ORCS: An Oblivious Routing Congestion Simulator

Timo Schneider, Torsten Hoefler, and Andrew Lumsdaine
{timoschn,htor,lums}@cs.indiana.edu

Open Systems Laboratory, Indiana University
150 S Woodlawn Ave, Bloomington, IN 47405, USA

February 18, 2009

1 Introduction

Bisection Bandwidth, as defined by Hennessy and Patterson in [4] as the bandwidth between the two equal sized halves of the network for the worst case partition, is widely used as a theoretical model for network performance. This model gives an upper bound for the minimal bisection bandwidth, as experienced by applications, of a network, as it does not take the used routing scheme into account. It has been proven that oblivious static routing, where there is one fixed path through the network for each $(source, destination)$ pair, is suboptimal for various network topologies [8]. However, oblivious routing is easy to implement and delivers low latencies because no computation is needed to route packets, since the routes can be determined off-line. Therefore, it is used by several high performance networks [1, 6, 10]. InfiniBand is one of the interconnection fabrics that use oblivious static routing. In [5] we showed that the effective bisection bandwidth, that can be measured for adequate communication patterns, is significantly lower than the bandwidth predicted by the bisection bandwidth model.

In our experiments none of the examined InfiniBand networks was able to deliver more than 61% of the bisection bandwidth, due to network congestion. Other effects that would deteriorate performance, for example flow control mechanisms, have not been taken into account, we only studied congestion by simulating traffic patterns and verified the correctness of our model by measurements.

To study the effect of congestion on large scale clusters, a part of the FASTOS II project[1], we developed a framework to simulate the congestion in oblivious destination based routed networks. In this work, we will explain our simulator and the related tools that we used. The design of our simulator framework is modular, so it could be extended to simulate different traffic patterns (a variety of them is already predefined as shown in Section 2.4), or using different approaches to present the data gathered during the simulation runs as those described in Section 2.5.

We will continue with a brief explanation of the routing scheme used by InfiniBand and why networks using such routing strategies might not deliver full bisection bandwidth for applications, even if the network topology is theoretically capable of doing so. Section 2 documents how our simulator can be used, and will give examples for analysis that can be performed with this software package. In Section 3, we will explain the usage of tools that enable the user to study the different routing algorithms supported by OpenSM, the InfiniBand subnet manager, with our simulator. We conclude with a description of the simulator implementation that will enable users to gain a deeper insight on how the simulator works in Section 4.

1.1 InfiniBand Routing

The InfiniBand standard [6] does not define a particular network topology, switches can be connected in an arbitrary way to each other. The switches have a simple static routing table which is programmed by a central entity, the

[1] http://www.science.doe.gov/grants/LAB07_23.html

subnet manager (SM). The SM uses special packets to explore the network topology in the initialization phase. It then computes the routing table for every switch. These routing tables are static, which means they do not change until the SM re-runs the configuration phase, which is done for example, in case of failing links.

If a hosts wants to send a packet to another host, this packet is marked with its destination and send to a switch the sending host is connected to. This switch can determine which port has to be used to send this packet further along with a simple lookup operation on the linear forwarding table (LFT). This table lists exactly one outgoing port for every destination. If messages to a certain destination are never routed to a particular switch by other switches, this switch does not have to list outgoing ports for that destination. Because of the described structure of the forwarding tables, there is only on path $p = [e_1, ...e_n]$, where e_i are the used physical links (or edges, in graph terminology), between each sender and receiver pair (s, r) that is used to send data, even if there are multiple paths between them in the network. This means, the bandwidth between two disjoint sets of hosts is not only determined by the number of links going from one partition two another, but also by the way the routes are distributed along the physical links. Researchers have pointed out that this problem could be mitigated by using adaptive, non-oblivious routing schemes [9, 3]. However it has also been shown that the features provided by InfiniBand to support different paths between a single sender receiver pair (LMC and Virtual Lanes), where the sending node determines which one should be used, are not sufficient to improve network performance significantly [2]. Our simulator does not support these InfiniBand features.

If two data streams have to share a physical link, the bandwidth experienced by each stream is of course lower than it would have been if each data stream would have been routed along a disjoint set of edges. This effect is called congestion. We analyzed the effect of congestion on bandwidth and latency in [5].

2 Our Simulator

To explore the effect of congestion as described in Section 1.1 in large real world network topologies, we developed a simulator program. This simulator takes a network topology (the used input file format is described in Section 2.1) and a traffic pattern (the predefined patterns are described in Section 2.4) to simulate the communication described in this pattern as it would happen in the given InfiniBand network. As a result, the impact of congestion on this communication pattern can be observed. We use various different metrics to assess this impact, our metrics will be described in Section 2.5. These metrics do not change the simulation, they describe the way which is used to reduce the data gathered during the simulation run into the final result.

2.1 Network Topology Input Format

Networks can be described as directed graphs $G = (V, E)$ where V is the set of vertices. Each vertex resembles one node in the network, either a host or a switch. Each edge $(u, v) \in E$ resembles a uni-directional physical link. Since the physical InfiniBand links are full-duplex there should be an edge $u \leftarrow v$ if, and only if, $v \leftarrow u$ exists in the network topology graph. Note that such a graph could be a multigraph, since there can be more than one physical link between two network nodes.

The network topology for the simulated network is given to the simulator as a directed graph in the dot format. The dot format is described in detail in [7]. An example for a graph in the dot format is given in Listing 1. Our simulator differentiates between two types of nodes: hosts and switches. All nodes that have a name starting with the letter "H" are recognized as hosts, all other nodes are regarded as switches.

The routing in a network determines the path that traffic between two hosts has to take. So for every pair (u, v) of vertices in the network graph there is a list of edges $e_1, ..., e_n \in E$ that will be used to get traffic from u to v. InfiniBand employs distributed static routing, that means if host H1 sends a packet to H2 there is exactly one path[2] that this packet will take. Since the routing is static this path is determined solely by the sending and receiving node and is not adapted over time due to congestion or other factors. The routing is called distributed because a sending host doesn't have to compute the complete path

[2] If LMC is used there can be a fixed number of different paths, selected by the source node for each transmission, usually in a round-robin scheme. Note that the current version of our simulator does not support LMC.

```
digraph network {
    "S1" -> "H1" [ comment = "H1" ];
    "S1" -> "H2" [ comment = "H2" ];
    "S1" -> "H3" [ comment = "H3" ];
    "S1" -> "H4" [ comment = "H4" ];

    "H1"  -> "S1" [ comment = "*" ];
    "H2"  -> "S1" [ comment = "*" ];
    "H3"  -> "S1" [ comment = "*" ];
    "H4"  -> "S1" [ comment = "*" ];

    "S1" -> "S11" [ comment = "H5" ];
    "S1" -> "S11" [ comment = "H6" ];
    "S1" -> "S12" [ comment = "H7" ];
    "S1" -> "S12" [ comment = "H8" ];

    "S11" -> "S1" [ comment = "H1,H2" ];
    "S11" -> "S1" [ comment = "H3,H4" ];
    "S12" -> "S1" [ comment = "H1,H2" ];
    "S12" -> "S1" [ comment = "H3,H4" ];
}
```

Listing 1: *Network topology with routing information*

to the receiving host, it just has to determine which outgoing physical link has to be used to get to the next node. This next node will then continue to send the data to its destination.

As shown in the example, not only the network topology is described in the dot file, but it also contains routing information. An edge in a dot graph can have arbitrary information attached. We add the routing information in comments attached to these edges. For example the definition for the edge between S11 and S1 in Listing 1 contains the comment string "H1,H2": this means that packets that have been routed to S11 are supposed to take this edge to reach the next hop if and only if their final destination is one of the hosts H1 or H2. Hosts are commonly connected to exactly one switch with one physical link. That means that all traffic emitted by a host has to use the respective edge in the network graph since there is no other way to leave the host. In that case all hosts (except the sending one) would have to be listed in the comment string for such edges. To reduce the input file size, we introduced the "*" character which matches any host in the network. This is shown in lines 7–10 in Listing 1.

```
# ibdiagnet -v -o .
# ibnetdiscover -s > ibndisc.out
$ get_network_graph ibdiagnet.fdbs
    ibndisc.out > topo.dot
```

Listing 2: *Extracting network topology and routing information from a InfiniBand network and transforming those into a simulator input file*

2.2 Obtaining Network Topology and Routing from InfiniBand Networks

If one wants to examine an existing InfiniBand installation with our simulator the network topology and routing information has to be provided in the format described in Section 2.1. Writing such files by hand is tiresome and error prone. Luckily the InfiniBand software stack provides tools to extract all the needed information from a working network installation. These tools, `ibdiagnet` and `ibnetdiscover`, can be used in conjunction with the `get_network_graph` script in our simulator package[3]. An example session to illustrate the usage of these tools is given in Listing 2.

Note that ibdiagnet and ibnetdiscover have to be executed with superuser privileges. Both tools are part of the OFED[4] software package. The ibdiagnet tool will write a variety of output files named ibdiagnet.* into the current working directory, where ibnetdiscover writes to stdout and therefore its output was redirected to a file called ibndisc.out. Our conversion script has to be called with two command line arguments, the ibdiagnet output file and the ibnetdiscover output file. The resulting dot graph is written to stdout and can be redirected into a file as shown in Listing 2.

2.3 Using the Simulator

In this section we will list the different command line options for our simulator and explain how they change the simulator's behaviour. In general, the simulator is invoked by `orcs [OPTIONS]`. The following options are valid:

`-h, --help` List all supported command line options

[3]Which can be obtained form http://www.unixer.de/ORCS/

[4]Which can be obtained from http://www.openfabrics.org/

`-V, --version` Prints the simulators version number
`--printptrn` Prints the communication pattern for every level. This is useful to check that the pattern looks as expected. For each level, the pattern is printed as a list of "sender → receiver" pairs. Note that the pattern will stay constant for every simulation iteration, the mapping is done independently. So it is not useful to use this flag for more than one iteration.
`--printnamelist`: Prints the used subset of hosts. This is only useful if the communicator size is smaller than the number of hosts in the input file.
`-v, --verbose` Prints a message for every level that has been simulated. This can be used to keep track of lengthy simulations with a large number of iterations or a big network input file.
`-s, --commsize=INT` This option controls the number of hosts that should be used for the simulation. If this parameter is 0, which is the default, then the simulator will use all available hosts. Note that some patterns only work for an even number of nodes. In this case the given number is diminished by one.
`-n, --num_runs=INT` With this option you can control the number of simulation iterations that will be performed. If the simulator is run with p MPI processes and is supposed to do c iterations each process will do $\lceil \frac{c}{p} \rceil$ iterations. By default the simulator only performs one simulation run.
`-p, --ptrn=STRING` Selects the pattern to use for the simulation. All valid patterns are described in detail in Section 2.4. If this option is omitted, the "bisect" pattern is used.
`--subset=STRING` Available options for this parameter are "rand" and "linear_bfs". If the "rand" option is used and the communicator size is smaller than the number of hosts in the topology input file, the hosts are randomly selected from all hosts available. This is intended to mimic a scenario where jobs of different size and runtime are assigned to a cluster by the batch system as soon as enough nodes become available. This will lead to "fragmentation"—a new job could get a fairly random selection of nodes. This influences the congestion the data transfers done by this job will experience since the probability for two hosts sharing the same leaf switch (where no congestion will happen) is smaller. With the "linear_bfs" option (the default) the simulator will perform a breadth-first search and order all nodes in their exploration order. The first nodes (specified by the communicator size) will then be used. This ensures that the minimal number of leaf switches are used.
`--metric=STRING` This option specifies which metric is to be used to measure the influence of congestion on the simulated communication patterns. The available metrics are described in detail in Section 2.5.
`-l, --ptrn_level=INT` This parameter can be used to only simulate a certain level of a particular pattern. Note that the number and also the amount of communication inside a level depends on the communicator size for most patterns. The different levels of a communication pattern with a certain number of participating nodes can be observed with the printptrn option.
`-i, --input_file=STRING` This option specifies the location of the network topology input file. It's format is specified in Section 2.1. If the input is to be read from STDIN, the user can specify `-` as the input file name or this option can be omitted completely (since reading from STDIN is the default behaviour).
`-o, --output_file=STRING` This option works similar as the option for the input file and specifies where simulation results should be written to.

2.4 Communication Patterns

In this section, we will describe the different communication patterns that are available in our current simulator implementation. The pattern is selected with the `-p, --ptrn=STRING` command line option. Valid parameters are: "rand", "null", "bisect", "bisect_fb_sym", "tree", "bruck", "gather", "scatter", "ring", "recdbl", "2neighbor", "4neighbor", "6neighbor" and "ptrnvsptrn". The communication patterns for our simulators can have multiple levels. Each level is simulated independently, data transmissions in one level do not influence transmissions in other levels. Each level is defined as a set of sender and receiver pairs, all data transmissions defined in one level will happen in parallel and all messages in our simulation are of equal length. So a communication pattern can be defines as a list of sets of sender-receiver pairs, for example the pattern $[\{(0,1),(2,3)\},\{(1,0),(3,2)\}]$ describes a communication between four nodes which has two levels. In level 0 two nodes send a message to

the other two nodes, in level 1 data is sent in the opposite direction. Note that the numbers in the communication pattern do not correspond to any particular node in the network topology input file. The mapping of nodes in the communication pattern to hosts in the network is done randomly before every simulation iteration.

The **rand** pattern: in this communication pattern every node sends a single datastream to one randomly selected node and receives a single datastream from one randomly selected node. So the communication pattern for a communicator size of n is defined as: $\{(s_0,\ r_0),\ ...,\ (s_{n-1},\ r_{n-1})\}$ where
$(s_i = i)\ \wedge\ (0 \leq r_i \leq n-1)\ \wedge\ (\forall i \neq k : r_i \neq r_k)$

The **null** pattern: this communication pattern is only useful in conjunction with the ptrnvsptrn communication pattern. In the null pattern no communication happens. It is defined as: $\{\}$ for all communicator sizes.

The **bisect** pattern splits the hosts in the network in equally sized halves. Each node in the first half sends a data stream to a node in the second half. If the number of hosts in the network is odd, and therefore we can not form equal sized partitions, one host is ignored. So the pattern is defined as $\{(s_0,\ r_0),\ ...,\ (s_k,\ r_k)\}$ where $k = \lfloor \frac{n}{2} \rfloor - 1$ and $s_i = 2i + 1$ and $r_i = 2i$.

The **bisect_fb_sym** is similar to the bisect pattern described before. While the flow of data was unidirectional in that pattern because there was a "sender" and a "receiver" partition, traffic is send in both directions in this pattern. The definition of the pattern is $\{(s_0,\ r_0),\ ...,\ (s_k,\ r_k), (s\prime_0,\ r\prime_0),\ ...,\ (s\prime_k,\ r\prime_k)\}$ with $k = \lfloor \frac{n}{2} \rfloor - 1$ and $s_i = 2i + 1$ and $r_i = 2i$, where $s\prime_i = r_i$ and $r\prime_i = s_i$.

The **tree** pattern simulates a binomial tree. The pattern consists of $\lceil log_2 n \rceil$ levels, where n is the communicator size. Figure 6 shows an example of a binomial tree with eight nodes. In each level $l \in \{0, ..., \lceil log_2 n \rceil - 1\}$ the communication "distance" is determined by $d = 2^l$. The pair $(i, i + d)$ is a member of level l if and only if $i + d < n$. So the pattern is defined as $[\{(s_0^0, r_0^0), ...\}, ..., \{(s_i^{\lceil log_2 n \rceil - 1}, r_i^{\lceil log_2 n \rceil - 1}), ...\}]$ with $s_i^l = i$ and $r_i^l = i + 2^l$ for all i, l where $i + 2^l < n$.

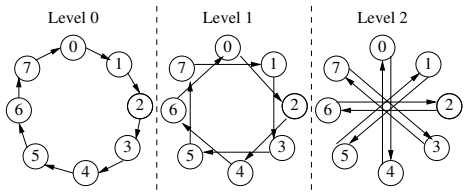

Figure 1: *Bruck pattern with eight nodes*

The **bruck** pattern simulates a pattern where each node is sending and receiving a message in every one of the $0 \leq l \leq < \log_2 n \rceil$ levels the pattern consists of, where n is the communicator size. Figure 1 shows an example of a bruck pattern with 8 nodes. The pattern is defined as $[\{(s_0^0, r_0^0), ..., (s_n^0, r_n^0)\}, ..., \{(s_i^{\lceil log_2 n \rceil - 1}, r_i^{\lceil log_2 n \rceil - 1}), ...\}]$ where $s_i^j = i$ and $r_i^j = i + 2^j mod n \rfloor$.

In the **gather** a single node receives a message from all other $n - 1$ nodes in the communicator. This pattern consists of a single level. So the pattern is defined as $\{(s_1, r_1), ..., (s_{n-1}, r_{n-1})\}$ where $s_i = i$ and $r_i = 0$.

The **scatter** pattern is very similar to the gather pattern described before, this time a single node sends a message to all other $n - 1$ nodes in the communicator. All communication happens in one level. So the pattern is defined as $\{(s_1, r_1), ..., (s_{n-1}, r_{n-1})\}$ where $s_i = 0$ and $r_i = i$.

The **ring** pattern describes a communication scheme where one node sends a message to another node, this node passes the data to the next node which has not participated in the communication yet, and so on, until all nodes received a message. In the last step the node which received the data most recently sends a message to the node which started the communication and thereby closes the ring. Because of the dependencies (we assume node k can not send the data before it received the message from node $k - 1$), each data transfer occurs in a single level. That means this pattern can never generate congestion by itself, but it is still useful in conjunction with the "ptrnvsptrn" pattern which we describe below.

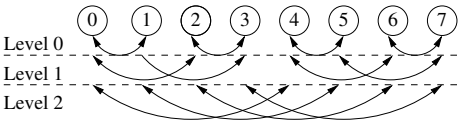

Figure 2: *Recursive doubling communication scheme for a communicator size of eight. The dotted lines indicate the level in which a certain communication happens.*

```
weight 1: 14 of the 64 connections
weight 2: 44 of the 64 connections
weight 3: 6 of the 64 connections
BW: 0.593750
```

Listing 3: *Example for the output of a simulation with the hist_max_cong metric: all connections established over the course of the simulation are treated equally and sorted in histogram bins according to their congestion*

For a communicator size of n the ring pattern is defined as $[\{(s_0^0, r_0^0)\}, ..., \{(s_0^{n-1}, r_0^{n-1})\}]$ with $s_0^j = j$ and $r_0^j = (j+1) mod n$.

The **recdbl** pattern consists of $\lceil \log_2 n \rceil$ levels for a communicator size of n. In every level $0 \leq l < \lceil \log_2 n \rceil$ hosts k and $k + 2^l$ exchange messages if, and only if, $\lceil \frac{k}{2^l} \rceil$ is even and $k + 2^l < n$. In Figure 2 we give an example for a recursive doubling communication pattern on a communicator with eight nodes.

The **Xneighbor** patterns simulate a communication scheme where every node sends and receives a message two or from X of its neighbors. We implemented this nearest-neighbor scheme for a one-, two-, and three-dimensional arrays, so there is a 2neighbor pattern as well as a 4neighbor and 6neighbor pattern. All communication in this pattern happens in a single level. When construct a pattern we start at node 0 and connect it to the nodes $1..X+1$ then we go to node 1 and connect it to the next X nodes that do not have enough neighbors already. We proceed with this scheme until we can not connect any two nodes without introducing a node with degree greater than X.

The **ptrnvsptrn** pattern is different from the ones described above, as it is not a single pattern but it enables the merge of two of the previously described patterns. It does so by adding the pattern given with the `--frstptrn` (or `-f`) option to the pattern given by the `--secptrn` (or `-c`) option. The communicator size for the first pattern is set with the command line option `-z` or its long form `--part_commsize`. The communicator size for the second pattern is the usual communicator size (indicated by `--commsize`), minus the communicator size for the first pattern. This pattern can be used to study the influence of two different communication schemes running simultaneously on the same network.

2.5 Congestion and Performance Metrics

Our simulator has five different ways to interpret the congestion data obtained by simulation of the traffic patterns described in the previous section. These different metrics determine the way output that is generated from the congestion maps for every simulation run. The metrics are called "sum_max_cong", "hist_max_cong", "hist_acc_band", "dep_max_delay", and "get_cable_cong". They will be explained in detail in this section.

The **sum_max_cong** metric adds the maximal congestion that occurred on any used route in every level in a communication pattern. This sum of maximal congestions is recorded as the result for one simulation run. After the last run a histogram of the results is printed. This metric is based on the assumption that every level has to be completed before the next level can start. Therefore the slowest message in every level determines the time needed to complete the entire pattern.

The **hist_max_cong** metric examines every single route used by any sender/receiver pair in a pattern. The maximal congestion along every route is saved in a histogram, regardless of the level it occurred in. Note, that in our model, the maximal congestion along a route determines the bandwidth of the simulated data transfer. We still simulate the different levels independently of each other. As a result, this metric prints a histogram of the congestions observed by every single data transfer simulated. It also calculates also the fraction of the

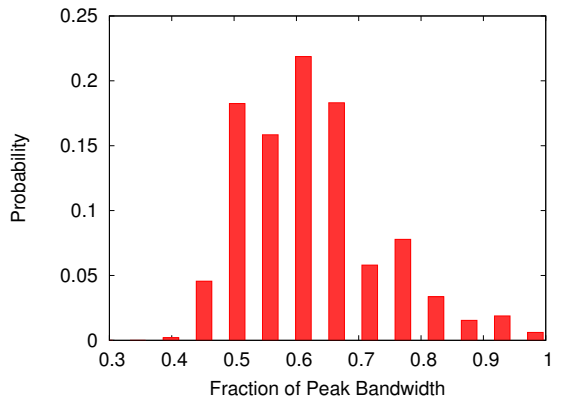

Figure 3: *A visualization of a simulation with the hist_acc_band metric: the probability distribution for the bandwidth achieved by a bisection bandwidth measurement pattern for a random mapping—it can be observed that the mapping has a large impact on the resulting performance.*

peak bandwidth that was achieved, using a simple linear congestion model where a congestion of two means half of the peak bandwidth. For example, if we simulate the bruck pattern defined above on a communicator with 16 nodes, this pattern will have 4 levels, with 16 message each. So all together 64 messages are transmitted. Therefore the results for this metric could look as shown in Listing 3. In this simulation run 14 of the 64 connections had been uncongested, while 44 connections experienced a congestion of two and six connections had two share at least one link with two other connections that were used simultaneously. For this simulation, only one iteration, with one mapping has been performed. If more iterations would have been done, the results of these would have been reported in the same histogram. So this metric does not distinguish between different levels or simulation runs (and mappings, as every simulation iteration uses a different mapping)—all established connections are treated equally.

The **hist_acc_band** is similar to the hist_max_cong metric described above: It treats all connections in all levels equally, determines their congestion factor and computes the fraction of peak bandwidth experienced by all the connections in one simulation run. However, the results of different simulation runs are not mixed. Every simulation run results in one number, the fraction of peak bandwidth for this particular run. These results are stored in a histogram and reported at the end. Since every simulation iteration used a different mapping of the communication tasks described in the pattern to the hosts in the network that perform them, this metric is suitable to analyse the significance of the mapping for a particular pattern and network. If the histogram consists only of a single peak, the mapping is rather insignificant because all mappings resulted in a similar bandwidth. If the range of observed bandwidth values is high, it is important to pay attention to the mapping or to choose a less mapping-sensitive communication scheme if possible, in order to get good performance. Figure 3 shows a visualisation of the histogram that resulted from 10000 simulation runs of the bidirectional bisection bandwidth measurement pattern, bisect_fb_sym, on a 16 node fat-tree with full bisection bandwidth. You can see the probability distribution for the bandwidth if a random mapping is simulated. It shows that full bisection bandwidth can be achieved, but only with a very small number of mappings, compared to the number of possible mappings.

The **dep_max_delay** metric is intended to be used together with the ptrnvsptrn communication pattern. This pattern merges two communication patterns by concatenating them level by level. For example if the first pattern is consists of $(0,1),(0,2),(0,3)$ (scatter with four nodes) and the second pattern is $(0,1),(2,3)$ (bisect pattern with four nodes) the concatenation would be $(0,1),(0,2),(0,3),(4,5),(6,7)$. The first pattern is used unchanged, the node numbers in the second pattern are increased by the largest node number in the first pattern, then the second pattern is merged with the first pattern. See Section 2.4 for usage information for this pattern. This metric only examines the congestion in the first pattern, which is defined via the `--ptrnfst` command line switch. It simulates the whole pattern. Than it weights every edge in the communication graph with the congestion factor of the route used for this communication. The length (sum of the weights along the edges) of the longest path through the communication

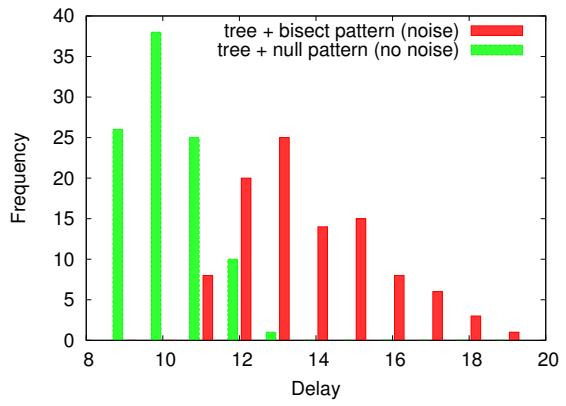

Figure 4: *A simulation with 100 iterations of the tree pattern on 800 hosts in a 1142-host fat-tree network. The rest of the nodes did not communicate (null pattern) in the "noiseless" case and used the bisect pattern in the "noise" case. This additional communication can double the execution time for the tree pattern.*

graph is reported as the result. We call this number the "delay" the pattern experienced because of congestion. This yields a single result for every simulation iteration. All delays are reported as a histogram at the end of the simulation. By comparing the histogram of a simulation with a pattern that uses the network to a histogram of a simulation with the null pattern (as the second part of the pattern), the influence of the "noise" induced by the second pattern can be studied. Figure 4 shows a visualization of the different histograms when this metric is used for 100 simulation runs of a tree pattern on 800 nodes of a 1142-host fat-tree network. The bisect pattern was used to generate "noise" on the rest of the hosts. In the noiseless case we utilized the null pattern. The graph shows how often a certain delay was observed in both cases.

The **get_cong_map** metrics aim is to help identifying bottlenecks in the network topology. After a simulation step is completed the cable congestion map contains the congestion factor for every edge that was used during this simulation step. With this metric the cable congestion map entries will be saved and further congestion entries, for the next level or simulation run, will be added. So after all simulation runs are finished we have recorded the sum of congestions that occurred over the entire simulation for every edge in the network topology graph. Edges that have never been used will have a congestion sum of zero. Then we will scale the congestion sums so that they become a number between zero and one by dividing them through the maximum of the recorded congestion sums. As a result this metric will print the network topology graph used as an input file, but every edge will be augmented with an additional property named, the scaled congestion sum. Also every edge will be given a color between green and red. The edges with the least relative congestion will be colored in green, where the edges that have a comparatively high congestion sum will be colored in red.

3 Evaluating other routing algorithms

With the tools explained in Section 2.2 one can obtain a topology graph and also routing information from a working InfiniBand Network[5]. However, OpenSM, the subnet manager for InfiniBand, currently supports five different routing algorithms. If one seeks to study the congestion behavior of those different algorithms, this could be done by reconfiguring the network once for each algorithm. This would require exclusive access to the cluster in question, since the InfiniBand network is not usable during reconfiguration. This makes it difficult to simulate large scale InfiniBand networks with different routing algorithms.

Luckily the OFED software stack also includes a tool called ibsim which simulates an InfiniBand network. It does so by exchanging the userspace library used by the subnet manager to send and receive management information to hosts and switches with a modified version of this library which emulates a network that can be specified by the user. This tool is limited in the amount of hosts it can emulate in its default configuration, but we were able to increase these limits with very small changes to the source code. We included these tools in our sim-

[5]Note that the tools we describe are Linux specific - they might work differently or not at all on Windows operating systems.

ulator software package, accompanied by some wrapper scripts that make the installation and usage easier. Note that the installation and usage of these tools does not need root privileges. The tools can be installed by executing the `install.sh` script. The ibsim tool will automatically be patched to be able to simulate up to 20480 hosts.

To use the ibsim tool to generate routing information for a network topology we need the network topology in a specific format, similar to the output of the ibnetdiscover tool. Our simulator package comes with a tool to convert the network topology graphs in the dot format described in Section 2.1 into the format used by ibsim. This program is called **dot2osm** and it can be used in the following way:

```
$ cd dot2osm
$ ./dot2osm input.dot output.osm
```

where input.dot contains the network topology in the dot format and the output, the same network topology in the OpenSM specific format, will be written to output.osm. Note that the input graph does not have to contain routing information. Any directed graph $G = (V, E)$ with the properties that there is an edge $(v, u) \in E \; \forall (u, v) \in E$ and every node in the graph that represents a host has exactly one outgoing edge can be used. This restriction is a consequence of InfiniBand links being full duplex. Now that we have a topology input file for ibsim we can use it to emulate this network:

```
$ ./start_ibsim.sh output.osm
```

wait for the line "Network simulator ready" to appear before you proceed with the next step, do not terminate the simulator until the routing information is stored in a file in the last step. The ibsim tool now uses a special version of the libumad library, which was preloaded via LD_PRELOAD by the start_ibsim script, to emulate an InfiniBand network with the specified topology. If OpenSM is started with our start_osm.sh script, the same version of the library will be loaded and OpenSM "thinks" it is managing this network, even if the machine used for this process does not have an InfiniBand card built in. Our startscript takes one parameter, the name of the routing algorithm that should be used. To use the default algorithm, which is called "minhop" do:

```
$ ./start_osm.sh minhop
```

the other supported routing algorithms are "updn", "ftree", "lash", and "dor". OpenSM supports another routing mechanism, the "file" mode. However, in this mode the user has to supply the routing information in the form of linear forwarding tables (lft), so it is of limited use for our purpose. Note that some routing algorithms, i.e., ftree only work for special topologies—if OpenSM detects that the selected routing algorithm does not work for the network topology it will fall back to the default algorithm (minhop) and a warning message will be emitted. Wait for OpenSM to print the message "SUBNET UP" before you proceed to the next step.

Now OpenSM has routed the simulated network, but the routing information is stored only in internal data structures of the network-emulation-library (if we would have used a real InfiniBand network, this information would have been stored in the switches). To obtain the routing information the `dump_routes.sh` script can be used:

```
$ ./dump_routes.sh > routing.lft
```

This tool dumps the linear forwarding table stored in every (simulated) switch in the network. These tables are big, hard to read and do not contain the network topology information. So to use them for simulations or other experiments we have to convert them into a directed graph with routing information. This is done with the lftsdump2dot script.

This script uses the network topology description in the OpenSM specific format which was created earlier, we called it output.osm in our example, together with the routing information obtained in the last step, which we called routing.lft, to generate a network topology graph with routing information as described in Section 2.1. The script is used in the following way:

```
$ cd lftsdump2dot
$ ./lftsdump2dot output.osm routing.lft >
    graph.dot
```

After this step graph.dot will contain a representation of the network topology contained in output.osm combined with the routing information from routing.lft which is compatible to our simulator and therefore can be used for further evaluation.

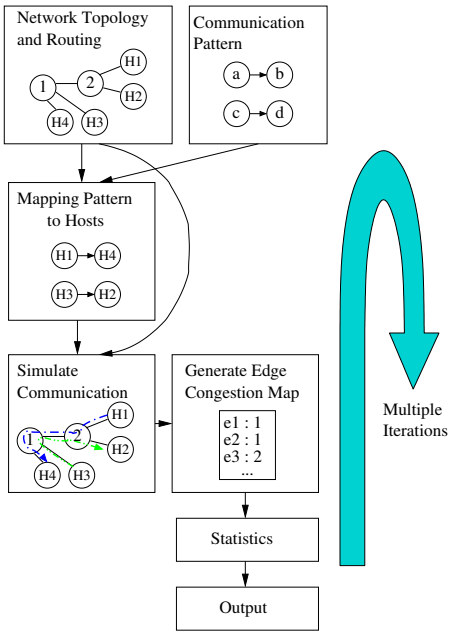

Figure 5: *Scheme of our simulator implementation*

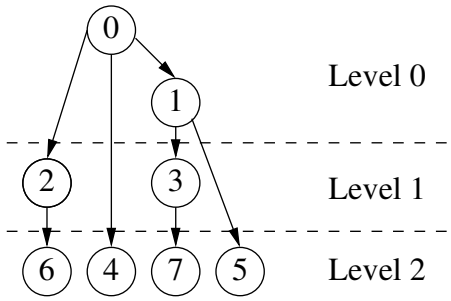

Figure 6: *Binomial Tree with eight nodes. Levels are indicated by dotted lines. If a receiving node is in level k the message to him is sent in this level and does not interfere with messages sent in other levels.*

4 The Implementation of the Simulator

In Figure 5 you can see a simplified scheme of how our simulator works. The network topology and routing information is retrieved from the input file. The format of this file is described in Section 2.1.

The traffic pattern is generated by the function `genptrn_by_name`. As explained in Section 2.3, the simulator offers a wide selection of traffic patterns that can be simulated. The traffic pattern is represented by a set of pairs of integers. For example a unidirectional bisection communication pattern with six modes can be represented by $\{(0,1),(2,3),(4,5)\}$. Every node is either sending to or receiving from exactly one other node. All messages are of uniform length in our simulations. All messages in one set are started simultaneously in our simulations. This is of course unacceptable for simulating communication patterns with dependencies or patterns where a single node sends multiple messages one after another. Therefore, we introduced levels in our communication patterns. A communication pattern with multiple levels then becomes a list of sets of pairs of integers. A binomial tree pattern could be modeled as $[\{(0,1)\},\{(0,2),(1,3)\},\{(0,4),(1,5),(2,6),(3,7)\}]$. A graphical description of this communication pattern is given in Figure 6. This pattern has three different levels as the graph has a degree of three, so there are three messages send by the root node in a sequential manner. We assume that level k starts after level $k-1$ is completed. So messages send in different levels do not interfere with each other.

To simulate the congestion induced by a certain traffic pattern we have to map the nodes in the communication pattern described above to hosts in the network. This is done in two stages: at first we have to know how many nodes from the network topology file should be used for the simulation. This also determines how many nodes the communication pattern should contain. We call this number the communicator size. But even if we know how many nodes to use, if it is desired to use less nodes than there are in the topology input file we do not know which we should use. The simulator currently supports two modes to select nodes: It can randomly choose p nodes from the n nodes available in the input file. This option aims to model a situation where jobs of various sizes and run-times are started by a batch system, after a while this will lead to fragmentation. The second mode will perform a breadth-first search from the first node defined in the input file and orders all hosts by the time at which the

had been found. Then the first p hosts in this list are used for the simulation. This ensures that we use the minimal amount of leaf switches possible. Before each simulation iteration the order of hosts in the list of used hosts is randomized by the function `shuffle_namelist`. Their position in this list maps them to nodes in the communication pattern.

The actual simulation of the communication pattern is carried out by the function `simulation_with_metric`. This function iterates over the communication pattern for each level and determines the route the traffic between each pair of hosts has to take by calling the function `find_route`. This function returns the list of edges that have been used for the specific host pair. All these edges are then added to the cable congestion map. This map counts, for every edge in the network topology, how often it has been used in the current level—it determines the congestion along each edge. The congestion for each edge is set to zero at the beginning of each level. After the simulation for one level is completed the cable congestion map contains the congestion for every edge during this level.

After the cable congestion map is filled at the end of one level we have to determine the maximal congestion for each route that was taken by a simulated stream of data. In our model this is the factor that determines the bandwidth with which this particular datastream was transmitted. This is done in the function `insert_into_bucket_maxcon2` by simply regenerating the route for each host pair in the communication pattern, as described above. Afterwards we iterate over the used edges and record the maximal congestion we see for all the used edges in the cable congestion map.

The rest of the statistical analysis process heavily depends on the used metric. The currently implemented metrics are described in Section 2.5.

The congestion induced by a communication pattern heavily depends on the mapping of the nodes in the communication pattern to the hosts in the network topology. Therefore one has to perform a large number of simulations, all of which with different random mappings, and average the results. Since the single simulation runs are independent of each other this can be easily parallelized. This is done with MPI. If the simulator is started with p MPI processes and has to execute c simulation runs (determined by the `--num_runs` command line parameter), every process will perform $\lceil \frac{c}{p} \rceil$ simulation iterations and send the results to the master process which will then process all results and generate the appropriate output.

Acknowledgments

This work was supported by the Department of Energy project FASTOS II (LAB 07-23), a grant from the Lilly Endowment and a gift the Silicon Valley Community Foundation on behalf of the Cisco Collaborative Research Initiative.

References

[1] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W.K. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, 1995.

[2] J. Flich, MP Malumbres, P. Lopez, and J. Duato. Improving Routing Performance in Myrinet Networks. In *Proc. of Int. Parallel and Distributed Processing Symp*, 2000.

[3] P. Geoffray and T. Hoefler. Adaptive Routing Strategies for Modern High Performance Networks. In *16th Annual IEEE Symposium on High Performance Interconnects (HOTI 2008)*, pages 165–172. IEEE Computer Society, Aug. 2008.

[4] J.L. Hennessy, D.A. Patterson, D. Goldberg, and K. Asanovic. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2003.

[5] T. Hoefler, T. Schneider, and A. Lumsdaine. Multistage Switches are not Crossbars: Effects of Static Routing in High-Performance Networks. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing*. IEEE Computer Society, Oct. 2008.

[6] TM InfiniBand. Trade Association, InfiniBand TM Architecture. Specification Volume 1. Release 1.0, 2000.

[7] E. Koutsofios and S.C. North. Drawing graphs with dot. *AT&T Labs–Research, Murray Hill, NJ, March*, 1999.

[8] F.T. Leighton. *Introduction to Parallel Algorithms and Architecture*. Morgan Kaufmann Publishers.

[9] JC Martinez, J. Flich, A. Robles, P. Lopez, and J. Duato. Supporting adaptive routing in InfiniBand networks. In *Parallel, Distributed and Network-Based Processing, 2003. Proceedings. Eleventh Euromicro Conference on*, pages 165–172, 2003.

[10] MD Schroeder, AD Birrell, M. Burrows, H. Murray, RM Needham, TL Rodeheffer, EH Satterthwaite, and CP Thacker. Autonet: a high-speed, self-configuring local area network usingpoint-to-point links. *Selected Areas in Communications, IEEE Journal on*, 9(8):1318–1335, 1991.