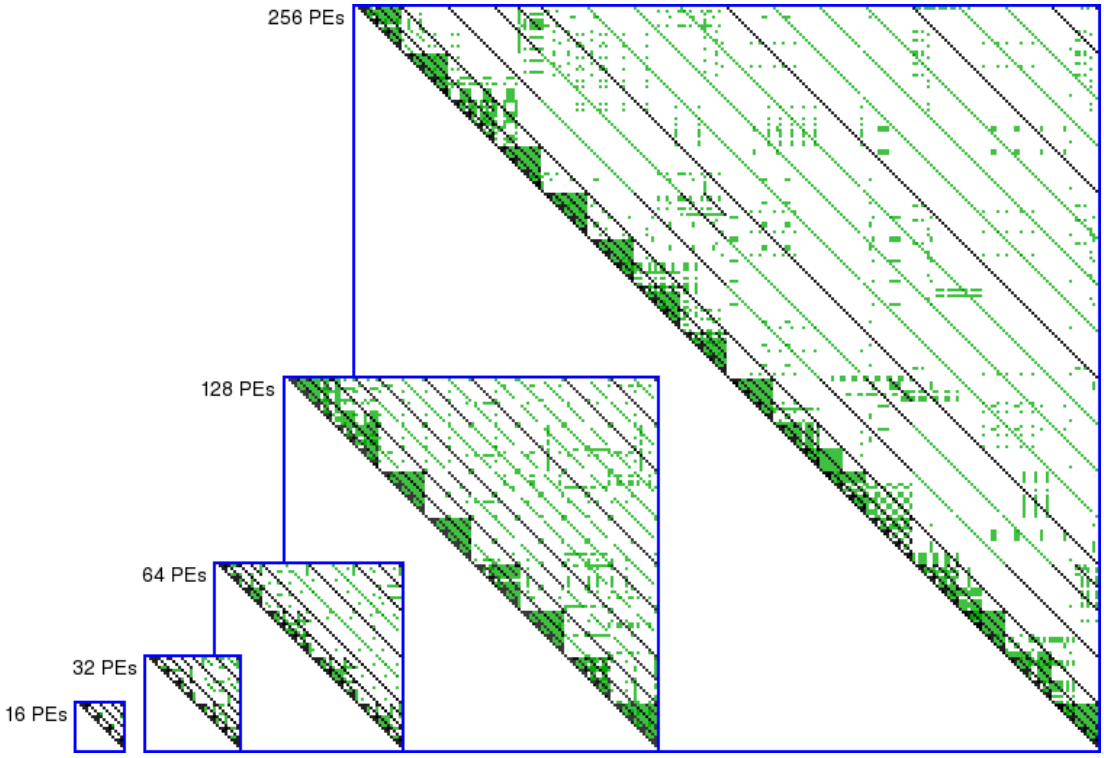

Figure 4.19: Solutions for Hypercube plus Single Torus with ± 1 offsets, $\eta = 3$

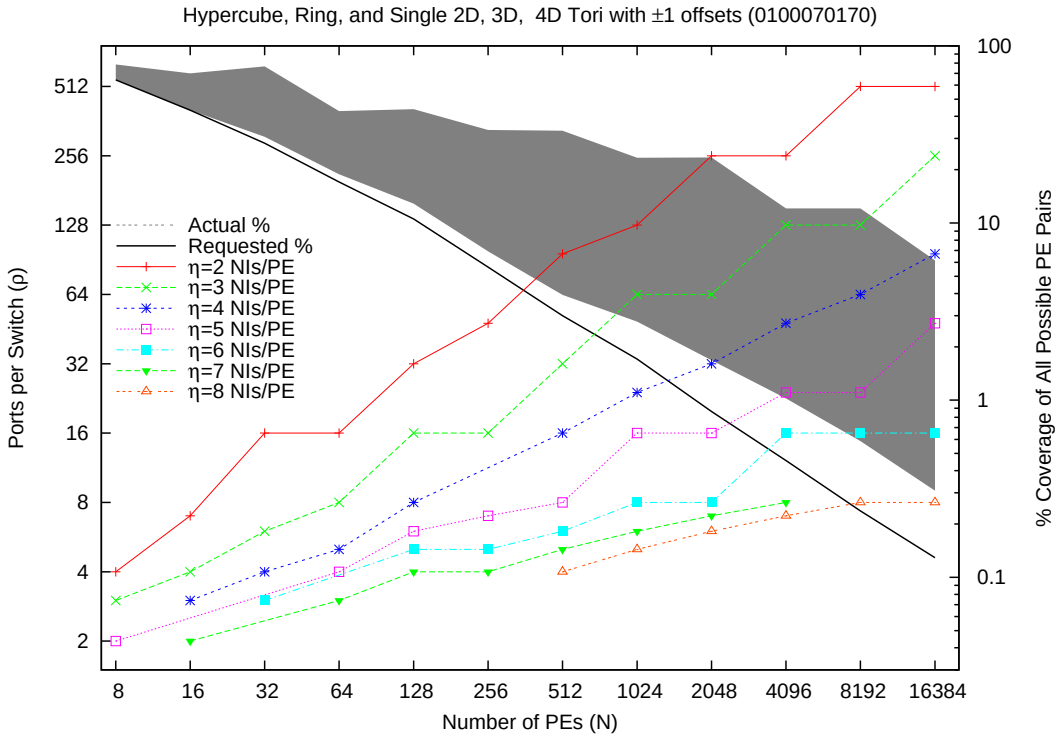

Figure 4.20: Scaling of Hypercube plus Single Torus with ± 1 offsets

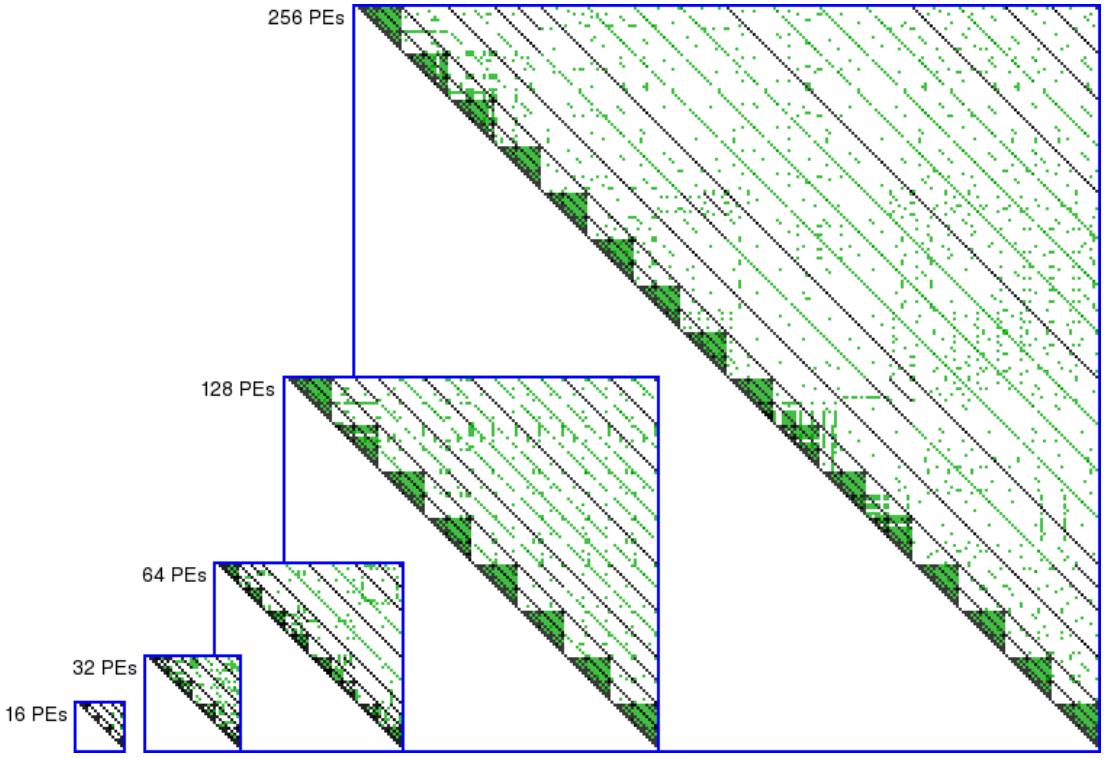

Figure 4.21: Solutions for Hypercube plus Single Torus with $\pm 2^k$ offsets, $\eta = 3$

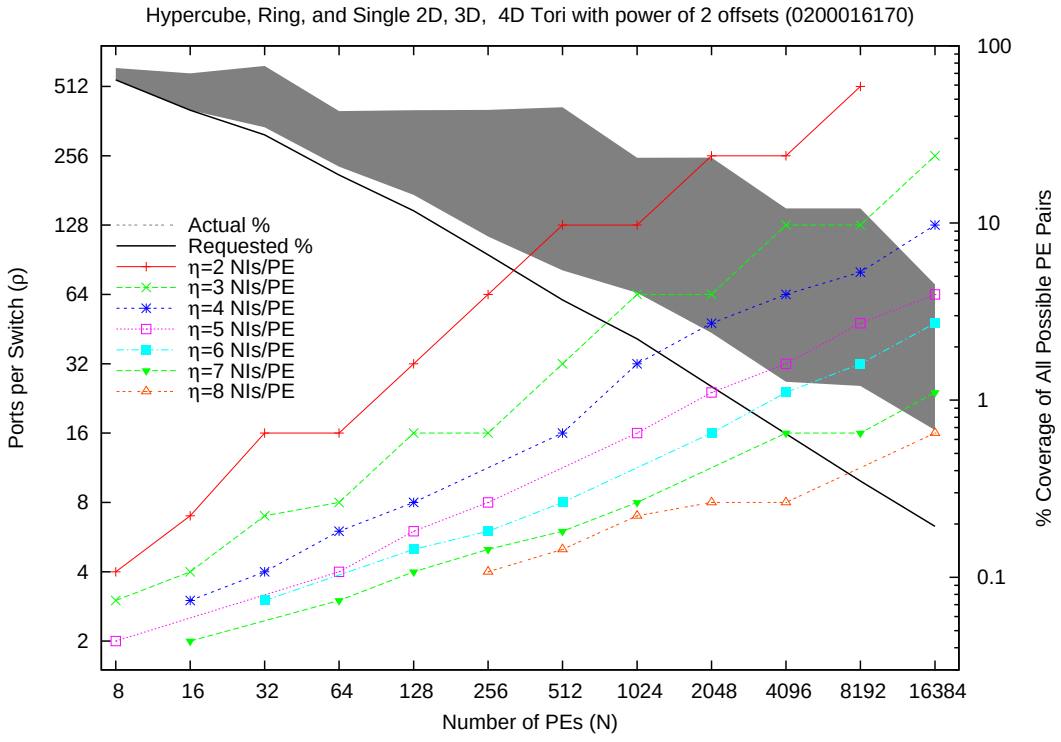

Figure 4.22: Scaling of Hypercube plus Single Torus with $\pm 2^k$ offsets

4.2.2 Hypercube plus Tori with Multiple Factorizations

Sample solutions and the scaling results for the following combination of communication patterns that include multiple factorizations for each torus sub-pattern are shown in Figure 4.23 and Figure4.24:

- Hypercube
- Ring with distance 1 offsets in X
- Multiple 2D torus with distance 1 offsets in X, or Y
- Multiple 3D torus with distance 1 offsets in X, Y, or Z
- Multiple 4D torus with distance 1 offsets in W, X, Y, or Z

For this combination of patterns, the 1024 PE with $\eta = 4$ case needed 32-port switches for the minimum solution found. The same pattern with the addition of the power of 2 offset neighbors for the 2D, 3D, and 4D torus sub-patterns has sample solutions shown in Figure 4.25 and scaling results shown in Figure 4.26. With this change in patterns, the 1024 PE with $\eta = 4$ case needed 48-port switches for the minimum solution found.

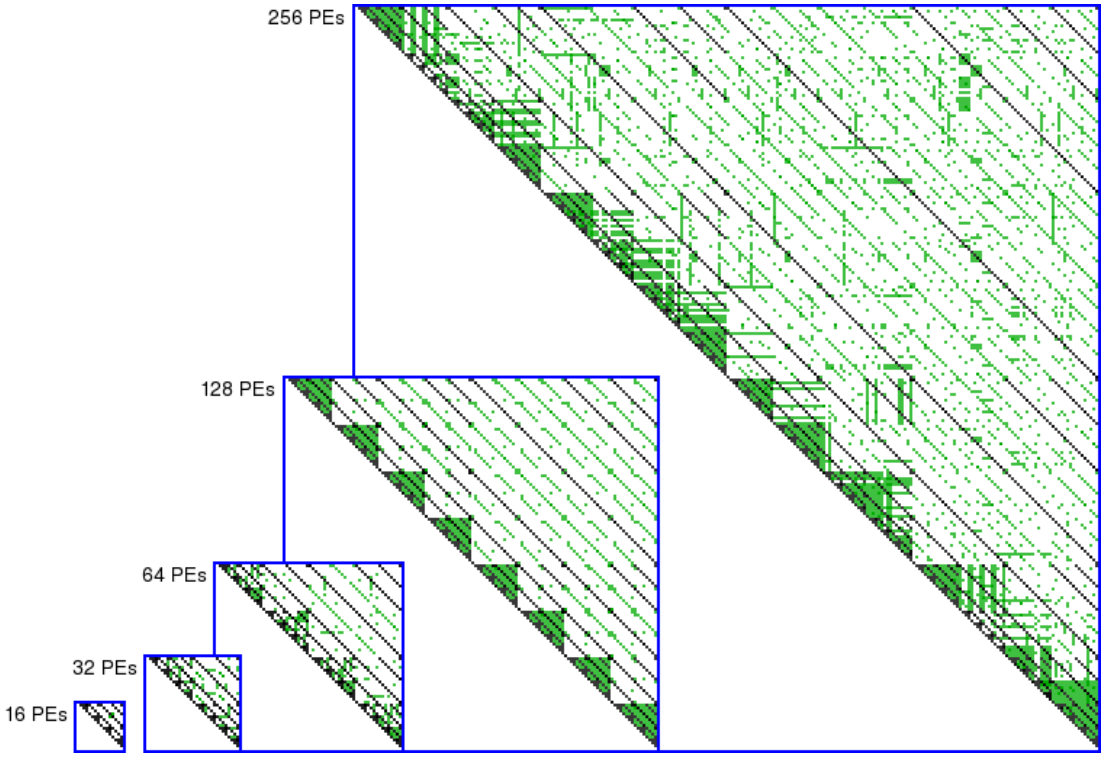

Figure 4.23: Solutions for Hypercube and Multiple Tori with ± 1 offsets, $\eta = 3$

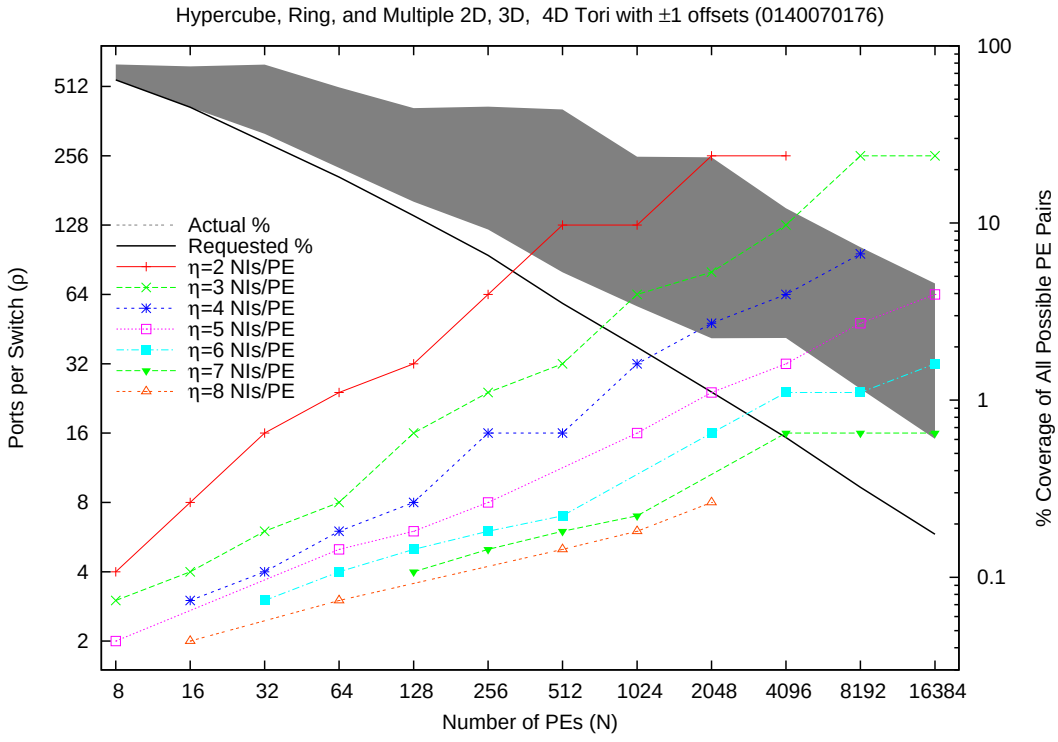

Figure 4.24: Scaling of Hypercube and Multiple Tori with ± 1 offsets

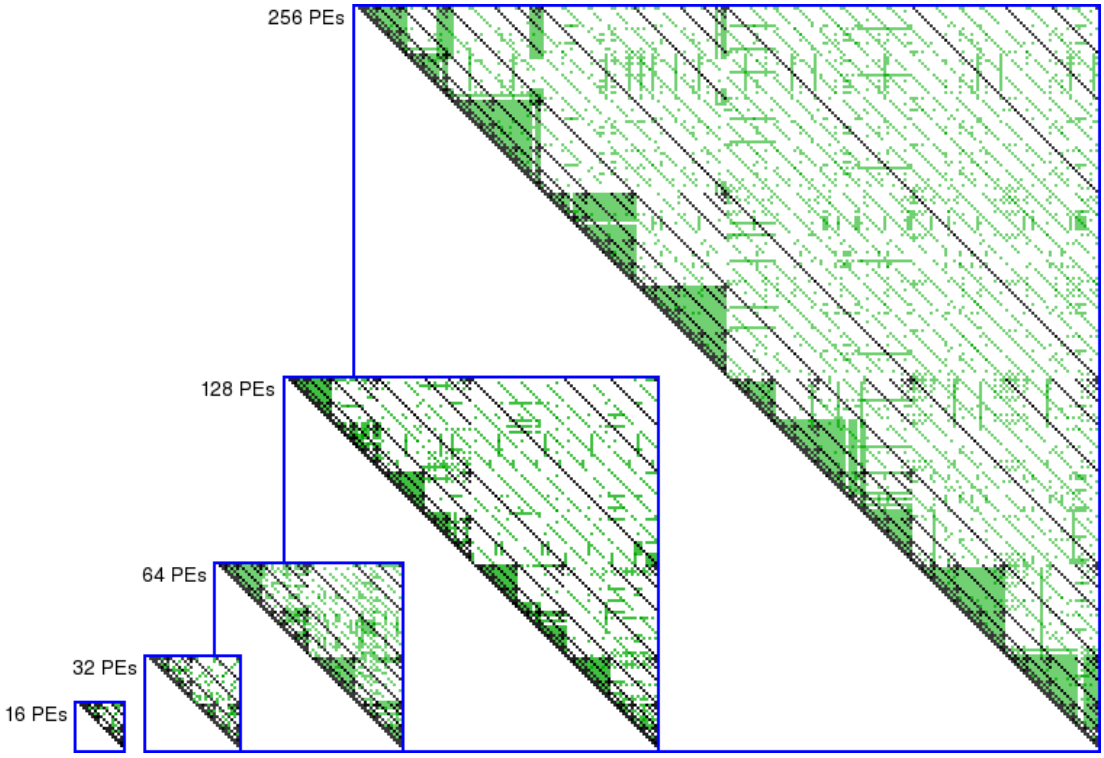

Figure 4.25: Solutions for Hypercube and Multiple Tori with $\pm 2^k$ offsets, $\eta = 3$

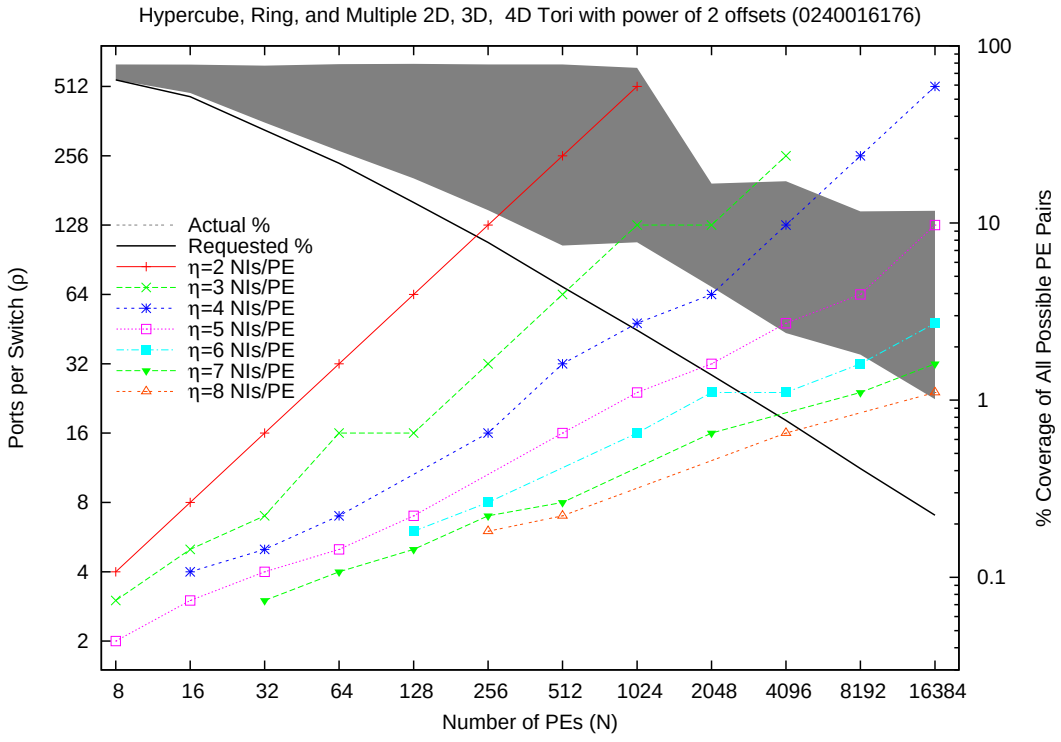

Figure 4.26: Scaling of Hypercube and Multiple Tori with $\pm 2^k$ offsets

4.2.3 Special Patterns plus Hypercube and Tori with Multiple Factorizations

If the preceding examples didn't cover the communication patterns of a target application, it is likely that application needs a pattern tailored specifically for it. As an example of two such patterns that are found in the literature are the bit-reversal pattern and the perfect shuffle and inverse-shuffle patterns, which were discussed in Section 2.3.1. Sample solutions and scaling results for the following combination of communication patterns that include multiple factorizations for each torus sub-pattern are shown in Figure 4.27 and Figure 4.28:

- Bit-reversal
- Perfect Shuffle and Inverse-Shuffle
- Hypercube
- Ring with distance 1 offsets in X
- Multiple 2D tori with distance 1 offsets in X, or Y
- Multiple 3D tori with distance 1 offsets in X, Y, or Z
- Multiple 4D tori with distance 1 offsets in W, X, Y, or Z

For this combination of patterns, the 1024 PE with $\eta = 4$ case needed 64-port switches for the minimum solution found. The same pattern with the addition of the power of 2 offset neighbors for the 2D, 3D, and 4D torus sub-patterns has sample solutions shown in Figure 4.29 and scaling results shown in Figure 4.30. With this change in patterns, the 1024 PE with $\eta = 4$ case needed 80-port switches for the minimum solution found.

As can be seen in the two scaling figures, it may be appropriate to consider an additional NI per PE rather than using larger switches. For example, the 1024 PE with $\eta = 5$ case in Figure 4.28 needed 32-port switches for the minimum solution found, which is comparable to the $\eta = 4$ case without the special purpose patterns. Similarly, the 1024 PE with $\eta = 5$ case in Figure 4.30 needed 48-port switches, which is also the same width switches as needed for the case without the special purpose patterns with $\eta = 4$.

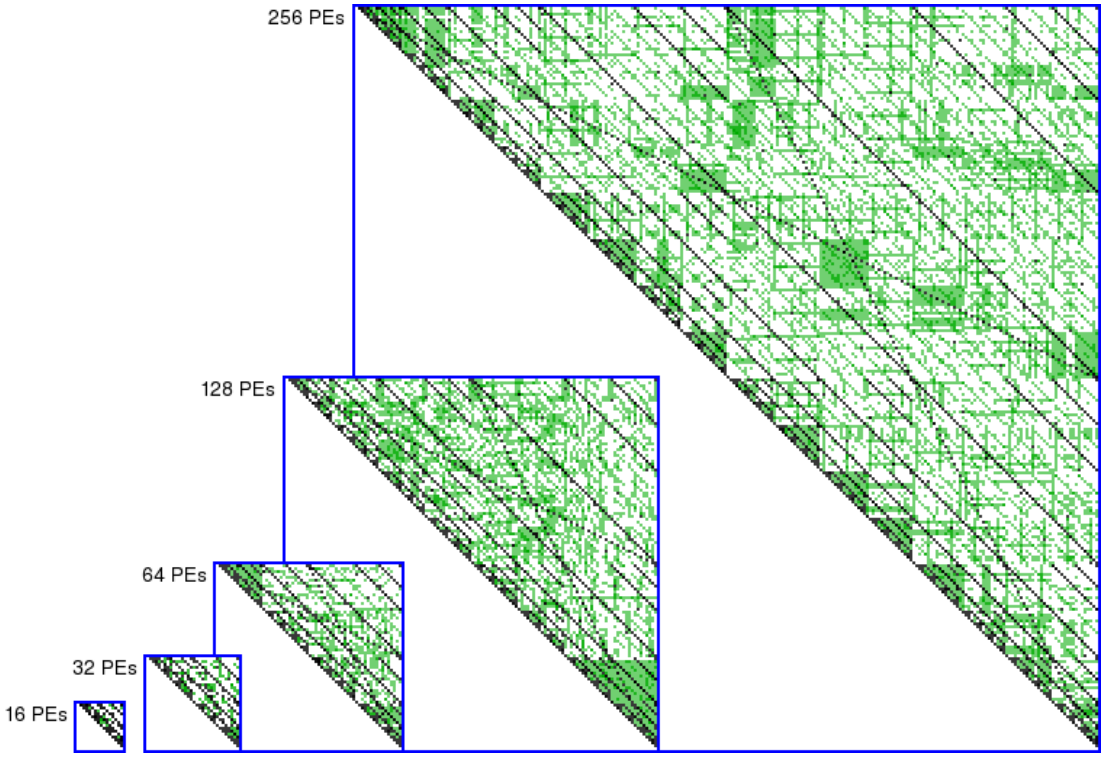

Figure 4.27: Solutions for Bit-reversal, Shuffle, Hypercube, and Tori with ± 1 offsets, $\eta = 3$

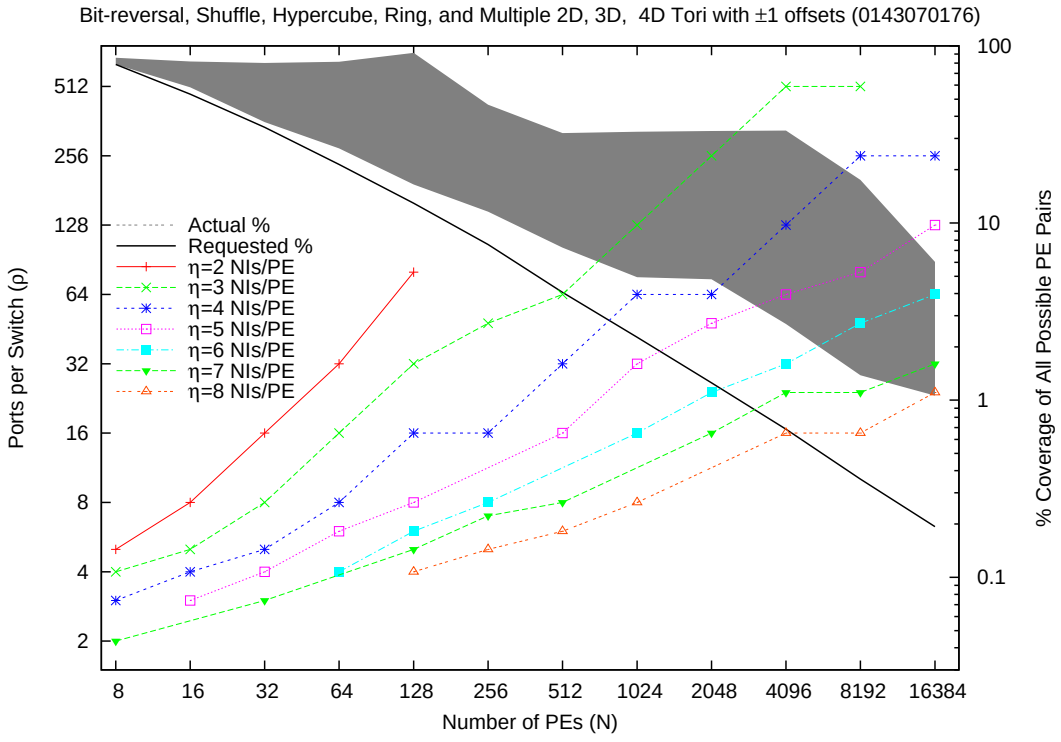

Figure 4.28: Scaling of Bit-reversal, Shuffle, Hypercube, and Tori with ± 1 offsets

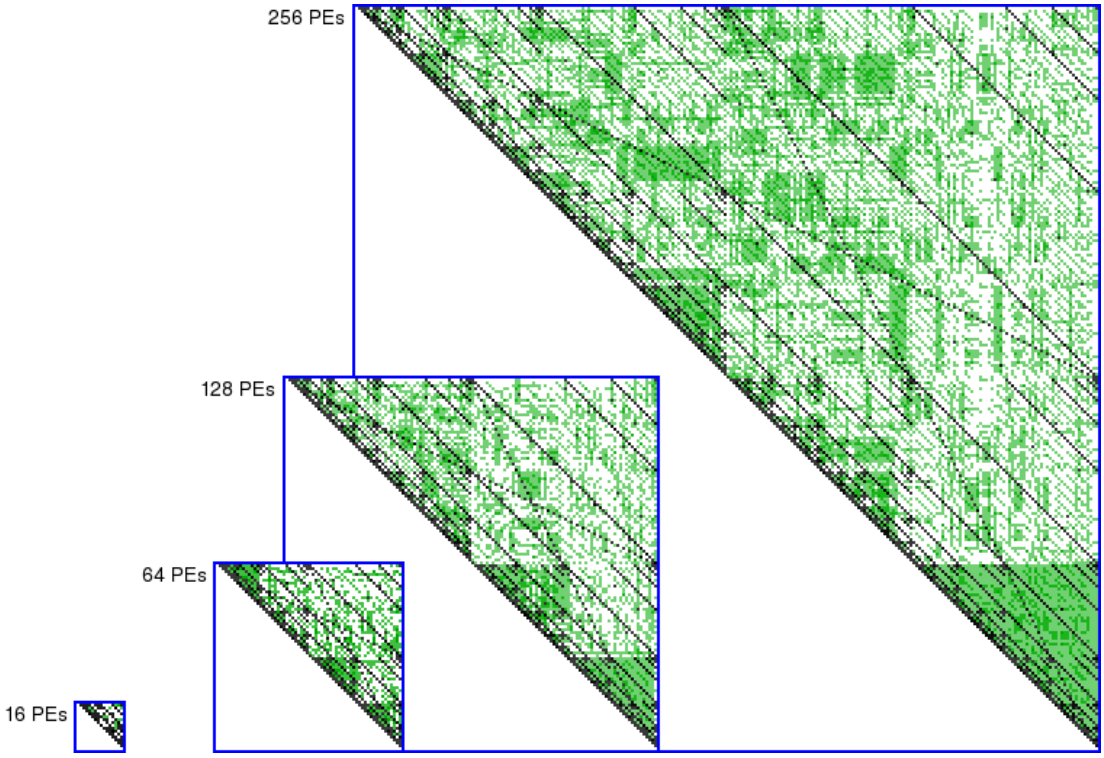

Figure 4.29: Solutions for Bit-reversal, Shuffle, Hypercube, and Tori with $\pm 2^k$ offsets, $\eta = 3$

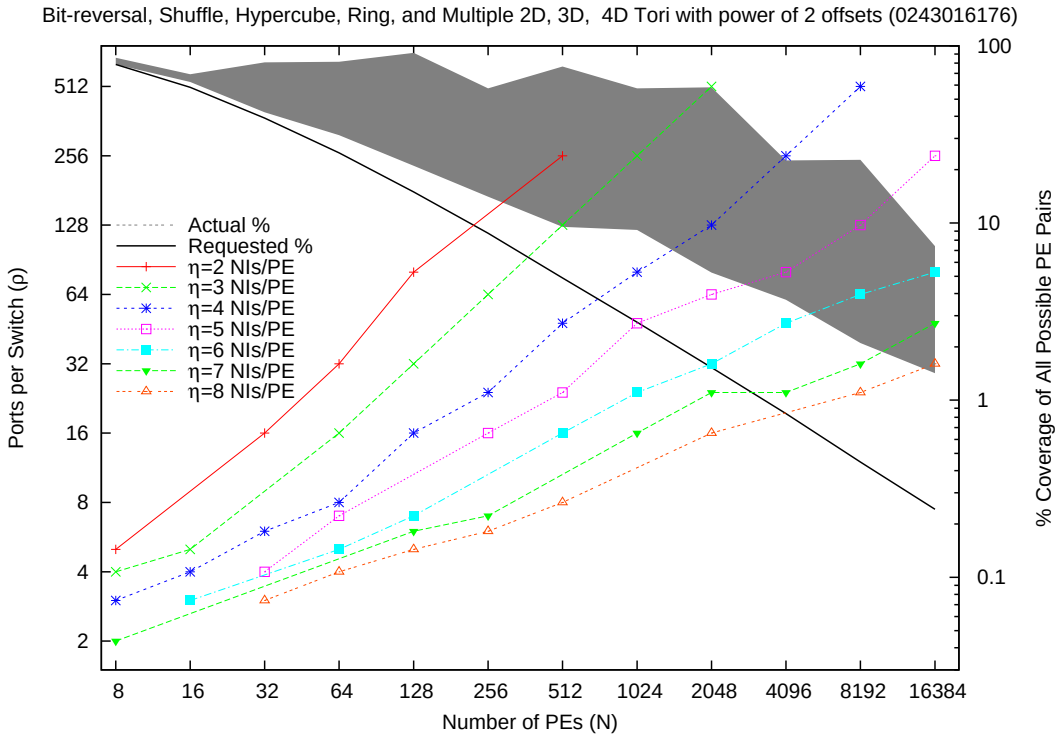

Figure 4.30: Scaling of Bit-reversal, Shuffle, Hypercube, and Tori with $\pm 2^k$ offsets

4.3 A Large Sparse FNN Example

Sparse FNN networks for very large systems, covering a wide range of communication patterns, can be built using commodity network hardware. The primary limitation in making larger designs appears to be simply the time and memory requirements of the Sparse FNN design tools. As the machines available to run the design tools become faster and contain more memory, even larger Sparse FNN designs can be found in a reasonable amount of time. The efficiency of the design tool programs have been dramatically improved since the initial Sparse FNN Heuristic was developed, enabling the design of dramatically larger Sparse FNNs.

The largest Sparse FNN designs we have thus far created contain 65,536 PEs. Not coincidentally, 65,536 is the same number of nodes that are in the BlueGene/L Supercomputer at the Lawrence Livermore National Laboratory. BlueGene/L is the fastest machine in the world today (March 2006) as measured by the HPL benchmark[64], achieving over 280 TFLOPS of performance; no other machine on the Top500 list[64] has close to as many nodes, so $N = 65,536$ seems a good upper bound on what people might be designing in the near future. Although the BlueGene/L's nodes contain two PEs each, the primary data network for BlueGene/L is a 65,536 element 3D torus factored as $64 \times 32 \times 32$ with the traditional six neighbors per node. As discussed in Section 4.1.3 and shown in Figure 4.12, designing a Sparse FNN that supports just a single 3D torus with six neighbors per PE is not interesting. Thus, the 65,536 PE example Sparse FNN design was selected to cover both a 3D torus with six neighbors per PE and the hypercube pattern. For comparison purposes, Figure 4.31 shows sample solutions of this pattern combination for several smaller machines. The scaling results for this combined pattern are shown in Figure 4.32 for 8 to 16,384 PEs[1].

To find solutions to this $N = 65,536$ problem, the parallel Sparse FNN GA described in Section 3.4 was run on KASY0 for 56 hours, with the search space limited to $5 \leq \eta \leq 6$ and $\rho \in \{24, 32, 48, 64, 96, 128\}$. These limits were selected based on the scaling results already found for this pattern as shown in Figure 4.32. During this run, the GA found a solution for each of the six switch sizes: solutions with $\eta = 5$ were found for $\rho \in \{48, 64, 96, 128\}$ and solutions with $\eta = 6$ were found for $\rho \in \{24, 32\}$. Solutions with $\eta = 5$ and $\rho \in \{64, 96, 128\}$ were found in the first six hours of the run. Over the next four hours, 15,343 individuals in the GA were evaluated as potential solutions with $\eta = 5$ and $\rho = 48$ before the program gave up and tried solving the problem with $\eta = 6$. Over the next seven hours of the run, the program found solutions with $\eta = 6$ for each of the remaining switch sizes, $\rho \in \{24, 32, 48\}$. The GA returned to the $\eta = 5$ and $\rho = 48$ problem, and after an additional 14 hours and 36,885 evaluated individuals[2], the GA found a solution with $S = 6,827$ switches. The remaining time of almost 25 hours was spent attempting to solve the problem for $\eta = 5$ and $\rho = 32$, in which 71,669 individuals were evaluated without finding a solution.

[1]The figure does not include the 65,536 PE designs so that comparisons to the other figures in this chapter would be easier because the axes are scaled identically.

[2]The time to evaluate an individual varies considerably depending on how early in the heuristic a failed design is detected. As the gene pool evolves towards designs that are closer to working, each individual takes longer to evaluate.

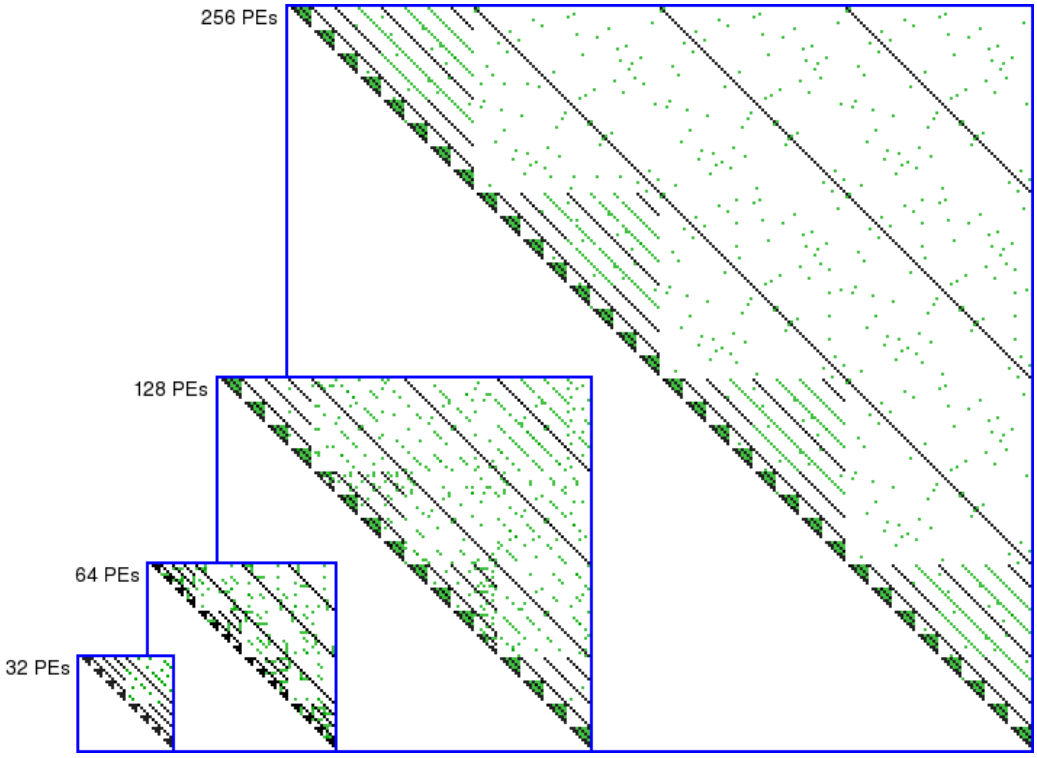

Figure 4.31: Solutions for Hypercube and Single 3D Torus with ± 1 offsets, $\eta = 3$

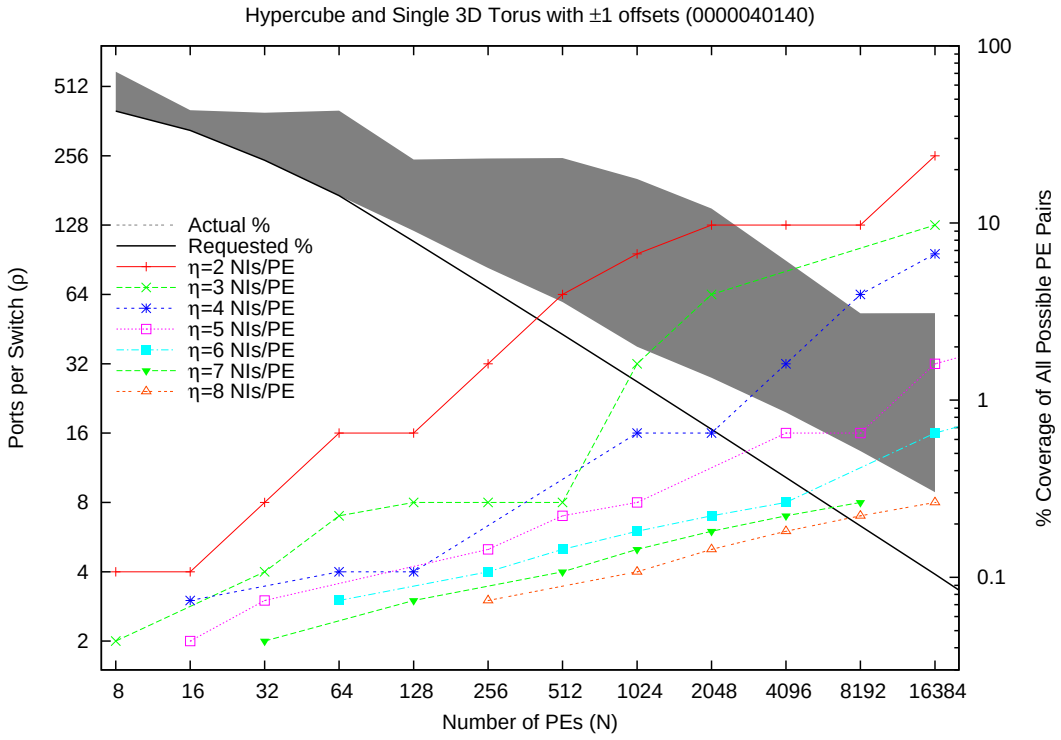

Figure 4.32: Scaling of Hypercube and Single 3D Torus with ± 1 offsets

Figure 4.33 shows a Design/Solution Map for the 65,536 PE solution with $\eta = 5$ and $\rho = 48$, much like the figures showing smaller sample Sparse FNN solutions throughout this chapter. In the upper right is a representation of the requested communication patterns, the Design, shown in black. Also in the upper right is a representation of the extra coverage achieved by the Sparse FNN solution, shown in green. The vertical axis goes from PE 0 at the top to PE 65535 at the bottom. Similarly, the horizontal axis goes from PE 0 on the left to PE 65535 on the right. Each coordinate in the figure represents the single-switch-hop connectivity of a pair of PEs. Because the links in the network are assumed to be bidirectional, the coverage if shown in the lower left would be a mirror image along the diagonal.

Due to the large scale of this example, it is impossible for the figure to show individual PE pair connections at any reasonable pixel resolution. So, each resolvable pixel in Figure 4.33 is actually a summation of the connectivity of a 64×64 PE region, representing the connectivity of 4,096 PE pairs. Near the upper left corner, a 64×64 pixel region is outlined by a blue box. That region is shown at almost full scale in Figure 4.34, which covers some of the connectivity for 4,096 PEs. This representative region is slightly off the diagonal axis to show more of the actual area representing coverage by the Sparse FNN design. In that figure, a blue box outlines yet another 64×64 pixel region, which is shown at full scale in Figure 4.35. This last figure shows a 256 PE section, where each connection can be distinctly seen.

For this large example the total number of possible PE pairs is 2,147,450,880. The requested communication pattern has 622,592 pairs – just 0.029% of all possible pairs! The actual design covers 7,529,833 pairs, or 0.351% of the possible pairs, including all the requested pairs. From one perspective, this coverage is 12 times more than requested, which may seem excessive. Yet, the solution has 285 times less coverage than a Universal FNN, and would thus be considerably less expensive to construct than a Universal FNN. More importantly, the example $\eta = 5$ solution would be less expensive than any of the $\eta = 6$ solutions. As of May 2006, the component cost for constructing the 5 NIs/PE solution using 48-port Gigabit Ethernet switches would be around $9.7 million. Although the 6 NIs/PE solution found using 24 port Gigabit Ethernet switches has a tighter coverage of only 0.21% of possible PE pairs, its component cost today would be around $10.2 million. Because the commodity prices of individual components change significantly over time, there is no general way of selecting the "best" Sparse FNN solution for a given problem.

Clearly, the Sparse FNN design technology presented in this dissertation is able to find interesting alternative network designs for machines at the largest scales currently being built. For this $N = 65,536$ problem, a weekend of runtime on a machine worth $40,000 to design a network that would cost on the order of $10 million is sufficient for an academic example. However, a much greater amount of computation time and power would be appropriate to apply towards finding a Sparse FNN design that would actually be implemented at these scales. It is not clear how much the example Sparse FNN design could be improved upon, but certainly the greater amounts of computational power that routinely will be able to be invested in designing larger machines is likely to result in even better designs.

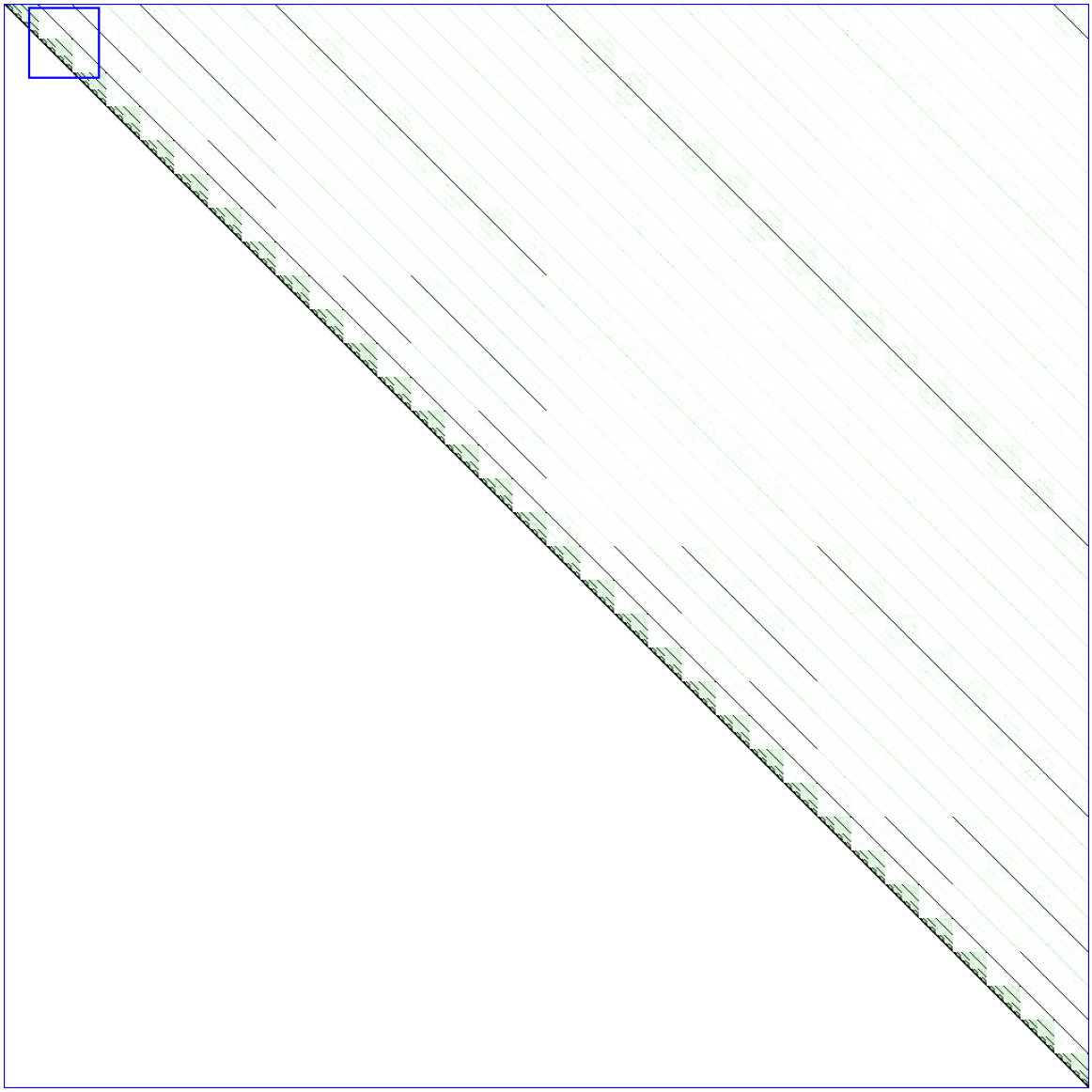

Figure 4.33: A scaled Design/Solution Map for $N = 65,536$, $\eta = 5$, and $\rho = 48$

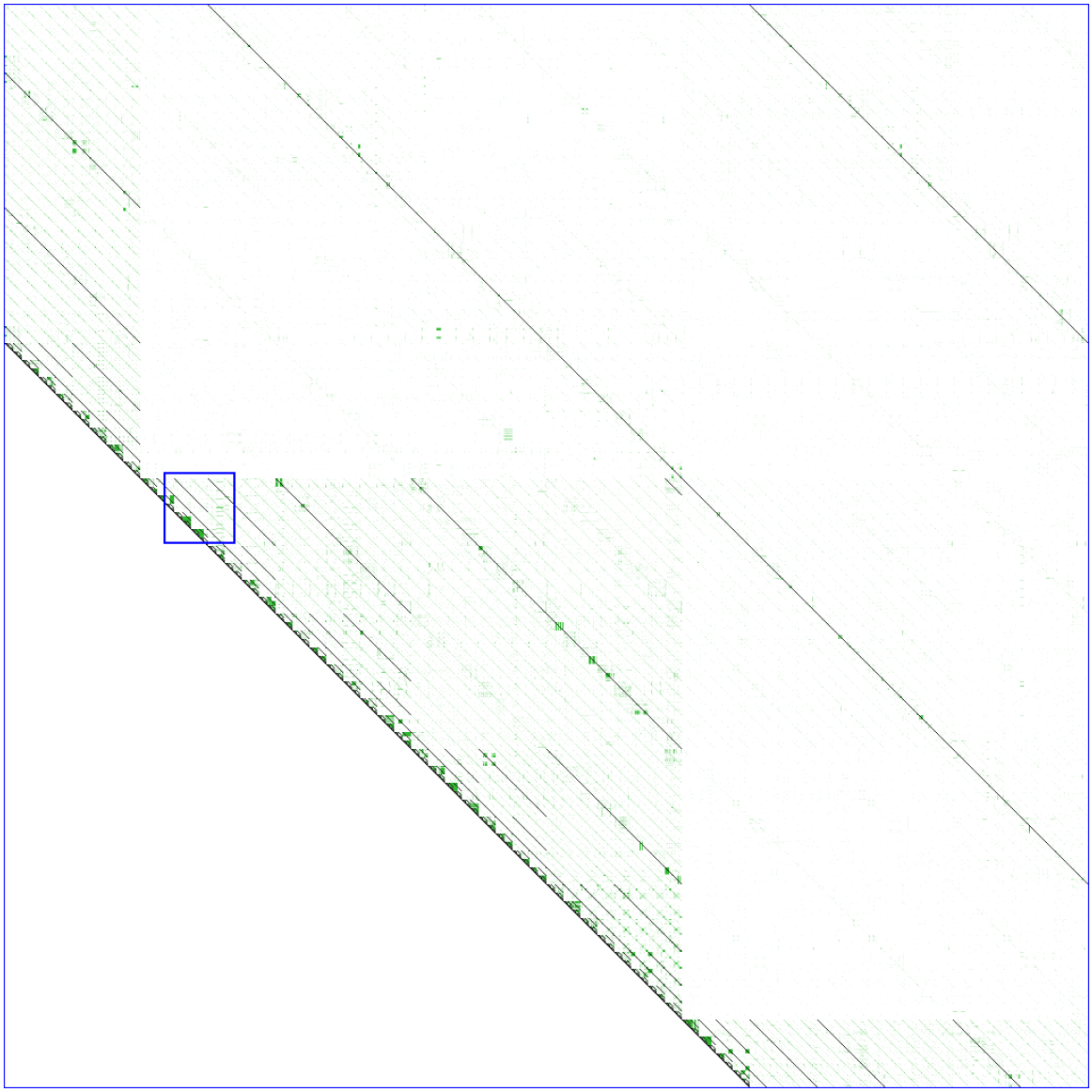

Figure 4.34: A representative 4,096-PE region of the $N = 65,536$ Design/Solution Map

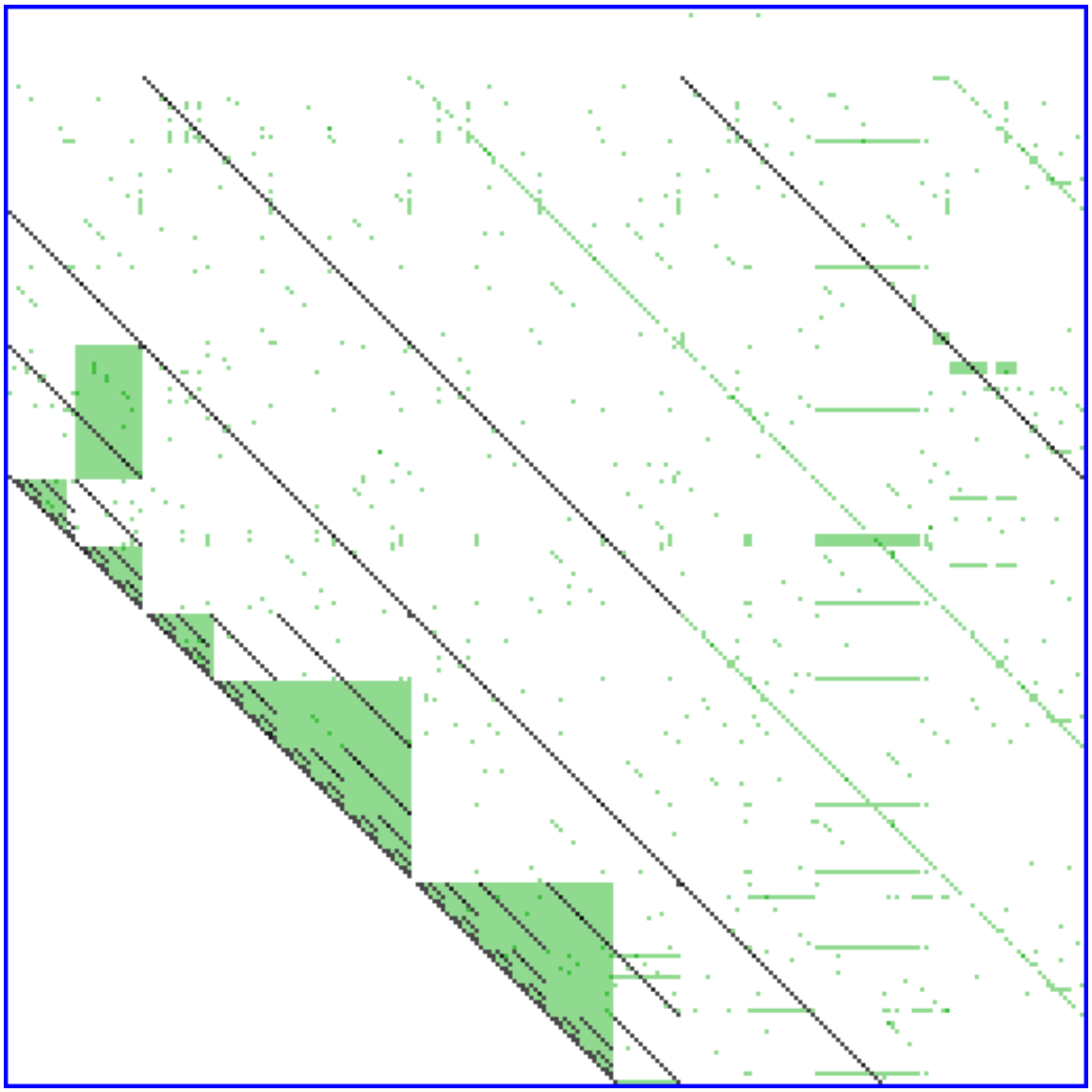

Figure 4.35: A representative 256-PE region of the $N = 65,536$ Design/Solution Map

Chapter 5

Message Routing and Practical Details for Implementing FNNs

In the preceding chapters we covered FNNs primarily from a perspective that was independent of the details of the underlying network hardware and protocols that one would use in a real FNN implementation. In this chapter, we cover issues that can not be so easily separated from these implementation details. For this work, we choose commodity Ethernet networking hardware and IP protocols for the target implementations, as well as the Linux OS as the base software platform. The primary detail that this chapter covers is: how do the parallel program's messages get routed in an FNN? Following that discussion are subsections on Linux runtime support, network booting of PEs, IP Multicast, and the potential for fault tolerance on FNNs. The final section discusses alternatives to using Ethernet to implement FNNs.

5.1 Point to Point Message Routing on FNNs

The two primary issues in routing point to point packets on an FNN are simply how to get from one PE to another within an arbitrary topology, and how to utilize the bandwidth available in a FNN when there are multiple paths between PE pairs. For Sparse FNNs the first issue must also deal with PE pairs that are not connected to a common switch.

In traditional IP networking, the task of routing a packet from one host to another is broken down into several steps. The first step is the selection or assignment of IP addresses to the hosts (PEs). Once the source and destination IP addresses are known, a routing table is consulted as the second step to determine if the destination is on a local network, or if the packet must be sent through a gateway host. Either way, the third step is to find the link-layer MAC address of the next host in the route, be it a gateway or the final destination of the packet. On Ethernet-like link layers, the Address Resolution Protocol (ARP) is used to obtain the next host's MAC address from its IP address. In the process of this third step, the outgoing network interface is also selected so that the packet can reach the next host directly. The fourth step is to actually transmit the packet, and then

if the packet is not yet at its final destination, the process is repeated from step two. By a sequence of these four steps, a message can be delivered from one host to another on any properly configured IP network.

It is constructive to view the first three steps using the terminology of tuples, where a 2-tuple is an ordered collection of two values, i.e. an ordered pair. For a FNN, each PE is assigned a sequential number, called a PE number, and each message has an associated 2-tuple of source and destination PE numbers. The first three steps of IP message routing described above can be viewed as translating tuples from one address space into another. Step one translates from a PE number into an IP address. Step two translates from a destination IP address, into a local-network IP address, e.g. the next "hop". Step three translates from the local-network IP address to the link layer MAC address. Thus, prior to step one, we have a 2-tuple of source and destination PE numbers, and after step three we have a 2-tuple of source and destination MAC addresses.

Thus ultimately, the problem to be solved for routing a point to point packet on a FNN is to translate from the initial PE number 2-tuple into an appropriate MAC address 2-tuple. The following subsections discuss several alternative ways of performing these tuple translations, with various trade-offs for performance, portability and ease of implementation.

5.1.1 IP Layer Technique for Routing Messages on FNNs

The first alternative is to fully leverage the traditional IP networking layer. Each NI in the FNN is assigned a unique IP address, such that NIs connected to the same switch have IP addresses that are members of an IP subnet dedicated to that switch. A small representative example of this method is shown in Figure 5.1. Given a destination IP address, the normal IP routing mechanisms can perform all the work to select the proper NI to egress from, and to obtain the destination NI's MAC address using the ARP system. Thus, effectively, the FNN routing software only needs to translate from a PE number tuple, to an IP address tuple.

This tuple translation is done using a hostname to IP address table. Each PE number is associated with a hostname, for example PE 42 would be assigned the hostname k42. Each PE would be given a custom "hosts" file that would store its PE number to IP address tuple translations. When a PE A first communicates with a neighboring PE B, it will perform a hostname lookup to get an IP address of PE B. This hostname lookup is not done per packet, and is usually done at most once per invocation of an application.

This technique has the advantage that a sequence of packets between a pair of PEs in a FNN should not be delivered out of order under normal operating conditions, because only a single path between a PE pair will be used. A disadvantage is that a maximum of a single unit of bandwidth will be used between any pair of PEs, even if there are multiple single-hop paths available between the PE pairs. Another disadvantage is that the ARP system won't necessarily get the proper[1] answer

[1]One may wonder how could the ARP system not get the proper MAC address for a given IP address, if each IP address is uniquely assigned to a single NI. However, for the applicable RFCs, it is not defined as to which object owns an IP address, be it a NI and/or the host that the NI belongs to. Thus, at least for Linux, the default choice was made that the host owns all the IP addresses of all its NIs so that in the typical ad-hoc networking setup, things would "just work".

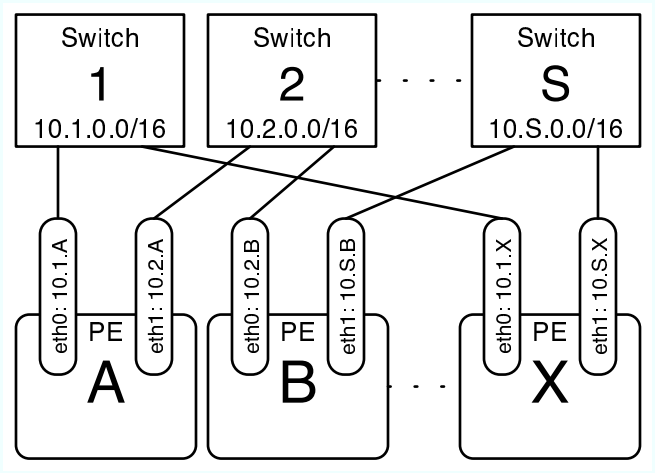

Figure 5.1: Multiple IP addresses per PE (one per NI)

if there are switch to switch connections in the FNN, such as when there are up-links to a top-level switch. This problem can be alleviated by pre-loading the ARP cache with permanent entries. This solution adds the burden of collecting and maintaining a database of all the MAC addresses used in the FNN, which would have to be updated each time a NI is replaced. Also, the Linux in-kernel ARP cache is limited to 256 entries, which limits the size of the cluster that can effectively use this technique to fewer than 256 PEs. This 256 entry limit could be changed by modifying the Linux kernel source.

The primary disadvantage of this IP-layer technique is due to a general assumption found in many High Performance Computing (HPC) software packages. There is an assumed one-to-one and onto mapping of IP addresses and PE numbers. In other words, the assumption is that PE A and PE B would use the same IP address to refer to PE X, and thus it is safe to exchange IP addresses between PEs to be used as their identifiers. With this IP layer FNN routing technique, that assumption is not true. In the example shown in Figure 5.1, one can see that PE A would use IP address 10.1.X to talk to PE X, while PE B would use 10.S.X instead. For this scheme to work, each PE must perform its own hostname/PE number to IP address translation. A custom "hosts" file per PE can be used to store these tuple translations. The LAM/MPI software layer was modified by the author to support this context-sensitive addressing. The patches were fully incorporated into the main LAM/MPI software distribution by July 2001 in the 6.5 version series. An investigation of what would be needed to similarly make Parallel Virtual Machine (PVM) support this technique was performed by a colleague, but the level of effort required was deemed excessive.

Unfortunately for FNNs with switch to switch connections, this choice means that when an ARP broadcast packet is sent asking for the MAC address that can reach a particular IP address, a Linux based PE with default configurations would respond with several different answers, one for each NI that the ARP packet arrived on. The requesting PE might not choose the ARP response that results in the shortest path between the two PEs.

5.1.2 ARP Cache Technique for Routing Messages on FNNs

Another alternative is to actively rely on preloaded ARP cache entries on each node to perform the IP to MAC address translations. Each PE would be assigned a single IP address, and although each NI in a particular PE would have the same IP address, they would still have unique MAC addresses. In Linux, a neighbor cache entry includes the egress NI name in addition to the destination MAC address. This technique solves the problem of multi-homed PEs with respect to the HPC software. However, this method still has the problem of collecting and maintaining a database of all the MAC addresses used in the FNN, which would have to be updated each time a NI is replaced. Also, again, this technique is limited to 256 PEs due to the ARP cache size limits in the Linux kernel. And, this approach does not take advantage of the bandwidth available between PE pairs that have multiple single switch paths in the FNN.

5.1.3 Link Layer Technique for Routing Messages on FNNs

A third alternative method for tuple translation from PE numbers to MAC addresses is to push the entire task down into the link layer of the network stack. Effectively, the PE numbers are directly used as IP addresses, and the entire FNN appears as a single IP subnet to the network stack as shown in Figure 5.2. In addition, the NIs in the FNN are assigned custom MAC addresses that are derived from the PE number and the device name/number, e.g. eth0, eth1, etc. In this method, the IP software layer communicates through a virtual network device shown as `bond0` in the figure. The ARP system is replaced by a FNN link-layer routing algorithm that translates directly from PE number tuples to these custom MAC address tuples. If there are multiple single switch paths between a PE pair, on a per packet basis, alternating MAC addresses can be selected by the FNN link-layer routing algorithm. In a sense, this technique is very similar to the ARP cache technique described above, but instead of fixed one-to-one and onto translations, this technique is able to utilize the bandwidth of multiple single switch paths between PE pairs. This method also avoids the other disadvantages of the previous methods. Our link-layer routing algorithm can be written without a 256 PE limit, and can avoid having to maintain a database of MAC addresses for all the NIs in the machine, because the MAC addresses are predominantly computed values.

The primary disadvantage of this method is the difficulty in implementing the virtual network device software which would have to also replace the ARP system functionality. The standard Linux channel bonding module was not capable of accomplishing this technique without extensive modification. The channel bonding module works as a virtual networking device, e.g. `bond0`, with which the upper layers of the networking stack communicate. IP packets are given to the `bond0` device, and then based on the bonding mode, the `bond0` device selects which “enslaved” NI will actually send the packet. This egress selection phase would need to be changed for a FNN, because which NIs are valid choices are dependent on the FNN wiring pattern and on the destination PE. On the receiving side, the Linux channel bonding code in the networking stack makes all the packets that arrive via an “enslaved” NI appear to come from the “master” device, e.g. `bond0`. In most

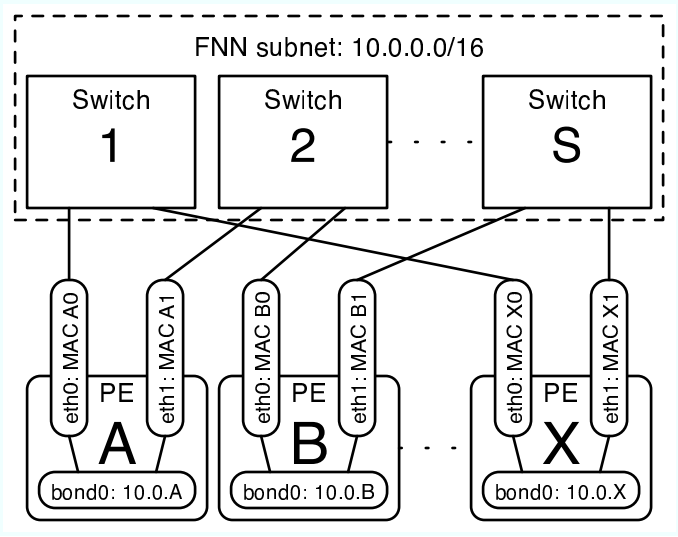

Figure 5.2: Single IP address per PE

Algorithm 11 C code to compute a NI's custom MAC address

```
MACaddrByte[0] = 2; /* Set the locally administered bit */
MACaddrByte[1] = nodeID >> 24;
MACaddrByte[2] = nodeID >> 16;
byte = MACaddrByte[3] = nodeID >> 8;
byte += MACaddrByte[4] = nodeID;
byte ^= (NI_Number & 0x7) << 5;
MACaddrByte[5] = byte;
```

bonding modes, the enslaved NIs on a PE are assigned a common MAC address, obtained from the first NI added to the bond. This cloning of MAC addresses allows the bond module to use the standard ARP system to translate from IP addresses to MAC addresses[2]. However, for a FNN each NI needs to have a unique MAC address, and thus the ARP system functionality would need to be replaced, as well as replacing the MAC address cloning feature of the bonding module with code to do our custom MAC address assignments. To initially implement this method, the author added a new mode called `fnn-routing` to the Linux channel bonding module. This new mode accomplished three primary things. First, when NIs are enslaved to the bonding module, they are assigned computed MAC addresses rather than a cloned MAC address. Second, the egress device selection phase was replaced by a FNN routing table lookup. Lastly, a Linux kernel `/proc` interface was added so that this routing table could be populated from a user space program.

The C code snippet in Algorithm 11 shows how the MAC addresses are computed, where

[2]This scheme of cloning of MAC addresses introduced its own set of problems that have plagued naive Beowulf cluster implementors for years. Basically, unless the network switches are properly configured to handle cloned MAC addresses, or the network is wired such that no individual switch could ever notice the cloned MAC addresses, things won't work well at all.

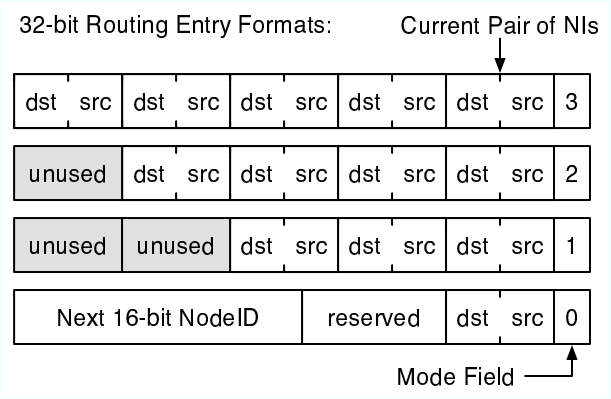

Figure 5.3: Routing Table Entry Formats

`nodeID` is the 32-bit IPv4 address of a PE, with the nominally 16-bit PE number as its least significant bits. The format of these computed MAC addresses was selected to meet two key requirements. First the computed MAC address must not conflict with any other MAC addresses that might be reachable on the same broadcast domain. This requirement is satisfied by setting the locally administered flag bit in each MAC, so that they do not conflict with any vendor supplied MAC addresses. Because the FNN is presumed to be isolated from any campus or corporate network by at least a router or head node, these locally administered MAC address would also not conflict with any other special MAC addresses that might be used by the local IT/Networking staff/department. Second, the MAC addresses must work well with commodity Ethernet switches. Commodity Ethernet network switches tend to use hash tables to maintain their MAC address to switch port mappings. We discovered that the hash functions in a variety of commodity Ethernet switches do not use the entire 48 bits of the MAC address. To avoid overflowing hash buckets in the switches, it was found to be sufficient for packets flowing through an individual switch to differ in the least significant eight bits of the MAC address. Thus an encoding was chosen that causes the last byte of the MAC address to be potentially different if either the PE number or NI number is different between two destinations. With a design target of supporting 2^{16} PEs, it is impossible to guarantee that the least significant byte of a computed MAC address is unique across the entire FNN.

The routing table within each PE consists of a 32-bit entry for each PE in the FNN, as shown in Figure 5.3. Nominally, this entry contains up to five 6-bit fields, followed by a mode field in the least significant two bits. Each 6-bit field is a 2-tuple of source and destination NI numbers. The mode field indicates if there are 5, 4, 3, or only 1 valid 6-bit fields. The case with only one valid 6-bit field allows room for the upper 16 bits of the entry to contain the PE number of an intermediary PE that must be used to reach the destination when using through-routing as discussed in Section 5.1.4. Thus, the destination MAC is reconstructed from the last 6-bit field, plus the PE number of the destination, or the PE number of the specified intermediary node. The source NI 3-bit number is used to select the outgoing NI, as well as the source MAC address. To facilitate load balancing over multiple NIs, the 6-bit fields are shifted down a position, and the used 6-bit field is placed at

the beginning of the list. This encoding scheme can accommodate 1, 2, 3, 4, or 5 NIs in a sequence, where the two-NI case uses four 6-bit entries, two for each NI used. The single NI case can either use the through-routing mode, or any of the other modes, with the single 6-bit entry replicated as many times as needed.

This encoding scheme also allows the designer to take advantage of imbalanced NI speeds on the source and destination, such as a server with a Gigabit Ethernet NI, and a regular PE with only 100 Mbit/s NIs. On the server side, the same Gigabit Ethernet can be selected to send packets to up to 5 different NIs on the client. And the client can send out packets through up to 5 NIs while selecting the server's single Gigabit Ethernet NI. Many different situations can be handled, all from how the individual routing entries are constructed. If more than 8 NIs are used per PE in a FNN ($\eta > 8$), this encoding scheme would need to be expanded to use more than 6 bits per NI pair. Also, if there are more than 65,536 PEs, the through-routing encoding would need to be modified to accommodate more than 16 bits for the intermediary PE number. This (temporary) restriction is consistent with the largest machine size explored to date (in Section 4.3).

5.1.4 Through Routing for Sparse FNNs

For Sparse FNNs, a given PE pair might not have a connection to a common switch. To send messages between these non-neighboring PEs, a route through one or more intermediary PEs should be selected. Ideally, the selected route should be the shortest possible, and thus involve the fewest intermediary PEs. Also, to maximize performance of these non-neighboring communications, it is desirable for two simultaneously active through-routed paths to not share any intermediary PEs. Unfortunately it is rather difficult to guarantee that no intermediary PEs are shared. One obvious approach to reduce the chance of conflicting paths would be to assign intermediary PEs evenly across all the through-routed paths.

At boot time, each PE selects its routes to other PEs based on its position in the graph representing the Sparse FNN. Specifically, the PE executes a a breadth first search on the graph to construct a breadth first tree (BFT) rooted at the current PE that reaches all the non-neighbor PEs. Then, a pass for each non-neighbor PE is performed where the BFT is incrementally rebalanced to spread out the load on the intermediary PEs. This rebalancing is done by finding all the possible first intermediary PEs, and then, using a pseudo random number generator, select one of the candidate PEs to be the parent in the BFT for the non-neighbor PE. The pseudo random number generator is a simple equation set up so that, for a PE pair that needs only one intermediary PE, the same intermediary PE will be picked for both communication directions. For PE pairs that are further apart, no effort is made to guarantee the same path is used in both directions.

Once these paths are selected, the current PE's routing table is filled such that for each non-neighbor PE, the first intermediary PE on the path to the destination is recorded. In effect, these intermediary PEs will then act as gateways that are closer to the destination. This scheme would work with any of the tuple translation techniques discussed above.

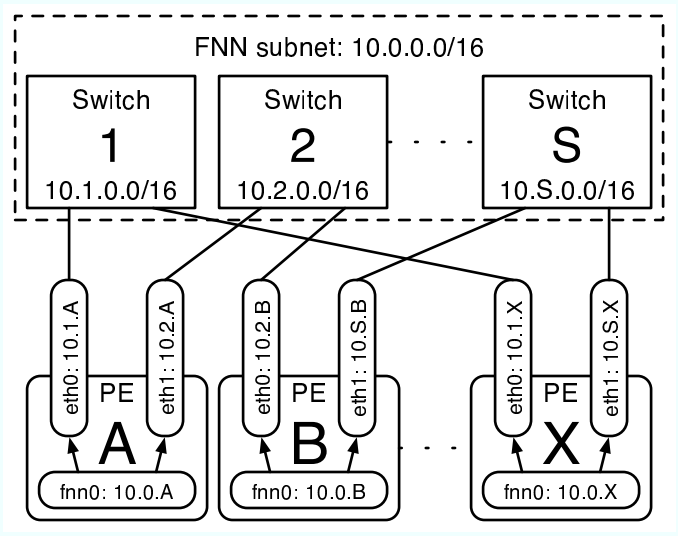

Figure 5.4: Combined scheme with an IP address per PE and an IP address per NI

5.2 FNN Runtime Support for Linux

The FNN runtime support for Linux consists of a loadable kernel module, a user space FNN router program, and a data file that describes the specific FNN wiring pattern, along with appropriate scripts to configure the FNN at PE boot time. This implementation uses a combination of the message routing techniques described in Sections 5.1.1, 5.1.3, and 5.1.4. The technique employed assigns a unique IP address to each PE, and also assigns a unique IP address to each NI in each PE such that there is a unique IP subnet for each switch[3] in the FNN, as well as an IP subnet for the entire FNN, as shown in Figure 5.4.

This technique is accomplished with an improved implementation of the link layer routing described in Section 5.1.3 that does not use the Linux channel bonding module. Instead, the author developed a standalone kernel module that implements a "send only" virtual network device, `fnn0`. Although based on the original code for the `fnn-routing` bonding mode, this module does not enslave any Ethernet devices, though it does still assign computed MAC addresses to the NIs used in the FNN. It also forwards packets to the appropriate Ethernet devices based on the information in the FNN routing table, as described previously. One of the primary effects of not enslaving the Ethernet devices is that when a packet arrives at a PE, the networking stack does not make it appear to have arrived via that PE's `fnn0` device. Other than firewall rules and the ARP system, the Linux IP network stack ignores the information about which network device the packets arrived on. Because the PEs in a parallel machine are not likely to be running a software firewall, and because the ARP

[3]For large FNNs with more than 255 switches, the IP addresses assigned to the individual NIs and the subnets assigned to each switch would need to be constructed in a different manner than shown in the figure. Because the IP address of each individual NI is only used in special situations, the specifics of the encoding scheme is not important, as long as each switch is assigned a unique subnet, and each NI attached to that switch is assigned a unique IP address within that subnet.

system was intentionally bypassed for this link-layer routing approach, the fact that packets never arrive via the `fnn0` device is not noticed by any IP layer software.

This runtime support that combines the link layer and IP layer routing techniques helps solve two complications of using FNNs in real systems. The first complication is how to support network booting of PEs within a FNN. The second complication is how to deal with broadcast and multicast protocols within a FNN. These complications and their solutions are discussed in the following two subsections.

5.2.1 Techniques for Initial PE Identification when a PE Network-Boots on a FNN

It is common practice to build cluster computers with diskless PEs, where the PE obtains its OS from a server via a network-booting scheme. During the initial phases of the network-boot of a PE, the software/firmware running on the booting PE will not be aware of the FNN. In other words, the FNN runtime support software just described is not encoded in the network boot firmware of a PE. Thus, the other PEs and/or servers in the FNN must be able to respond appropriately to some kinds of non-FNN traffic. There are many schemes for booting a computer over a network, with the Pre-boot eXecution Environment (PXE) boot ROM as the most common method. The network boot ROM in the client contacts a boot server using either the Dynamic Host Configuration Protocol (DHCP) or the Bootstrap Protocol (BOOTP), and in so doing also uses the Address Resolution Protocol (ARP). Thus, the FNN runtime software must support ARP, DHCP, and BOOTP in some way.

There are three scenarios that need to be dealt with to support these protocols on a FNN. The first scenario is the easy case where each PE's boot NI is connected to a broadcast domain that reaches the boot server directly through a single NI. The server simply needs to respond to the requests via that same NI. The second scenario is the case where the boot server has multiple NIs connected to the broadcast domain on which PE's send their boot requests. In this case, the server must select a single NI to use when responding. The third scenario is the case where the boot server does not share a broadcast domain with the PE that attempts to network boot. In this case, another PE must intercept the request and forward it to the boot server.

We solve this network boot problem by assigning each NI in the FNN a unique IP address, separate from the PE's `fnn0` IP address. These NI-specific IP addresses allow normal non-FNN protocols to work through individual NIs, totally oblivious to the `fnn0` virtual network device. This technique directly solves the problem for the first two scenarios, when a server's DHCP (Dynamic Host Configuration Protocol) daemon is configured to listen on each of its individual NIs that are reachable by each network boot capable NI of the PEs. To solve the third scenario, it should be sufficient to set up designated PEs in the FNN as DHCP/BOOTP proxy servers, and then guarantee that these proxy PEs are booted prior to any of their client PEs. Selection of these proxy PEs could be done similarly to the selection of intermediary PEs via the through routing scheme described in Section 5.1.4, with the primary boot server acting as the head of the breadth first tree.

5.2.2 Options for IP Multicast and Ethernet Broadcast Support

Because the combined technique for FNN routing described above has unique IP addresses per NI, and each PE could act as an IP Multicast router, any of a multitude of dynamic IP Multicast routing techniques found in the networking literature could be utilized. Specifically, the Protocol Independent Multicast (PIM) Dense mode routing scheme should be sufficient to enable the use of IP Multicast within the FNN. If IP Multicast on a FNN is performance critical, static IP Multicast routes could be configured to reduce the overhead of dynamically configuring routes.

The author briefly investigated how to support Ethernet broadcast across an entire Sparse FNN without a common physical broadcast domain. Because the FNN runtime support uses the IP address of outgoing packets to index into the FNN routing table, it was not going to be a simple matter to directly support routing Ethernet packets that did not contain an IP header. Although it was expected that this problem would need to be solved to allow the BOOTP protocol to work for network booting the PEs, as discussed above, Ethernet broadcast support was ultimately not required. Thus the primary use and motivation for supporting Ethernet broadcasts across the entire Sparse FNN was eliminated. The one known network protocol that remains which would benefit from support for Ethernet broadcast would be UDP broadcast packets. Because UDP broadcast packets are IP packets, it should be easier to augment the current FNN runtime support to handle UDP broadcast packets directly, as described below.

The most promising approach for UDP broadcast support would be to use a minimum spanning tree (MST) of broadcast domains that covers the entire FNN. Each node in this MST would be an Ethernet broadcast domain including all the PEs reachable by a single physical broadcast. These MST nodes would be connected by edges labeled with the set of PEs that are common between a pair of broadcast domains. For each edge in this MST, a single gateway PE would be deterministically selected for forwarding packets between the two broadcast domains. With a deterministic method for constructing this MST and for selecting the gateway PEs, each PE in the Sparse FNN would on its own be able to arrive at the same MST based on the FNN wiring table. Because the UDP broadcast packet would contain a source IP address that included the source PE number, each gateway PE receiving the UDP packet could determine where in the MST the packet came from, and thus whether that gateway PE needed to rebroadcast a copy of the packet out one or more of its links. This way the UDP broadcast packets would be routed without encountering any loops, by simply following all outgoing edges of the MST until it reached the leaves of the tree. The one remaining question is what should the initiating PE do with the UDP broadcast packet in the first place. The PE should not broadcast the packet out all of its NIs because doing so could cause the packet to simultaneously arrive at multiple nodes in the MST, which would cause the packet to be sent throughout the MST multiple times. Instead, if the initiating PE is not a gateway PE of the MST, it should deterministically select one NI to use to broadcast the packet, thus selecting a single node in the MST for the broadcast to start from. If the initiating PE is a gateway PE, then it should send the packet to both of the broadcast domains that the gateway connects in the MST.

5.2.3 Overhead of the FNN Runtime software

When a PE in a FNN is ready to send a packet to another PE, the packet is handed over to the virtual network device `fnn0` before it leaves the machine. On our KASY0 cluster, which is discussed in Section 6.2, we ran experiments to determine what level of overhead was added by the `fnn0` device. For each outgoing packet, we measured how much latency can be attributed to the FNN runtime software. The CPU performance register counters were used to get timing results within a few tens of CPU clock cycles. When a PE sent one packet each to each PE of KASY0, the overhead per packet was 2,324 CPU clock cycles or about 1,120 ns when averaged over a thousand rounds. For repeated packet sends to the same destination from one PE, the overhead measured on KASY0 was about 210 CPU clock cycles or about 100 ns when averaged over a thousand packets. The much smaller overhead for the repeated sends is clearly from the routing table entry staying in the processor's L1 cache (64 KB), while the larger overhead value comes from having to pull a not-recently-used routing table entry into the CPU's cache.

5.3 Options for Fault Tolerance and New Communication Patterns on Sparse FNNs

Due to the typically large number of available paths between PEs in a Sparse FNN, especially when including paths through intermediary PEs, it is clear that with some work, Sparse FNNs should be able to be fairly fault tolerant. Also, with the diversity of connectivity in a typical Sparse FNN, it would seem that there should be a way to remap PE numbers on an existing Sparse FNN to support a new communication pattern that was unanticipated at the time the original Sparse FNN was designed.

As initially proposed for this Ph.D. work, it was assumed that the GA developed for designing Sparse FNNs could be easily modified into a re-targeting tool for the above purposes. This assumption was based on the structure of the Universal FNN GA which used DNA that was a direct representation of the network wiring pattern. As discussed in Section 3.4, the DNA used by the Sparse FNN GA as developed is *not* a direct representation of a network wiring pattern. Thus, the Sparse FNN GA can not take as input an already existing wiring pattern. It is possible that a new GA could be developed to find PE renumberings for an existing Sparse FNN that would either avoid specific faults or support a new communication pattern. However, it appears that such a GA would not be very practical, because the time to completion of a GA could very well exceed the time to simply fix a fault in a machine. As for the case where an unanticipated communication pattern needs to be supported, there exist a variety of techniques in the literature for mapping one network topology onto another with bounds on the amount of dilation. Studying and implementing these remapping techniques were beyond the scope of the current work.

As implemented, the runtime support software for Sparse FNNs leaves open the possibility for a user space fault-tolerance daemon to reconfigure the FNN routing table for a PE on demand. To

implement this daemon, there would need to be a method for detecting and identifying a fault in the network. Once the fault has been identified, a revised FNN description file that reflects the new state of the working network hardware could be created. Once created, this file would need to be propagated to the PEs, and then used to rebuild the FNN routing tables for each PE. Further study of this approach or other dynamic fault tolerance approaches is warranted, though such studies are beyond the scope of the current work.

5.4 InfiniBand (IB), Myrinet, QsNet, SCI, and Other Link-technology Alternatives

The previous sections of this chapter discussed various details that were specific to Ethernet implementations of FNNs. Here we discuss how these same concepts could be developed and implemented for other types of networking hardware. There are four key requirements of any candidate link technology that must be true for their use in FNNs:

1. Allows multiple independently routable NIs per PE
2. Availability of switches or routers with a reasonably large number of ports
3. A flexible packet routing scheme that allows for route selection using a lookup table
4. An addressing scheme that supports the total number of PEs in the machine

Because a FNN is built with as many NIs as it has total switch ports, there also needs to be a reasonable balance between the cost of the NIs and the cost per port of the switches. In contrast, typical (non-FNN) switched network topologies have many fewer NIs than the total number of switch ports in the network, which makes the cost of the NIs less important to the total cost. Network technologies such as Myrinet[7, 44] have NIs which tend to be much more expensive than the average port cost on a switch. Thus, Myrinet is not likely to be an economically effective candidate for FNN implementations, although, there appear to be no technical limitations preventing the use of Myrinet.

The lack of wide switches for commodity link technologies such as FireWire/IEEE1394[25] and USB[65] generally precludes their use in FNNs. The 63 node address limitation of IEEE1394 might also hinder its use for FNNs. For the various custom link technologies used in many supercomputer architectures with directly connected topologies, such as the Cray T3D[17] and the IBM BlueGene/L[24] architectures, the custom routing chips at each node have very little routing flexibility and could not be used without modification in a FNN.

The Quadrics QsNet[9, 44, 56] link technology should be applicable to FNNs because it has wide switches and support for multiple NIs per PE. It is unknown if the software layer that interfaces with the NIs is flexible enough for use in FNNs.

The Scalable Coherent Interface (SCI)[31] link technology employed in the Dolphin WulfKit[21] products is at first glance not applicable to FNNs due to its typical use in directly connected torus

topologies. However, 8-port switches are available for SCI that would allow the implementation of FNNs using SCI link technology.

The InfiniBand (IB)[9, 34, 44] link technology appears to have the full flexibility for use in FNNs. There are a variety of multi-port IB Host Channel Adapter (HCA) cards can be configured as independent NIs, and there are a variety of switch widths available from many vendors.

For all of the link technology alternatives, it appears the primary limitation for use in a FNN would be the level of difficulty in configuring the software layer(s) that interface with the NIs and that control the route selection for the network. Because these various alternative link-technologies have not yet been used in a FNN, the author can not guarantee that some firmware or proprietary software limitation would not preclude their use in a FNN. However, having talked with various vendors and examined the documentation for these technologies, the author believes there are no such limitations for Myrinet, InfiniBand, Dolphin/SCI, and Quadrics QsNet. Especially significant, is that for all four of those high speed link technologies, there are open-source software drivers available for use with the Linux OS. Such open-source software drivers should allow the implementation of the needed FNN runtime routing software, if the native software's configuration tables, etc. do not have the flexibility to directly support FNNs.

Chapter 6

Some Real FNN Implementations

In the preceding chapter, various possible Sparse FNN designs were presented. This chapter discusses two real parallel machines, one built with a Universal FNN and the other with a Sparse FNN. In addition to their network designs, both machines are remarkable for their achieved price/performance ratio on real applications.

6.1 The KLAT2 Supercomputer with the First Universal FNN

Figure 6.1 shows KLAT2 (Kentucky Linux Athlon Testbed 2)[19], a 64 PE supercomputer with the world's first Universal FNN. Its FNN consisted of nine 31-port switches ($\rho = 31$) and had four NIs per PE ($\eta = 4$). KLAT2 was built in the Spring of 2000, and was the first general purpose supercomputer to achieve over a GFLOPS of performance for each $1000 spent on the machine.

The significance of the KLAT2 machine in regards to this dissertation is not in its awards[16, 30] or performance characteristics. Rather, KLAT2 was a machine which facilitated the study of FNNs, which inspired the thesis for Sparse FNNs. The network on KLAT2 had much more connectivity than was nominally usable. To write an application that would keep all the wires busy at once would have been a rather difficult challenge. This challenge is clear when we look at a graphical representation of KLAT2's Universal FNN.

It is natural to think of both design constraints and solutions in terms of a square connectivity matrix, with node sources listed down the left side and sinks listed across the top, that shows how many links worth of bandwidth are requested/dedicated to that directed pairwise communication. Although such a graph for a design specification can be completely asymmetric, because all commonly used network hardware employs bidirectional cabling, no directly useful information is lost if the matrix is folded along the diagonal; the bandwidth requested for A→B is made equal to that requested for B→A by giving both the maximum value of either. Taking advantage of this property, we can represent the design requirements and solution in a single square matrix: the lower left triangle defines the requirements while the upper right triangle shows the bandwidth delivered by the solution. This matrix is trivially shown in graphical form as a square image in which the color (gray shade) of each point corresponds to the number of links required or dedicated.

Figure 6.1: Kentucky Linux Athlon Testbed 2 (KLAT2)

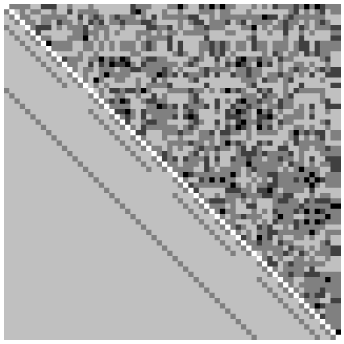

Figure 6.2: KLAT2's FNN Design/Solution Map

Figure 6.2 is a representation similar to the connectivity matrices elsewhere in this dissertation. However, instead of showing the desired connectivity as black pixels in the upper right triangle, the desired connectivity is shown in the lower left triangle. Because a Universal FNN implicitly requires that all PE pairs have single-switch latency, this figure instead emphasizes the number of links, e.g. bandwidth, between each PE pair, represented as shades of gray.

The white center line represents nodes talking to themselves, which neither requires nor uses network bandwidth. The lower left triangle specifies complete connectivity with a single unit bandwidth per pair and, additionally, two or more units bandwidth for the communication patterns shown in a somewhat darker gray. KLAT2's network actually delivers as much as four units of bandwidth per pair (a black pixel corresponds to four units of bandwidth), entirely covering the single-unit requirement region. Although KLAT2's design does not quite cover the two-unit requirement region with two or more units of bandwidth, it comes very close to covering it with an average of more

Figure 6.3: Kentucky ASYmmetric Zero (KASY0)

than two units bandwidth per pair. This shortfall is because, at the time KLAT2 was designed, our FNN design tool favored a higher average over complete coverage of the two-or-more region.

Several higher-level properties of KLAT2's network are easily visible in this graphic. One is the asymmetric nature of KLAT2's network design; the upper right triangle is a random-looking pattern of one to four units bandwidth per pair. Additionally, when one views KLAT2's network in this way it seems clear that the network is seriously over designed – there are many low-importance pairs that are given high-bandwidth coverage. Suppose that we remove the constraint that all pairs must have at least one unit of reserved, single-hop latency, bandwidth. Our concern is thus shifted to finding a design which covers all node pairs that we expect will have significant communications between them. This shift in design constraints is how the basic concept of Sparse FNNs was formed.

6.2 The KASY0 Supercomputer with the First Sparse FNN

In the Summer of 2003 we built a cluster supercomputer that would demonstrate the Sparse FNN concept in a real system, in addition to giving us a powerful machine for use in our lab. The KASY0 (Kentucky ASYmmetric Zero) supercomputer[35, 47] shown in Figure 6.3 has 128 PEs and a Sparse FNN using three NIs/PE ($\eta = 3$) and a total of seventeen 23-port switches ($\rho = 23$).

6.2.1 KASY0's Hardware

The 128 PEs in KASY0 were constructed from interchangeable parts from the commodity PC industry, each of which contained these items:

- One Retail AMD Athlon XP 2600+ (2.075GHz clock, 256 KB L2 cache, 333 MHz FSB)
- One 512MB PC2700 DDR SDRAM DIMM (Crucial part #CT6464Z335)
- One BioStar M7VIT Pro motherboard with onboard Fast Ethernet Network Interface
- Two Linksys LNE100TX v4.1 Fast Ethernet NICs
- One Codegen 6042L case with 400W power supply, plus two 80mm fans

To meet the design goals of KASY0, we considered a variety of designs for the PEs to achieve the best price/performance ratio using a fixed budget for the total machine cost. Although processors were available with faster peak GFLOPS numbers, higher memory bandwidth, and/or lower memory latency, the price premium for those alternatives would have reduced the size of the machine by more than the gains in individual PE performance. A parallel supercomputer is a design that converts great PE price/performance into great raw performance.

KASY0 is able to achieve a very good price/performance ratio, in part, because of its low cost. Including all parts, shipping, and assembly labor[1] the total cost was $39,604.31. The remarkable thing about KASY0's price is that, while network hardware is often the dominant cost for a system of its size (128 plus 4 spare nodes), less than 11% of the system cost went for the network hardware. The AMD Athlon XP 2600+ processors were more than 35% of the total system cost; memory was 21%. In fact, the annual electric bill for operating KASY0 is about the same as the cost of KASY0's network.

6.2.2 KASY0's Sparse FNN

KASY0's Sparse FNN was designed to cover the following communication patterns:

- Hypercube
- Bit reversal
- Ring with distance-1 offsets
- Single 2D torus (16×8) with full row and column adjacency
- Single 3D torus ($8 \times 4 \times 4$) with adjacency to all PEs that differ in only one dimension

The lines in Figure 6.4, starting with a number followed by ":" specify the actual wiring pattern for KASY0's Sparse FNN: the first number is the switch number and the remaining numbers on each line are the node numbers connected to that switch. While this table of numbers may be an exact description of KASY0's wiring pattern, it is not particularly helpful in revealing patterns or other

[1]The students who helped assemble KLAT2 were volunteers who donated their time. The students were compensated for their efforts with $188 worth of food total, and an immeasurable amount of education about the internals of PCs and their construction.

```
 0:   0  15  16  31  32  47  48  63  64  79  80  95  96 111 112 120 121 122 123 124 125 126 127
 1:   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  24  40  56  72  88 104 120
 2:   7  23  39  55  71  87 103 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127
 3:   2  10  18  26  33  34  35  37  42  43  44  45  46  50  58  66  74  82  90  98 106 114 122
 4:   5  13  21  29  37  45  53  61  69  77  81  83  84  85  86  89  92  93  94 101 109 117 125
 5:   1   9  17  25  33  41  49  57  64  65  67  68  70  73  75  76  78  81  89  97 105 113 121
 6:   4  12  16  18  19  20  22  23  27  28  30  36  44  52  60  68  76  84  92 100 108 116 124
 7:   3  11  19  27  35  43  51  59  67  75  83  91  96  98  99 102 103 104 107 109 110 115 123
 8:   6  14  22  30  38  46  48  50  53  54  55  56  59  60  62  70  78  86  94 102 110 118 126
 9:  32  36  38  39  40  41  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63
10:   2  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  69  71  72  73  74  79
11:  64  65  66  67  68  69  70  71  72  74  75  76  77  78  79  80  82  87  88  90  91  93  95
12:   5  17  20  21  24  26  29  31  80  81  82  83  84  85  86  87  88  89  90  91  92  94  95
13:   7   8  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 100 101 105 111
14:  66  73  77  85  96  97  98  99 100 101 102 103 104 105 106 107 108 109 110 111 112 117 119
15:  49  51  52  54  57  58  61  62  63  65  93  97  99 106 107 108 113 114 115 116 118 119 127
16:   0   1   3   4   6   8   9  10  11  12  13  14  25  28  34  42
```

Figure 6.4: KASY0's Switch Connection List

interesting features of the design. To that end, Figure 6.5 shows a graphical Design/Solution Map for KASY0's Sparse FNN. In the upper right is a representation of the actual coverage achieved by the Sparse FNN solution, with black indicating coverage of the requested pattern, and green indicating coverage of extra non-requested PE pairs. The vertical axis goes from PE 0 at the top to PE 127 at the bottom. Similarly, the horizontal axis goes from PE 0 on the left to PE 127 on the right. Each coordinate in the figure represents the single-switch-hop connectivity of a pair of PEs. Because the links in the network are bidirectional, the coverage if shown in the lower left would be a mirror image along the diagonal. The intensity of the colors in the upper right indicate the number of single-switch paths that connect each PE pair, with darker colors indicating more available paths.

6.2.3 KASY0's Performance

KASY0's theoretical peak performance numbers are 531 GFLOPS and 1.06 TFLOPS, respectively, for 64/80-bit[2] and 32-bit floating point. Real applications will achieve lower numbers. This section summarizes performance results for the HPL benchmark and the POV-Ray benchmark.

A well-known reference for supercomputer performance is the Top500[64], which lists the 500 supercomputers that obtain the highest GFLOPS speed executing the HPL (High Performance Linpack) benchmark program. Performance on HPL depends partly on the theoretical peak GFLOPS of the processors, but also on the parallel implementation and efficiency of the network that allows the processors to work together. In the June 2003 Top500 list, most systems use expensive, specialized, network hardware. The machines explicitly listed as using standard 100Mb/s Fast Ethernet achieved an average of less than 8.5% of peak. The average for the systems listed as using Gigabit Ethernet is better, at about 30% of peak. In contrast, KASY achieved 187.3 GFLOPS, over 35% of peak using a double-precision version of HPL. For the HPL benchmark, KASY0 achieved a price/performance ratio of $0.21/MFLOPS (64/80-bit).

[2]The floating-point registers and internal results have 80 bits of precision, though each value in RAM only has 64-bits of storage.

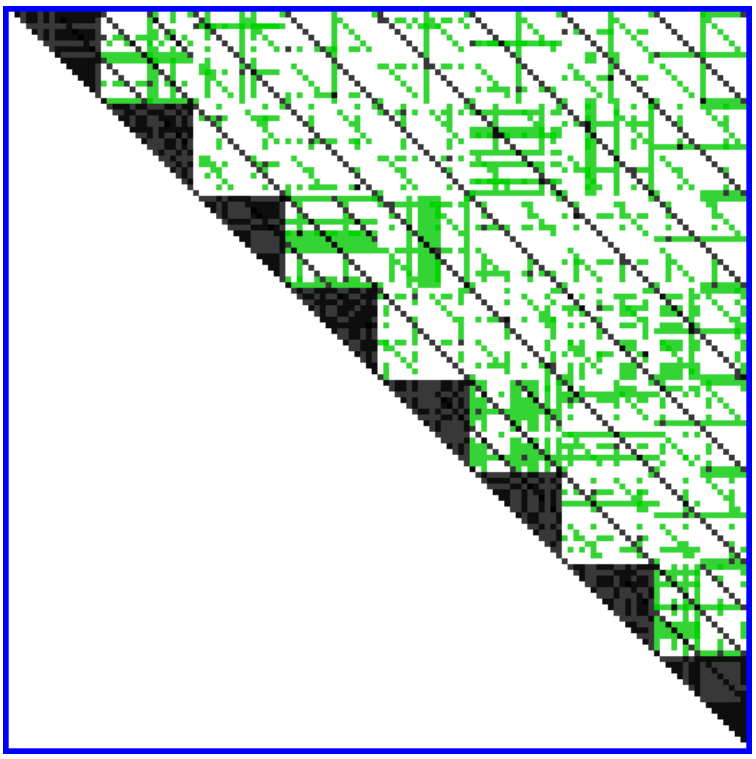

Figure 6.5: KASY0's Design/Solution Map

Building on our work from KLAT2's Gordon Bell submission[30], we wrote a newly tuned SGEMM core[3] that uses the 3DNow! multi-media instruction set of the Athlon. Using this newly tuned SGEMM core, KASY0 gets 482.6 GFLOPS on a single-precision version of the Linpack benchmark. That is over 45% of theoretical peak performance and less than $82 per 32-bit GFLOPS, or an astounding price/performance ratio of $0.082/MFLOPS (32-bit).

As of August 22, 2003, KASY0 set a new world record for rendering the complex benchmark image shown in Fig.6.6 using the Persistence of Vision Raytracer (POV-Ray)[54]. Executing `pvmpovray 3.5c` on KASY0 to render the standard `benchmark.pov` scene took 72 seconds, which beat the previous record of 107 seconds set on August 1, 2003. This POV-Ray benchmark has a communication pattern commonly found in manager-worker parallel codes, with many small messages between a central manager node and individual worker nodes. This asymmetric communication pattern was best supported by placing the manager process on KASY0's boot server, yielding two switch-hop latency to any worker node in the cluster. Despite recent submissions from other systems, KASY0 *still* holds the world record on this benchmark as of April 6, 2006 – more than two and a half years after setting the record![4]

[3] Our tuned 3DNow! SGEMM core is available in the 3.6 and later versions of ATLAS[68].

[4] We are certain that there are machines that easily could beat KASY0's record. However, the records list the system cost, and the benchmark code's structure is such that systems using conventional network designs that can beat KASY0 would outrun it by a very small margin at much higher cost. There is a newer machine costing significantly more than KASY0 proudly positioned below it on the list.

Figure 6.6: Standard POV-Ray 3.5 Benchmark image

6.2.4 Scalability of KASY0's Supported Communication Patterns

The two torus sub-patterns in KASY0's Sparse FNN fall into the $O(\sqrt[D]{N})$ category discussed in Section 2.3.3, so they do not scale as well as the various sets discussed previously in Chapter 4. Figure 6.7 shows example solutions to this combination of patterns for 16, 32, 64, 128, and 256 PEs. This more difficult scaling property can be seen in Figure 6.8, which shows summary results for this combination of patterns on other size machines.

At the time KASY0 was designed, our knowledge of how Sparse FNNs scale was limited. The design tool was a cruder and much slower version of the non-GA based heuristic described in Section 3.3. The selection of communication patterns for the KASY0 design was aimed at determining how many "awkward" patterns we could cover using 24-port Fast Ethernet switches, which were the cheapest per port at the time. In retrospect, including $O(\sqrt[D]{N})$ scaling patterns in KASY0's Sparse FNN probably was not justified; certainly, the codes KASY0 usually runs do not need them.

With the improved design technology discussed in this dissertation, a Sparse FNN supporting the same patterns specified for KASY0 could be built using 16-port switches. Similarly, the runtime support software was in very early development when KASY0 was built, so it was safer to connect each switch in the Sparse FNN to a top level switch to allow direct access to each PE from a single manager machine. Thus, KASY0's Sparse FNN was designed as if the switches were 23-ports each, reserving the 24th port to be an uplink to a top level switch.

Even with the above caveats, KASY0's Sparse FNN was dramatically cheaper than any other conceivable network of comparable performance for a 128 PE machine at the time KASY0 was built. It cost only $39,604.31 to build KASY0, with only 11% of the total cost spent on the network. Yet, achieved performance on real applications and benchmarks that clearly demonstrated the superior effectiveness of its remarkably inexpensive network. We also were able to use KASY0 very effectively to further develop and refine the Sparse FNN technologies that are the core of this dissertation.

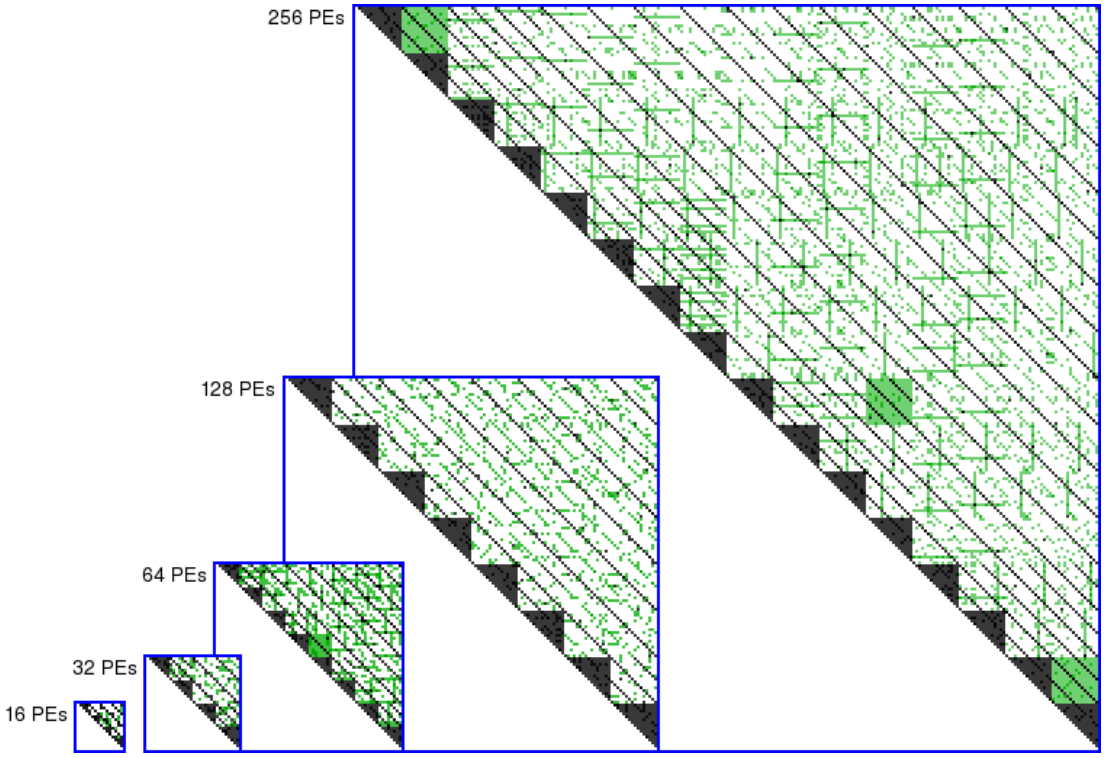

Figure 6.7: $\eta = 3$ NIs/PE Solutions for KASY0's supported communication patterns

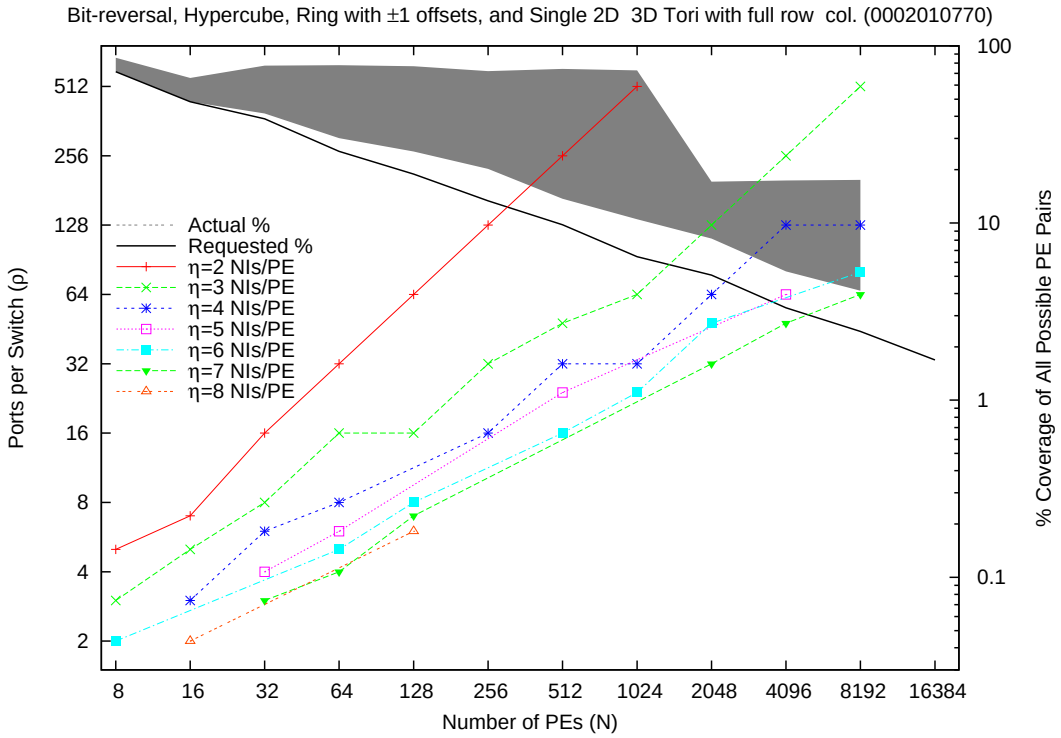

Figure 6.8: Scaling of KASY0's supported communication patterns

Chapter 7

Adoptions and Future Advancement of the Technology

Beyond our laboratory's KLAT2 and KASY0 clusters, FNN technology has been adopted and used in clusters in the Mechanical Engineering Department at the University of Kentucky, Keele University, Cornell University, Utah State University, Xavier University, and the University of Louisville. There are probably others as well; both Linux Labs (a cluster vendor and developer of system software) and the Massachusetts Institute of Technology have discussed with us their interest in using FNN technology. Various versions/generations of the FNN runtime support software have been deployed on clusters using FNNs at the sites listed. The FNN support software also can improve performance of more traditional networks, especially those using channel bonding.

As of this writing, KASY0 still is the only *Sparse* FNN implementation. However, there has been very recent interest in deploying Sparse FNN based systems at other institutions. It is only with the completion of this dissertation that we are releasing the support in a sufficiently polished form that Sparse FNN design and implementation will be possible without substantial consulting assistance from our research group.

There are several areas for future research that have been made apparent by the work reported in this dissertation:

- The design and characteristics of Fractional FNNs, defined in Section 2.4, are worth exploring. There is great potential to achieve a significant fraction of the performance of a Sparse or Universal FNN at a dramatically lower cost. Further, Fractional FNNs are more amenable to the process of automatically creating designs based on empirical evaluation of an application code's execution.

- An area deserving additional research would be the class of networks that are not strictly FNNs yet utilize knowledge of the expected communication patterns in their design. Specifically, networks that guarantee at most k switch hops between selected PE pairs, where $k > 1$, would not be Sparse FNNs, yet they would give further flexibility in the cost/performance trade-off decisions when designing a particular machine.

- Work on implementing fault tolerance in a FNN is an area ripe for further research. Fault tolerance is a major concern for machines of the size that Sparse FNNs have the most impact.
- The current trend towards multiple processor cores per chip and the use of multiple chips per machine node in a parallel machine indicates the need to explore the effects on communication patterns that occur within a node, as well as between nodes.
- Is it practical to increase the pair synergy between different communication patterns by selecting alternate PE numberings for the various patterns, and/or alternate factorizations of the grid/torus patterns? Increasing pair synergy may be a key technique toward both support of fault tolerance and improved efficiency using multiple-core/multiprocessor nodes.
- The implementation of runtime support for FNNs on another high speed link layer technology, such as InfiniBand, would be a very practical path towards making it possible for FNNs to be more widely used. Unfortunately, KASY0 was built in the last days when cost favored 100Mb/s Fast Ethernet, and the fact that KASY0 uses a "slow" implementation technology has sometimes blinded potential users from seeing the fundamental importance of Sparse FNNs. This problem is now quickly disappearing, as Gigabit Ethernet has become the accepted norm for supercomputer network implementation technology and it is fully supported by our current runtime software. As of the 26th Top500 list (November 2005), more than half the systems implement their primary interconnection network using Gigabit Ethernet technology.

We believe that, even if Sparse FNNs are not *the* answer, the era of hand-designed network topologies is coming to a close. From the complexities of high degree Cayley graphs to the asymmetries of FNNs, it seems clear that computational tools such as GAs will become standard engineering practice for design of future supercomputer networks. The computational power that can be applied towards the design problem will continue to advance. The sheer magnitude of the computations performed to design the networks presented in Chapter 4 would not have been available for the task a decade ago. In the decades that follow this work, current trends predict that all the above computations will be able to be replicated in a few hours (or less) on a commodity laptop computer. With such a low cost for taking this approach, the benefits of using a computer to design networks will make this approach irresistible.

Chapter 8

Conclusions

For scalable parallel programs, the set of PE pairs that communicate often is both predictable and small relative to the number of possible PE pairings. Exploiting this sparseness property can greatly enhance the design and implementation of networks for massively parallel supercomputers.

The sparseness of communicating pairs is rooted in the fact that each of the human-designed communication patterns commonly used in parallel programs has the property that the number of communicating pairs grows relatively slowly as the number of PEs is increased. Additionally, the number of pairs in the union of all communication patterns used in a suite of parallel programs grows surprisingly slowly due to pair synergy: the same pair often appears in multiple communication patterns. The detailed analysis of communication patterns presented in Section 2.3 clearly shows that the number of PE pairs actually communicating is very sparse, although the structure of the sparseness can be complex.

The exploitation of this sparseness can be accomplished in many ways. Here, our focus is on producing Sparse FNNs: network designs which use the sparseness of communicating PE pairs to provide single-switch latency and full wire bandwidth for each of the PE pairs specified. Sparse FNNs achieve these performance properties despite using relatively few network interfaces per PE and switches that have far fewer ports than there are PEs. The Sparse FNN design problem is discussed in Chapter 3, the runtime support needed to make it work is described in Section 5.2, and Section 6.2 overviews a working prototype (KASY0) which not only demonstrated the claimed properties, but also set world records for its price/performance and performance on a specific application (the POV-Ray 3.5 benchmark rendering problem [54]).

The concept of matching the network topology to the expected communication pattern for an application is not new; JPL's Big Viterbi Decoder[15] is an example of this concept. However, covering a few patterns with single-hop latency and full link bandwidth was done either by constructing a network that was literally the union of the networks for the individual patterns or by finding ways to map other topologies onto the hardware topology (e.g., embedding a mesh in a hypercube). The contribution of Sparse FNNs is that they view covering many patterns as a single problem, creating a network that is a cover of the union rather than the union of the covers. For two-port switches, there would be no difference: a two-port switch is essentially equivalent to a wire, covering just a

single pair. However, switches with three or more ports allow the network design to be cheaper – a three-port switch implements 3 pairings, four-port implements 6, and 48-port implements 1,128.

Perhaps the reason Sparse FNNs were not invented earlier is that they would not have been feasible just a decade ago. The graph covering problem[28, 38] upon which Sparse FNN design is based, has no known solution algorithm that has less than exponential time complexity; it was necessary to develop new Genetic Algorithm (GA) technology to solve the design problem, and the computational power needed is beyond what would have been readily available a decade ago. Design problems small enough to be solved by hand are not large enough to have significant sparseness. The benefits of sparseness only become apparent for machines with at least 128 PEs and fairly wide switches, and both of these features have become common only in the last few years. However, it is clear that the future will be filled with machine design problems well-suited to Sparse FNN solutions.

Bibliography

[1] Douglas A. Aberdeen, Jonathan Baxter, and Robert Edwards. 98 cents/Mflops/s, Ultra-Large-Scale Neural-Network Training on a PIII Cluster. In *Proceedings of the IEEE/ACM SC2000 conference*, Dallas, Texas, November 2000.

[2] Sheldon B. Akers and Balakrishnan Krishnamurthy. A group-theoretic model for symmetric interconnection networks. *IEEE Transactions on Computers*, 38(4):555–566, April 1989.

[3] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing*. The Benjamin/Cummings Publishing Company, Inc., 1994.

[4] BBN Advanced Computers Inc. *Butterfly Products Overview*, October 1987.

[5] Rudolf Berrendorf, Heribert C. Burg, Ulrich Detert, Ruediger Esser, Michael Gerndt, and Renate Knecht. Intel paragon XP/S - architecture, software environment, and performance. Technical Report KFA-ZAM-IB-9409, 1994.

[6] D.W. Blevins, E.W. Davis, R.A. Heaton, and J.H. Reif. BLITZEN: A highly integrated massively parallel machine. *Journal of Parallel and Distributed Computing*, 8:150–160, February 1990.

[7] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, 1995.

[8] P. A. Boyle, D. Chen, N. H. Christ, M. A. Clark, S. D. Cohen, C. Cristian, Z. Dong, A. Gara, B. Joó, C. Jung, C. Kim, L. A. Levkova, X. Liao, G. Liu, R. D. Mawhinney, S. Ohta, K. Petrov, T. Wettig, and A. Yamaguchi. Overview of the QCDSP and QCDOC computers. *IBM Journal of Research and Development*, 49(2/3):351–365, March/May 2005.

[9] Ron Brightwell, Doug Doerfler, and Keith Underwood. A Comparison of 4X InfiniBand and Quadrics Elan-4 Technologies. In *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, September 2004.

[10] Jehoshua Bruck, Ching-Tien Ho, Shlomo Kipnis, Eli Upfal, and Derrick Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1143–1156, November 1997.

[11] C. J. Burgess and A. G. Chalmers. The optimisation of irregular multiprocessor computer architectures using genetic algorithms. Technical Report CSTR-96-006, Department of Computer Science, University of Bristol, March 1996.

[12] Charles Clos. A study of non-blocking switching networks. *Bell Systems Technical Journal*, 32(2):406–424, March 1953.

[13] Charles J. Colbourn and Jeffery H. Dinitz, editors. *The CRC Handbook of Combinatorial Designs*, pages 260–265,419–423. CRC Press, 1996.

[14] O. Collins, F. Pollara, S. Dolinar, and J. Statman. Wiring Viterbi Decoders (Splitting de-Bruijn Graphs). In *TDA Progress Report 42-96 (October-December 1988)*, pages 93–103. Jet Propulsion Laboratory, February 1989. NASA Code 310-30-72-88-01.

[15] Oliver Collins, Sam Dolinar, Robert McEliece, and Fabrizio Pollara. A VLSI decomposition of the deBruijn graph. *Journal of the ACM*, 39(4):931–948, 1992.

[16] Computerworld Honors for the KLAT2 Supercomputer. http://www.cwhonors.org/Search/his_4a_detail.asp?id=4298. 2001.

[17] Cray Reasearch Inc. *Cray T3D System Architecture Overview*, 1993.

[18] Hank Dietz and Tim Mattox. Inside the KLAT2 Supercomputer: The Flat Neighborhood Network and 3DNow! *http://arstechnica.com/cpu/2q00/klat2/klat2-1.html*, June 2000.

[19] H.G. Dietz and T.I. Mattox. KLAT2's Flat Neighborhood Network. In *Proceedings of the 4th Annual Linux Showcase, Extreme Linux Track*, Atlanta, GA, October 2000.

[20] H.G. Dietz and T.I. Mattox. Compiler Techniques For Flat Neighborhood Networks. In S.P. Midkiff, J.E. Moreira, M. Gupta, S. Chatterjee, J. Ferrante, J. Prins, W. Pugh, and C.-W. Tseng, editors, *Languages and Compilers for Parallel Computing, 13th International Workshop (LCPC 2000)*, volume 2017 of *Lecture Notes in Computer Science*, pages 244–259, IBM Watson Research Center, Yorktown, NY, 2001. Springer-Verlag.

[21] Dolphin Interconnect Solutions, Inc. *D200 Series – WulfKit High-Performance Parallel Computing Clustering Solution*, 2004.

[22] José Duato, Sudhakar Yalamanchili, and Lionel Ni. *Interconnection Networks: An Engineering Approach*. IEEE Computer Society Press, 1997.

[23] Reuven Elbaum and Moshe Sidi. Topological design of local-area networks using genetic algorithms. *IEEE/ACM Transactions on Networking*, 4(5):766–778, 1996.

[24] Adiga et al. An overview of the BlueGene/L supercomputer. In *Proceedings of the IEEE/ACM SC2002 conference*, Baltimore, MD, November 2002.

[25] FireWire/IEEE 1394 Trade Association. http://www.1394ta.org.

[26] FNN (Flat Neighborhood Network). http://aggregate.org/FNN/.

[27] GAMMA. http://www.disi.unige.it/project/gamma/.

[28] Daniel M. Gordon, Greg Kuperberg, and Oren Patashnik. New constructions for covering designs. *Journal of Combinatorial Designs*, 3(4):269–284, July 1995.

[29] S. K. S. Gupta, C.-H. Huang, P. Sadayappan, and R. W. Johnson. Implementing fast Fourier transforms on distributed-memory multiprocessors using data redistributions. *Parallel Processing Letters*, 4(4):477–488, 1994.

[30] Th. Hauser, T.I. Mattox, R.P. LeBeau, H.G. Dietz, and P.G. Huang. High-Cost CFD on a Low-Cost Cluster. In *Proceedings of the IEEE/ACM SC2000 conference*, Dallas, TX, November 2000. Received Gordon Bell Prize Honorable Mention, Price/Performance category.

[31] Hermann Hellwagner and Alexander Reinefeld, editors. *SCI: Scalable Coherent Interface, Architecture and Software for High-Performance Computer Clusters*, volume 1734 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[32] Debra Hensgen, Raphael Finkel, and Udi Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1):1–17, February 1988.

[33] R. Hoare, H. Dietz, T. Mattox, and S. Kim. Bitwise aggregate networks. In *Proceedings of The Eighth IEEE Symposium on Parallel and Distributed Processing (SPDP '96)*, New Orleans, LA, October 1996.

[34] InfiniBand Trade Association. *InfiniBand(TM) Architecture Specification Release 1.2*, October 2004.

[35] KASY0 (Kentucky ASYmmetric Zero). http://aggregate.org/KASY0/.

[36] Sunil Kim and Alexander Veidenbaum. On shortest path routing in single stage shuffle-exchange networks. In *Proc. 7th Annual ACM Symposium on Parallel Algorithms and Architectures SPAA'95*, pages 298–307, Santa Barbara, California, 1995.

[37] KLAT2 (Kentucky Linux Athlon Testbed 2). http://aggregate.org/KLAT2/.

[38] La Jolla Covering Repository. http://www.ccrwest.org/cover.html.

[39] Vijay Lakamraju, Israel Koren, and C.M. Krishna. Filtering Random Graphs to Synthesize Interconnection Networks with Multiple Objectives. *IEEE Transactions on Parallel and Distributed Systems*, 13(11):1139–1149, November 2002.

[40] T.L. Lau and E.P.K. Tsang. Applying a mutation-based genetic algorithm to processor configuration problems. In *8th IEEE Conference on Tools with Artificial Intelligence (ICTAI'96)*, Toulouse, France, November 1996.

[41] Tung Leng Lau. *Guided Genetic Algorithm*. PhD thesis, Department of Computer Science, University of Essex, United Kingdom, 1999.

[42] Charles E. Leiserson. Fat-trees: Universal networks for hardware efficient supercomputing. *IEEE Transactions on Computers*, C-34(10):892–901, October 1985.

[43] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St Pierre, David S. Wells, Monica C. Wong-Chan, Shaw-Wen Yang, and Robert Zak. The network architecture of the Connection Machine CM-5. *Journal of Parallel and Distributed Computing*, 33(2):145–158, 1996.

[44] J. Liu, B. Chandrasekaran, J. Wu, W. Jiang, S. Kini, W. Yu, D. Buntinas, P. Wyckoff, and D. K. Panda. Performance Comparison of MPI Implementations over InfiniBand, Myrinet and Quadrics. In *Proceedings of the ACM/IEEE SC2003 Conference*, November 2003.

[45] Junichiro Makino, Eiichiro Kokubo, and Toshiyuki Fukushige. Performance evaluation and tuning of GRAPE-6 - towards 40 "real" Tflops. In *Proceedings of the ACM/IEEE SC 2003 Conference (SC'03)*, Phoenix, AZ, November 2003.

[46] MasPar Corp. *MasPar System Overview*, pn 9300-0100-2790 edition, July 1990.

[47] Timothy I. Mattox, Henry G. Dietz, and William R. Dieter. Sparse Flat Neighborhood Networks (SFNNs): Scalable Guaranteed Pairwise Bandwidth and Unit Latency. In *Proceedings of the Fifth Workshop on Massively Parallel Processing (WMPP'05) held in conjunction with the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005)*, Denver, CO, USA, April 2005.

[48] E. H. McKinney. Generalized Birthday Problem. *The American Mathematical Monthly*, 73(4):385–387, April 1966.

[49] Message Passing Interface Forum, http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html. *MPI: A Message-Passing-Interface Standard*, May 1994.

[50] nCUBE Corporation. *Technical Overview: nCUBE 2 Supercomputers*, 1990.

[51] Lionel M. Ni and Philip K. McKinley. A Survey of Wormhole Routing Techniques in Direct Networks. *IEEE Computer*, pages 62–76, February 1993.

[52] Kari J. Nurmela. Constructing combinatorial designs by local search. Research Report A27, Helsinki University of Technology, Digital Systems Laboratory, Espoo, Finland, November 1993.

[53] Kari J. Nurmela and Patric R. J. Östergård. Constructing covering designs by simulated annealing. Technical Report B10, Helsinki University of Technology, Digital Systems Labora tory, Espoo, Finland, January 1993.

[54] Official POV-Ray Benchmarks. http://www.haveland.com/index.htm?povbench/index.php.

[55] Scott Pakin, Mario Lauria, and Andrew Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of the IEEE/ACM SC95 Conference*, December 1995.

[56] Fabrizio Petrini, Wu chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The Quadrics network: High-performance clustering technology. *IEEE Micro*, 22(1):46–57, January 2002.

[57] M. R. Samatham and D. K. Pradhan. The de Bruijn multiprocessor network: A versatile parallel processing and sorting network for VLSI. *IEEE Transactions on Computers*, 38(4):567–581, 1989.

[58] Howard Jay Siegel. *Interconnection Networks for Large-Scale Parallel Processing*. McGraw-Hill Publishing Company, 1990.

[59] Evan Speight, Hazim Abdel-Shafi, and John K. Bennett. Realizing the performance potential of the Virtual Interface Architecture. In *Proceedings of the 13th ACM International Conference on Supercomputing (ICS)*, June 1999.

[60] Craig Stanfill. Communications Architecture in the Connection Machine System. Technical Report HA87-3, March 1987.

[61] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I:11–14, Oconomowoc, WI, 1995.

[62] Thinking Machines Corporation. *Connection Machine Model CM-2 Technical Summary*, November 1990.

[63] Thinking Machines Corporation. *Connection Machine CM-5 Technical Summary*, November 1992.

[64] Top500 Supercomputers. http://top500.org/.

[65] Universal Serial Bus (USB). http://www.usb.org/.

[66] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture (ISCA'92)*, May 1992.

[67] T. J. Warwick. *A GA Approach To Constraint Satisfaction Problems*. PhD thesis, Department of Computer Science, University of Essex, United Kingdom, February 1995.

[68] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing*, 27(1-2):3–25, January 2001.

[69] Junming Xu. *Topological Structure and Analysis of Interconnection Networks*. Kluwer Academic Publishers, 2001.

[70] Jung-Lok Yu, Moon-Sang Lee, and Seung-Ryoul Maeng. An Efficient Implementation of Virtual Interface Architecture using Adaptive Transfer Mechanism on Myrinet. In *Proceedings of the Eighth International Conference on Parallel and Distributed Systems (ICPADS'01)*, June 2001.

Vita of Timothy Ian Mattox

Date of Birth: October 21, 1971

Place of Birth: Lynchburg, Virginia

Education

1997	M.S.E.E., Electrical and Computer Engineering, Purdue University
1993	B.S.C.E.E., Electrical and Computer Engineering, Purdue University

Professional Experience

1999-2006	Graduate Research Assistant, lead student researcher in the KAOS Lab Department of Electrical and Computer Engineering, University of Kentucky
1994-2005	Research Exhibitor at the annual IEEE/ACM Supercomputing Conference The Aggregate.org consortium, Purdue University & University of Kentucky
1996-1999	Graduate Teaching Assistant, various graduate & undergraduate computer courses School of Electrical and Computer Engineering, Purdue University
1994-1996	Graduate Research Assistant, student researcher in the Parallel Processing Lab School of Electrical and Computer Engineering, Purdue University
1992,1993	Summer Undergraduate Research Internship Engineering Research Center for Intelligent Manufacturing Systems, Purdue University
1989,1990	Summer Internships: CAD Operator/Designer & System Analyst/Programmer Simplimatic Engineering Co., Lynchburg, VA

Awards and Distinctions

2000	Gordon Bell Prize Honorable Mention, Price/Performance Category[5]
2000	SC2000 HPC Games, Most Innovative Hardware Prize
1997	Magoon Award for Excellence in Teaching, Purdue University
1994-1999	Active member of Eta Kappa Nu Electrical Engineering honor society
1989	2nd place in Zoology at the 40th International Science and Engineering Fair “A Computer Simulation of Biological Evolution”

List of Publications

[1] Timothy E. Dowling, Mary E. Bradley, Edward Colón, John Kramer, Raymond P. LeBeau, Grace C.H. Lee, Timothy I. Mattox, Raul Morales-Juberías, Csaba J. Palotai, Vimal K. Parimi, and Adam P. Showman. The EPIC Atmospheric Model with an Isentropic/Terrain-Following Hybrid Vertical Coordinate. *Icarus (in press)*, 2006.

[2] Timothy I. Mattox, Henry G. Dietz, and William R. Dieter. Sparse Flat Neighborhood Networks (SFNNs): Scalable Guaranteed Pairwise Bandwidth and Unit Latency. In *Proceedings of the Fifth Workshop on Massively Parallel Processing (WMPP'05) held in conjunction with the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005)*, Denver, CO, USA, April 2005.

[3] Th. Hauser, T.I. Mattox, R.P. LeBeau, H.G. Dietz, and P.G. Huang. Code optimizations for complex microprocessors applied to CFD software. *SIAM Journal on Scientific Computing*, 25(4):1461–1477, 2004.

[4] H.G. Dietz and T.I. Mattox. Compiler optimizations using data compression to decrease address reference entropy. In Bill Pugh and Chau-Wen Tseng, editors, *Languages and Compilers for Parallel Computing, 15th International Workshop (LCPC 2002)*, volume 2481 of *Lecture Notes in Computer Science*, College Park, MD, USA, 2005. Springer-Verlag.

[5] Th. Hauser, T.I. Mattox, R.P. LeBeau, H.G. Dietz, and P.G. Huang. High-Cost CFD on a Low-Cost Cluster. In *Proceedings of the IEEE/ACM SC2000 conference*, Dallas, TX, November 2000. Received Gordon Bell Prize Honorable Mention, Price/Performance category.

[6] H.G. Dietz and T.I. Mattox. KLAT2's Flat Neighborhood Network. In *Proceedings of the 4th Annual Linux Showcase, Extreme Linux Track*, Atlanta, GA, October 2000.

[7] H.G. Dietz and T.I. Mattox. Compiler Techniques For Flat Neighborhood Networks. In S.P. Midkiff, J.E. Moreira, M. Gupta, S. Chatterjee, J. Ferrante, J. Prins, W. Pugh, and C.-W. Tseng, editors, *Languages and Compilers for Parallel Computing, 13th International Workshop (LCPC 2000)*, volume 2017 of *Lecture Notes in Computer Science*, pages 244–259, IBM Watson Research Center, Yorktown, NY, 2001. Springer-Verlag.

[8] Hank Dietz and Tim Mattox. Inside the KLAT2 Supercomputer: The Flat Neighborhood Network and 3DNow! *http://arstechnica.com/cpu/2q00/klat2/klat2-1.html*, June 2000.

[9] H.G. Dietz, T.I. Mattox, and G. Krishnamurthy. The Aggregate Function API: It's not just for PAPERS anymore. In Z. Li, P.-C. Yew, S. Chatterjee, C.-H. Huang, P. Sadayappan, and D. Sehr, editors, *Languages and Compilers for Parallel Computing, 10th International Workshop (LPCP'97)*, volume 1366 of *Lecture Notes in Computer Science*, pages 277–291, Minneapolis, MN, 1998. Springer-Verlag.

[10] Timothy I. Mattox. Synchronous aggregate communication architecture for MIMD parallel processing. Master's thesis, School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN, August 1997.

[11] H.G. Dietz and T.I. Mattox. Managing polyatomic coherence and races with replicated shared memory. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 53–58, March 1997.

[12] R. Hoare, T.I. Mattox, and H. Dietz. TTL-PAPERS 960801: The Modularly Scalable, Field Upgradable, Implementation of Purdue's Adapter for Parallel Execution and Rapid Synchronization. Technical Report http://aggregate.org/AFN/960801/Index.html, School of Electrical Engineering, Purdue University, West Lafayette, IN, 1997.

[13] R. Hoare, H. Dietz, T. Mattox, and S. Kim. Bitwise aggregate networks. In *Proceedings of The Eighth IEEE Symposium on Parallel and Distributed Processing (SPDP '96)*, New Orleans, LA, October 1996.

[14] H.G. Dietz, R. Hoare, and T. Mattox. A fine-grain parallel architecture based on barrier synchronization. In A. Reeves, editor, *1996 International Conference on Parallel Processing*, volume I Architecture, pages 247–250, Bloomington, IL, August 1996. IEEE Computer Society Press.

[15] Henry G. Dietz, T. M. Chung, and Timothy I. Mattox. A parallel processing support library based on synchronized aggregate communication. In C.-H. Huang, P. Sadayappan, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, 8th International Workshop (LCPC'95)*, volume 1033 of *Lecture Notes in Computer Science*, pages 254–268, Columbus, OH, USA, 1996. Springer-Verlag.

[16] H.G. Dietz, T.M. Chung, T. Mattox, and T. Muhammad. A synchronization and aggregate communication library for PAPERS clusters. Technical Report http://aggregate.org/TechPub/TR19950131/tr950131.html, School of Electrical Engineering, Purdue University, West Lafayette, IN, January 1995.

[17] H.G. Dietz, T.M. Chung, T.I. Mattox, and T. Muhammad. Purdue's Adapter for Parallel Execution and Rapid Synchronization: The TTL-PAPERS Design. Technical Report http://aggregate.org/TechPub/ICPP95/icpp95.html, School of Electrical Engineering, Purdue University, West Lafayette, IN, January 1995.

[18] H.G. Dietz, T. Muhammad, and T. Mattox. TTL Implementation of Purdue's Adapter for Parallel Execution and Rapid Synchronization. Technical Report http://aggregate.org/TechPub/super4.pdf, School of Electrical Engineering, Purdue University, West Lafayette, IN, December 1994.

[19] Henry G. Dietz, William E. Cohen, T. Muhammad, and Timothy I. Mattox. Compiler techniques for fine-grain execution on workstation clusters using PAPERS. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D.A. Padua, editors, *Languages and Compilers for Parallel Computing, 7th International Workshop (LCPC'94)*, volume 892 of *Lecture Notes in Computer Science*, pages 31–45, Ithaca, NY, 1995. Springer-Verlag.

[20] H.G. Dietz, T. Muhammad, J.B. Sponaugle, and T. Mattox. PAPERS: Purdue's Adapter for Parallel Execution and Rapid Synchronization. Technical Report TR-EE 94-11, School of Electrical Engineering, Purdue University, West Lafayette, IN, USA, March 1994.