Compiler Techniques For Flat Neighborhood Networks

H. G. Dietz and T. I Mattox

College of Engineering, Electrical Engineering Department
University of Kentucky
Lexington, KY 40506-0046
http://aggregate.org/
hankd@engr.uky.edu

Abstract. A Flat Neighborhood Network (FNN) is a new interconnection network architecture that can provide very low latency and high bisection bandwidth at a minimal cost for large clusters. However, unlike more traditional designs, FNNs generally are not symmetric. Thus, although an FNN by definition offers a certain base level of performance for random communication patterns, both the network design and communication (routing) schedules can be optimized to make specific communication patterns achieve significantly more than the basic performance. The primary mechanism for design of both the network and communication schedules is a set of genetic search algorithms (GAs) that derive good designs from specifications of particular communication patterns. This paper centers on the use of these GAs to compile the network wiring pattern, basic routing tables, and code for specific communication patterns that will use an optimized schedule rather than simply applying the basic routing.

1 Introduction

In order to use compiler techniques to design and schedule use of FNNs, it is first necessary to understand precisely what a FNN is and why such an architecture is beneficial. Toward that, it is useful to briefly discuss how the concept of a FNN arose. Throughout this paper, we will use KLAT2 (Kentucky Linux Athlon Testbed 2), the first FNN cluster, as an example. Though not huge, KLAT2 is large enough to effectively demonstrate the utility of FNNs: it unites 66 Athlon PCs using a FNN consisting of 264 NICs (Network Interface Cards) and 10 switches.

There are two reasons that the processors of a parallel computer need to be connected: (1) to send data between them and (2) to agree on global properties of the computation. As we discussed in [1], the second functionality is not well-served using message-passing hardware. Here, we focus on the first concern. Further, we will restrict our discussion to clusters of PCs, since few people will have the opportunity to design their own traditional supercomputer's network.

In the broadest terms, we need to distinguish only six different classes of network topologies (and two minor variations on the last). These are shown in Fig. 1.

The ideal network configuration would be one in which each processor is directly connected to every other node, as shown in Fig. 1a. Unfortunately, for an N-node system this would require N-1 connections for each node. Using standard motherboards and NICs, there are only bus slots for a maximum of 4-6 NICs. Using relatively expensive 4-interface cards, the upper bound could be as high as 24 connections; but even that would not be usable for a cluster with more than 25 nodes.

Accepting a limit on the number of connections per node, direct connections between all nodes are not possible. However, it is possible to use each node as a switch in the network, routing through nodes for some communications. In general, the interconnection pattern used is equivalent to some type of hypermesh; in Fig. 1b, a degree 2 version (a ring) is pictured. Because NICs are generally cheaper than switches, this structure minimizes cost, but it also yields very large routing delays – very high latency.

To minimize latency without resorting to direct connections, the ideal network would connect a high-bandwidth NIC in each node to a single wire-speed switch, as shown in Fig. 1c. For example, using any of the various Gb/s network technologies (Gigabit Ethernet [2], Myricom's Myrinet [3], Giganet's CLAN [4], Dolphin's SCI [5]), it is now possible to build such a network. Unfortunately, the cost of a single Gb/s NIC exceeds the cost of a typical node, the switch is even more expensive per port, and wide switches are not available at all. Most Gb/s switches are reasonably cheap for 4 ports, expensive for 8 or 16 ports, and only a few are available with as many as 64 ports. Thus, this topology works only for small clusters.

The closest scalable approximation to the single switch solution substitutes a hierarchical switching fabric for the single switch, as shown in Fig. 1d. Some Gb/s technologies allow more flexibility than others in selecting the fabric's internal topology; for example, Gb/s Ethernet only supports a simple tree whereas Giganet CLAN can use higher-performance topologies such as fat trees – at the expense of additional switches. However, any switching fabric will have a higher latency than a single switch. Further, the bisection bandwidth of the entire system is limited to the lesser of the bisection bandwidth of the switch(es) at the top of the tree or the total bandwidth of the links to the top switch(es). This is problematic for Gb/s technologies because the "uplinks" that interconnect switches within the fabric are generally the same speed as the connections used for the NICs; thus, half of the ports on each switch must be used for uplinks to achieve the maximum bisection bandwidth.

Fortunately, 100Mb/s Ethernet switches do not share this last problem: wire-speed 100Mb/s switches often have Gb/s uplinks. Thus, it is possible to build significantly wider switch fabrics that preserve bisection bandwidth at a relatively low cost. The problem is that a single 100Mb/s NIC per node does not provide enough bandwidth for many applications. Fig. 1e shows the standard

Fig. 1a. Direct connections

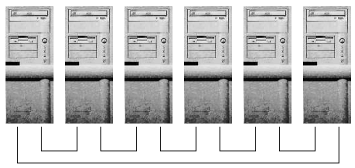

Fig. 1b. Switchless connections

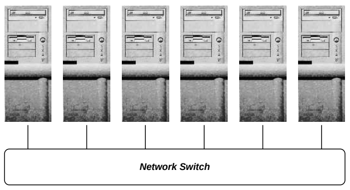

Fig. 1c. Ideal switch

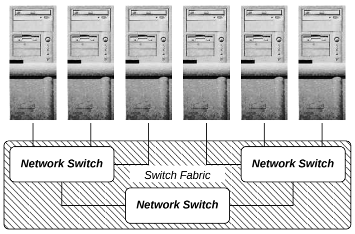

Fig. 1d. Switch fabric

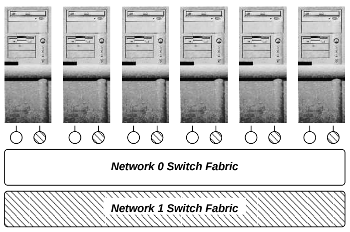

Fig. 1e. Channel bonding

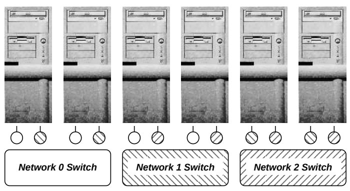

Fig. 1f. FNN

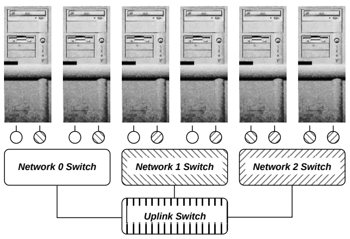

Fig. 1g. FNN with uplink switch

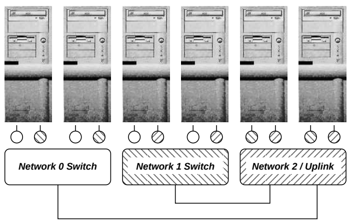

Fig. 1h. FNN with folded uplink

Fig. 1. Network topologies used in connecting cluster nodes

Linux-supported solution: use multiple NICs per node, connect them to identical fabrics, and treat the set of NICs in each node as a single, parallel, NIC. The software support for this, commonly known as “channel bonding,” was the primary technical contribution of the original Beowulf project. Unfortunately, the switch fabric latency is still high and building very large clusters this way yields the same bisection bandwidth and cost problems discussed for Gb/s systems built as shown in Fig. 1d. Further, because channel-bonded NICs are treated as a single wide channel, the ability to send to different places by simultaneously using different NICs is not fully utilized.

The Flat Neighborhood Network (FNN), shown in Fig. 1f, solves all these problems. Because switches are connected only to NICs, not to other switches, single switch latency and relatively high bisection bandwidths are achieved. Cost also is significantly lower. However, FNNs do cause two problems. The first is that some pairs of nodes only have single-NIC bandwidth between them with the minimum latency, although extra bandwidth can be obtained with a higher latency by routing through nodes to “hop” neighborhoods. The second problem is that routing becomes a very complex issue.

For example, the first two machines in Fig. 1f have two neighborhoods (subnets) in common, so communication between them can be done much as it would be for channel bonding. However, that bonding of the first machine’s NICs would not work when sending a message to the third machine because those nodes share only one neighborhood. Even without the equivalent of channel bonding, routing is complicated by the fact that the apparent address (NIC) for the third machine is different depending on which node is sending to it; the first machine would talk to the first NIC in the third node, but the last machine would talk to the second NIC. Further, although this very small example has some symmetry which could be used to simplify the specification of routing rules, that is not generally true of FNNs.

At this point, it is useful to abstract the fully general definition of a FNN: a network using a topology in which all important (usually, but not necessarily, all) point-to-point communication paths are implemented with only a single switch latency. In practice, it is convenient to augment the FNN with an additional switch that connects to the uplinks from the FNN’s switches, since that switch can provide more efficient multicast support and I/O with external systems (e.g., workstations or other clusters). This second-level switch also can be a convenient location for “hot spare” nodes to be connected. The FNN with this additional uplink switch is shown in Fig. 1g.

In the special case that one of the FNN switches has sufficient ports available, it also is possible to “fold” the uplink switch into one of the FNN switches. This folded uplink FNN configuration is shown in Fig. 1h. Although the example’s 4-port switches would not be wide enough to be connected as shown in this figure, if the switches are wide enough, it always is possible to design the network so that sufficient ports are reserved on one of the FNN switches.

Thus, FNNs scale well, easily provide multicast and external I/O, and offer high performance at low cost. A more detailed evaluation of the performance of FNNs (especially in comparison to fat trees and channel bonding) is given in [9]. Independent of and concurrent with our work, a group at the Australian National University created a cluster (*Bunyip* [10]) whose network happens to have the FNN properties, and their work confirms the performance benefits.

The problem with FNNs is that they require clever routing. Further, their performance can be improved by tuning the placement of the paths with extra bandwidth so that they correspond to the communication patterns that are most important for typical applications. In other words, FNNs require compiler technology for analysis and scheduling (routing) of communication patterns in order to achieve their full performance. Making full use of this technology for both the design and use of FNNs yields significant benefits.

Bunyip [10] uses a hand-designed symmetric FNN. However, the complexity of the FNN design problem explodes when a system is being designed with a set of optimization criteria. Optimization criteria range from information about relative importance of various communication patterns to node physical placement cost functions (intended to reduce physical wiring complexity). Further, many of these criteria interact in ponderous ways that only can be evaluated by partial simulation of potential designs. It is for these reasons that our FNN design tools are based on genetic search algorithms (GAs).

2 The FNN Compiler

The first step in creating a FNN system is the design of the physical network. Logically, the design of the network is a function of two separate sets of constraints: the constraints imposed by physical hardware and those derived from analysis of the communications that the resulting FNN is to perform. Thus, the compiler's task is to parse specifications of these constraints, construct and execute a GA that can optimize the design according to these constraints, and finally to encode the resulting design in a form that facilitates its physical construction and use.

The current version of our network compiler uses:

- A specification of how many PCs, the maximum number of NICs per PC (all PCs do not have to have the same number of NICs!), and a list of available switches specified by their width (number of ports available per switch). Additional dummy NICs and/or switches are automatically created within the program to allow uneven use of real NICs/switch ports. For example, KLAT2's current network uses only 8 of 31 ports on one of its switches; the other switch ports appear to be occupied by dummy NICs that were created by the program.
- A designer-supplied evaluation function that returns a quality value derived by analysis of specific communication patterns and other performance measures. This function also marks problem spots in the proposed network configuration so that they can be preferentially changed in the GA process.

In the near future, we expect to distribute a version of the compiler which has been enhanced to additionally include:

– A list of switch and NIC hardware costs, so that the selection of switches and NIC counts also can be automatically optimized.
– A clean language interface for this specification.

Currently, we modify the GA-based compiler itself by including C functions that redefine the search parameters.

2.1 The GA Structure

The GA is not a generic GA, but is highly specialized to the problem of designing the network. The primary data structure is a table of bitmasks for each PC; each PC's bitmask has a 1 only in positions corresponding to each neighborhood (switch) to which that PC has a NIC connected. This data structure does not allow a PC to have multiple NICs connected to the same neighborhood, however, such a configuration would add nothing to the FNN connectivity. Enforcing this constraint and the maximum number of NICs greatly narrows the search space.

Written in C, the GA's bitmask data structure facilitates use of SIMD-within-a-register parallelism [6] when executed on a single processor. It also can be executed in parallel using a cluster. KLAT2's current network design was actually created using our first Athlon cluster, Odie – four 600MHz Athlon PCs.

To more quickly converge on a good solution, the GA is applied in two distinct phases. Large network design problems with complex evaluation functions 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 will make the full search converge faster.

The simplified cost weighting only values basic FNN connectivity, making each PC directly reachable from every other. The problem is made smaller by dividing both the PC count and the switch port counts by the same number while keeping the NICs per PC unchanged. For example, a design problem using 24-port switches and 48 PCs is first scaled to 2-port switches and 4 PCs; if no solution is found within the alloted time, then 3-port switches and 6 PCs are tried, then 4-port switches and 8 PCs, etc. 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.

If no solution is found for any of the scaled-down problems, the GA is directly applied to the full-size problem.

2.2 The Genetic Algorithm Itself

The initial population for the GA is constructed for the scaled-down problem using a very straightforward process in which each PC's NICs are connected to the lowest-numbered switch that still has ports available and is not connected to the same PC via another NIC. Additional dummy switches are created if the process runs out of switch ports; similarly, dummy NICs are assigned to virtual PCs to absorb any unused real switch ports. The resulting scaled-down initial FNN design satisfies all the constraints except PC-to-PC 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 checks only connectivity, not switch port usage, NIC 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 PCs is a simple matter of bitwise AND of the two bitmasks; counting the ones in that result measures the available bandwidth. Which higher-level 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. Since we cannot know what the precise optimum design's value will be for 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 PC and making its set of NIC connections match those for that PC in the other parent. This forced match process works by exchanging NIC connections with other PCs (which may be real or dummy PCs) in the child that had the desired NIC connections. Thus, the resulting child has properties taken from both parents, yet always is a complete specification of the NIC 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 the number of NICs/PC and the number of ports for each switch.

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

1. The first mutation technique swaps individual NIC-to-switch connections between PCs selected at random.
2. The second mutation technique simply swaps the connections of one PC with those of another PC, essentially exchanging PC numbers.

Thus, the mutation operators are also closed and preserve the basic NIC and switch port 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.

2.3 The FNN Compiler's Output

The output of the FNN compiler is simply a table. Each line begins with a switch number followed by a `:`, which is then followed by the list of PC numbers connected to that switch.

This list is given in sorted order, but for ideal switches, it makes no difference which PCs are connected to which ports, provided that the ports are on the correct switch. It also makes very little difference which NICs within a PC are connected to which switch. However, to construct routing tables, it is necessary to know which NICs are connected to each switch, so we find it convenient to also order the NICs such that, within each PC, the lowest-numbered NIC is connected to the lowest-numbered switch, etc.

We use this simple text table as the input to all our other tools. Thus, the table could be edited, or even created, by hand.

3 The FNN Translators

Once the FNN compiler has created the network design, there are a variety of forms that this design must be translated into in order to create a working implementation. For this purpose, we have created a series of translators.

3.1 Physical Wiring

One of the worst features of FNNs is that they are physically difficult to wire. This is because, by design, they are irregular and often have very poor physical locality between switches and NICs. Despite this, wiring KLAT2's PCs with 4 NICs each took less than a minute per cable, including the time to neatly route the cables between the PC and the switches.

The trick that allowed us to wire the system so quickly is nothing more than color-coding of the switches and NICs. As described above, all the ports on a switch can be considered interchangeable; it doesn't matter which switch port a NIC is plugged into. Category 5 cable, the standard for Fast Ethernet, is available in dozens of colors at no extra cost. Thus, the problem is simply how to label the PCs with the appropriate colors for the NICs it contains.

For this purpose, we created a simple program that translates the FNN switch connection representation into an HTML file. This file, which can be loaded into any WWW browser and printed, contains a set of per-PC color-coded labels that have a color patch for each NIC in the PC showing which color cable, and hence which switch, should be connected. KLAT2's wiring, and the labels that were used to guide the physical process, are shown in Fig. 2.

For KLAT2, it happens that half of our cables were transparent colors; the transparent colors are distinguished from the solid colors by use of a double triangle. Of course, a monochrome copy of this paper makes it difficult to identify specific colors, but the color-coding of the wires is obvious when the color-coded labels are placed next to the NICs on the back of each PC case, as you can see them in the photo in Fig. 2.

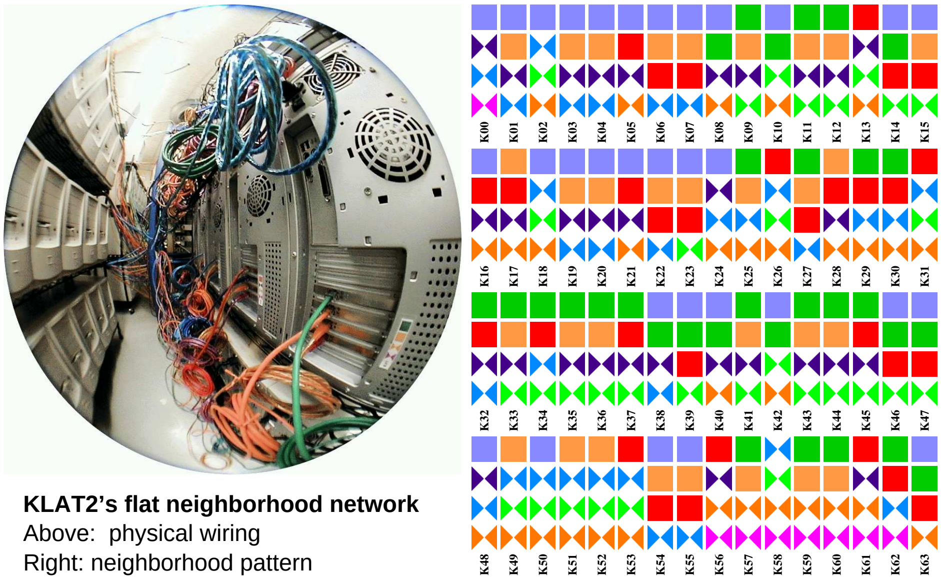

Fig. 2. FNN wiring of KLAT2's 64 PCs with 4 NICs each

3.2 Basic Routing Tables

In the days of the Connection Machines (CMs), Thinking Machines employees often could be heard repeating the mantra, "*all* the wires *all* the time." The same focus applies to FNN designs: there is tremendous bandwidth available, but only when all the NICs are kept busy. There are two ways to keep all the wires busy. One way is to have single messages divided into pieces sent by all the NICs within a PC, as is done using channel bonding. The other way is to have transmission of several messages to different destinations overlap, with one message per NIC. Because FNNs generally do not have sufficient connectivity to keep all the wires busy using the first approach, the basic FNN routing centers on efficient use of the second.

Although IP routing is normally an automatic procedure, the usual means by which it is automated do not work well using a FNN. Sending out broadcast requests to find addresses is an exceedingly bad way to use a FNN, especially if an uplink switch is used, because that will make all NICs appear to be connected rather than just the ones that share subnets. Worse still, some software systems, such as LAM MPI [7, 8], try to avoid the broadcasts by determining the locations of PCs once and then passing these addresses to all PCs. That approach fails because each PC actually has several addresses (one per NIC) and the proper one to use depends on which PC is communicating with it. For example, in Fig. 1f, the first PC would talk to the third PC via its address on subnet 1, but the last PC would talk to it via the address on subnet 3. Thus, we need to construct a unique routing table for each PC.

To construct these routing tables, we must essentially select one path between each pair of PCs. According to the user-specified communication patterns, some PC-to-PC paths are more important than others. Thus, the assignments are made in order of decreasing path importance. However, the number of alternative paths between PCs also varies, so among paths of equal importance, we assign the paths with the fewest alternatives first.

For the PC pairs that have only a single neighborhood in common, the selection of the path is trivial. Once that has been done, the translator then examines PC pairs with two neighborhoods in common, and tries to select the the path whose NICs have thus far appeared in the fewest assigned paths. The process then continues to assign paths for pairs with three, then four, etc., neighborhoods in common. The complete process is then repeated for the next most important pairs, and so forth, until every pair has been assigned a path.

KLAT2's current network was designed partially optimizing row and column communication in an 8x8 logical configuration of the 64 processors (the two hot spares are on the uplink switch). Although the translator software actually builds a shell script that, when executed, builds the complete set of host routing tables (actually, pre-loading of each ARP cache), that output is too large to include in this paper. A shorter version is simply a table that indicates which subnets are used for each pairwise communication, as shown in Fig. 3.

3.3 Advanced Routing Tables

As discussed above, in a typical FNN, many pairs of PCs will share multiple neighborhoods. Thus, additional bandwidth can be achieved for a single message communication by breaking the message into chunks that can be sent via different paths and sending the data over all available paths simultaneously – the FNN equivalent of channel bonding. What makes FNN advanced routing difficult is that, unlike conventional channel bonding, the FNN mechanism must be able to correctly manage the fact that NICs are bonded only for a specific message destination rather than for all messages.

For example, in Fig. 2, PC **k00** is connected to the blue, transparent purple, transparent blue, and transparent magenta neighborhoods. The second PC, **k01**, shares three of those neighborhoods, replacing the transparent magenta with orange. The third PC, **k02**, has only two neighborhoods in common with **k00**: blue and transparent blue. Thus, when **k00** sends to **k01**, three of its NICs can be used to create one wider data path, but when sending from **k00** to **k02**, only

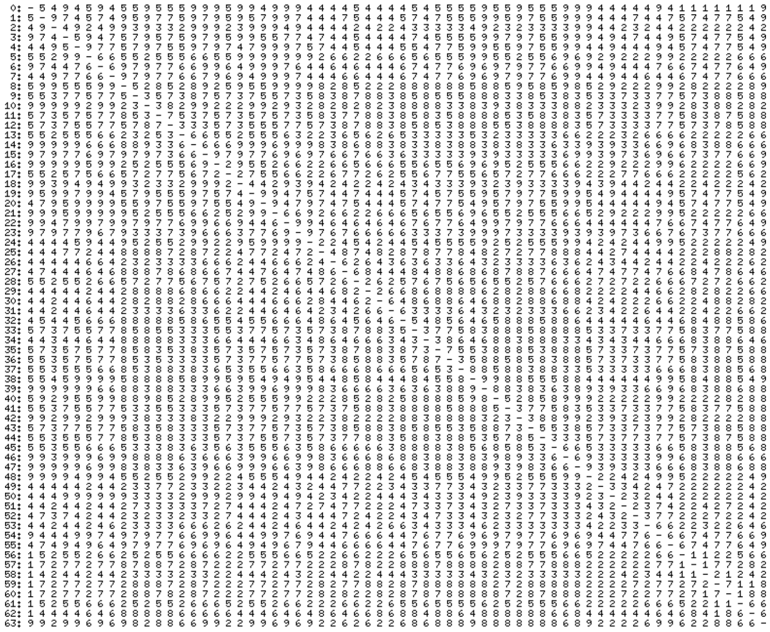

Fig. 3. Basic FNN routing for KLAT2

two NICs can be used together. If **k00** needs to send a message to **k63**, there is only one neighborhood in common and only one NIC can be used.

Although sending message chunks through different paths is not trivial, the good news is that the selection of paths can be done locally (within each PC) without loss of optimality for any permutation communication. By definition, any communication pattern that is a permutation has only one PC sending to any particular PC. Because there is no other sender targeting the same PC, and all paths are implemented directly through wire-speed switches, there is no possibility of encountering interference from another PC's communication. Further, nearly all Fast Ethernet NICs are able to send data at the same time that they are receiving data, so there is no interference within the NIC from other messages being sent out. Of course, there may be some memory access interference within the PCs, but that is relatively unimportant.

A simple translator can encode the FNN topology so that a runtime procedure can determine which NICs to specify as the sources and destinations. This is done by translating the switch neighborhood definitions into a table of NIC tuples. Each tuple specifies the NIC numbers in the destination PC that correspond to each of the NICs in the source PC. For example, the routing from **k00** to **k01** would be represented by a tuple of 1-3-4-0 meaning that **k00**'s first NIC is routed to **k01**'s first NIC, **k00**'s second NIC is routed to the third NIC of **K01**, the third NIC of **k00** is routed to the fourth NIC of **k01**, and the final value of 0 means that the fourth NIC of **k00** is not used.

To improve caching and simplify lookup, each of the NIC tuples is encoded as a single integer and a set of macros to extract the individual NIC numbers from that integer. Extraction of a field is a shift followed by a bitwise AND. With this encoding, the complete advanced routing table for a node in KLAT2 is just 128 bytes long.

Using standard Ethernet hardware, the routing by NIC numbers would require the ARP cache in each machine to translate these addresses into MAC hardware addresses. This is easily done for small clusters, but can become less efficient if very large ARP caches are required. Thus, it may be more practical to lookup MAC addresses directly rather than NIC numbers. The result is a modest increase in table size. For KLAT2, the MAC-lookup table would require 1.5K bytes.

3.4 Problems and Troubleshooting

Unfortunately, the unusual properties of FNNs make it somewhat difficult to debug the system. Although one might expect wiring errors to be common, the color coding essentially eliminates this problem. Empirically, we have developed the following list of FNN problems and troubleshooting techniques:

- The numbering of NICs depends on the PCI bus probe sequence, which might not be in an obvious order as the PCI bus slots are physically positioned on the motherboard. For example, the slots in the FIC SD11 motherboards are probed in the physical order 1-2-4-3. Fortunately, the probe order is consistent for a particular motherboard, so it is simply a matter of determining this order using one machine before physically wiring the FNN.
- If the FNN has an uplink switch, any unintended broadcast traffic, especially ARPs, can cripple network performance. Looking at the Ethernet status lights, it is very easy to recognize broadcasts; unfortunately, a switch failure also can result in unwanted broadcast traffic. Using a network analyzer and selectively pulling uplinks makes it fairly easy to identify the source(s) of the broadcasts. Typically, if it is a software problem, it will be an external machine that sent an ARP into the cluster. This problem can be fixed by appropriately adjusting ARP caches or by firewalling – which we strongly recommend for clusters.
- Application-level software that assumes each machine has a single IP/MAC address independent of the originating PC will cause many routes to go through the FNN uplink switch, whereas normal cluster-internal communications do not use the uplink switch. All application code should use host name lookup (e.g., in the local ARP cache) on each node.

Given that the system is functioning correctly with respect to the above problems, physical wiring problems (typically, a bad cable or NIC) are trivially detected by failure of a ping.

4 Performance

Although the asymmetry of FNNs defies closed-form analysis, it is possible to make a few analytic statements about performance. Using KLAT2, we also have preliminary empirical evidence that the benefits predicted for FNNs actually are delivered.

4.1 Latency and Pairwise Bandwidth

Clearly, the minimum latency between any pair of PCs is just one switch delay and the minimum bandwidth available on any path is never less than that provided by one NIC (i.e., 100Mb/s unidirectional, 200Mb/s bidirectional for Fast Ethernet). The bandwidth available between a pair of PCs depends on the precise wiring pattern, however, it is possible to compute a tight upper bound on the average bandwidth as follows.

PCs communicate in pairs. Because no PC can have two NICs connected to the same switch, the number of ways in which a pair of connections through an *S*-port switch can be selected is $S*(S-1)/2$. Only switch ports that are connected to NICs count. Similarly, if there are P PCs, the number of pairs of PCs is $P*(P-1)/2$. If we sum the number of connections possible through all switches and divide that sum by the number of PC pairs, we have a tight upper bound on the average number of links between a PC pair. Since both the numerator and denominator of this fraction are divided by 2, the formula can be simplified by multiplying all terms by 2.

For example, KLAT2's network design described in this paper uses 4 NICs, 31 ports on each of 8 switches and 8 ports on the ninth, and has 64 PCs (the two "hot spare" PCs are placed on the uplink switch). Thus, we get about 1.859 bidirectional links/pair. In fact, the FNN design shown for KLAT2 achieves precisely this average pairwise bandwidth. Using 100Mb/s Ethernet, that translates to 371.8Mb/s bidirectional bandwidth per pair.

An interesting side effect of this formula is that, if some switch ports will be unused, the maximum average pairwise bandwidth will be achieved when all but one of the switches has all its ports used. Thus, the GA naturally tends to result in FNN designs that facilitate the folded uplink configuration.

4.2 Bisection Bandwidth

Bisection bandwidth is far more difficult to compute because the bisection is derived by dividing the machine in half in the worst way possible and measuring the maximum bandwidth between the halves. A reasonable upper bound on the bisection bandwidth is clearly the total number of NICs times the number of PCs times the unidirectional bandwidth per NIC; for KLAT2, this is 4*64*100, or 25.6Gb/s.

Generally, bisection bandwidth benchmarks measure performance using a permutation communication pattern, but which pairwise communications are used is not specified and it can make a large difference which PCs in each half are paired. If we select pairwise communications between the two halves using a random permutation, the expected bisection bandwidth can be computed using the average bandwidth available per PC, computed as described above. For KLAT2, this would yield 371.8Mb/s*32, or 11.9Gb/s.

Of course, the above computations ignore the additional bandwidth available by hopping subnets using either routing through PCs or the uplink switch. Although a folded uplink switch adds slightly more bisection bandwidth than an unfolded one, it is easy to see that a non-folded uplink switch essentially adds bisection bandwidth equal to the number of non-uplink switches used times the bidirectional uplink bandwidth. For KLAT2's 9 switches with Fast Ethernet uplinks, an unfolded uplink switch adds 1.8Gb/s to the 11.9Gb/s total, yielding 13.7Gb/s. However, the routing techniques described in this paper normally ignore the communication paths that would route through the uplink switch.

Further complicating the measurement of FNN bisection bandwidth is the fact that peak bisection bandwidth for a FNN is not necessarily achievable for *any* permutation communication pattern. The ability of multiple NICs to function simultaneously within each PC makes it far easier to achieve high bisection bandwidth using a pattern in which many PCs *simultaneously* send messages to various destinations through several of their NICs. We know of no standard bisection bandwidth benchmark that would take advantage of this property, yet the performance increase is easily observed in running real codes.

4.3 Empirical Performance

The FNN concept is very new and we have not yet had time to fully evaluate its performance nor to clean-up and release the public-domain compiler and runtime software that we have been developing to support it. Thus, we have not yet run detailed network performance benchmarks. However, KLAT2's FNN has enabled it to achieve very high performance on several applications.

At this writing, a full CFD (Computational Fluid Dynamics) code [11], such as normally would be run on a shared-memory machine, is running on KLAT2 well enough that it is a finalist for a Gordon Bell Price/Performance award. KLAT2 also achieves over 64 GFLOPS on the standard LINPACK benchmark (using ScaLAPACK with our 32-bit floating-point *3DNow!* SWAR extensions).

Why is performance so good? The first reason is the bandwidth. As described above, KLAT2's FNN has about 25Gb/s bisection bandwidth – an ideal 100Mb/s switch the full width of the cluster would provide no more than 6.4Gb/s bisection bandwidth, and such a switch would cost far more than the FNN. Although Gb/s hardware can provide higher pairwise bandwidth, using a tree switch fabric yields less than 10Gb/s bisection bandwidth at an order of magnitude higher cost than KLAT2's FNN.

Additional FNN performance boosts come from the low latency that results from having only a single switch delay between source and destination PCs and from the semi-independent use of multiple NICs. Having four NICs in each PC allows for parallel overlap in communications that the normal Linux IP mechanisms would not provide with channel bonding or with a single NIC. Further, because each hardware interface is buffered, the FNN communications benefit from greater buffered overlap.

5 Conclusion

In this paper, we have introduced a variety of compiler-flavored techniques for the design and use of a new type of scalable network, the Flat Neighborhood Network (FNN).

The FNN topology and routing concepts make it exceptionally cheap to implement – the network hardware for KLAT2's 64 (plus 2 spare) nodes cost about 8,000 dollars. It is not only much cheaper than Gb/s alternatives that it outperforms, but also is cheaper than a conventional 100Mb/s implementation would have been using a single NIC per PC and a cluster-width switch.

The low cost and high performance are not an accident, but are features designed using a genetic search algorithm (GA) to create a network optimized for the specific communications that are expected to be important for the parallel programs the system will run. Additional compiler tools also were developed to manage the relatively exotic wiring complexity and routing issues. With these tools, it is easy and cost-effective to customize the system network design at a level never before possible.

KLAT2, the first FNN machine, is described in detail at:
`http://aggregate.org/KLAT2/`

References

1. Dietz, H., Chung, T., Mattox, T.: A Parallel Processing Support Library Based On Synchronized Aggregate Communication. In: Languages and Compilers for Parallel Computing, Springer-Verlag, New York (1996) 254-268
2. The Gigabit Ethernet Alliance, `http://www.gigabit-ethernet.org/`
3. Myricom, Inc., `http://www.myri.com/`
4. Giganet CLAN, `http://www.giganet.com/products/indexlinux.htm`
5. Dolphin SCI (Scalable Coherent Interconnect), `http://www.dolphinics.com/`
6. Fisher, R., Dietz, H.: The Scc Compiler: SWARing at MMX and 3DNow!. In: Carter, L., Ferrante, J. (eds.): Languages and Compilers for Parallel Computing, Springer-Verlag, New York (2000) 399-414
7. The LAM MPI Implementation, `http://www.mpi.nd.edu/lam/`
8. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard. Rice University, Technical Report CRPC-TR94439, April 1994.
9. Dietz, H., Mattox, T.: KLAT2's Flat Neighborhood Network. In: proceedings of the Fourth Annual Linux Showcase (ALS2000, Extreme Linux track) Atlanta, GA, October 12, 2000
10. Aberdeen, D., Baxter, J., Edwards, R.: 92cents/MFlops/s, Ultra-Large-Scale Neural-Network Training on a PIII Cluster. In: IEEE/ACM proceedings of SC2000, Dallas, TX, November 4-10, 2000
11. Hauser, Th., Mattox, T., LeBeau, R., Dietz, H., Huang, G.: High-Cost CFD on a Low-Cost Cluster. In: IEEE/ACM proceedings of SC2000, Dallas, TX, November 4-10, 2000