On LID Assignment In InfiniBand Networks

Wickus Nienaber
Dept. of Computer Science
Florida State University
Tallahassee, FL 32306
nienaber@cs.fsu.edu

Xin Yuan
Dept. of Computer Science
Florida State University
Tallahassee, FL 32306
xyuan@cs.fsu.edu

Zhenhai Duan
Dept. of Computer Science
Florida State University
Tallahassee, FL 32306
duan@cs.fsu.edu

ABSTRACT

To realize a path in an InfiniBand network, an address, known as Local IDentifier (LID) in the InfiniBand specification, must be assigned to the destination and used in the forwarding tables of intermediate switches to direct the traffic following the path. Hence, path computation in InfiniBand networks has two tasks: (1) computing the paths, and (2) assigning LIDs to destinations (and using the LIDs in the forwarding tables to realize the paths). We will refer to the task of computing paths as *routing* and the task of assigning LIDs as *LID assignment*. Existing path computation methods for InfiniBand networks integrate these two tasks in one phase. In this paper, we propose to separate routing and LID assignment into two phases so as to achieve the best performance for both routing and LID assignment. Since the routing component has been extensively studied and is fairly well understood, this paper focuses on LID assignment whose major issue is to minimize the number of LIDs required to support a routing. We prove that the problem of realizing a routing with a minimum number of LIDs is NP-complete, develop a number of heuristics for this problem, and evaluate the performance of the heuristics through simulation. Our results demonstrate that by separating routing from LID assignment and using the schemes that are known to achieve good performance for routing and LID assignment separately, more effective path computation methods than existing ones can be developed.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Network Topology, Network Communication; C.2.3 [**Network Operation**]: Network Management

General Terms

Algorithms, Measurement, Performance

ANCS'07, December 3–4, 2007, Orlando, Florida, USA.

Keywords

InfiniBand, LID Assignment, NP-Complete

1. INTRODUCTION

The InfiniBand architecture (IBA) is an industry standard architecture for interconnecting processing nodes and I/O devices [6]. It is designed around a switch-based interconnect technology with high-speed point-to-point links. IBA offers high bandwidth and low latency communication and can be used to build many different types of networks including I/O interconnects, system area networks, storage area networks, and local area networks. IBA has been widely adopted in large scale high performance computing (HPC) systems: many of the top 500 fastest supercomputers listed in the June 2007 release [12] use IBA as the interconnect technology.

An InfiniBand network is composed of one or more subnets connected by InfiniBand routers. Each subnet consists of processing nodes and I/O devices connected by InfiniBand switches. We will use the general term *machines* to refer to processing nodes and I/O devices at the edge of an InfiniBand network. This paper considers the communication within a subnet. A subnet is managed by a subnet manager (SM). By exchanging subnet management packets (SMPs) with the subnet management agents (SMAs) that reside in every InfiniBand device in a subnet, the SM discovers the subnet topology (and topology changes), computes the paths between each pair of machines based on the topology information, configures the network devices, and maintains the subnet.

InfiniBand requires the paths between all pairs of machines to be dead-lock free and deterministic. These paths must then be realized with a destination based routing scheme. Specifically, machines are addressed by local identifiers (LIDs). Each InfiniBand packet contains in its header the source LID (SLID) and destination LID (DLID) fields. Each switch maintains a forwarding table that maps the DLID to one output port. When a switch receives a packet, it parses the packet header and performs a table lookup using the DLID field to find the output port for this packet. The fact that one DLID is associated with one output port in the forwarding table implies that (1) the routing is deterministic; and (2) each DLID can only direct traffic in one direction in a switch.

Destination based routing limits the paths that can be realized. Consider the paths from nodes 4 and 5 to node 0 in Figure 1. Assume that node 0 is associated with only one LID, the paths $4 \rightarrow 3 \rightarrow 1 \rightarrow 0$ and $5 \rightarrow 3 \rightarrow 2 \rightarrow 0$ cannot be supported simultaneously: with one LID for node 0, the traffic toward node 0 in node 3 can only follow one direction. To overcome this problem and allow more flexible routes, IBA introduces a concept called LID Mask Control (LMC) [6], which allows multiple LIDs to be associated with each machine. Using LMC, each machine can be assigned a range of LIDs (from $BASELID$ to $BASELID + 2^{LMC} - 1$). Since LMC is represented by three bits, at most $2^{LMC} = 2^7 = 128$ LIDs can be assigned to each machine. By associating multiple LIDs with one machine, the paths that can be supported by the network are more flexible. For example, the two paths in Figure 1 can be realized by having two LIDs associated with node 0, one for each path. Nonetheless, since the number of LIDs that can be allocated (to a node or in a subnet) is limited, the paths that can be used in a subnet are still constrained, especially for medium or large sized subnets.

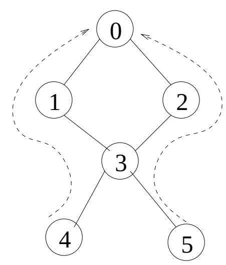


Figure 1: An example

The use of destination based routing with multiple LIDs for each machine complicates the path computation in InfiniBand networks. In addition to finding the paths between machines, the SM must assign LIDs to machines and compute the forwarding tables that realize the paths. Hence, the path computation in an InfiniBand network is logically composed of two tasks: the first task is to compute the dead-lock free deterministic paths for each pair of machines; and second task is to assign LIDs to machines and compute the forwarding tables for realizing the paths determined in the first task. We will use the terms *routing* and *LID assignment* to refer to these two tasks. The term *routing* may also refer to the set of paths computed.

Since the IBA specification [6] does not give specific algorithms for path computation, this area is open to research and many path computation schemes have been proposed. Existing path computation schemes [1, 2, 3, 5, 8, 9, 10] are all based on the Up*/Down* routing [11], which is originally an adaptive dead-lock free routing scheme. Moreover, all of these schemes integrate the Up*/Down* routing, path selection (selecting deterministic paths among potential paths allowed by the Up*/Down* routing), and LID assignment in one phase. While these schemes provide practical solutions, there are some notable limitations. First, since Up*/Down* routing, path selection, and LID assignment are integrated, these schemes cannot be directly applied to other dead-lock free routing schemes, such as L-turn [4], that may have better load balance properties. Second, the quality of the paths selected by these schemes may not be the best. In fact, the load balancing property of the paths is often compromised by the LID assignment requirement. For example, the fully explicit routing [9] restricts the paths to each destination such that all paths to a destination can be realized by one LID (avoiding the LID assignment problem). Notice that load balancing is one of the most important parameters that determine the performance of a routing system and is extremely critical for achieving high performance in an InfiniBand network. Third, the performance of LID assignment in these schemes is not clear. Since LID assignment is integrated with routing and path selection in all existing schemes, the LID assignment problem itself is not well understood.

We propose to separate routing from LID assignment, which may alleviate the limitations discussed in the previous paragraph: the separation allows routing to focus on finding paths with good load balancing properties and LID assignment to focus on its own issues. Among the two tasks, routing in system area networks that require dead-lock free and deterministic paths has been extensively studied and is fairly well understood. There exist dead-lock free adaptive routing schemes, such as Up*/Down* routing [11] and L-turn routing [4], that can be used to identify a set of candidate paths. Path selection algorithms that can select dead-lock free deterministic paths with good load balancing properties from candidate paths have also been developed [7]. Applying these algorithms in InfiniBand networks can potentially result in better paths being selected than those selected by the existing path computation schemes developed for InfiniBand. However, in order to apply these routing schemes, LID assignment, which has never been studied independently from other path computation components before, must be investigated. This is the focus in this paper.

LIDs are limited resources. The number of LIDs that can be assigned to each node must be no more than 128. In addition, the 16-bit SLID and DLID fields in the packet header limit the total number of LIDs in a subnet to be no more than $2^{16} = 64K$. For a given routing (a set of paths), one can always use a different LID to realize each path. Hence, the number of LIDs needed to realize a routing is no more than the number of paths. However, using this simple LID assignment approach, a system with more than 130 machines cannot be built: it would require more than 129 LIDs to be assigned to a machine in order to realize the (more than 129) paths from other machines to this machine. Hence, the major issue in LID assignment is to minimize the number of LIDs required to realize a given routing. Minimizing the number of LIDs enables (1) larger subnets to be built, and/or (2) more paths to be supported in a given subnet. Supporting more paths is particularly important when multi-path routing [14] or randomized routing is used. In the rest of this paper, we use the term *LID assignment problem* to refer to the problem of realizing a routing with a minimum number of LIDs.

We further the theoretical understanding of LID assignment by proving that the LID assignment problem is NP-complete. We develop three types of heuristics for this problem and evaluate the proposed heuristics through simulation. These heuristics allow existing methods for finding load balance dead-lock free deterministic paths to be applied in InfiniBand networks. Practically, we demonstrate that by separating routing from LID assignment and using the schemes that are known to achieve good performance for routing and LID assignment separately, more effective path computation methods than existing ones can be developed. In many cases, especially for reasonably large systems, the new methods compute paths that (1) have better load balancing properties, and (2) can be realized with a smaller number of LIDs.

The rest of the paper is organized as follows. In Section 2, we introduce the notations and formally define the LID assignment problem. We prove the NP-completeness of the LID assignment problem in Section 3. In Section 4, we describe the proposed heuristics. Section 5 evaluates the proposed heuristics and the overall performance of various path computation schemes. Finally, Section 6 concludes the paper.

2. PROBLEM DEFINITION

An InfiniBand subnet consists of *machines* connected by *switches*. A *node* refers to either a switch or a machine. InfiniBand allows both regular and irregular topologies. The techniques developed in this paper are mainly for irregular topologies. The links are bidirectional; a machine can have multiple ports connecting to one or more switches; and multiple links are allowed between two nodes. We model an InfiniBand network as a directed multi-graph, $G = (V, E)$, where E is the set of directed *edges* and V is the set of switches and machines. Let M be the set of machines and S be the set of switches. $V = M \cup S$. Let there exist n links between two nodes u and v. The links are numbered from 1 to n. The n links are modeled by $2n$ direct edges $((u, v), i)$ (or $u \xrightarrow{i} v$) and $((v, u), i)$ (or $v \xrightarrow{i} u$), $1 \leq i \leq n$. The i-th link between nodes u and v is modeled by two direct edges $((u, v), i)$ and $((v, u), i)$. An example InfiniBand topology is shown in Figure 2. In this example, switches $s0$ and $s1$ are connected by two links; machine $m3$ is connected to two switches $s1$ and $s2$.

A *path* $p = u \xrightarrow{i_0} a_1 \xrightarrow{i_1} a_2 \xrightarrow{i_2} ... \xrightarrow{i_{n-1}} a_n \xrightarrow{i_n} v$ consists of a set of directed edges $\{u \xrightarrow{i_0} a_1, a_1 \xrightarrow{i_1} a_2, ..., a_n \xrightarrow{i_n} v\}$. $NODE(p) = \{u, a_1, a_2, ..., a_n, v\}$ is the set of nodes that the path $p = u \xrightarrow{i_0} a_1 \xrightarrow{i_1} a_2 \xrightarrow{i_2} ... \xrightarrow{i_{n-1}} a_n \xrightarrow{i_n} v$ goes through. $SRC(p) = u$ is the source of path p and $DST(p) = v$ is the destination of path p. A path $p = u \xrightarrow{i_0} a_1 \xrightarrow{i_1} a_2 \xrightarrow{i_2} ... \xrightarrow{i_{n-1}} a_n \xrightarrow{i_n} v$ is *end-to-end* when $SRC(p) = u \in M$ and $DST(p) = v \in M$. In this case, path p is said to be an *end-to-end* path. For example, the dark line in Figure 2 shows an end-to-end path $m0 \xrightarrow{1} s0 \xrightarrow{2} s1 \xrightarrow{1} s2 \xrightarrow{1} m4$. A *routing* R is a set of end-to-end paths, $R = \{p_1, p_2, ...\}$.

InfiniBand realizes each path through destination based routing. In Figure 2, we show the entries in the forwarding tables that realize two paths $m0 \xrightarrow{1} s0 \xrightarrow{2} s1 \xrightarrow{1} s2 \xrightarrow{1} m4$ (the solid dark line) and $m1 \xrightarrow{1} s0 \xrightarrow{1} s2 \xrightarrow{1} m4$ (the dotted dark line). This example assumes that LIDs 4 and 5 are assigned to machine $m4$ and the entries are illustrated with a random forwarding table format: each table entry is of the form $(DLID, output\_port)$. As shown in the example, path $m0 \xrightarrow{1} s0 \xrightarrow{2} s1 \xrightarrow{1} s2 \xrightarrow{1} m4$ is realized by having entry $(DLID = 4, output\_port = 2)$ in the forwarding table in switch $s0$, $(DLID = 4, output\_port = 3)$ in $s1$, and $(DLID = 4, output\_port = 3)$ in $s2$. Once the forwarding tables are installed, machine $m0$ can send packets to $m4$ following this path by making $DLID = 4$ in the packet header. Note that the physical installation of the forwarding table in different switches is performed by the SM in the path distribution phase, which is beyond the scope of this paper.

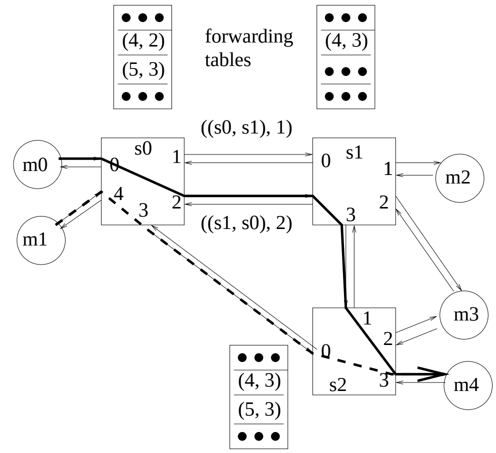


Figure 2: An InfiniBand network topology (LIDs 4 and 5 are assigned to m4)

To realize a path p towards a destination v, a LID LID_v that is associated with the node v must be used and an entry in the form of $(LID_v, output\_port)$ must be installed in each of the intermediate switches along the path. Once LID_v is associated with one *output_port* in a switch, it cannot be used to realize other paths that use different output ports in the same switch. We will use the term *assigning LID_v to path p* to denote the use of LID_v to realize path p. In the example in Figure 2, LID 4 is assigned to path $m0 \xrightarrow{1} s0 \xrightarrow{2} s1 \xrightarrow{1} s2 \xrightarrow{1} m4$ and LID 5 is assigned to path $m1 \xrightarrow{1} s0 \xrightarrow{1} s2 \xrightarrow{1} m4$.

Since different destinations are assigned different ranges of LIDs in InfiniBand networks, the number of LIDs required for realizing a routing is equal to the sum of the number of LIDs required for each destination. In other words, the **LID assignment problem for a routing can be reduced to the LID assignment problem for each individual destination**. Let $R = \{p_1, p_2, ...\}$ be a routing and $D = \{d | \exists p_i \in R, DST(p_i) = d\}$ be the set of destinations in R. Let $d \in D$ be a destination node in some paths in R, $R_d = \{p | p \in R \ and \ DST(p) = d\}$. We have $R = \cup_{d \in D} R_d$. Let the minimum number of LIDs needed for realizing R_d be L_d and the minimum number of LIDs needed for realizing R be L. Since LIDs for different destination nodes are independent of one another, $L = \sum_{d \in D} L_d$. We will call LID assignment for each R_d the *single destination LID assignment problem*. In the rest of the paper, we will focus on the single destination LID assignment problem. Unless specified otherwise, all paths are assumed to have the same destination. Next, we will introduce some definitions and lemmas that lead to the formal definition of the single destination LID assignment problem.

Definition 1: Two paths p_1 and p_2 (with the same destination) are said to have a *split* if there exists a node $a \in NODE(p_1) \cap NODE(p_2)$, $a \xrightarrow{i} b \in p_1$ and $a \xrightarrow{j} c \in p_2$, such that either $i \neq j$ or $b \neq c$.

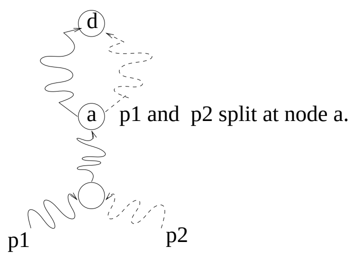
Basically, two paths have a split when (1) both paths share an intermediate node, and (2) the outgoing links from the intermediate node are different. Figure 3 (a) shows the case when two paths have a split.

Lemma 1: When two paths p_1 and p_2 have a split, they must be assigned different LIDs. When two paths p_1 and

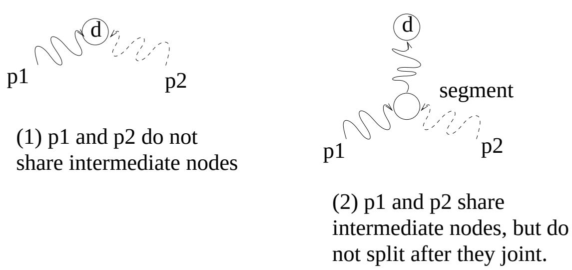
p_2 do not have any split, they can share the same LID (be assigned the same LID).

Proof: We will first prove the first proposition in this lemma: when two paths p_1 and p_2 have a split, they must be assigned different LIDs. Let p_1 and p_2 be the two paths that have a split. From Definition 1, there exists a node $a \in NODE(p_1) \cap NODE(p_2)$, $a \xrightarrow{i} b \in p_1$ and $a \xrightarrow{j} c \in p_2$, such that either $i \neq j$ or $b \neq c$. Consider the forwarding table in node a. When either $i \neq j$ or $b \neq c$, $a \xrightarrow{i} b \in p_1$ uses a different port from $a \xrightarrow{j} c \in p_2$. Since one LID can only be associated with one output port in the forwarding table, two LIDs are needed in switch a to realize the two directions. Hence, p_1 and p_2 must be assigned different LIDs.

Now consider the second proposition: when two paths p_1 and p_2 do not have any split, they can share the same LID (be assigned the same LID). Let p_1 and p_2 be the two paths that do not have a split. There are two cases. The first case, shown in Figure 3 (b) (1), is when the two paths do not share any intermediate nodes. The second case, shown in Figure 3 (b) (2), is when two paths share intermediate nodes, but do not split after they join. For both cases, the same LIDs can be used in forwarding table of the switches along both paths to realize both paths, and the two paths can be assigned the same LIDs. □



(a) The case when two LIDs are needed



(b) the cases when one LID can be shared

Figure 3: The cases when a LID can and cannot be shared between two paths

It must be noted that the statements "p_1 can share a LID with p_2" and "p_1 can share a LID with p_3" do not imply that "p_2 can share a LID with p_3". Consider paths $p_1 = m2 \rightarrow s1 \rightarrow s2 \rightarrow m4$, $p_2 = m0 \rightarrow s0 \rightarrow s1 \rightarrow s2 \rightarrow m4$, and $p_3 = m_1 \rightarrow s0 \rightarrow s2 \rightarrow m4$ in Figure 2. Clearly, p_1 can share a LID with p_2 and p_1 can share a LID with p_3, but p_2 and p_3 have a split at switch $s0$ and cannot share a LID. The following concept of *configuration* defines a set of paths that can share one LID.

Definition 2: A *configuration* is a set of paths (with the same destination) $C = \{p_1, p_2, ...\}$ such that no two paths in the set have a split.

Lemma 2: All paths in a configuration can be realized by one LID.

Proof: Let l be a LID. Consider any switch, s, in the system. This switch can either be used by the paths in the configuration or not used. If s is used by some paths, by the definition of configuration, all paths that pass through s must follow one outgoing port in switch s, *port*, (otherwise, the paths have a split at s and the set of paths is not a configuration). Hence, the entry $(DLID = l, output\_port = port)$ can be shared by all paths using s. If s is not used by any paths in the configuration, no entry is needed in the forwarding table to realize the paths in the configuration. Hence, LID l can be used in the switches along all paths in configuration to realize all of the paths. □

Definition 3 (Single destination LID assignment problem ($SD(G, d, R_d)$)): Let the network be modeled by the multi-graph G, d be a node in G, $R_d = \{p_1, p_2, ...\}$ be a single destination routing (for all $p_i \in R_d$, $DST(p_i) = d$). The single destination LID assignment problem is to find a function $c : R_d \rightarrow \{1, 2, ..., k\}$ such that (1) $c(p_i) \neq c(p_j)$ for every pair of paths p_i and p_j that have a split, and (2) k is minimum.

Let $c : R_d \rightarrow \{1, 2, ..., k\}$ be a solution to $SD(G, d, R_d)$. Let $R_d^i = \{p_j | c(p_j) = i\}$, $1 \leq i \leq k$. By definition, R_d^i is a configuration; $R_d = \cup_{i=1}^k R_d^k$; and $R_d^i \cap R_d^j = \phi$, $i \neq j$. Thus, $SD(G, d, R_d)$ is equivalent to the problem of partitioning R_d into k disjoint sets R_d^1, R_d^2, ..., R_d^k such that (1) each R_d^i is a configuration, and (2) k is minimum. When the disjoint sets R_d^1, R_d^2, ..., R_d^k are found, the routing R_d can be realized by k LIDs with one LID assigned to all paths in R_d^i, $1 \leq i \leq k$.

$SD(G, d, R_d)$ states the optimization version of this problem. The corresponding decision problem, denoted as $SD(G, d, R_d, k)$, decides whether there exists a function $c : R_d \rightarrow \{1, 2, ..., k\}$ such that $c(p_i) \neq c(p_j)$ for every pair of paths p_i and p_j that have a split.

Since InfiniBand realizes multiple LIDs for each destination using the LID Mask Control (LMC) mechanism, the actual number of LIDs assigned to each destination must be a power of two, up to 128. Hence, if the solution to $SD(G, d, R_d)$ is k, the actual number of LIDs assigned to d is $2^{\lceil lg(k) \rceil}$. For example, when $k = 4$, $2^{\lceil lg(k) \rceil} = 4$; when $k = 5$, $2^{\lceil lg(k) \rceil} = 8$.

3. NP-COMPLETENESS

Theorem 1: $SD(G, d, R_d, k)$ is NP-complete.

Proof: We first show that $SD(G, d, R_d, k)$ belongs to NP problems. Suppose that we have a solution for $SD(G, d, R_d, k)$, the verification algorithm first affirms the solution function $c : R_d \rightarrow \{1, 2, ..., k\}$. It then checks for each pair of paths $p1$ and $p2$, $c(p1) = c(p2)$, that they do not have a split. It is straightforward to perform this verification in polynomial time. Thus, $SD(G, d, R_d, k)$ is an NP problem.

We prove that $SD(G, d, R_d, k)$ is NP-complete by showing that the graph coloring problem, which is a known NP-complete problem, can be reduced to this problem in polynomial time. The graph-coloring problem is to determine the minimum number of colors needed to color a graph. The k-coloring problem is the decision version of the graph coloring problem. A k-coloring of an undirected graph $G = (V, E)$ is a function $c : V \rightarrow \{1, 2, ..., k\}$ such that $c(u) \neq c(v)$ for every edge $(u, v) \in E$. In other words, the numbers 1, 2, ..., k represent the k colors, and adjacent vertices must have different colors.

The reduction algorithm takes an instance $< G, k >$ of the k-coloring problem as input. It computes the instance $SD < G', d, R_d, k >$ as follows. Let $G = (V, E)$ and $G' = (V', E')$. The following vertices are in V'.

- The destination node $d \in V'$.
- For each $u \in V$, two nodes $n_u, n_{u'} \in V'$.
- For each $(u, v) \in E$, a node $n_{u,v} \in V'$. Since G is an undirected graph, (u, v) is the same as (v, u) and there is only one node for each $(u, v) \in E$ (node $n_{u,v}$ is the same as node $n_{v,u}$).

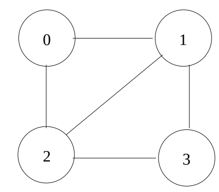
The edges in G' are as follows. For each n_u, let nodes n_{u,i_1}, n_{u,i_2}, ..., n_{u,i_m} be the nodes corresponding to all node u's adjacent edges in G. The following edges: (n_u, n_{u,i_1}), (n_{u,i_1}, n_{u,i_2}), ..., $(n_{u,i_{m-1}}, n_{u,i_m})$, (n_{u,i_m}, n_{u_2}), $(n_{u'}, d)$ are in E'. Basically, for each node $u \in G$, there is a path in G' that goes from n_u, through each of the nodes in corresponding to the edges adjacent to u in G, then through $n_{u'}$ to node d.

Each node $u \in V$ corresponds to a path p_u in R_d. p_u starts from node n_u, it goes through every node in G' that corresponds to an edge adjacent to u in G, and then goes to node $n_{u'}$, and then d. Specifically, let n_{u,i_1}, n_{u,i_2}, ..., n_{u,i_m} be the nodes corresponding to all node u's adjacent edges in G, $p_u = n_u \xrightarrow{1} n_{u,i_1} \xrightarrow{1} n_{u,i_2} ... \xrightarrow{1} n_{u,i_m} \xrightarrow{1} n_{u'} \xrightarrow{1} d$. *The idea is to construct an instant of G', d, and R_d such that $p_u, p_v \in R_d$ have a split if and only if u and v are adjacent nodes.* From the construction of p_u, we can see that if nodes u and v are adjacent in G ($(u, v) \in E$), both p_u and p_v go through node $n_{u,v}$ and have a split at this node. If u and v are not adjacent, p_u and p_v do not share any intermediate node, and thus, do not have a split.

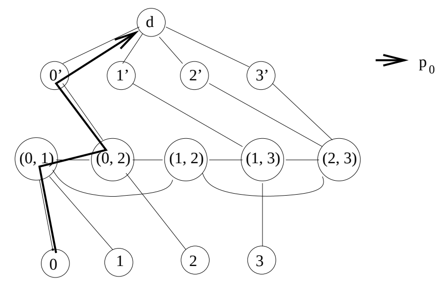
Figure 4 shows an example of the construction of G', d and R_d. For the example G in Figure 4 (a), we first create the destination node d in G'. The second and fourth rows of nodes in Figure 4 (b) corresponds to the two nodes $n_{u'}$ and n_u for each node $u \in V$. The third row of nodes corresponds to the edges in G. Each node u in G corresponds to a path p_u in R_d, $R_d = \{p_0, p_1, p_2, p_3\}$, where $p_0 = n_0 \xrightarrow{1} n_{0,1} \xrightarrow{1} n_{0,2} \xrightarrow{1} n_{0'} \xrightarrow{1} d$, $p_1 = n_1 \xrightarrow{1} n_{0,1} \xrightarrow{1} n_{1,2} \xrightarrow{1} n_{1,3} \xrightarrow{1} n_{1'} \xrightarrow{1} d$, $p_2 = n_2 \xrightarrow{1} n_{0,2} \xrightarrow{1} n_{1,2} \xrightarrow{1} n_{2,3} \xrightarrow{1} n_{2'} \xrightarrow{1} d$, and $p_3 = n_3 \xrightarrow{1} n_{1,3} \xrightarrow{1} n_{2,3} \xrightarrow{1} n_{3'} \xrightarrow{1} d$. The path p_0 that corresponds to node 0 in Figure 4 (a) is depicted in Figure 4 (b). It can easily see that in this example, $p_u, p_v \in R_d$ have a split if and only if u and v are adjacent nodes.

To complete the proof, we must show that this transformation is indeed a reduction: the graph G can be k-colored if and only if $SD(G', d, R_d, k)$ has a solution.

First, we will show the sufficient condition: if G can be k-colored, $SD(G', d, R_d, k)$ has a solution. Let $c : V \rightarrow \{1, 2, ..., k\}$ be the solution to the k-coloring problem. We can partition R_d into $R_d^i = \{p_u | c(u) = i\}$. Let $p_u, p_v \in R_d^i$. Since $c(u) = c(v)$, nodes u and v are not adjacent in G. From the construction of G', d, and R_d, p_u and p_v do not have split. By definition, R_d^i is a configuration. Hence, R_d can be partitioned into k configurations R_d^1, R_d^2, ..., R_d^k and $SD(G', d, R_d, k)$ has a solution.



(a) An example graph for graph coloring



(b) The corresponding graph for LID assignment

Figure 4: An example of mapping G to G'

Now, we will show the necessary condition: if $SD(G', d, R_d, k)$ has a solution, G can be k-colored. Since $SD(G', d, R_d, k)$ has a solution, R_d can be partitioned into k configurations R_d^1, R_d^2, ..., and R_d^k. Let $p_u, p_v \in R_d^i$, $1 \leq i \leq k$. Since R_d^i is a configuration, p_u does not have split with p_v in G'. From the construction of G', d, R_d, u and v are not adjacent in G. Hence, all nodes in each configuration can be colored with the same color and the mapping function $c : V \rightarrow \{1, 2, ..., k\}$ can be defined as $c(u) = i$ if $p_u \in R_d^i$, $1 \leq i \leq k$. Hence, if $SD(G', d, R_d, k)$ has a solution, G can be k-colored. □

4. LID ASSIGNMENT HEURISTICS

Since the LID assignment problem is NP-complete, we resort to heuristic algorithms for solving the problem. All of our heuristics are based on the concept of **minimal configuration set**, which is defined next.

Definition 4: Given a single destination routing $R_d = \{p_1, p_2, ...\}$, the set of configurations $MC = \{C_1, C_2, ..., C_k\}$ is a **minimal configuration set** for R_d if and only if all of the following conditions are met:

- each $C_i \in MC$, $1 \leq i \leq k$, is a configuration;
- each $p_i \in R_d$ is in exactly one configuration in MC;
- for each pair of configuration C_i and $C_j \in MC$, $i \neq j$, there exist $p_x \in C_i$ and $p_y \in C_j$ such that p_x and p_y have a split.

The configuration set is minimal in that there do not exist two configurations in the set that can be further merged. From Lemma 2, all paths in one configuration can be realized by 1 LID. Hence, assume that $MC = \{C_1, C_2, ..., C_k\}$ is a **minimal configuration set** for routing R_d, the routing R_d can be realized by k LIDs. All of the heuristics attempt to minimize the number of LIDs needed by finding a minimal configuration set for a routing.

4.1 Greedy heuristic

For a given routing R_d, the greedy LID assignment algorithm creates configurations one by one, trying to put as as many paths into each configuration as possible to minimize the number of configurations needed. This heuristic repeats the following process until all paths are in the configurations: create an empty configuration (current configuration), check each of the paths in R_d that has not been included in a configuration whether it has a split with the paths in the current configuration, and greedily put the path in the configuration (when the path does not split with any paths in the configuration). The algorithm is shown in Figure 5. Each configuration (or path) can be represented as an array of size $|V|$ that stores for each node the outgoing link from the node (in a configuration or a path, there can only be *one* outgoing link from each node). Using this date structure, checking whether a path has a split with any path in a configuration takes $O(|V|)$ time (line (5) in Figure 5); and adding a path in a configuration also takes $O(|V|)$ time (line (6)). The loop at line (4) runs for at most $|R_d|$ iterations and the loop at line (2) runs for at most k iterations, where k is the number of LIDs allocated. Hence, the complexity of the algorithm is $O(k \times |R_d| \times |V|)$, where k is the number of LIDs allocated, R_d is the set of paths, and V is the set of nodes in the network.

(1) MC = ϕ, k = 1
(2) **repeat**
(3) $\quad C_k = \phi$
(4) $\quad$ **for each** $p \in R_d$
(5) $\qquad$ **if** p does not split with any path in C_k **then**
(6) $\qquad\quad C_k = C_k \bigcup \{ p \}$, $R_d = R_d - \{ p \}$
(7) $\qquad$ **end if**
(8) $\quad$ **end for**
(9) $\quad$ MC = MC $\bigcup \{ C_k \}$, k = k + 1
(10) **until** $R_d = \phi$

Figure 5: The greedy heuristic

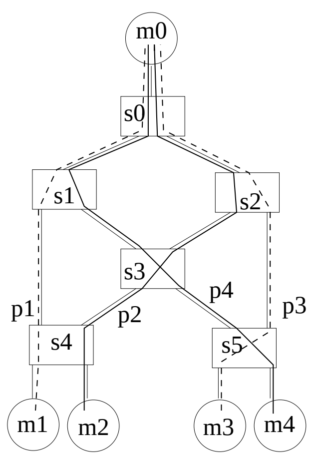


Figure 6: An example of LID assignment

We will use an example to show how the greedy heuristic algorithm works and how its solution may be sub-optimal. Consider realizing $R_{m0} = \{p_1, p_2, p_3, p_4\}$ in Figure 6, where $p_1 = m1 \xrightarrow{1} s4 \xrightarrow{1} s1 \xrightarrow{1} s0 \xrightarrow{1} m0$, $p_2 = m2 \xrightarrow{1} s4 \xrightarrow{1} s3 \xrightarrow{1} s2 \xrightarrow{1} s0 \xrightarrow{1} m0$, $p_3 = m4 \xrightarrow{1} s5 \xrightarrow{1} s2 \xrightarrow{1} s0 \xrightarrow{1} m0$, and $p_4 = m3 \xrightarrow{1} s5 \xrightarrow{1} s3 \xrightarrow{1} s1 \xrightarrow{1} s0 \xrightarrow{1} m0$. The greedy algorithm first creates a configuration and puts p_1 in the configuration. After that, the algorithm tries to put other paths into this configuration. The algorithm considers p_2 next. Since p_1 and p_2 split at switch $s4$, p_2 cannot be included in this configuration. Now, consider p_3. Since p_3 and p_1 do not have any joint intermediate nodes, p_3 can be included in the configuration. After that, since p_4 splits with p_3 at switch $s5$, it cannot be included in this configuration. Thus, the first configuration will contain paths p_1 and p_3. Since we have two paths p_2 and p_4 left unassigned, new configurations are created for these two paths. Since p_2 and p_4 split at switch $s3$, they cannot be included in one configuration. Hence, the greedy algorithm realizes R_{m0} with three configurations: $C_1 = \{p_1, p_3\}$, $C_2 = \{p_2\}$, and $C_3 = \{p_4\}$. Thus, 3 LIDs are needed to realize the routing with the greedy heuristic. Although $MC = \{C_1, C_2, C_3\}$ is a minimal configuration set, the solution is not optimal: R_{m0} can be partitioned into two configurations: $C'_1 = \{p_1, p_4\}$ and $C'_2 = \{p_2, p_3\}$ and only two LIDs are needed to realize the routing.

4.2 Split-merge heuristics

For a given routing R_d, the greedy algorithm tries to share LIDs as much as possible by considering each path in R_d: the minimal configuration set is created by merging individual paths into configurations. The split-merge heuristics use a different approach to find the paths that share LIDs. This class of heuristics has two phases: in the first phase, R_d is split into configurations; in the second phase, the greedy heuristic is used to merge the resulting configurations into a minimal configuration set, which is the final LID assignment. In the split phase, the working set initially contains one item R_d. In each iteration, a node is selected. Each item (a set of paths) in the working set is partitioned into a number of items such that each of the resulting items does not contain paths that split in the node selected (the paths that split in the selected node are put in different items). After all nodes are selected, the resulting items in the working set are guaranteed to be configurations: paths in one item do not split in any of the nodes. In the worst case, each resulting configuration contains one path at the end of the split phase and the split-merge heuristic is degenerated into the greedy algorithm. In general cases, however, the split phase will produce configurations that include multiple paths. It is hoped that the split phase will allow a better starting point for merging than individual paths. The heuristic is shown in Figure 7. Using a linked list to represent a set and the data structure used in the greedy algorithm to represent a path, the operations in the loop from line (4) to (7) can be done in $O(|R_d||V|)$ operations: going through all $|R_d|$ paths and updating the resulting set that contains each path with $O(|V|)$ operations. Hence, the worst case time complexity for the whole algorithm is $O(|V|^2|R_d| + k|V||R_d|)$.

Depending on the order of the nodes selected in the split phase, there are variations of this split-merge heuristic. We consider two heuristics in our evaluation, the *split-merge/S* heuristic that selects the node used by the smallest number of paths first, and the *split-merge/L* heuristic that selects the node used by the largest number of paths first.

/* splitting */
(1) $S = \{R_d\}$, $ND = V$
(2) **repeat**
(3) Select a node, a, in ND;
(4) **for** each $S_i \in S$ **do**
(5) partition paths in S_i that splits at node a into multiple sets S_i^1, S_i^2, ...,S_i^j
(6) $S = (S - \{S_i\}) \cup S_i^1 \cup ... \cup S_i^j$;$ND = ND - \{a\}$
(7) **end for**
(8) **until** $ND = \phi$
/* merging */
(9) apply the greedy heuristic on S.

Figure 7: The split-merge heuristic.

4.3 Graph coloring heuristics

This heuristic converts the LID assignment problem into a graph coloring problem. First, a split graph is built. For all paths p_i, where $p_i \in R_d$, there exists a node n_{p_i} in the split graph. If p_i and p_j have a split with each other, where p_i, $p_j \in R_d$, an edge (n_{p_i}, n_{p_j}) is added in the split graph. After all paths $p_i \in R_d$ have been compared with all other paths $p_j \in R_d$, where $i \neq j$, a complete split graph is created. It can be easily shown that if the split graph can be colored with k colors, R_d can be realized with k LIDs: the nodes assigned the same color correspond to the nodes assigned the same LID. This conversion allows heuristics that are designed for graph coloring to be applied to the LID assignment. If we take the example from Figure 6, the corresponding split graph is shown in Figure 8. Node $p1$ has an edge with node $p2$ as they split at $s4$, node $p2$ has an additional edge with $p4$ as they split at $s3$. Finally, $p3$ has an edge with $p4$ as they split with each other at $s5$. This results in the split graph shown in Figure 8.

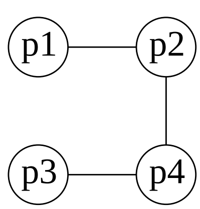


Figure 8: The split graph for Figure 6

While other graph coloring algorithms can be applied to color the split graph, we use a simple coloring heuristic in this paper. In our heuristic, the graph is colored by applying the colors one-by-one. Each color is applied as follows (starting from a graph with no color):

- Step 1: select a node to color;
- Step 2: remove all nodes that are adjacent to the node selected in Step 1;
- Step 3: if there exist other nodes that are not removed or colored, goto Step 1.

After a color is applied, all nodes that are colored are removed from the graph. Uncolored nodes (removed in Step 2) are restored to form a reduced graph to be colored in the next round. The process is repeated until all nodes are colored. As an example from Figure 8 node $p2$ could be chosen first in step 1. In step 2 nodes $p1$ and $p4$ are removed as they share an edge with $p2$. In step 3 a single node $p3$, remains so the steps are repeated starting at step1. The remaining node $p3$ is chosen in step 1, with no nodes remaining, we obtain a configuration $C_1 = \{p2, p3\}$. C_1 is colored and nodes $p1$ and $p4$ are restored to the graph. We repeat the steps again as above and obtain the final configuration $C_2 = \{p1, p4\}$. We color C_2 and the coloring is complete.

The heuristic is embedded in the selection of a node to color in Step 1. We consider two coloring based heuristics in this paper: the most split path first heuristic (color/L) when the node in the split graph with the largest nodal degree is selected (node $p2$ or node $p4$ in Figure 8); and the least split path first heuristic (color/S) when the node in the split graph with the smallest nodal degree is selected (node $p1$ or node $p3$ in Figure 8). The worst case time complexity for computing the split graph is $O(|R_d|^2|V|)$. After the graph is created, the complexity for coloring is $O(k \times |R_d|^2)$.

5. PERFORMANCE STUDY

We study the performance of the LID assignment heuristics as well as the performance of different path computation schemes using various random irregular topologies. We report results on systems with 128, 256, and 512 machines and 16, 32, and 64 switches. Specifically, the configurations include: 128 machines/16 switches, 256 machines/16 switches, 512 machines/16 switches, 128 machines/32 switches, 256 machines/32 switches, 512 machines/32 switches, 128 machines/64 switches, 256 machines/64 switches, and 512 machines/64 switches. We will use the notion X/Y to represent the system configuration with X machines and Y switches. For example, 128/16 denotes the configuration with 128 machines and 16 switches. Each random irregular topology is generated as follows. First, a random switch topology is generated using the *Georgia Tech Internetwork Topology Models (GT-ITM)* [15]. The average nodal degree is 8 for all three cases (16, 32, and 64 switches). After the switch topology is generated, the machines are randomly distributed among the switches with a uniform probability distribution. Note that the topologies generated by GT-ITM are not limited to Internet-like topologies, this package can generate random topologies whose connectivity follows many different probability distribution. For each type of topologies, we produce 32 different random topologies and report the average results on the 32 random instances. We have performed experiments on other random topologies, the results have a similar trend.

5.1 Performance of LID assignment heuristics

The LID assignment heuristics evaluated include *greedy*, *split-merge/L* where the node used by the largest number of paths is selected first in the split phase, *split-merge/S* where the node used by the smallest number of paths is selected first, *color/L* that is the most split path first heuristic (paths that split with the largest number of other paths are colored first), and *color/S* that is the least split path first heuristic (paths that split with the least number of other paths are colored first). To save space, we will use notion *s-m/L* to represent *split-merge/L* and *s-m/S* to represent *split-merge/S*.

The effectiveness of the heuristics may be affected by the types of paths used for LID assignment even though our LID assignment schemes work with any routing schemes including multi-path routing and non dead-lock free routing and do not make any assumption about routing. In the evaluation, we consider two Up*/Down routing based schemes that guarantee to produce deadlock free routes. The first scheme is called the *Shortest Widest* scheme. In this scheme, the routing between each pair of machines is determined as follows. First, Up*/Down* routing (the root node is randomly selected to build the tree for Up*/Down* routing) is applied to limit the paths that can be used between each pair of machines. After that, a shortest-widest heuristic is used to determine the path between machines. This heuristic determines the paths between machines one by one. At the beginning, all links are assigned a weight of 1. When a path is selected, the weight on each link in the path is increased by 1. For a given graph with weights, the shortest-widest heuristic tries to select the shortest path between two nodes (among all paths allowed by the Up*/Down* routing). When there are multiple such paths, the one with the smallest weight is selected. The second routing scheme is called the *Path Selection* scheme. In this scheme, the paths are determined as follows. First, Up*/Down* routing is applied to limit the paths that can be used between each pair of machines. After that, a k-shortest path routing algorithms [13] is used to find a maximum of 16 shortest paths (following the Up*/Down* routing rules) between each pair of nodes. Note that some pairs may not have 16 different shortest paths. After all paths are computed, a path selection algorithm [7] is applied to select one path for each pair of machines. The path selection algorithm follows the most loaded link first heuristic [7], which repeatedly removing paths that use the most loaded link in the network until only one path for each pair remains. It has been shown in [7] that the most loaded link first heuristic is effective in producing load balancing paths. Both the shortest widest scheme and the path selection scheme compute one path for each pair of machines.

Table 1 depicts the performance of the heuristics when they are applied to the paths computed using the shortest widest scheme. The table shows the average of the total number of LIDs assigned to all machines. Each number is the average of 32 random instances. In computing the LIDs allocated for each node, LID mask control is taken into consideration: each node is assigned a power of 2 LIDs. We obtain the following observations from the experiments. First, the performance differences among the heuristics for the 16-switch configurations are very small. The performance difference between the best and the worst heuristics is less than 1%. The fact that five different heuristics, all computing minimal configuration sets for a routing in very different ways, yield similar performance suggests that for the paths computed by the shortest-widest scheme on networks with a small number of switches, other LID assignment schemes will probably have similar performance. Second, as the subnet becomes larger, the performance difference also becomes larger, even though the absolution difference is still small (less than 10%). For example, on the 64-switch configurations the performance differences between the best and the worst heuristics are 8.4% for 128 machines, 5.5% for 256 machines, and 4.9% for 512 machines. These results indicate that as the network becomes larger, the impact of selecting a good LID assignment heuristic becomes more significant.

conf.	greedy	s-m/S	s-m/L	color/S	color/L
128/16	478.7	478.9	477.3	479.3	476.4
256/16	1044.3	1045.4	1041.5	1047.7	1039.2
512/16	2218.3	2220.1	2211.8	2220.4	2208.5
128/32	451.5	453.9	452.9	461.3	443.0
256/32	1078.8	1084.7	1079.0	1100.0	1062.4
512/32	2428.7	2440.2	2425.8	2461.0	2392.1
128/64	422.8	427.7	427.0	441.5	407.4
256/64	1015.5	1022.2	1019.3	1044.6	990.6
512/64	2325.8	2338.4	2330.1	2385.1	2274.4

Table 1: The average of the total number of LIDs allocated (shortest widest)

Among the proposed heuristics, the split-merge approach has a very similar performance to the greedy algorithm. Thus, the higher complexity in the split-merge approach cannot be justified. The most split path first heuristic (color/L) is consistently better than all other heuristics while the least split path first (color/S) is consistently worse than other heuristics. This indicates that color/L is effective for this problem while color/S is not. The trend is also observed when the path selection scheme is used to compute paths.

Table 2 shows the results for the paths computed by the path selection scheme. Each number in the table is the average (over 32 random instances) of the total number of LIDs allocated to all machines for each configuration. There are several interesting observations. First, the performance differences among different heuristics are much larger than the cases with the shortest widest scheme. On the 16-switch configurations, the performance differences between the best and the worst heuristics are 24.7% for 128 machines, 24.8% for 256 machines, and 23.3% for 512 machines. For larger networks, the differences are more significant. On the 64-switch configurations, the performance differences are 30.1% for 128 machines, 30.0% for 256 machines, and 27.5% for 512 machines. This indicates that for the paths computed with the path selection scheme, which are more diverse than those computed by the shortest-widest routing, a good LID assignment heuristic significantly reduces the number of LIDs needed. The good news is that color/L consistently achieves the best performance in all cases, which indicates that this is a robust heuristic that performs well for different situations. Second, comparing the results for paths computed by the shortest widest routing (Table 1) with those computed by path selection (Table 2), we can see that when the number of machines is small (128 machines with 32 and 64 switches), the paths computed by the shortest widest scheme requires less LIDs to realize than the paths computed by the path selection scheme assuming the same LID assignment heuristic. However, when the number of machines is larger (256 and 512), the paths computed from the shortest-widest scheme requires more LIDs. This shows that routing can have a significant impact on the LID requirement.

In summary, depending on the routing method, LID assignment heuristics can make a significant difference in the number of LIDs required. The color/L heuristic consistently achieves high performance in different situations. The results also indicate that routing has a significant impact on the LID requirement, which argues for the separation of routing and LID assignment.

conf.	greedy	s-m/S	s-m/L	color/S	color/L
128/16	520.9	524.2	514.0	581.2	466.0
256/16	951.3	952.7	935.0	1062.6	851.2
512/16	1829.2	1852.8	1823.0	2038.7	1653.2
128/32	540.3	546.7	539.3	611.3	466.0
256/32	1006.7	1018.2	1002.2	1130.8	887.2
512/32	1904.0	1920.3	1895.7	2115.8	1688.7
128/64	528.0	541.1	530.5	599.4	460.5
256/64	1054.9	1092.9	1068.1	1197.9	921.4
512/64	2019.9	2075.4	2043.4	2278.6	1786.6

Table 2: The average of the total number of LIDs allocated (path selection)

conf.	Fully Explicit		Renaming		Separate	
	load	LIDs	load	LIDs	load	LIDs
128/16	4.34	128	3.84	477.8	3.70	466.0
256/16	8.65	256	7.52	1044.9	7.35	851.2
512/16	17.95	512	15.46	2213.3	14.91	1653.2
128/32	3.29	128	3.01	448.1	2.75	466.0
256/32	6.89	256	6.26	1079.2	5.8	887.2
512/32	14.71	512	13.24	2422.8	12.37	1688.7
128/64	3.29	128	3.01	420.0	2.75	460.5
256/64	6.15	256	5.72	1011.8	5.13	921.4
512/64	11.36	512	10.55	2323.4	9.54	1786.6

Table 3: The maximum link load and the number of LIDs required

5.2 Overall performance of various path computation methods

We compare a new path computation scheme that separates routing from LID assignment with existing path computation schemes for InfiniBand including destination renaming [5] and fully explicit routing [9]. The new path computation scheme, called *separate*, uses the path selection scheme described in the previous subsection for routing and color/L for LID assignment. The fully explicit routing [9] selects paths such that one LID is sufficient to realize all paths to a destination. Hence, at the expense of the load balancing property of the paths computed, this method requires the least number of LIDs among all path computation schemes. The destination renaming [5] scheme uses a shortest path algorithm to select paths that follow Up*/Down* routing rules. It assigns LIDs as the paths are computed. Both destination renaming and fully explicit routing are currently used [3]. All three schemes compute one path for each pair of machines.

We evaluate the performance of the path computation methods with two parameters: (1) the number of LIDs required, and (2) the load balancing property of the paths. We measure the load balancing property as follows. We assume that the traffic between each pair of machines is the same and measure the *maximum link load* under such a traffic condition. In computing the maximum link load, we normalize the amount of data that each machine sends to all other machines to be 1. Under our assumption, the load of a link is proportional to the number of paths using that link. A good load balance routing should distribute traffic among all possible links and should have small maximum link load values in the evaluation.

Table 3 shows the results for the three on routing and LID assignment schemes different configurations. The results are the average of 32 random instances for each configuration. As can be seen from the table, the fully explicit routing uses one LID for each machine, and thus, it requires a smallest number of LIDs. However, it puts significant constraints on the paths that can be used and the load balancing property is the worst among the three schemes: the maximum link load of fully explicit routing is much higher than other schemes. For example, on 128/16, the maximum link load with fully explicit routing is 17% higher than that with *Separate*; on 512/64, it is 19% higher. Destination renaming, which is more comparable to our proposed new scheme, has a better load balancing property than fully explicit routing. Our proposed scheme, *Separate*, has the best load balancing property in all cases, which can be attributed to the effectiveness of the path selection algorithm [7]. Moreover, for reasonably large networks (256 and 512 machines), *separate* also uses a smaller number of LIDs than destination renaming. For example, for the 512 machines/64 switches case, in comparison to destination renaming, the separate scheme reduces the maximum link load by 10.6% while decreasing the number of LIDs needed by 25.4%. This indicates that *separate* has much better overall performance than destination renaming: it reduces the maximum link load and uses a smaller number of LIDs simultaneously. This demonstrates the effectiveness of separating routing from LID assignment, as well as the effectiveness of the color/L LID assignment heuristic.

6. CONCLUSION

In this paper, we propose to separate routing from LID assignment in the path computation in InfiniBand networks. We prove that the problem of finding the smallest number of LIDs for realizing a routing is NP-complete. We develop a set of LID assignment heuristics and show that color/L is consistently the most effective heuristic among all proposed schemes in different situations. Depending on the routing method, color/L can be very effective in reducing the number of LIDs needed. We also demonstrate that the techniques developed in this paper can be used with the existing schemes that find dead-lock free and deterministic paths with good load balancing properties to obtain efficient path computation schemes for InfiniBand networks. We must note that our proposed path computation scheme, which separates routing from LID assignment, has a higher computation complexity than existing ones. Hence, it is more suitable to be used to compute the initial network configuration than to deal with incremental network changes.

Acknowledgement

This work is supported in part by National Science Foundation (NSF) grants: CCF-0342540, CCF-0541096, and CCF-0551555.

7. REFERENCES

[1] A. Bermudez, R. Casado, F. J. Quiles, T.M. Pinkston, and J. Duato, “Evaluation of a Subnet Management Mechanism for InfiniBand Networks.” *Proc. of the 2003 IEEE International Conference on Parallel Processing* (ICPP), pages 117–124, Oct. 2003.

[2] A. Bermudez, R. Casado, F. J. Quiles, and J. Duato, “Use of Provisional Routes to Speed-up Change Assimilation in InfiniBand Netrworks.” *Proc. of the*

2004 IEEE International Workshop on Communication Architecture for Clusters (CAC'04), page 186, April 2004.

[3] A. Bermudez, R. Casado, F. J. Quiles, and J. Duato, "Fast Routing Computation on InfiniBand Networks." *IEEE Trans. on Parallel and Distributed Systems*, 17(3):215-226, March 2006.

[4] M.Koibuchi, A. Funahashi, A. Jouraku, and H. Amano, "L-turn Routing: An Adaptive Routing in Irregular Networks." *Proc. of the 2001 International Conference on Parallel Processing* (ICPP), pages 383-392, Sept. 2001.

[5] P. Lopez, J. Flich, and J. Duato, "Deadlock-Free Routing in InfiniBand through Destination Renaming." *Proc. of the 2001 International Conference on Parallel Processing* (ICPP), pages 427-434, Sept. 2001.

[6] InfiniBand^{TM} Trade Association, *InfiniBand* ^{TM} *Architecture Specification*, Release 1.2, October 2004.

[7] M. Koibuchi, A. Jouraku and H. Amano, "Path Selection Algorithm: The Stretegy for Designing Deterministic Routing from Alternative Paths." *Parallel Computing*, 31(1):117-130, 2005.

[8] J. C. Sancho, A. Robles, and J. Duato, "A New Methodology to Compute Deadlock-Free Routing Tables for Irregular Networks." *Proc. of the 4th Workshop on Communication Architecture and Applications for Network-Based Parallel Computing*, Jan. 2000.

[9] J. C. Sancho, A. Robles, and J. Duato, "Effective Strategy to Computing Forwarding Tables for InfiniBand Networks." *Proc. of the International Conference on Parallel Processing* (ICPP), pages 48-57, Sept. 2001.

[10] J. C. Sancho, A. Robles, and J. Duato, "Effective Methodology for Deadlock-Free Minimal Routing in InfiniBand Networks." *Proc. of the 2002 International Conference on Parallel Processing* (ICPP), pages 409-418, 2002.

[11] M. D. Schroeder, A. D. Birrell, M. Burrow, H. Murray, R. M. Needham, T. L. Rodeheffer, "Autonet: a High-speed Self-configuring Local Area Network Using Point-to-Point Links." *IEEE JSAC*, 9(8): 1318-1335, 1991.

[12] Top 500 supercomputer sites. http://www.top500.org

[13] J. Y. Yen. "Finding the k shortest loopless paths in a network." *Management Science*, 17(11), July 1971.

[14] X. Yuan, W. Nienaber, Z. Duan, and R. Melhem, "Oblivious Routing for Fat-Tree Based System Area Networks with Uncertain Traffic Demands." *ACM Sigmetrics*, pages 337-348, June 2007.

[15] E. W. Zegura, K. Calvert and S. Bhattacharjee, "How to Model an Internetwork." *IEEE Infocom '96*, pages 594-602, April 1996.