Efficient Myrinet Routing

Thomas W. Finley*

April 18, 2002

1 Introduction

In parallel computing over a network of workstations one frequently needs to exchange data between nodes on the network. This project's aim is to provide an efficient strategy for routing information over a network. This paper is written with Myrinet in mind, a network technology developed by Myricom Inc.

Nodes communicate by sending packets: like cars, packets travel along roads (links), make turns at intersections (switches), and presumably at some point get to where they have to go (the destination node). However, like many other networks there is the potential for contention when two packets need to use the same resource.

In parallel computing applications, nodes on a network know to whom they want to send and how much they want to send. If one of the nodes on the network is aware of this information, it is possible for it to make decisions regarding the routing of communications to avoid contention, so communications are completed as quickly as possible. The time spent sharing information between nodes can exceed time spent in calculation. Therefore, any strategy that reduces this time yields significantly faster parallel computations.

After a brief overview of Myrinet, this paper presents a form of a typical Myrinet network topology; how it works and how contention may be avoided; presents a fast algorithm to route communications over that network; and finally presents C++ code for a rather quick implementation of this algorithm.

2 Myrinet

A Myrinet [MyPr98] consists of nodes connected through a collection of links and switches. Myricom calls switches in a Myrinet crossbars. There are ports on crossbars, and one port on each node through which data may pass. If the maximum number of ports a crossbar can have is n, then it is called an n-port crossbar. Connections between two ports are links. Links are like simple pipes: data enters through one end and appears at the other end. Figure 1 shows a possible Myrinet.

*This work is supported by NSF VIGRE grant number DMS-9983320.

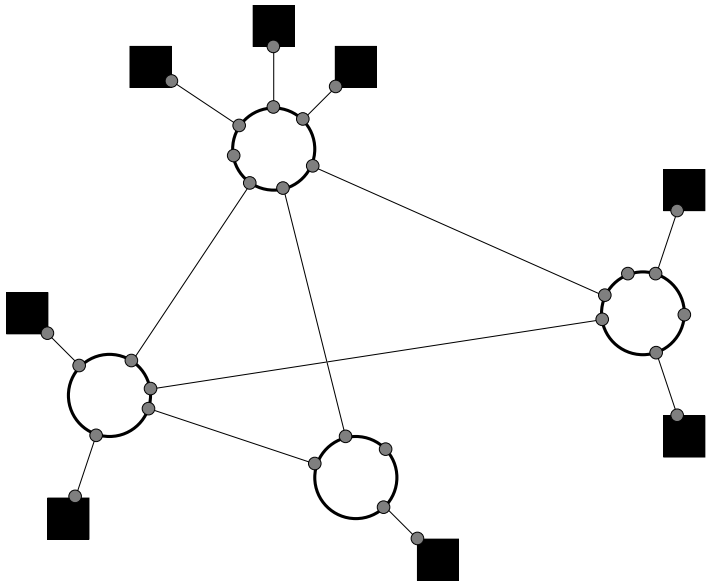

Figure 1: This is an example of a possible Myrinet. The black boxes are the nodes. The large white circles are the switches/crossbars. The smaller greyed in circles on the crossbars and nodes are ports. The lines that connect ports are links. Some ports are not connected to a link: not all ports need to have a link connected for the network to function. This is an illustration of a possible topology based on the Myrinet ANSI standard, but no one would actually build a Myrinet like this.

On a Myrinet, information is exchanged between nodes across the network through packets. As a packet passes through links it sometimes runs into a switch. The packet contains bytes at its header that the switch uses to determine where to direct the packet. The route information inside a packet completely determines the routing of that packet, so once the packet leaves a send node its path is already determined. Each switch "eats" the first header route byte, and uses the eaten byte to determine where to send the packet. The switch sends the packet minus this route byte through the indicated port. This route may lead to another switch, or ultimately some node on the network which receives this packet. The route information is stripped from the packet byte by byte the packet passes through switches; all that remains at the receiving node is the packet itself.

For an n-port-crossbar, the n ports are indexed 0 through $n-1$. Recall that when a packet reaches a switch, the first byte in the packet determines how that packet is routed within the switch. The routing is done through relative addressing: if a packet arrives at a switch through port a and wants to exit through port b, inside the route byte the quantity $b-a$ is encoded, which will direct the packet along its desired route. The two most significant bits in the route byte are always $\texttt{10}_2$, while the six least significant bits together form this "relative port address" as a signed two's complement number. For example, if a packet arrives at a switch through port 1 and wants to exit through port 12, the route byte will be $\texttt{10001011}_2 = \texttt{0x8B}$. If a packet arrives at a switch through port 7 and wants to exit through port 4, the route byte will be $\texttt{10111101}_2 = \texttt{0xBD}$.

All links are full duplex links, i.e., data may flow in both directions without contention.

One consequence of this is that a node could receive data at the same time that it is sending.[1] Also, if in a switch a packet flows from port A out to port B, another packet may simultaneously come in from port B without contention, and similarly a third packet can go out through port A. However, two packets cannot travel in the same direction on the same link without contention.

Myricom's low level message passing API, GM [MyGM00], is intelligent enough to deal with contention without error. GM limits the size of each packet sent[2] to prevent a single packet from monopolizing the network. In the event of contention on a port, these small packets are interleaved with other packets to ensure reliable packet delivery, much like time sharing on a processor. However, if contention is avoided, data is transmitted along the network at the maximum speed without the need to share links with other packets.

Dealing with contention intelligently can be difficult since Myrinet does not have any prescribed regular topology. Provided one has the money to pay for a nonstandard configuration, it is possible to connect these crossbar switches through any sort of linking arrangement and the network will function just fine when communications are handled through GM. With GM, one of the nodes, called the mapper node, runs a mapper program that derives the topology of the network. Based on that topology, this mapper node assigns route tables to each node detected on the Myrinet. Each node's route table has an entry for every other node. When sending a message from a node M to a node N, GM makes use of a route table by prepending the packet with the route in N's entry in M's route table. Unfortunately as a consequence routes do not adapt to current packets on the network to avoid contention: a route will only change when the mapper is run. This inflexibility of the routes makes contention difficult to avoid.

It is difficult to find efficient routes quickly on a general Myrinet. However, in spite of the fact that Myrinet is flexible enough to allow irregular topologies, the most popular "ready made" configurations of Myrinet are based on a network topology specifically designed for making finding contention-free routes possible and fairly easy [MySN01]. We limit our discussion to these types of networks.

3 The Network

3.1 Vital statistics

The topology of the general network to which this paper's routing algorithm may be applied is pictured in Figure 2.

The network in Figure 2 links a number of sending nodes to an equal numbers of receiving nodes. The lines represent links along which data may flow. These links are one-way links where data may flow only from the left end to the right end. The seemingly "disconnected" links to the left and the right each indicate a single connection to either a sender (on the left) or a receiver (on the right). Finally, the circles represent switches. Each switch has a series of ports to which links are connected: ports connected to the "exit" end of a link are

[1] Due to technical restrictions of the computers themselves and of Myricom PCI cards, in some configurations this is impossible to accomplish. However, this is not a shortcoming of the network itself.

[2] The default packet size over GM is 4 kilobytes.

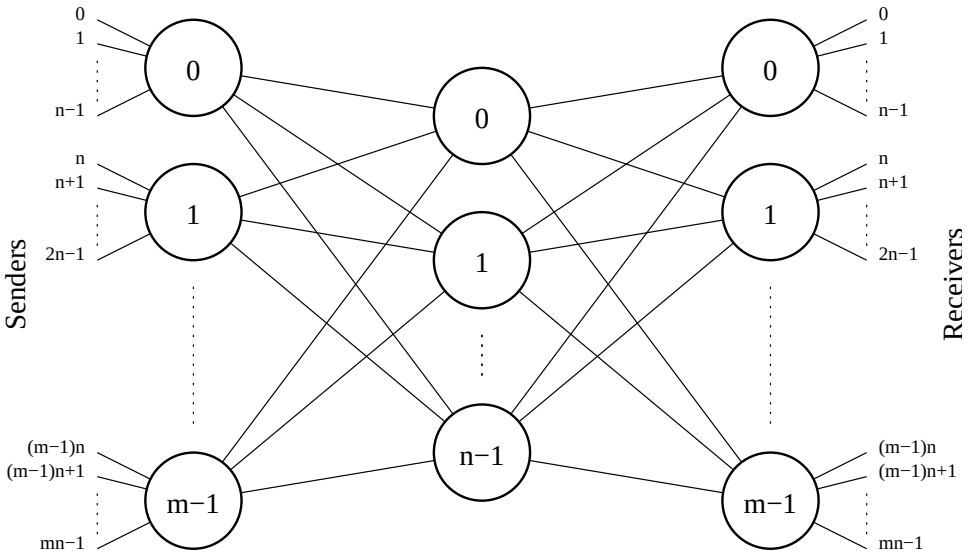

Figure 2: The general network described in this paper.

called receive ports, while those connected to the "entrance" end of a link are called send ports.

In a switch, packets arriving at a receive port may be routed to any of the send ports, unless that send port is already in use for transmission of a different packet. Contention occurs when two packets are transmitted along the same link. Two packets may use the same switch without contention so long as they do not arrive through the same port, or attempt to leave through the same port.

Send switches are those switches to which senders are connected via a single link, and receive switches are defined similarly. In Figure 2, the left column of switches are send switches, while the right column are the receive switches. As shown in the diagram, there are exactly n nodes connected to each receiver and sender switch. Due to details of the algorithm n must itself be some power of two. As also shown in the diagram, there are exactly m send switches and as many receiver switches.

Aside from the sender and receiver switches, there are the switches in the central column of switches. These are called route switches. Why route switches? Every sender is connected to a single send switch, and every receiver is connected to a single receive switch. Given a packet from a specific sender to a specific receiver, note that the choice of send and receive switches for this packet is prescribed since the sender and receiver nodes are each themselves only on one send and receive switch. There is no other choice but to use those two switches. However, there is a choice about which of the route switches the data will pass through. Regarding the routing of a packet from our sender to a receiver, this is the only choice that we have, hence the name route switch.

Those familiar with network topologies may recognize this sort of network as a 3-stage rearrangeably-non-blocking Clos network (or simply rearrangeable Clos network) with the added stipulation that the number of nodes connected to a switch is a power of two [Cl53].

3.2 Relation to Myrinet

As an interlude, we mention how this network relates to the more popular Myrinet topologies.

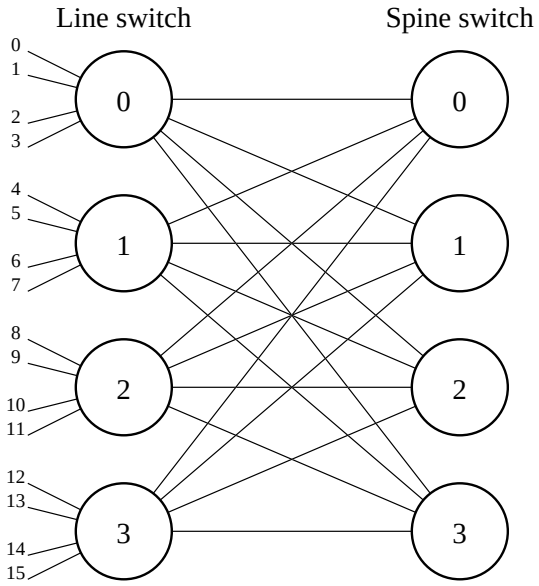

Figure 3: An M2M-OCT-SW8 switch.

The simplest non-trivial variant of a Myrinet is that shown in Figure 3, an M2M-OCT-SW8 switch [CS97]. This component is long outdated, but the current crop of Myricom's networks has the same general topology [MySN01]. Figure 3 gives the false impression that this Myrinet is only half of the network that we described, but it is actually the same network.

Consider a packet moving from one node to another. A packet first encounters one of the switches labeled a line switch, then it proceeds to a spine switch, the analogy of our route switch. Remember that links in a Myrinet are full-duplex: if we consider each full-duplex link to really consist of two unidirectional links, one proceeding one direction and one in the other direction, we note that in going to the spine switch we have only used those links whose direction is from left to right. Those from right to left have not been touched.

The network described in this paper has been adapted to Myrinet by folding the network about the middle route switches [CLS]. In terms of our network, the senders and receivers are really the same nodes, and indeed the so-called send and receive switches are physically the same switch in a Myrinet.

The Myrinet shown is a bit more powerful than our three-stage network. If a packet has a send and receive switch with the same index, that is the same switch in the Myrinet and the packet can simply be routed directly to its destination from that switch. However, there is nothing wrong with routing "intra-line-switch" packets through the spine.

3.3 Permutations

As mentioned, the only choice to be made in the routing of a single packet is the choice of the route switch through which that packet passes. Routing only becomes an interesting problem when there are a large number of simultaneous communications. Our goal is to

avoid contention completely. If a set of simultaneous communications contains two communications from a single sender, then those two communications must use the same link since that sender is connected to the network only through a single link: contention becomes unavoidable. Therefore, avoidance of contention is only possible when a set of simultaneous communications does not contain any communication where any sender appears more than once. A similar line of reasoning applies to the receivers.

One of the properties of this type of network, which Clos proved in the more general case of the rearrangeable Clos network, is that it is possible to find contention free routes for communications in a set of communications disjoint with respect to senders and receivers.

Putting these two together, given a set of pairs of communicating hosts, to find a contention free route it is both necessary and sufficient that these pairs be disjoint with respect to both senders and receivers.

A set of simultaneous communications from senders to receivers where any sender appears at most once and any receiver appears at most once is commonly referred to as a permutation. For mathematical abstraction, consider a permutation p as a mapping from the set of sending nodes S to the set of receiving nodes R, that is, $p : S \to R$. $p(i) \in R$ is the receiver to which the sender i is sending. Naturally this mapping is injective since no receiver may appear more than once as explained earlier.

3.4 Bipartite multigraph abstraction of a permutation

In order to find a route for each communication represented in a partition, we first consider the permutation as a bipartite multigraph.

Let W_s be the set of send switches, and W_r be the set of receive switches. Our bipartite multigraph $G = (V_s, V_r, E)$, where V_s is the set of left party nodes, V_r is the set of right party nodes, and E is the set of edges.

There is a bijective mapping $\nu_s : W_s \to V_s$, so that each sender switch corresponds to a left-party vertex in the bipartite multigraph. Likewise, there is a bijective mapping $\nu_r : W_r \to V_r$.

Also, we define a mapping $\sigma_s : S \to W_s$ so that for a sender $i \in S$, $\sigma_s(i) \in W_s$ is simply the sender switch to which the sender node i is connected. We define a similar mapping $\sigma_r : R \to W_r$ so that for a receiver $i \in R$, $\sigma_r(i) \in W_r$ is the receiver switch to which the receiver node i is connected.

As an example of σ_s and σ_r, in the diagram of Figure 2, if we have nodes and switches enumerated as integral quantities, then a mapping from nodes to their switches (either sender or receiver) the switch to which a node is connected would simply be the node index divided by n without the remainder.

The set of edges E in G is defined as $E = \{(\nu_s(\sigma_s(i)), \nu_r(\sigma_r(p(i))), i) | i \in S\}$, where, for each edge $(l, r, s) \in E$, $l \in V_s$ and $r \in V_r$ are the left and right vertices between which the edge is drawn, and $s \in S$ is an extra piece of information attached to an edge, the sender of the communication that corresponds to this edge in the graph.

Put more intuitively and concisely (albeit inaccurately), consider the send and receive switches in our network as opposing parties of vertices. Every communication between nodes is represented as an edge between the vertices which correspond to the switches to which

those nodes are connected. We also associate with each edge the sender in the communication from which that edge was generated.

3.5 Routing and its ties to edge coloring

First, we must clarify what makes routes contention free routes in this case. Since there is exactly one link between any one send switch and any one route switch, and exactly one link between any one route switch and any one receive switch, if we have a sender node s sending to receiver node r, with each on sending and receiving switches $S = \sigma_s(s)$ and $R = \sigma_r(r)$ respectively, and with their communications going through route switch K, then any other communicating pair with a sender connected to sender switch S or a receiver connected to receiver switch R cannot use K, or this communication will have to use one of the links that the original communication used. With both communications proceeding immediately from S to K, or both communications proceeding from K to R, such a collision would be unavoidable since there is only one link for each of those possible routes.

In summary, for any communication, any other communication that shares either the sending or receiving switch cannot be routed through the same route switch without contention.

We transfer this to our constructed bipartite multigraph. Each edge in the multigraph corresponds to a communication, and both vertices for each edge corresponds to that edge's communication's send and receive switches. We must assign a route switch to each edge so that no two adjacent edges, i.e., no two edges that share a vertex in common, are assigned the same route switch. Note that if you simply replace "route switch" with "color" in that last sentence, you have the exact definition of the classic edge coloring problem.

Therefore, in order to find routes for the communications in a permutation, we must find an edge coloring for the bipartite multigraph that represents that partition. Those communications whose edges have the same colors will have their communications routed through the same route switch.

There are two important facts about edge coloring relevant here. One is that the chromatic index[3] of a bipartite multigraph is equal to the degree of the graph [Ga95]. Each sender may appear only once in a permutation, there are only n senders on each switch, and each switch corresponds to a vertex, so the maximum degree of any bipartite multigraph we construct is bounded at n. As consequence we may always color any bipartite multigraph with exactly n colors. Since colors correspond to a choice of route switch, this is why it is possible to find routes for any permutation using exactly n route switches.

3.6 An example permutation and its routes

We see a specific example of the sort of network described shown in Figure 4. In this example we instantiate $m = 3$ and $n = 4$. For our permutation p which describes what set of communications we have to find routes for, I show the twelve communications in duples, with the first element as the index of the send node and the second element the index of the

[3]The chromatic index of a graph is the minimum number of colors needed to edge color the graph.

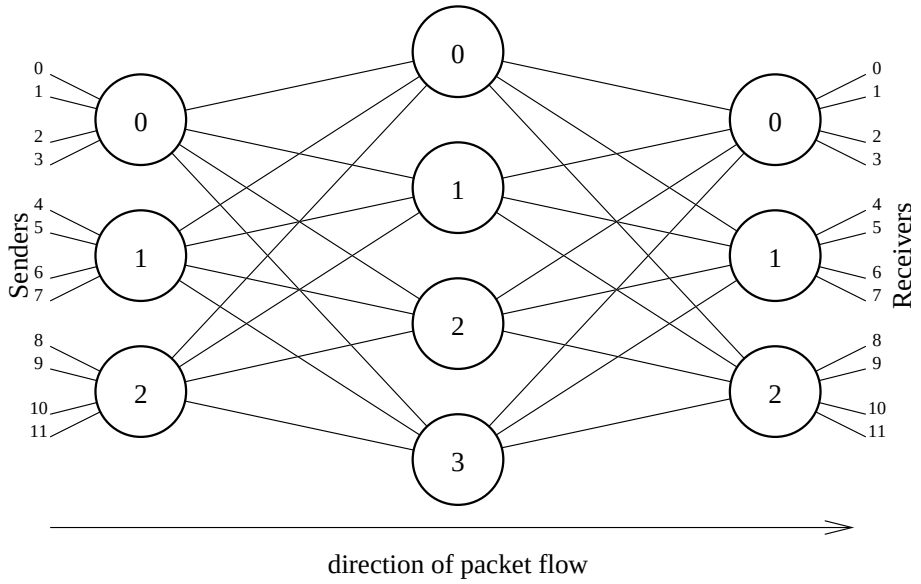

Figure 4: The topology of the network used in this example. As it pertains to the general form of network, here $m = 3$ and $n = 4$.

receive node.

$$(0,6),(1,4),(2,11),(3,0),(4,3),(5,1),(6,2),(7,8),(8,5),(9,7),(10,10),(11,9)$$

In this permutation, for example, the pair $(2, 11)$ signifies that sender 2 is sending to receiver 11.

We map the permutation on to a graph as described earlier. For the sake of convenience I consider the vertices of the left party to be indexed exactly as their corresponding send switches, and likewise with the right party and the receive switches. Then, the graph $G = (V_s, V_r, E)$ is $V_s = \{0, 1, 2\}$, $V_r = \{0, 1, 2\}$, and these are the edges:

$$\begin{aligned} E \;=\; & \{(0,1,0)_{(0,6)}, (0,1,1)_{(1,4)}, (0,2,2)_{(2,11)}, (0,0,3)_{(3,0)}, (1,0,4)_{(4,3)}, (1,0,5)_{(5,1)}, \\ & (1,0,6)_{(6,2)}, (1,2,7)_{(7,8)}, (2,1,8)_{(8,5)}, (2,1,9)_{(9,7)}, (2,2,10)_{(10,10)}, (2,2,11)_{(11,9)}\} \end{aligned}$$

Note that the subscripts are not actually included, but are only there so that the reader may at a glance determine the communication that generates each edge.

We now have the partition p expressed as a bipartite multigraph. We have three vertices in each party, since there are three send and receive switches. The graph is shown in Figure 5. Each vertex shows which of the original elements in P it corresponds to.

Then, the coloring algorithm is applied. The degree of the graph is 4, so we can achieve an edge coloring with 4 colors. An example of a coloring is also shown in Figure 5 (there are typically many possible colorings). We enumerate the colors 0 through 3, where 0 colored edges are the solid lines, 1 the dashed, 2 the dotted, and 3 the dotted and dashed. Then define C_i as the set of pairs of communicating hosts such that the edges they correspond to in the bipartite graph have color i. Since we have preserved the sender of each communication's edge, we are able to tell every sender the route switch to use in sending its packet. In the

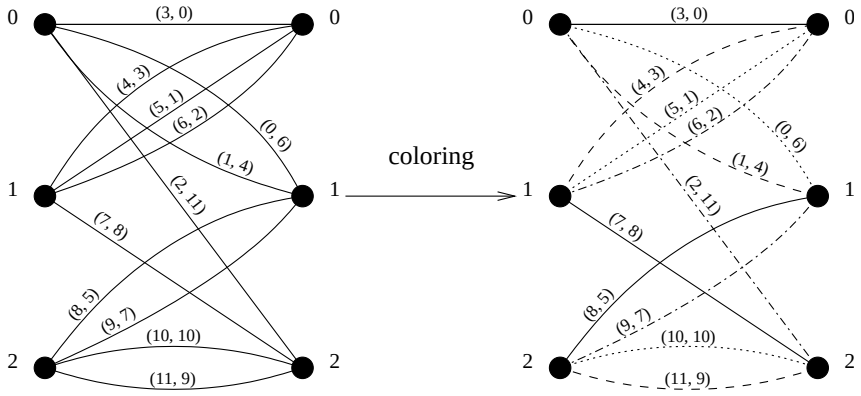

Figure 5: The routing example of the bipartite graph abstraction for the P described in the example. Each edge has the element of P that it represents labeling it. A valid coloring of this graph is also shown, with each different coloring represented as a different style of line.

course of the actual computation we would only have the sender, but for the reader we present both sender and receiver. The C_i sets are shown below.

$$\begin{aligned} C_0 &= \{(3,0),(7,8),(8,5)\} \\ C_1 &= \{(1,4),(4,3),(11,9)\} \\ C_2 &= \{(0,6),(5,1),(10,10)\} \\ C_3 &= \{(2,11),(6,2),(9,7)\} \end{aligned}$$

Each communication in C_i goes through routing switch i. This completes the routing procedure. If desired, one can check these routes against the original graph to ensure that there are indeed contention free routes.

4 Bipartite Multigraph Edge Coloring using Euler Splits

4.1 Workings of the Euler split

An Euler split of a graph can be applied to any regular bipartite graph of even degree. The result of the Euler split is that a graph is divided into two graphs, which can be thought of as having the same number of vertices but each has half the number of edges of the original graph. Indeed, the degree of the resulting graph is exactly half the degree of the original graph. The graph is also regular just like the original graph [CH82].

In more precise terms, if we define $G = (V_l, V_r, E)$ as a regular bipartite multigraph of degree $2n$, where V_l are the vertices in the left party and V_r are the vertices in the right party, with both vertex sets disjoint. (Note that since the graph is regular and bipartite there must be an equal number of vertices in both parties.) A split results in two graphs $G_1 = (V_l, V_r, E_1)$ and $G_2 = (V_l, V_r, E_2)$, where $E_1 \cap E_2 = \varnothing$ and $E_1 \cup E_2 = E$.

How is this split accomplished? We first start with Euler partitions[4]. Cole defines an Euler partition as a process where a graph is divided into distinct sets of open and closed Euler paths [COS00]. Note that the Euler paths described by an Euler partition is not an Euler paths for the entire multigraph, but each partition is a subgraph of the original graph on which an Euler path can be found, and the partition is the list of edges one can traverse to follow such an Euler path in the subgraph of the partition (that is, the edges of the subgraph are ordered in the order for an Euler path). Every edge in any Euler partition is distinct from any other edge in any other Euler partition of this graph. In a graph, a path is an ordered succession of edges so that one can start at some start vertex.

In spite of the dense prose above, the process for finding Euler partitions is rather straightforward. We hop from our start vertex to a vertex in the alternate party with the two vertices connected via an edge. We remove the edge from the graph and put it in the partition list. We then jump from this new vertex back over to the other party of vertices, again via an edge, removing the edge and putting it in the partition list. Which edge is picked is irrelevant. This process continues. We only stop when we're forced to, that is, when our travels bring us to a vertex that has no more edges. Those familiar with graph theory and of Euler's Königsberg bridge problem in particular will recognize that this "finishing" vertex must be vertex we started with, since this graph is regular with even degree; i.e., each Euler partition defines an Euler cycle.

After this partition has been finished, we pick another start vertex–which vertex doesn't matter, so long as it has some edges left connected to it–and begin the process again. We stop forming Euler partitions when there are no more edges left in our original graph.

To form the Euler split, we simply iterate through all the partitions we formed. The partitions are little more than lists of edges: we iterate through these lists, and place alternating edges in each of the graphs G_1 and G_2 as defined above.

In practice Euler partitions are unnecessary. One may skip directly to the step of breaking alternate edges into their respective subgraphs as the partitions are being formed. Euler partitions are only useful in simplifying the proof of correctness of this algorithm.

If one has a graph structure where one can remove, add, and iterate vertex-wise through edges in a graph in constant time per edge, the process of applying this algorithm to a graph with E edges takes time of order $\Theta(E)$[5].

4.2 What this all has to do with edge coloring

Suppose that we have the case where the degree of our original regular bipartite multigraph is a power of two: that is, we have a bipartite multigraph where every vertex has degree 2^n, where $n \in \mathbb{Z}, n \geq 0$. Then, if we apply an Euler split to this graph, we result in two regular bipartite multigraphs with degree 2^{n-1}. If we then apply two Euler splits to these two graphs, the result is four regular bipartite multigraphs with degree 2^{n-2}. Since we started with a graph whose degree was a power of 2, we can recursively apply these splits, and the only time we reach a graph with odd degree, when we have to stop, is when the degree is 1.

[4]This definition of Euler partition should not be confused with the more common definition of Euler partitions one finds in number theory.

[5]$f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1 and c_2 such that $n \geq n_0$, $c_1 g(n) \leq f(n) \leq c_2 g(n)$ [CLR90]

At this point we have 2^n different graphs, each of degree one. Suppose, for each of these graphs, we color each edge a color unique to that graph. Since each of the colors is unique to the split graphs, and each has degree 1, no vertex can have two incident edges with the same color. If we assign the colors to the corresponding edges in the original graph, we have a 2^n degree graph with a proper 2^n coloring on its edges.

This was demonstrated to take time $\Theta(E \log D)$, where D is the degree of the graph [CH82]. Note that this constraint of a 2^n degree bipartite multigraph is for convenience sake, as recently an order $\Theta(E \log D)$ algorithm was discovered, using an innovation that these Euler splits can be applied to a graph, even if it does not have even degree [COS00]. However, the current state of Myrinet does not require this.

Given the nature of *why* we are coloring, we are guaranteed to have an edge coloring problem on a bipartite multigraph where we may use c colors, where c is a power of two (even if the degree of the graph happens to be lower). The extra book-keeping and other operations necessary to use the algorithm in the general case are not unreasonable, but they are extra complications that are unnecessary for the problem we are attempting to solve.

Note the use of this Euler split algorithm to color the graph as shown in Figure 6. The eagle eyed will note the similarity of the original graph to that of Figure 5 concerning our routing over a Clos network example, and that the final leaves of the recursion have generated graphs whose corresponding edges in the graph of Figure 5 ended up being colored the same color.

4.3 Implementation

A sample prototype implementation of this routing algorithm takes one of the communication partitions, and derives a route for each communication within that partition. The C++ code for the implementation is provided with rather full comments in Appendix A.

5 Future Work

Let us now consider possible future development of this strategy.

First of all, this code only finds routes, but it does not interface at all with Myrinet. In some sense this would not be appropriate given the range of possible situations in which this sort of algorithm could be applied, but practical applications according to its intended purpose would yield more data on its benefits and costs.

Secondly there is the limited subset of networks to which this may be applied. This method for routing was developed in response to a request for a better algorithm for routing over the Math department's Myrinet cluster at Duke University. We have a small Myrinet of only 24 computers. In the current distributions of Myrinet hardware, this algorithm can be applied just as easily to a network connecting up to 128 computers, and with one small modification it may be applied to a network of up to 256, and can be adapted upward with some more effort. It is not worth examining these networks and how to apply the algorithm. However, that is the very problem: the method as it is is tremendously ad hoc in that it must be tailored to the network.

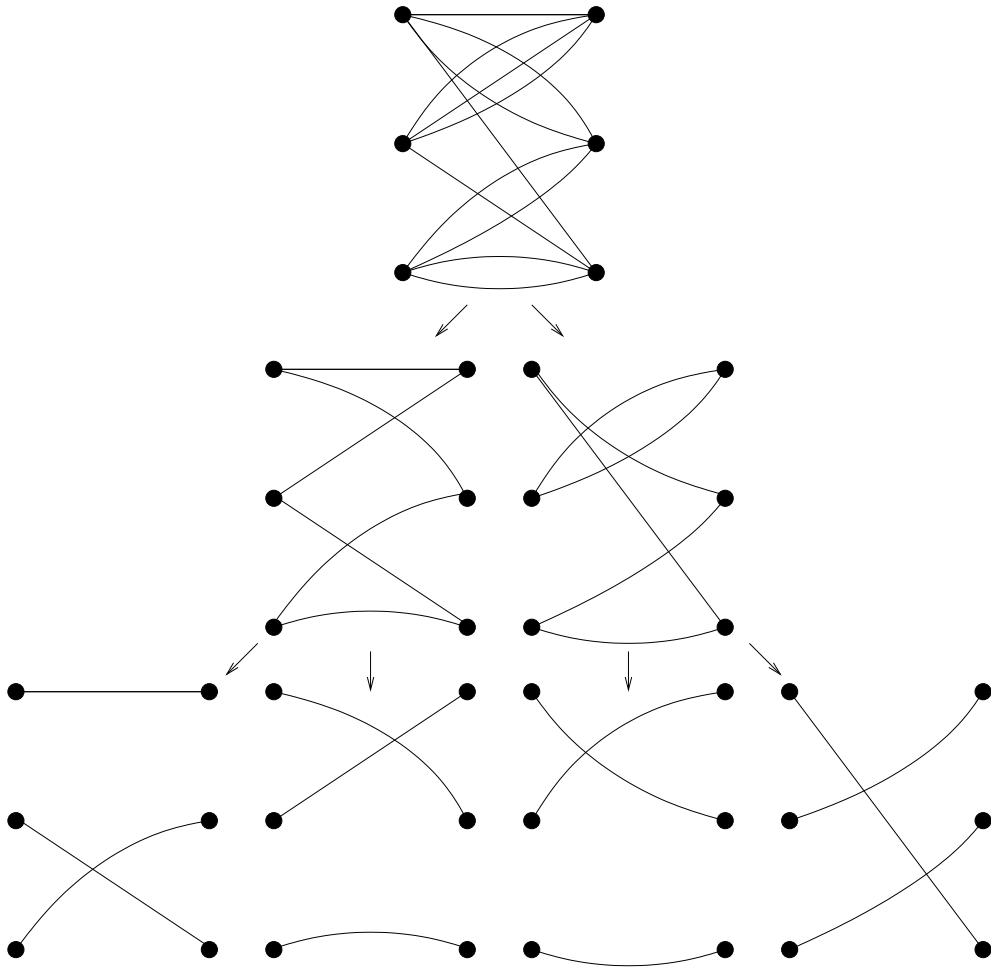

Figure 6: Recursive Euler splits on a graph of degree 2^2.

Nonetheless, suppose we accept this limitation. Surely the network will not change its topology on us, right? However, suppose the network does change without warning, that is, suppose there is a failure within the network. If a node is disconnected, then we may simply ignore the fact that it exists, but what if one of the route switches fails, or some link fails somewhere? The existing method for routing over a network will adapt to that failure. This algorithm will not adapt, and in many cases could not even be applied on the net topology depending on the exact nature of a failure.

Another problem is that sending messages with user-defined routes is not easily or cleanly done within the context of Myricom's own message passing API, GM. GM allows a program to change route tables, but in a multiprogrammed environment this leads to synchronization and performance issues if multiple programs attempt to change the tables. Suffice to say, these prescribed routes cannot be easily changed. If there is to be any user-defined route, the GM source must be modified to allow for this functionality.

6 Acknowledgements

I would like to thank Professor William Allard of Duke's mathematics department for helping me learn about Myrinet; providing feedback, encouragement, and direction for my efforts; and for his many helpful critiques of my writing.

References

[Be65] V. Beneš, *Mathematical Theory of Connecting Networks and Telephone Traffic*, Academic Press Publishing Company, New York (1965).

[CCI99] R. Canonico, R. Cristaldi, G. Iannello, *A scalable flow control algorithm for the Fast Messages communication library*, Communication, Architecture, and Applications for Network-Based Parallel Computing (1999), 77-90.

[Cl53] C. Clos, *A study of non-blocking switching networks*, Bell System Technical Journal 32 (2 March 1953), 406–424

[CS97] D. Cohen, C. Seitz, *M2M-OCT-SW8: Octal 8-Port Myrinet-SAN Switch, Recommended Topologies*. Myricom, Inc. White Papers (3 July 1997).

[CH82] R. Cole, J. Hopcroft, *On edge coloring bipartite graphs*. SIAM J. Comput 11 (1982), 540–548.

[COS00] R. Cole, K. Ost, S. Schirra, *Edge coloring bipartite multigraphs in $O(E \log D)$ time*, Combinatorica 21(1) (2001), 5–12.

[CLR90] T. Cormen, C. Leiserson, R. Rivest, *Introduction to Algorithms*, MIT Press, McGraw Hill, Cambridge Massachusetts (1990).

[FW77] S. Fiorini, R. J. Wilson, *Edge-colourings of graphs*, Research notes in math. no. 17, Pitman, London, San Francisco, Melbourne (1977).

[FMLD00] J. Flitch, M. P. Malumbres, P. López, J. Duato, *Improving routing performance in Myrinet networks*, Proceedings of the Int'l Parallel and Distributed Processing Symposium 2000 (May 2000).

[Ga95] F. Galvin, *The list chromatic index of a bipartite multigraph*, Journal of Combinatorial Theory, Ser.B 63 (1995), 153–158.

[Ho81] I. Holyer, *The NP-completeness of edge coloring*, SIAM J. Comput. 10 (1981), 718–720.

[La76] E. Lawler, *Combinatorial optimization: networks and matroids*, Holt Rinehart and Winston, New York (1976).

[MG92] J. Misra, D. Gries, *Constructive proof of Vizing's theorem*, Information Processing Letters, 41(3) (1992), 131–133.

[MyGM00] Myricom, *The GM message passing system*, http://www.myri.com/scs/GM/doc/gm.pdf (18 July 2000).

[MySN01] Myricom, *Guide to Myrinet-2000 switches and switch networks*, http://www.myri.com/myrinet/m3switch/guide/index.html (1 July 2001).

[MyPr98] Myricom, *Myrinet-on-VME protocol specification draft standard*, http://www.myri.com/open-specs/myri-vme-d11.pdf, 1.1 (31 August 1998).

[PF00] F. Petrini, W. Feng, *Buffered coscheduling: A new methodology for multitasking parallel jobs on distributed systems*, Proceedings of the Int'l Parallel and Distributed Processing Symposium 2000 (May 2000).

[Ro98] K. H. Rosen, *The handbook of discrete and combinatorial mathematics*, CRC Press, New York (December 1998).

[CLS] E-mail correspondence with Charles L. Seitz, chuck@myri.com (11 July 2001).

[Tu84] A. Tucker, *Applied combinatorics*, 3rd ed., John Wiley and Sons, New York (1994).

A Code

This is C++ code that implements the routing procedure described in this paper. There are two major parts to the code, `main.C`, which randomly generates permutations and finds routings for them, and `graph.H` and `graph.C`, which are the brawn behind the finding of the routings. Code is commented with the aim of being rather easy to follow for one familiar with this paper.

The implementation's only unusual feature is the method for coloring the graph, which is worth a few words. Each vertex in the graph represents its edges as elements in an array; dynamic structures are unnecessary since the maximum number of edges on each vertex is tightly bounded to some low number. In the case of current versions of Myrinet, for example, due to technical restrictions the maximum degree is fixed at 8.

Each vertex, both left and right, has such an array of edges. Edges connect a left and right node, so each edge is noted twice, once in the left node, once in the right node. The index of the same edge in either node will vary between the two vertices. However, note the properties of an index: each edge at each vertex cannot share its edge list index with any other edge at that vertex: one edge would overwrite the other. This is very similar to the restrictions placed on edge colors in edge coloring a graph. If one can make every edge have both the same index position in both the left and right vertex, then the index at which that edge is stored then becomes the color! The only thing the coloring algorithm does is ensure this index symmetry. Each edge's color is then implied by its position in the index array.

This implementation is also pretty fast. To give a loose idea of speed I present some timing information. This was timed on Compaq XP 1000 computers with an Alpha (212634) chip with clock speed of 667 Mhz with 4 MB of L3 cache. What is timed is the section of code that takes a partition and returns a route. The other sections of code involved in generating the random cases and reporting the results are not timed.

Time for a network with 1024 nodes with $2^3 = 8$ nodes per switch averaged 385 microseconds per route. A network with 1024 nodes with $2^6 = 64$ nodes per switch averaged 720 microseconds. A network with 32 nodes with 8 nodes per switch averaged 12.1 microseconds. A network with 32 nodes with 2 nodes per switch averaged 5.8 microseconds. Given that a batch of communications often takes on the order of seconds, this time cost is not considerable.

A.1 `graph.H`

```
#ifndef _GRAPH_H
#define _GRAPH_H

/**
 * The Graph class is not a general purpose graph.  It is rather a
 * very special case of graph, a bipartite multigraph with equal
 * numbers of vertices on each side, AND with a maximum degree that is a
 * power of two, AND it is only useful for purposes of edge coloring
 * when the graph is full and regular.  It was built with these
 * restrictions in mind.
 *
```

```
 * This is not intended to be an incredibly unbreakable piece of code:
 * it is assumed that the parameters passed into functions are
 * restricted to what their descriptions say they should be.  There is
 * NO error checking on parameters for speed reasons: proper external
 * usage should run all checks themselves.
 *
 * Here's example usage in coloring a graph:
 *
 * - First, you instantiate a Graph.  If you want a graph with 3 vertices
 * with 4 edges per vertex, you say "new Graph(2, 3)".
 *
 * - Then, you repeatedly add edges.  Each edge must also have a
 * "data" parameter passed in to identify it.
 *
 * - Once all edges are added, color() is called.  This makes the
 * graph regular and applies an algorithm that colors the bipartite
 * graph.
 *
 * - One can then extract which edges are colored a certain way by
 * using the "dataForColor" method, which, given a color, puts all of
 * the "data" parameters into an array and returns how many edges are
 * that color.
 *
 * - If one wants to reuse the structure, call empty() to restore the
 * graph to the virgin state.
 */

class Graph {
public:
  Graph(unsigned int degreePower, unsigned int vertices);
  ~Graph();
  void addEdge(const unsigned int left, const unsigned int right,
               const unsigned int data);
  void printGraph() const;
  void color();
  unsigned int dataForColor(unsigned int * data,
                            const unsigned int color) const;
  void empty();

  // Returns the log_2 of the maximum degree of any vertex.
  inline unsigned int degreePower() const { return myDegreePower; }
  // Returns the maximum degree of any vertex.
  inline unsigned int degree() const { return myDegree; }
  // Returns the number of vertices in either party (both parties have
  // equal numbers of vertices).
  inline unsigned int vertices() const { return myVertices; }
  // Returns the number of edges presently in this graph.
```

```
  inline unsigned int edges() const { return myEdges; }
private:
  struct Edge {
    unsigned int to;    // Which vertex in the opposite party?
    unsigned int index; // Which indexed vertex in the opposite party?
    unsigned int data;  // The "data."
    bool real;          // Is this a real or "filler" edge?
  };

  struct Vertex {
    unsigned int degree;
    Edge * edges;
  };

  void addEdge(unsigned int left, unsigned int right,
               unsigned int data, bool real);
  inline void fill();

  void recurseSplit(unsigned int bottom, unsigned int size);
  inline void eulerSplit(const unsigned int bottom, const unsigned int size);

  inline void swapRight(unsigned int vertex, unsigned int a, unsigned int b);
  inline void swapLeft(unsigned int vertex, unsigned int a, unsigned int b);

  Graph::Vertex *leftParty, *rightParty;
  unsigned int myDegreePower, myDegree, myVertices, myEdges;
};

#endif // _GRAPH_H
```

A.2 graph.C

```
#include "graph.H"
#include <stdio.h>

/**
 * Construct a bipartite multigraph.  For n = degreePower, this
 * construct a graph with degree 2^n.  The parameter "vertices"
 * indicates the number of vertices in either the left or right party.
 */
Graph::Graph(unsigned int degreePower, unsigned int vertices) {
  myDegree = 1<<(myDegreePower = degreePower);
  myVertices = vertices;
  myEdges = 0;

  // Allocate whatever memory needs to be allocated.
  // Allocate the verticess.
  leftParty = new Graph::Vertex[myVertices << 1];
```

```
  rightParty = leftParty + myVertices;
  // Allocate the edges, and set them into the vertices.
  Graph::Edge * edges = new Graph::Edge[myVertices << (myDegreePower+1)];
  for (Graph::Vertex * ni = leftParty; ni<(leftParty+(myVertices<<1)); ni++) {
    ni->degree = 0;
    ni->edges = edges;
    edges += myDegree;
  }
}

/**
 * Frees all malloced data declared in the constructor.
 */
Graph::~Graph() {
  // Due to the nature of how we block malloced the data, this should
  // be sufficient.
  delete[] leftParty[0].edges;
  delete[] leftParty;
}

/**
 * The addEdge function adds an edge between the indicated left and
 * right user vertices.  The left and right index must be between 0
 * inclusive and the maximum number of vertices noninclusive.
 *
 * The data number is a unique identifier for this vertex: when the
 * color() algorithm is finished, one uses the dataForColor() to
 * extract this data.
 */
void Graph::addEdge(const unsigned int left, const unsigned int right,
                    const unsigned int data) {
  addEdge(left, right, data, true);
}

/**
 * This is the same as the above addEdge, except that there is the
 * option for making a vertex a "filler" vertex by setting the real
 * parameter to false, and that this is a private function.  A filler
 * vertex acts just like a regular vertex, but is simply flagged as
 * not being real.  As the name filler implies, it is simply filler so
 * that the Euler split algorithm is gauranteed to work correctly.
 */
void Graph::addEdge(const unsigned int left, const unsigned int right,
                    const unsigned int data, const bool real) {
  // Get the pointers for a convinient reference.
  Graph::Vertex *l = leftParty + left, *r = rightParty + right;
  Graph::Edge *le = l->edges+l->degree, *re = r->edges+r->degree;
```

```
  // Where is each edge going to in the opposite party?
  le->to = right;
  re->to = left;
  // Have the edges reference each other.
  le->index = r->degree;
  re->index = l->degree;

  // Is this a real (i.e., non filler) vertex?
  re->real = le->real = real;

  // Up the degree on each vertex.
  l->degree++;
  r->degree++;

  // Bookkeeping with regards to edges.
  myEdges++;

  // Unique identifier.  Only the left vertex needs to know about this,
  // since when extracting data according to color we only scan the
  // left vertices.
  le->data = data;
}

/**
 * Colors the graph.  At the lowest level, each vertex has an array
 * with edges.  The left vertices have their arrays with edges, as do
 * the right verticess, so EVERY edge in this bipartite multigraph
 * appears twice, once in its left vertex and one in its right vertex.
 * Naturally, the index of the edge in either array varies.
 *
 * The way the coloring algorithm works is that it performs a rather
 * quick algorithm that works in O(N*D log D) time.  The only thing
 * this algorithm does is gaurantee that for any edge, if the entry
 * for the edge at its left vertex is at index I, the entry for the
 * edge at its right vertex is at index I as well.
 *
 * The reason that this "symmetric ordering" is a proper coloring
 * algorithm lies in the definition of edge coloring: an edge coloring
 * is a proper edge coloring iff for any edge connecting vertices A
 * and B, its color is not the same as any other edge emanating from A
 * or B.  This index "I" which this edge is present at in both its own
 * vertices A and B can be thought of as the index for the edge
 * itself, and surely this index must be unique at both vertices,
 * since otherwise we'd have edges overwriting each other.
 *
 * In this course of a single coloring this will be called once.
```

```
 */
void Graph::color() {
  // Make sure we're in good state, that is, that the graph is regular
  // and of degree power of two.
  fill();
  // In the special case where our degree is 1, any processing is
  // unnecessary.  (We de need the fill for color extraction,
  // however.)
  if (myDegree==1) return;
  // Do recursive splits.
  recurseSplit(0, myDegree);
  // Readjust the degrees to the correct state.
  for (Graph::Vertex *n = rightParty+myVertices-1; n>=leftParty; n--)
    n->degree = myDegree;
}

/**
 * Puts the data for all the colors into the "data" array.  Due to the
 * nature of the edge coloring problem, the maximum bound on the
 * number of elements written to the array is necessarily less than or
 * equal to the number of vertices, so the data parameter should point
 * to a block of memory of at least that size.  The color parameter is
 * a number that may range from 0 through edges-1 that indicates which
 * color we're extracting information for.  The return value is the
 * number of entries in this data array.
 *
 * This method will not fail under any circumstances, provided the
 */
unsigned int Graph::dataForColor(unsigned int * data,
                                 const unsigned int color) const {
  unsigned int entries=0;
  // After the end of the color sequence, the position of a vertex
  // within the edge array implies its color.
  for (unsigned int n=0; n<myVertices; n++)
    if (leftParty[n].edges[color].real)
      data[entries++] = leftParty[n].edges[color].data;
  return entries;
}

/**
 * Empties the graph.
 */
void Graph::empty() {
  for (Graph::Vertex *n = rightParty+myVertices-1; n>=leftParty; n--)
    n->degree = 0;
}
```

```
/**
 * The filler function does a pass through the graph, and makes sure
 * that the graph is full (all vertices have the maximum degree) and
 * regular (all vertices have equal degree).  Whatever edges need be
 * added are added as filler edges.  This function is intended for use
 * so that the Euler split algorithm can work without error.
 *
 * In this course of a single coloring this will be called once.
 */
inline void Graph::fill() {
  unsigned int l=0, r=0;
  while (myEdges < (myVertices<<myDegreePower)) {
    // Scan to the next non-empty left vertices.
    while (leftParty[l].degree == myDegree) l++;
    // Scan to the next non-empty right vertices.
    while (rightParty[r].degree == myDegree) r++;
    // Add the edge.
    addEdge(l, r, false);
  }
}

/**
 * A convinience function that swaps the places of edges in the edge
 * array on the right party.
 *
 * In the course of a single coloring, this function will be called
 * (1/2)*E*log D times.
 */
inline void Graph::swapRight(unsigned int vertex, unsigned int a,
                             unsigned int b) {
  if (a == b) return;

  // Do the actual swap using struct copies.
  Graph::Edge *ea = rightParty[vertex].edges + a,
    *eb = rightParty[vertex].edges + b;
  Graph::Edge temp = *ea;
  *ea = *eb;
  *eb = temp;

  // Change the references in the left party.
  leftParty[ea->to].edges[ea->index].index = a;
  leftParty[eb->to].edges[eb->index].index = b;
}

/**
 * A convinience function that swaps the places of edges in the edge
 * array on the left party.
```

```
 *
 * In the course of a single coloring, this function will be called
 * (1/2)*E*log D times.
 */
inline void Graph::swapLeft(unsigned int vertex, unsigned int a,
                            unsigned int b) {
  if (a == b) return;

  // Do the actual swap using struct copies.
  Graph::Edge *ea = leftParty[vertex].edges + a,
    *eb = leftParty[vertex].edges + b;
  Graph::Edge temp = *ea;
  *ea = *eb;
  *eb = temp;

  // Change the references in the left party.
  rightParty[ea->to].edges[ea->index].index = a;
  rightParty[eb->to].edges[eb->index].index = b;
}

/**
 * Recursively performs Euler splits on the graph.
 *
 * In the course of a single coloring, this function will be called
 * D-1 times.
 */
void Graph::recurseSplit(unsigned int bottom, unsigned int size) {
  // Split the graph into two "subgraphs" G_1 and G_2.
  eulerSplit(bottom, size);
  if (size <= 2) return;
  size >>= 1;
  recurseSplit(bottom, size);           // Recursively split G_1.
  recurseSplit(bottom + size, size);    // Recursively split G_2.
}

/**
 * Perform an Euler split on some section of the vertex edgespace.  That
 * makes no sense.  Essentially what the algorithm does is do repeated
 * Euler splits on the edges, starting with the population of ALL the
 * edges.  The edges on each vertex are rearranged within the array of
 * edges for each vertex: those in the first half of the split go into
 * the first half of the array, those in the second half of the split
 * go into the second half of the array.  This allows one to do an
 * Euler split on a smaller section of the edge population.
 *
 * For example, for coloring a degree 4 graph, one would do a split
 * with (0,4), and THEN do a split with (0,2) and (2,2).  At the end
```

```
 * of the process, the color would then be implied by the position of
 * the edge within the graph.
 *
 * In the course of a single coloring, this function will be called
 * D-1 times.
 */
inline void Graph::eulerSplit(const unsigned int bottom,
                              const unsigned int size) {
  // This is the number of edges that we have to do before our Euler
  // split is truly accomplished.
  unsigned int toProcess = size * myVertices, start=0, curr;

  // For each vertex we must know how many we need to do.  The "degree"
  // of each vertex will do nicely.
  for (unsigned int n=0; n<myVertices; n++)
    leftParty[n].degree = rightParty[n].degree = size;

  // While we still have more edges to process, continue.
  while (toProcess) {
    // Find a nice starting place.
    while (!leftParty[start].degree) start++;
    // Set the current vertex to the starting vertex.
    curr = start;

    // While our current vertex isn't a dead end...
    while (leftParty[curr].degree) {
      unsigned int L0, L1, R0, R1, to;

      // Which edge are we going to follow?
      L0 = bottom+((size-leftParty[curr].degree)>>1);
      // What is the original index of the corresponding right edge index?
      R0 = leftParty[curr].edges[L0].index;
      // Where SHOULD the right guy be?
      to = leftParty[curr].edges[L0].to;
      R1 = bottom+((size-rightParty[to].degree)>>1);
      // Move into the correct position.
      swapRight(to, R0, R1);

      // Decrement the degree of the vertex on the left.
      // We leave right alone for now for convinience of calculation.
      leftParty[curr].degree--;

      // Follow that edge over to a vertex in the right hand side.
      curr = to;

      // Find an edge in this right party vertex that has not been
      // split yet.
```

```
      R0 = bottom+((size+rightParty[curr].degree)>>1)-1;
      // What is the index of the correspoding left vertex for this edge?
      L0 = rightParty[curr].edges[R0].index;
      // Find an index in the edge array to move this edge to.
      to = rightParty[curr].edges[R0].to;
      L1 = bottom+((size+leftParty[to].degree+1)>>1)-1;
      // Move into the correct position.
      swapLeft(to, L0, L1);

      // We've processed two edges on the right side.
      rightParty[curr].degree-=2;
      // Follow the edge we just removed over to the other vertex.
      curr = to;
      // We've processed one edge on the left side.
      leftParty[curr].degree--;

      // In all, we've just done 2.
      toProcess -= 2;
    }
  }
}

/**
 * A very simple debugging function.  This will go through each of the
 * vertices in the left party, and for each vertex print out the vital
 * statistics for each vertex, and then its edges including the left
 * index then the right index of the vertices each edge connects.
 */
void Graph::printGraph() const {
  printf("THE GRAPH (das grafen)\n");
  for (unsigned int n = 0; n < myVertices; n++) {
    printf("%d : Address %p, edges %p\n",
           n, leftParty+n, leftParty[n].edges);
    for (unsigned int e = 0; e < leftParty[n].degree; e++) {
      printf("\t%4d -> %4d, at index %d%s\n",
             n, leftParty[n].edges[e].to, leftParty[n].edges[e].index,
             leftParty[n].edges[e].real ? "" : " FILLER");
    }
  }
}
```

A.3 main.C

```
// Set <algorithm> to use srand, rand.  We're not doing cryptography...
#define __STL_NO_DRAND48

// Set to 1 to have the permutation and the route output.
#define DO_OUTPUT 1
```

```
// The release version has the profiler disabled.
#define PROFILER 0

#include <algorithm>
#include <cstdlib>
#include <stdio.h>
#include <time.h>       // For the random seed.
#include "graph.H"

#if PROFILER
#include "profiler.H"
#endif

/**
 * This is test code that uses the Graph class to find routings for
 * randomly generated permutations.  The findRoute function is that
 * which finds routes, and its comments describes its functionality.
 * The number of tests to run, the number of nodes on each switch, and
 * the number of nodes.  Running the program will describe the command
 * line options.
 *
 * Thomas Finley
 */

char format[] = "%Xd";

void argumentError();
void handleArguments(int argc, char** argv);
void findRoute(int * to);
void generateCase(int * to, unsigned int free);

char * name;
// These globals are preinitialized to defaults.
unsigned int r=1,       // Runs
  n=24,                 // "Nodes" (computers)
  d=3;                  // Log_2 of the # of computers per line card.

/**
 * Prints out proper command-line-ed-ness of the program, then exits.
 */
void argumentError() {
  fprintf(stderr, "Usage: %s runs [nodes [d]]\n", name);
  fprintf(stderr, "  runs  - number of random cases to test\n");
  fprintf(stderr, "  nodes - number \"computers\" connected (default 24)\n");
  fprintf(stderr, "  d     - 2^d is the # of computers per line card "
          "(default d=3)\n");
  exit(-1);
```

```
}

/**
 * Call 1-877-BAD-CODE for more information on how you can further the
 * cause of spaghettiness in YOUR community.  Sets the global
 * parameters based on command line arguments.
 */
void handleArguments(int argc, char** argv) {
  if (argc < 2) argumentError();
  if (sscanf(argv[1], "%d", &r) != 1) argumentError();
  if (argc > 2) if (sscanf(argv[2], "%d", &n) != 1) argumentError();
  if (argc > 3) if (sscanf(argv[3], "%d", &d) != 1) argumentError();
}

/**
 * The parameter "to" passed in is an "n" length integer array, where
 * each element corresponds to a communication, telling which node the
 * node that corresponds to the array index is talking to.  For
 * example, if to[5] == 2, that means that node 5 wants to talk to
 * node 2, etc.  A negative quantity means that node is not talking to
 * anybody.
 *
 * The values are passed back in the passed in array.  Each element is
 * replaced with the index of the intermediate crossbar they are
 * supposed to use.  If the corresponding computer wasn't talking to
 * anyone, the negative number remains in its place.
 */
void findRoute(int * to) {
  // Graph vertices are the number of line cards needed if there are n
  // computers, and each line card connects to 2^d computers.  On the
  // default network with n=24 and d=3, 24 computers can connect to 3
  // line cards that each hold 8 computers each.  n=25 would require 4
  // line cards.  (Equivalently, graphVertices is the smallest number
  // of 2^d capacity containers that can contain n objects.)
  static unsigned int graphVertices = (n>>d) + ((n&((1<<d)-1)) ? 1 : 0);

  // This is a static declaration so we're confident that use of the
  // Graph structure is limited to this one function.
  static Graph c(d, graphVertices);

  // When we add edges as we do here, the left vertex in added edges
  // is the LINE CARD of the source, and similarly the right vertex is
  // the line card of the destination.  The numerical ID in the range
  // [0,n) of the actual sender is cast as a (void *) and passed in as
  // data since I don't care to allocate a special structure...
  for (unsigned int i=0; i<n; i++)
    if (to[i] >= 0)
```

```
      c.addEdge(i>>d, to[i]>>d, i);

  // This will color the graph with 2^d colors.
  c.color();

  // For each color, extracts the data, where data is the sender.
  for (int i=(1<<d)-1; i>=0; i--) {
    static unsigned int * array = new unsigned int[1<<d];
    for (int j=c.dataForColor(array, i)-1; j>=0; j--) {
      to[array[j]] = i;
    }
  }

  // At this point we ready the coloring structure for another run.
  c.empty();
}

/**
 * This will generate a sequence of communicating pairs, as the same
 * format as the "to" array described in the the findRoute function.
 * "to" should point to a preallocated array of integers of size n.
 * The "free" determines how many pairs of this list are NOT
 * communicating, that is, are inactive pairs.
 */
void generateCase(int * to, unsigned int free) {
  for (unsigned int i=0; i<n; i++)
    to[i] = i;
  random_shuffle(to, to+n);
  if (free) {
    // You bet I'm lazy.
    free = free>n ? n : free;
    for (unsigned int i=0; i<free; i++)
      to[i] = -1;
    random_shuffle(to, to+n);
  }
}

/**
 * The output format for this program depends (sort of) on having
 * columns of data line up, so the format string must have numbers
 * printed on equally spaced fields.  We check the maximum number of
 * digits for the maximum number, and set the character on the format
 * string appropriately.
 */
void setupFormatString() {
  unsigned int max = (n-1) > ((unsigned) 1<<d) ? (n-1) : (1<<d);
  unsigned int digits = 1;
```

```
  while (max /= 10) digits++;
  format[1] = '1'+digits;
}

int main(int argc, char** argv) {
  name = argv[0];
  handleArguments(argc, argv);
  srand(time(0));

  // A simple timer that is disabled for the release version.
#if PROFILER
  Profiler p;
#endif

  setupFormatString();

  int * to = new int[n];
  while (r--) {
    generateCase(to, 0);

#if DO_OUTPUT
    // This prints out the permutation.  A permutation is given as a
    // simple list of integers.  If n=to[i], then in this permutation
    // sender i sends to receiver n.
    printf("\n");
    for (unsigned int i=0; i<n; i++)
      printf(format, to[i]);
    printf("\n");
#endif

    // The use of the Profiler times the relevant sections.
#if PROFILER
    p.click();
#endif
    findRoute(to);
#if PROFILER
    p.click();
#endif

#if DO_OUTPUT
    // For each of the communications earlier printed, this will print
    // the route crossbar through which each communication will go.
    for (unsigned int i=0; i<n; i++)
      printf(format, to[i]);
    printf("\n");
#endif
  }
```

```
#if PROFILER
  fprintf(stderr, "Routing took %d user, %d system useconds\n",
    p.microseconds(true), p.microseconds(false));
#endif

  return 0;
}
```