Sparse Flat Neighborhood Networks (SFNNs): Scalable Guaranteed Pairwise Bandwidth & Unit Latency

Timothy I. Mattox, Henry G. Dietz, & William R. Dieter
ECE Department, University of Kentucky
453 F. Paul Anderson Tower, Lexington, KY 40506-0046
tmattox@engr.uky.edu, hankd@engr.uky.edu, dieter@engr.uky.edu

Abstract

Network performance for a particular application is determined by the latency and bisection bandwidth that are achieved for the set of specific communication patterns used by that application. The number of nodes with which each node might potentially communicate grows linearly as nodes are added, thus, network cost for large systems either becomes a large fraction of machine cost or performance suffers. However, performance-critical communication patterns commonly occurring in real parallel programs rarely require that each node directly communicate with every other node.

The number of node pairs actually communicating generally grows far slower than the expected $O(N^2)$. Thus, a carefully designed network for a massively parallel system can use relatively narrow switches while still providing single-switch latency and guaranteed pairwise bandwidth for performance-critical communications. This paper introduces Sparse Flat Neighborhood Networks (SFNNs), a variant of Flat Neighborhood Networks (FNNs) which are engineered from first principles to efficiently meet these detailed pairwise communication performance criteria.

1. Introduction

Parallel supercomputers, clusters, and server farms rely heavily on performance of a system area network to coordinate the actions of individual nodes. High performance computing systems with tens to tens of thousands of Processing Elements (PEs) have been built for many decades using a variety of network designs, and a vastly larger set of network designs can be found in the literature. Most research is focused on networks with endearing mathematical properties or simple routing schemes. In this paper, we suggest that network form should follow function: the best network is the design which, of all feasible networks, yields performance characteristics best matching the latency and bandwidth needs of your code while simultaneously satisfying the relevant cost, scalability, and reliability constraints. The problem in taking this seemingly obvious approach is that the design space is surprisingly large and complex.

The work presented in this paper does not consider the complete design space, but does search the relatively large and important space including all designs, symmetric or asymmetric, that could satisfy the constraints with single-switch latency. Even though only symmetric designs are commonly considered, for applications with complex communication patterns, the best topology is likely to be asymmetric. There are two reasons:

1. Even if only symmetric communication patterns are important to an application, the union of multiple symmetric patterns is generally asymmetric; thus, asymmetric communication patterns naturally occur in real scientific and engineering applications
2. The design space for asymmetric designs is far larger than the space for symmetric designs, with the corresponding result that the best asymmetric designs often are better than the best symmetric ones

The problem is simply that asymmetry is difficult for humans to manage and there were neither automated design tools nor the technology with which to efficiently implement them. The insights presented here, combined with genetic algorithm (GA) search technology, make Sparse Flat Neighborhood Networks (SFNNs) an extremely cost-effective and highly scalable way to achieve specific performance goals for user-specified communication patterns.

Section 2 defines and describes the properties of SFNNs, including several examples. The actual design process, as implemented by our tools, is described in Section 3. Although SFNNs can use ordinary routing procedures implemented within standard network hardware, Section 4 discusses the SFNN runtime support we developed to more efficiently utilize Ethernet-like networking. A brief conclusion appears in Section 5.

2. What Is An SFNN?

Although we plan to develop tools that will handle optimized design of fully general network topologies, we initially focused on the construction of design tools for an important and immediately useful class of networks that we named **Flat Neighborhood Networks (FNNs)**[2]. The FNN concept won a number of awards, including a 2000 Gordon Bell award honorable mention for price/performance[6], "Most Innovative Architecture" in the HPC Games Challenge at SC2000, and a Computerworld Smithsonian award as one of the six information technologies most advancing science in 2001. The FNN concept is based on a re-evaluation of the basic principles of interconnection networks. The ideal switched network should provide:

- Single hop latency for all pairwise communications.
- Full link bandwidth between any node pair independent of any other node pairs that may be communicating at the same time.

These properties are easily achieved using a network consisting of a single switch that has at least as many ports as there are nodes. However, in an FNN, the same properties can be extended to several times as many nodes as there are ports per switch.

Because using paths connecting one switch to another would introduce both additional latency and potential bandwidth bottlenecks, we ignore such paths for the purpose of satisfying the design constraints. An FNN is modeled as a non-recirculating single-stage network. The trick is to use multiple **Network Interfaces** (**NIs**, or **NICs** for NI Cards) in each node; thus, each node can be connected to multiple switches. The design problem is therefore determining to which switches each node should be connected.

The complexity of the FNN design problem is perhaps surprisingly very high. The problem is actually a minor variation on the well-known graph theory problem called **(v,k,t)-covering design**. The number of nodes corresponds to **v**, the number of ports per switch corresponds to **k**, and pairwise grouping implies that **t** would be equal to 2. The standard covering problem is still an open problem in mathematics. 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[14]. The FNN design problem differs from the standard covering problem primarily in that the number of network interfaces per node is constrained; there is no corresponding constraint on (v,k,t)-covering design. The FNN design problem also differs in that it incorporates optimization of a variety of more complex secondary characteristics, such as making the number of times each pair is covered (i.e., the conflict-free bandwidth between each pair) be tuned to approximate a specification of relative bandwidth requirements.

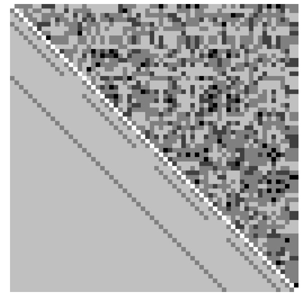

Figure 1. KLAT2's FNN Design/Solution Map

As our FNN design tools have matured, we also have developed better ways to graphically represent both design constraints and their solutions. 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.

The network in the world's first FNN-based supercomputer, **KLAT2 (Kentucky Linux Athlon Testbed 2)**[3], as built in Spring 2000, corresponds to the design/solution map shown in Figure 1. 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 (i.e., 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 than two units bandwidth per pair. This 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, viewing 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 is the basic concept of a **Sparse Flat Neighborhood Network (SFNN)**. An SFNN is an FNN, except the FNN property is ensured not for all node pairs, but only for explicitly noted node pairs. For node pairs not explicitly selected, communications would still be possible through intermediary nodes, or through additional switch-to-switch connections.

The first SFNN-based supercomputer is **KASY0 (Kentucky ASYmmetric zero)**, a 128-node cluster which was built in the Summer of 2003. The design specification and solution for KASY0's SFNN are shown in Figure 2.

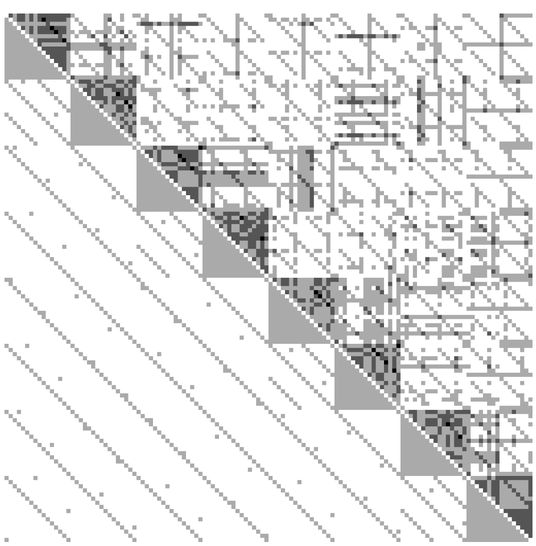

Figure 2. KASY0's SFNN Design/Solution Map

Although KASY0's network also is over designed, it is far more effective than KLAT2's network in placing bandwidth where it will be used. The "sparseness" of the design specification and of the solution do not mean that communications between node pairs represented by white pixels are impossible, rather, it implies that those pairs do not communicate enough to justify reserving bandwidth and ensuring single-hop latency. Thus, in the rare event that communications between such pairs do occur, latency may be somewhat more than a single hop and the available bandwidth may be dependent on the other traffic in the network at that time.

The result of this new design technology is that KASY0's 128-node SFNN cost only $4,200 – about half as much as KLAT2's 64-node FNN. One might be tempted to think that the price difference is due to network hardware price drops in the three-year period between construction of KLAT2 and KASY0, but the primary difference actually is the use of 3 NICs per node and 24-port switches in KASY0 instead of 4 NICs per node and more expensive 32-port switches as used in KLAT2.

Clearly, the concept of covering all desired communication patterns with single-hop latency reserved bandwidth has limited scalability. We are currently working on design tools that can maximize coverage without requiring that complete coverage of a design specification be achieved, and these new **Fractional Flat Neighborhood Network (FFNN)** design tools will remove all scaling issues and even can be used to optimize the design of more conventional networks, such as trees and fat trees. Of course, an FFNN does not have as desirable properties as an SFNN. Thus, the question we consider here is: how large a system can SFNN technology scale to?

2.1. Scalability

Although FNNs have proven very useful for moderate-sized systems, such as KLAT2, the most important property of SFNN technology is the fact that the design methodology allows scaling to very large systems. Using commodity networking components, we have created SFNN solutions for design problems with as many as 10,000 nodes. However, the primary reason that SFNNs scale is not really a property of SFNNs, but of communication patterns. A communication pattern defines a set of communicating pairs, most often, a *1:1* and *onto* mapping (*permutation*) of nodes onto nodes.

It is a simple fact of arithmetic that the number of other nodes each node might need to talk to grows linearly with the number of nodes. More precisely, in a system with *N* nodes, the number of possible communication pairs involving each node is 2(N-1) for unidirectional (ordered) pairs or *N-1* for bidirectional (unordered) pairs. This also corresponds to the well-known fact that a direct-connection (switch-less, single-link latency) network would require N(N-1) unidirectional wires or (N(N-1))/2 bidirectional links. Thus, network complexity seems to scale as $O(N^2)$. However, as we shall demonstrate in this paper by reviewing the various communication patterns commonly discussed in the parallel processing literature, most parallel programs require high performance on only a small fraction of the possible pairs for each node. As discussed in the following sections, the fraction actually used in typical applications sharply decreases as the number of nodes is increased!

O(1) Scaling Patterns. It is ironic that, despite the parallel processing community's concerns about the complexity of large scale networks, the most commonly used communication patterns in parallel programs are all patterns that have a constant number of pairs per node independent of the number of nodes. This observation is confirmed by the fact that many of the largest systems built, such as the Intel Paragon, Cray T3D, and ASCI RED, have successfully used networks with a simple 3D mesh topology.

The number of bidirectional communication pairs in which each node is involved when communicating with adjacent nodes within a 1D mesh is either one or two. For a toroidal 1D mesh (i.e., a ring), each node participates in two bidirectional communication pairs; a non-toroidal 1D mesh differs only in that the two end nodes participate in just one pair each. The per-node pair count is entirely unaffected by the total number of nodes.

The design space becomes significantly larger when 2D meshes are considered, because there may be multiple ways to factor the nodes into a 2D mesh. For example, a 32-node system could be viewed as 2x16, 4x8, 8x4, or 16x2. However, once a factorization is selected, the pair count per node is independent of the total number of nodes. Communicating with nodes that are adjacent by row or column yields between two and four pairs per node, with edge nodes in non-toroidal meshes having the lower pair counts. Including diagonally adjacent nodes simply changes the count to three to eight pairs, again with the lower values corresponding to edge nodes in non-toroidal meshes. Eight pairs may be a large fraction of all possible pairs in a small cluster, but it becomes a vanishingly small fraction of all possible pairs as the system design is scaled to thousands of nodes. The same is true of higher-dimensionality mesh adjacency pairs.

The conceptually most complex communication patterns commonly used in parallel programs are typically patterns consisting of a single pair per node. For example, the *bit-reversal* communication pattern[5] may employ a reasonably complex formula to determine which node each node will communicate with, but there is only a single pair involving each node. The same is true of *shuffle* [9]; it is also significant that, given bidirectional links, *inverse-shuffle* is implemented by the exact same pairing that *shuffle* uses. Notice further that all these patterns are permutations, so, with an appropriate network, each pattern can be implemented in a single message time-step.

O(LogN) Scaling Patterns. In addition to *O(1)* scaling patterns, nearly all programs contain some communication patterns that scale as *O(LogN)*. Fundamentally, *O(LogN)* 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, and Aggregate Function Network (AFN) hardware implements them directly within the network[7], 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 node communicates with the nodes whose numbers differ from the source node number by only a single bit position's value in the binary representation. For example, in a 32-node system, node 5 (binary 00101) would be paired with 1 (00001), 4 (00100), 7 (00111), 13 (01101), and 21 (10101). Clearly, there are no more than *ceiling*($Log_2 N$) bit positions, so there are at most that many nodes differing from any given node's number by precisely one bit position, and the number of pairs per node grows as *O(LogN)*.

It is important to note that many message-passing systems differentiate between what MPI[13] calls *all-reduce* and *reduce*. An *all-reduce* would suggest that all nodes should have their complete tree, whereas a *reduce* requires only the tree rooted at a specific point (typically, node 0). Thus, *reduce* can be implemented using an average of half as many pairs per node, although the root node will still require the full set of pairs. If an *all-reduce* is implemented using a reduce followed by a *broadcast* from the root node, rather than by directly performing N *reduces* simultaneously, then the complete tree of pairs is only needed for the root node. This is significant in that *reduce* is far more common than *all-reduce* in parallel programs.

O($N^{1/D}$) Scaling Patterns. D-dimensional *scatter*, *gather*, and *personalized all-to-all* communications involve each node interacting with every other node in its dimension. In a 2D space, each node would need to be paired with every node in the same row or column, yielding *O*($sqrt(N)$) pairs per node. The 3D case scales pairs per node by the cube root of N. The 1D case is the worst; all nodes are in the same dimension.

Superficially, it seems that all N-1 pairs are needed for each node in order to support 1D *personalized all-to-all*. However, such a communication pattern is not able to be accomplished in a single time-step unless N-1 messages can be simultaneously output by each node. With fewer than N-1 NICs, this is literally 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 would then be repackaged and retransmitted until each node 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. 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 that will be 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 your particular MPI.

The result is that these compound patterns are usually able to be efficiently implemented using primitive patterns that scale as *O(1)* or *O(LogN)*. Thus, the number of pairs per node scales approximately as *O(LogN)*, not as *O(N)*.

Pair Synergy. A set of parallel programs will generally require not just one communication pattern, but the union of all communication patterns used in any of the programs. Thus, it is natural to expect that the number of pairs for each node is actually the sum of the number of pairs for that node in each communication pattern. Of course, the sum of a fixed number of *O(1)* and *O(LogN)* values grows no faster than *O(LogN)*. However, the sum could still become too large for commodity switches.

In practice, the sum of pairs is a rarely-seen upper bound. A pair required by one pattern very often also is required by another pattern.

This type of synergy is very common among various mesh patterns and even reductions. For example, nearly all pairs required to support 1D adjacency also are required to support 2D adjacency; only 1D pairs involving nodes in edge positions in the 2D pattern are not covered by 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.

SFNNs are not the only type of network that can take advantage of this insight. For example, direct connection networks also can be made much simpler. However, the relatively high degree of switches (e.g., 24, 32, 48, 64, or even more ports) makes it feasible to cover design requirements for systems that are much larger than can be accommodated using the relatively low degree (e.g., 5 or 6 NICs) afforded by direct connections between nodes.

A Larger Example. The result of all the above is that SFNN networks for very large systems, covering a wide range of communication patterns, can be built using commodity network hardware. The largest such designs we have thus far created contain over 10,000 nodes. The primary limitation in making larger designs appears to be the inefficiency of the design tool itself in dealing with very large designs. The concepts appear to allow much larger systems to be built and improvements to our design tool already have been responsible for the jump from a maximum design size of about 1,000 nodes to over 10,000 using the same network hardware components. It is impractical to use a 10,000x10,000 design map scaled to fit within the margins of this paper, so our large design example is a 1,024-node design, shown in Figure 3. This design uses 48-port switches to cover all of the following:

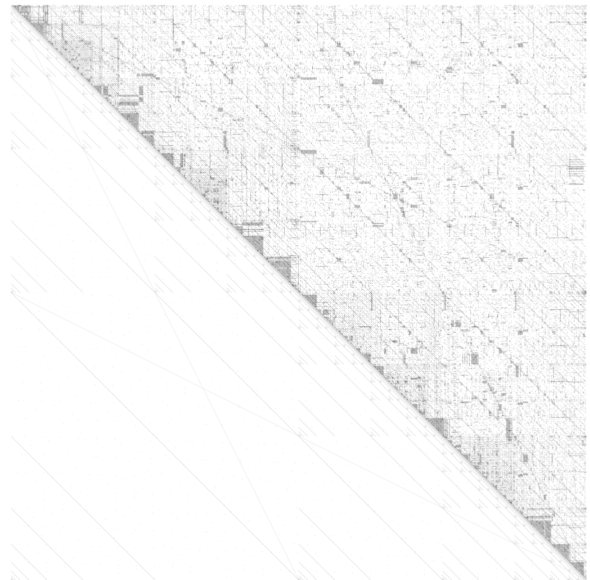

Figure 3. A 1,024-node Design/Solution Map

- 1D torus power of 2 offsets
- 2D torus power of 2 offsets in 512x2, 256x4, 128x8, 64x16, and 32x32 topologies
- 3D torus power of 2 offsets in 256x2x2, 128x4x2, 64x8x2, 64x4x4, 32x16x2, 32x8x4, 16x16x4, and 16x8x8 topologies
- Shuffle & inverse-shuffle
- Bit-reversal
- Hypercube
- 2D matrix transpose (32x32, one element per node)

The total number of possible pairs is 523,776. The union of all the specified patterns requests coverage of just 14,566 pairs; approximately 2.78% of all possible pairs! The actual design covers 102,883 pairs, or 19.6%, including all 14,566 requested pairs.

This system design requires 4,848 NICs (assuming no NIs are built-into the motherboards) and 101 switches. The total street cost for the network would thus be less than

$100,000 using name-brand 100 Mb/s Fast Ethernet hardware (~$500 per switch, ~$10 per NI+cable). Latency would be around 30 microseconds (i.e., a single Fast Ethernet switch delay) for any communication pattern that is covered. Bisection bandwidth measured on any permutation pattern covered would be greater than 102.4 Gb/s, with a total network bandwidth of no more than 484.8 Gb/s. If these performance numbers are not sufficient, *any* networking technology could be substituted for Fast Ethernet – Gb/s Ethernet, Infiniband, etc. – in which case a new design optimized for the constraints imposed by the networking technology should be created. Note that SFNN designs primarily use multiple NIs per node for connectivity reasons, and thus only require a node's motherboard to support full-bandwidth on a single NI at any one time.

Two properties are immediately visible from this moderately large example. As predicted by the above discussion, the design constraints are indeed very sparse despite covering a rich set of communication patterns. The second property immediately visible is that the design solution, though quite sparse, is significantly less sparse than the design constraints. It is not clear why the solution has a higher density: is the higher density necessary or is it the result of the design tool settling for a less-than-minimal solution? The answer seems to be a mix of both. As our SFNN design tool has improved, the density of the solutions found has significantly decreased (and the design sizes the tool can handle with similar network components have correspondingly increased). However, because an SFNN must completely cover the design constraint pairs, it is highly likely that some (small) fraction of the pairs only can be covered with relatively loose-fitting solutions. Our ongoing research into FFNNs already has shown that very small switches can be used to cover the vast majority of design constraint pairs without including a significant number of pairs outside the design constraints.

2.2. Asymmetry

It is significant that the vast majority of SFNN design solutions, including all three shown above, are asymmetric. In fact, by enumeration of all possible designs for very small design problems, it easily can be shown that the best asymmetric design is usually better than the best symmetric design for most sets of design constraints. Symmetric design constraints are usually more efficiently covered with somewhat asymmetric solutions. Just as the union of symmetric patterns generally is asymmetric, a cover constructed symmetrically using switches of a fixed width is unlikely to have precisely the same symmetry as the design constraints; thus, a symmetric cover would needlessly include additional pairs. Interestingly, many designers confuse symmetry with modularity; asymmetric designs easily can be constrained to have a specified modular structure without requiring that the complete design be symmetric. In summary, asymmetric design may require powerful design tools, but the benefits can be huge.

2.3. Multiprocessor Nodes

The above discussion assumes a single processor per node. However, assuming multiple processors per node simply makes the communication pair requirements for each node be computed using the sum of the processor communication requirements for all processors within that node. In itself, this tends to yield somewhat denser specifications, making the benefit in using multiprocessor nodes unclear: contrary to popular belief, multiprocessor nodes do not necessarily reduce network cost per processor. Because interprocessor communication within a node does not require use of an external network, it would be possible for an SFNN design tool to incorporate renumbering of processors within nodes as a mechanism to help minimize the external network hardware required to cover a design specified by processor pairs. Rather than numbering processors within nodes sequentially, a future SFNN design tool for systems with multiprocessor nodes might number processors in a random-looking order that minimizes the external network hardware needed to cover all specified pairs.

3. SFNN Design

A primary motivator for the development of SFNNs was the realization that Universal FNNs supported many more communication patterns than were needed. By designing for the known needed communication patterns, the network could be much less expensive to construct, yet still yield comparable performance for a given set of applications. To design SFNNs, it is necessary to specify the set of communication patterns that we wish the SFNN to support with guaranteed pairwise bandwidth and unit latency. This set can be represented by an NxN weighted connectivity matrix for N PEs, with the value at matrix element (x,y) being the relative importance of the communications between PE x with PE y. If the weights are restricted to zero and one, the matrix is in the same form as an adjacency matrix; two PEs are considered adjacent if they are connected to a common switch. There are three approaches for determining these connectivity matrices.

3.1. Specification Of Performance Constraints

One can search research publications to identify the complete set of communication patterns that are discussed. 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 alternative is to construct the 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 communications in the way that one might expect. A third approach, which we are developing primarily for FFNN design, involves automated determination of communication patterns using instrumented runs of target applications.

The current SFNN design 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 1 entry in the matrix, every other pair yields a 0. 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 Grids or Tori with various sub-patterns independently selectable:
  - Distance 1 offsets in X, Y, or Z
  - Distance 1 diagonals in 2D and 3D
  - Power of 2 offsets in X, Y, or Z
  - All PEs that differ in only one dimension (e.g. every PE in same row, column, etc.)

For 2D and 3D grids/tori, one can select to use just one balanced factorization of N, or all unique 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.

3.2. The Design Algorithm

Although SFNN design appears to be harder than the unsolved (v,k,t)-covering problem, good solutions can be created using a **Genetic Algorithm (GA)** very similar to that we developed for FNN design [2]. The genome representation used in both the FNN and SFNN GAs consists of a fixed-length bit string representing the set of PEs connected to each switch. The crossover and mutation operations are specially coded to ensure various invariant properties are met by all population members: the number of PEs, the maximum number of switches, the maximum number of NICs per PE, and the maximum number of connections per switch. This saves considerable time because the GA's metric evaluation is merely ranking feasible network designs, not also checking their feasibility. However, the GA would be slow to find FNNs, and even slower to find SFNNs, without a few extra tricks. The FNN design tool attempts to speed its search by first finding approximate solutions to a scaled-down problem and then scaling those solutions up to seed the full problem's initial population; the SFNN design tool also seeds its initial population with good designs, but by using a greedy "buddy connection" algorithm rather than scaling the problem.

3.3. Related Work In Network Design

There is a huge body of literature on interconnection network designs for parallel supercomputers, but very little on the design process.

Traditionally, the emphasis has been on selecting a "universal" topology with good mathematical properties and then mapping program communications onto that network architecture. Various mesh and hypercube architectures have been very popular in more traditional supercomputers, while cluster network designs most often are either trees or fat trees [11], perhaps also employing trunking or channel bonding [15] to enable use of wider, but lower-speed, switches. All of these network topologies share the property that they have straightforward scaling procedures, use relatively low-degree nodes, and require multiple routing hops to accomplish most communication patterns. Mesh and hypercube designs offer very simple routing algorithms and relatively easy use of alternative paths for load balancing or fault recovery. Trees are fully compatible with commodity network hardware. Fat trees offer the ability to maintain bisection bandwidth as the design is scaled, but at the cost of requiring routers rather than "dumb" switches to load balance across alternative paths through he thick portion of the tree. None of these designs directly addresses issues of latency for user-specified communication patterns.

The design process for Circulant graphs and other Caley graphs[17] is becoming a popular topic in interconnection research due to the low latency these graphs afford by using relatively high-degree nodes. However, this design process is again oblivious to performance requirements for user-specified communication patterns. The work presented in this paper is closely related to, and based on, our earlier work using genetic algorithms to design FNNs [2] which may be either symmetric or asymmetric. At the same time we were building KLAT2 (Spring 2000), a group at Australia's National University by hand created a symmetric FNN for the Bunyip supercomputer[1]. The work on filtering random graphs published in Fall 2002 by Lakamraju et al [10] is closely related to the FNN approach, but use of a random search is far less efficient than the genetic algorithm approach we have taken, they restricted their consideration to regular graphs of high-degree nodes, and the metrics that they considered do not directly correspond to performance on a user-specified set of communication patterns.

4. SFNN Runtime support

FNN and SFNN routing is closely related to Linux Channel Bonding [15], however, it is complicated by the fact that the set of NIs used is a function of the destination of each message. Another important difference is that it is often desirable to use uplink connections between switches for an SFNN to route messages between node pairs not covered by the design, which prohibits the duplication of MAC addresses (as used in channel bonding).

Our driver assigns each NI a unique local MAC address such that a single table lookup of a 32-bit entry determines all routes to be used between a particular pair of nodes. Our MAC-level mechanism can be used with both standard IP and technology-specific libraries; e.g., a GAMMA[4] interface could further reduce Gb/s Ethernet latency. Our reassignment of MAC addresses somewhat complicates the network boot process, which we have resolved by incorporating support for our drivers in *Warewulf[16].*

5. Conclusion

When we first demonstrated FNNs in KLAT2 in Spring 2000, the general reaction to the technology was strongly positive. Within a short time, the technology had won various awards and was being applied by both commercial[12] and academic[8] users. However, we knew that universal FNNs were just the "low-hanging fruit" of a much larger crop: the ability to automatically engineer an interconnection network to meet arbitrary performance criteria. SFNNs are a giant leap along this path, not just because they scale to much larger designs and are even more cost effective than FNNs, but because they represent the first time high-performance network design has been directly driven by a purely quantitative specification. Perhaps even more significantly, our support software now makes the fact that an SFNN is being used entirely transparent; it can be built using standard network hardware and even message-passing library routines need not know an SFNN is being used. The SFNN concept, and the tools developed to exploit it, allow a supercomputer systems designer more flexibility and control over the cost/performance trade-offs of the network for a suite of target applications.

References

[1] Douglas A. Aberdeen, Jonathan Baxter, and Robert Edwards. 98 cents/Mflops/s, Ultra-Large-Scale Neural-Network Training on a PIII Cluster. In *IEEE/ACM SC2000 Gordon Bell Prize Award in the Price/Performance category*, Dallas, Texas, November 2000.

[2] H.G. Dietz and T.I. Mattox. Compiler Techniques For Flat Neighborhood Networks. In *13th International workshop on Languages and Compilers for Parallel Computing*, pages 239–254, IBM Watson Research Center, Yorktown, New York, August 2000.

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

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

[5] 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.

[6] Th. Hauser, T.I. Mattox, R.P. LeBeau, H.G. Dietz, and P.G. Huang. High-Cost CFD on a Low-Cost Cluster. In *IEEE/ACM SC2000 Gordon Bell Prize Honarable Mention in the Price/Performance category*, Dallas, Texas, November 2000.

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

[8] KAGe. http://www.esci.keele.ac.uk/geophysics/kage/.

[9] 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.

[10] 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.

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

[12] LinuxLabs. http://www.linuxlabs.com/nimbus.html.

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

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

[15] 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.

[16] Warewulf. http://warewulf-cluster.org/.

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