An MPI Tool for Automatically Discovering the Switch Level Topologies of Ethernet Clusters

Joshua Lawrence Xin Yuan
Department of Computer Science, Florida State University, Tallahassee, FL 32306
{lawrence, xyuan}@cs.fsu.edu

Abstract

We present an MPI topology discovery tool for homogeneous Ethernet switched clusters. Unlike existing Ethernet topology discovery methods that rely on Simple Network Management Protocol (SNMP) queries to obtain the topology information, our tool infers the topology from end-to-end measurements. The tool works on clusters connected by managed and/or unmanaged Ethernet switches, and does not require any special privilege. We discuss the theoretical foundation of the tool, present the algorithms used, and report our evaluation of the tool.

1 Introduction

Ethernet is the most popular local area networking technology due to its low cost and high performance cost ratio. It provides the connectivity for many low-end high performance computing (HPC) clusters. We will use the term *Ethernet switched clusters* to refer to clusters with commodity workstations/personal computers connected by Ethernet switches. Such clusters include both dedicated HPC clusters and "casual" clusters that are formed by a set of random computers working for the same application.

Many HPC applications running on Ethernet switched clusters are message passing programs that use Message Passing Interface (MPI) [9] routines to perform communications. MPI defines the API for message passing and is the industrial standard for developing portable and scalable message passing programs. Two types of communication routines are specified in MPI: point-to-point routines that perform communications between two processes and collective communication routines that carry out communications among a group of processes. For an Ethernet switched cluster to efficiently support MPI applications, it is essential that the MPI communication performance is maximized.

The physical switch level topology of an Ethernet switched cluster strongly affects the performance of MPI collective operations. In particular, in an Ethernet switched cluster with multiple switches, using topology specific communication algorithms can drastically improve the performance over the topology unaware algorithms used in common MPI libraries such as MPICH [10] and OPEN MPI [11]. To achieve high MPI performance, automatic routine generators that take the topology information as input and automatically produce topology specific MPI collective routines have been developed [4, 5, 12, 13]. Unfortunately, for many clusters, especially the "casual" clusters, the switch level topologies are usually unknown. Having an automatic topology discovery tool may significantly benefit MPI applications on such systems.

In this paper, we present an MPI tool that automatically discovers the switch level topology of an Ethernet switched cluster. The tool infers the switch level topology based on end-to-end measurements. In comparison to the traditional Ethernet topology discovery methods/tools [1, 3, 6, 8] that rely on Simple Network Management Protocol (SNMP) queries to discover the topology, our tool offers several advantages. First, our tool does not require the switches to support SNMP and thus works on clusters connected by managed and/or unmanaged switches (the kind that does not support SNMP). Notice that unmanaged switches are much cheaper and are deployed much more widely than managed switches. Second, since our tool infers the topology based on end-to-end measurements, it does not require special privilege: anyone who can run MPI programs on a set of machines (that form a cluster) can use our tool to discover the switch level topology for the set of machines. The limitation of this tool is that it works best on homogeneous clusters with similar machines and switches, and may not be able to handle heterogeneous clusters. However, most MPI applications run on homogeneous clusters where this tool can effectively discover the topologies.

We tested the tool on various homogeneous clusters connected by fast Ethernet (100Mbps) or Giga-bit Ethernet (1Gbps) switches with different configurations. The results show that the tool is robust in discovering the switch level topology. With this tool, the automatic routine generators [4, 5, 12, 13] can use the topology information of a cluster and automatically produce efficient topology specific MPI collective routines. The tool can be downloaded at http://www.cs.fsu.edu/∼xyuan/MPI.

The rest of this paper is organized as follows: Section 2 discusses the related concepts in Ethernet switched clusters; Section 3 presents our topology discovery tool; Section 4 evaluates the components of the tool as well as the tool as a whole on different cluster configurations; Section 5 discusses the related work and Section 6 concludes the paper.

2 Ethernet switched clusters

An Ethernet switched cluster consists of workstations/personal computers connected by Ethernet switches. We will use the term *computers* to refer to workstations or personal computers. Since our tool works best on homogeneous clusters, we will assume that the clusters are homogeneous with the same type of computers, links, and switches. The links are fully duplex: computers connected to an Ethernet switch can send and receive at the full link speed simultaneously. Each machine is equipped with one Ethernet card. Ethernet switches use a spanning tree algorithm to determine forwarding paths that follow a tree structure [15]. Thus, the switch level physical topology of the network is always a **tree**.

The topology can be modeled as a directed graph $G = (V, E)$ with nodes V corresponding to switches and computers, and edges E corresponding to unidirectional channels. Let S be the set of all switches in the cluster and M be the set of all computers ($V = S \cup M$). Let $u, v \in V$, a directed **edge** $(u, v) \in E$ if and only if there is a link between nodes u and v. We will call the physical connection between nodes u and v **link** (u, v), which corresponds to two directed edges (u, v) and (v, u). Since the network topology is a tree, the graph is also a tree with computers being leaves and switches being internal nodes. Notice that a switch may also be a leaf, but will not participate in any communication and will be excluded from the topology. We define the *hop-count distance* between two end machines to be the number of switches in the path between them (the path is unique in a tree). Figure 1 shows an example cluster.

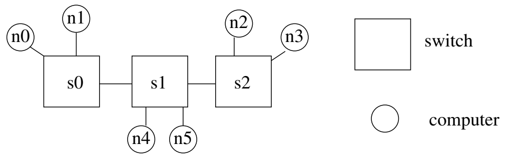


Figure 1. Ethernet switched cluster example

3 An automatic MPI topology discovery tool

The topology discovery tool is an MPI program that automatically infers the switch level topology from end-to-end measurements. Two key features in Ethernet switched clusters facilitate the determination of the switch level topology from end-to-end measurements.

- First, Ethernet switches use the store-and-forward switching mechanism. Thus, when a packet travels through a switch, a notable latency is introduced. Let the link bandwidth be B and the packet size be $msize$, the store-and-forward latency is at least $2 \times \frac{msize}{B}$ ($\frac{msize}{B}$ for receiving the packet from the input port and $\frac{msize}{B}$ for sending the packet to the output port). Consider 1Gbps Ethernet switches. When sending a 1000-byte packet ($msize = 1000B$), each switch adds at least $\frac{2\times1000\times8}{10^9}s = 16\mu s$. Such a latency is reflected in the round-trip time (RTT) between each pair of machines. Hence, we can measure the RTT between the pairs of machines and obtain a hop-count distance from these measurements.

- Second, the switch level topology of an Ethernet switched cluster is always a tree [15]. When the hop-count distance between each pair of computers is decided, the tree topology can be uniquely determined (we will prove this and give an algorithm to compute the tree topology from the hop-count distances).

The topology discovery process consists of two steps. First, the end-to-end round-trip time (RTT) between each pair of machines is measured and the hop-count distance is derived. The result of this step is a hop-count matrix with each entry in the matrix recording the hop-count between a pair of machines. Once the hop-count matrix is obtained, in the second step, the switch level (tree) topology is computed. In the following, we will first discuss how to compute the network topology from the hop-count matrix, and then describe how to obtain the hop-count matrix.

3.1 Computing the tree topology from the hop-count matrix

Let the number of machines in the cluster be N and let the machines be numbered from 0 to $N - 1$. The hop-count matrix, HC, is an $N \times N$ matrix. Each entry in the hop-count matrix, $HC_{i,j}$, $0 \le i \ne j \le N - 1$, records the hop-count distance from machine i to machine j.

Before we present the algorithm to compute the tree topology from the hop count matrix, we will show an example that a general topology (not a tree topology) cannot be uniquely determined from the hop-count matrix. Consider the 3-machine cluster in Figure 2 (a). Let us assume that the paths are bi-directional and $path(0, 1) = 0 \rightarrow s0 \rightarrow s1 \rightarrow 1$; $path(0, 2) = 0 \rightarrow s0 \rightarrow s3 \rightarrow s2 \rightarrow 2$; and $path(1, 2) = 1 \rightarrow s1 \rightarrow s2 \rightarrow 2$. Thus,

$$HC = \begin{bmatrix} 0 & 2 & 3 \\ 2 & 0 & 2 \\ 3 & 2 & 0 \end{bmatrix}.$$

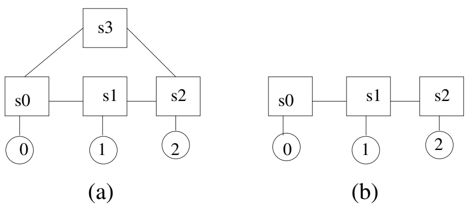


Figure 2. An example

This matrix is also the hop-count matrix for the topology shown in Figure 2 (b). This example shows that multiple (general) topologies can correspond to one hop-count matrix. Hence, a hop-count matrix cannot uniquely determine a corresponding topology. Fortunately, this is not that case when the topology is a tree. Next, we will show that the tree topology corresponding to a hop-count matrix is unique and can be derived from the hop-count matrix, which is the theoretical foundation of our tool.

Lemma 1: Let $G = (S \cup M, E)$ be a tree topology, S being the set of switches, M being the set of machines, and E being the set of edges. When $|S| \geq 2$, there exists a switch $s \in S$ such that s is connected to exactly one other switch.

Proof: We will provide a constructive algorithm to find such a switch (internal node) that connects to only one other switch. Starting from any switch, s_0. If this switch connects only to one other switch, then $s = s_0$ is found. Otherwise, pick any one of the switches connected to s_0. Let this switch be s_1. If s_1 connects only to one switch (s_0), then $s = s_1$ is found. Otherwise, s_1 connects to at least two switches. We will consider the switch that connects to s_1, but is not s_0. Let the switch be s_2. If s_2 connects to one switch (s_1), then the switch is found. Otherwise, there are at least two switches connected to s_2 with one not equal to s_1. We can proceed and consider that switch. This process is then repeated. Since the topology is a tree, each time a new switch will be considered (otherwise, a loop is formed, which contradicts the assumption that the topology is a tree). Since there are a finite number of switches in the topology, the process will eventually stop with the switch that connects to only one other switch being found. □

Definition 1: Let M be the set of machines, tree topology $G_1 = (S_1 \cup M, E_1)$ is said to be equivalent to tree topology $G_2 = (S_2 \cup M, E_2)$ if and only if there exists a one-to-one mapping function $F : S_1 \cup M \rightarrow S_2 \cup M$ such that (1) for all $u \in M$, $F(u) = u$; (2) for any $(u, v) \in E_1$, $(F(u), F(v)) \in E_2$; and (3) for any $(F(u), F(v)) \in E_2$, $(u, v) \in E_1$.

Theorem 1: Let M be the set of machines. Let $G_1 = (S_1 \cup M, E_1)$ and $G_2 = (S_2 \cup M, E_2)$ be two tree topologies connecting the machines M. Let $H(G)$ be the hop-count matrix for topology G. If $H(G_1) = H(G_2)$, then G_1 is equivalent to G_2.

Proof: We prove this by induction on the size of S_1 (the number of switches in G_1). Base case, when there is only one switch in S_1. In this case, we have $H(G_1)_{i,j} = H(G_2)_{i,j} = 1$, $0 \leq i \neq \jmath \leq N - 1$. If G_2 also has one switch, then G_1 and G_2 is equivalent since there is only one way to connect the N machines (each with one Ethernet card) to that switch. If G_2 has more than one switch, there are at least two switches in G_2 that are directly connected to machines (switches can only be internal nodes in the tree). Let s_1 and s_2 be two such switches in G_2 and let s_1 directly connect to machine m_1 and s_2 directly connect to machine m_2. We have $H(G_2)_{m_1,m_2} \geq 2$ since the path from machine m_1 to machine m_2 must pass through switches s_1 and s_2. This contradicts the fact that $H(G_2)_{m_1,m_2} = 1$. Hence, G_2 can only have one switch and is equivalent to G_1.

Induction case: assume that when $|S_1| \leq k$, when $H(G_1) = H(G_2)$, G_1 and G_2 are equivalent. Consider the case when $|S_1| = k + 1$. From Lemma 1, we can find a switch $s \in S_1$ that connects to only one other switch in G_1. This switch s must connect to at least one machine (otherwise it is a leaf, which is not allowed). Let the set of machines directly connect to s be M_s. Clearly, in G_2, there must exist a switch x directly connecting to all machines in M_s (otherwise, $H(G_1) \neq H(G_2)$). Moreover, switch x must connect to one other switch in G_2. We will prove this statement by contradiction.

Assume that switch x connects to more than one other switch in G_2. Let switches y and z be two such switches that directly connect to x. Let machine $x_m \in M_s$ be a machine directly connected to switch x. Since switches are internal nodes, there must exist a machine whose path to x_m goes through switch y. Let this machine be y_m and the switch directly connected to y_m be switch ly. Similarly, there exists a machine whose path to x_m goes through switch z. We denote this machine as machine z_m and the switch it directly connects to be switch lz. Let us assume that in G_1, machine y_m directly connects to switch t and machine z_m directly connects to switch r. Machine $x_m \in M_s$ directly connects to switch s in G_1. Figure 3 depicts the connectivity among these machines/switches in G_1 and G_2.

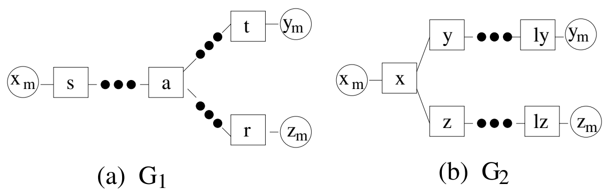


Figure 3. The connectivity of the related nodes in G_1 and G_2

In G_1 shown in Figure 3 (a), $path(x_m, y_m)$ and $path(x_m, z_m)$ share some switches (the first one is switch s). We denote the last switch in both paths be switch a. Hence, $path(x_m, y_m) = x_m \rightarrow s \rightarrow ... \rightarrow a \rightarrow ... \rightarrow t \rightarrow y_m$. $path(x_m, z_m) = x_m \rightarrow s \rightarrow ... \rightarrow a \rightarrow ... \rightarrow r \rightarrow z_m$. Since G_1 is a tree topology, there is a unique path between any two nodes. Hence, the unique path between y_m and z_m is $path(y_m, z_m) = y_m \rightarrow t \rightarrow ... \rightarrow a \rightarrow ... \rightarrow r \rightarrow z_m$. Since s only connects to one other switch, $a \neq s$ and $H(G_1)_{y_m,z_m} < H(G_1)_{x_m,y_m} + H(G_1)_{x_m,z_m} - 1$.

In G_2 shown in Figure 3 (b), $path(x_m, y_m) = x_m \rightarrow x \rightarrow y \rightarrow ... \rightarrow ly \rightarrow y_m$; $path(x_m, z_m) = x_m \rightarrow x \rightarrow z \rightarrow ... \rightarrow lz \rightarrow z_m$; and the unique path from y_m to z_m is $path(y_m, z_m) = y_m \rightarrow ly \rightarrow ... \rightarrow y \rightarrow x \rightarrow z \rightarrow ... \rightarrow lz \rightarrow z_m$. Hence, $H(G_2)_{y_m,z_m} = H(G_2)_{x_m,y_m} + H(G_2)_{x_m,z_m} - 1$. Under the assumption that $H(G_1) = H(G_2)$, we have $H(G_1)_{x_m,y_m} = H(G_2)_{x_m,y_m}$ and $H(G_1)_{x_m,z_m} = H(G_2)_{x_m,z_m}$. Hence,

$$\begin{aligned} H(G_1)_{y_m,z_m} &< H(G_1)_{x_m,y_m} + H(G_1)_{x_m,z_m} - 1 \\ &= H(G_2)_{x_m,y_m} + H(G_2)_{x_m,z_m} - 1 \\ &= H(G_2)_{y_m,z_m} \end{aligned}$$

This contradicts that fact that $H(G_1) = H(G_2)$. Hence, switch x can only connect to one other switch in G_2. Both G_1 and G_2 has the same sub-tree: $M_s \cup s$ in G_1 and $M_s \cup x$ in G_2. We can construct a reduced graph G_1^- by removing M_s from G_1 and changing switch s to a machine, and a reduced graph G_2^- by removing M_s from G_2 and changing switch x to a machine. Clearly, $H(G_1^-) = H(G_2^-)$ when $H(G_1) = H(G_2)$. From the induction hypothesis, G_1^- and G_2^- are equivalent. Using the one-to-one mapping function, f, that maps the switches in G_1^- to the switches in G_2^-, we can construct a one-to-one mapping function, g that maps switches in G_1 to the switches in G_2: $g(b) = f(b)$ for all $b \in S_1 - s$ and $g(s) = x$; and for all $u \in M$, $g(u) = u$. Clearly, (1) for any $(u, v) \in E_1$, $(g(u), g(v)) \in E_2$; and (2) for any $(g(u), g(v)) \in E_2$, $(u, v) \in E_1$. Thus, G_1 is equivalent to G_2. Hence, when $H(G_1) = H(G_2)$, G_1 is equivalent to G_2. $\square$

Theorem 1 states that the tree topology has a unique hop-count matrix. Next, we will present an algorithm that computes the tree topology from the hop-count matrix. The idea of the algorithm is as follows: when the hop count distances between all pairs of machines are 1's, the topology consists of one switch that connects to all nodes. When the hop-count distance for some pair of machines is larger than 1, there is more than one switch in the topology. In this case, the algorithm finds one switch that connects to only one switch in the topology (Lemma 1), and constructs the sub-tree that consists of this switch and attached machines to this switch. After that, the algorithm reduces the size of the problem by treating the newly found sub-tree as a single machine (with one less switch than the original problem), recomputes the hop-count matrix for the reduced graph, and repeats the process until all switches are computed. This algorithm is shown in Figure 4.

```
Input: the hop-count matrix, HC
Output: the tree topology stored in an
        array $parent[..]$. Switches are numbered
        from N to $N + |S| - 1$ in the result.

(1) $WorkHC = HC; size = N;$
(2) $for(i = 0; i < N; i++)\ NodeNum[i] = i;$
(3) $newswitch = N;$
(4) loop:
(5)   $largestd = max_{i,j}\{WorkHC_{i,j}\}$
(6)   if ($largestd == 1$) then
        /* the last switch left */
(7)     for ($i = 0$; $i < size$; $i++$)
          $parent[NodeNum[i]] = newswitch;$
(8)     exit; /* done */
(9)   end if
      /* More than two switches left */
(10)  Find a node $n$ such that there exists a $j$,
        $WorkHC_{n,j} == largestd$
(11)  Find all nodes whose distances to $n$ are 1's.
(12)  Let the set of nodes be $A$.
(13)  for all $i \in A$ parent[NodeNum[i]] = newswitch;
(14)  Reduce the tree by removing all nodes in $A$
        and adding a new leaf node newswitch,
        recompute $WorkHC$, and $NodeNum$.
(15)  newswitch ++
(16) end loop
```

Figure 4. Algorithm for tree topology

Figure 5 shows how the topology in Figure 1 is derived by the algorithm. The *WorkHC* stores the hop-count matrix in Figure 5 (a) in the first iteration. The nodes involved in the largest hop count distance are connected to switch $s0$ or $s2$. Assume that a node connected to $s0$ is selected. The sub-tree consisting of $n0$, $n1$, $s0$ is computed and the topology is reduced to Figure 5 (b) (*WorkHC* stores the hop-count distance for this topology in the second iteration). The newly added node is the blank circle. The hop-count distances for the reduced graph are derived from the old hop-count distances. For example, distance from the new node to n4 is equal to the distance from n0 to n4 minus 1. In the next iteration, the sub-tree of $n0$, $n1$, $s0$, $n4$, $n5$, and $s1$ is discovered and is reduced to Figure 5 (c), which is discovered in the last iteration.

3.2 Obtaining the hop-count matrix

The algorithm in Figure 4 derives the topology from the hop-count matrix. The remaining issue is to obtain the hop-count matrix. In theory, it is fairly straight-forward to ob-

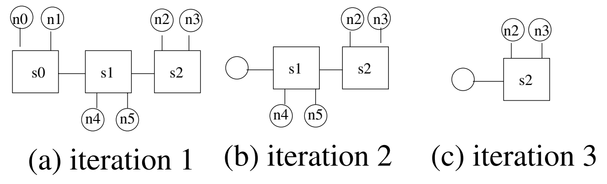


Figure 5. An example for topology calculation

tain the hop-count matrix from end-to-end measurements in a homogeneous cluster. The key issue is to filter out practical measurement anomalies, which may be caused by system noises, asynchronized system events, and other factors. However, automatically deriving the hop-count matrix from end-to-end measurements for a heterogeneous cluster is a much more difficult problem that we are still working on. Therefore, our tool works the best on homogeneous clusters. This tool uses three steps to obtain the hop-count matrix.

- Step 1: Raw RTT measurements. In this step, the tool uses *MPI_Send* and *MPI_Recv* to obtain the end-to-end RTT measurements. The main challenge is to ensure that the empirical measurement results are sufficiently accurate.
- Step 2: Data clustering. Statistical methods are applied to filter out the noises in the RTT measurements and to group the raw RTT measurement results into classes with each class corresponding to a hop-count distance.
- Step 3: Hop-count inference. Hop-count distances are inferred from the classified RTT results.

Raw RTT measurements

The RTT between each pair of machines is basically the ping-pong time between a pair of machines. Since the accuracy of the empirical measurements has a very significant impact on the effectiveness of the tool, our tool incorporates several mechanisms to ensure the ping-pong time measurements are accurate.

- RTT is measured with a reasonably large packet size. The default packet size in our tool is 1400 Bytes to ensure the latency from a 1Gbps (and 100Mbps) switch is greater than the measurement noise.
- Each RTT measurement is obtained by measuring multiple iterations of message round trips and using the average of the multiple iterations. Measuring multiple iterations increases the total time and improves the measurement accuracy. The tool uses a parameter $NUMRTT$ to control the number of iterations in each RTT measurement.
- A statistical method is incorporated to ensure that the measured results are consistent. To determine one RTT between a pair of machines, a set of X RTT samples are measured. The average and the 95% confidence interval of the samples are computed. If the confidence interval is small (the ratio between the interval and the average is smaller than a predefined value), the results are consistent and accepted. If the confident interval is large, the results are inconsistent and rejected. In this case, the number (X) of samples to be measured is doubled and the process is repeated. The RTT value is decided when the measurement results are sufficiently consistent or when a predefined large number (1000) of samples are taken (in this case, the measurement results may still be inconsistant, but we will use the results regardless). Having a threshold for the 95% confidence interval statistically guarantees the quality of the RTT measurements. The tool has two parameters related to the statistical method: *THRESHOLD* that is used to determine whether the 95% confidence interval is sufficiently small, and *INITPINGS* that is the number of samples taken initially.

The final measurement results are stored in the *RawRTT* array at the sender. After all *RawRTT* entries are measured, the results are gathered into machine 0, which will perform the later steps. Figure 6 shows an example *RawRTT* matrix that is obtained for the example topology in Figure 1.

$$RawRTT = \begin{bmatrix} 0.0 & .131 & .157 & .156 & .173 & .175 \\ .134 & 0.0 & .159 & .158 & .176 & .177 \\ .160 & .155 & 0.0 & .132 & .158 & .159 \\ .159 & .159 & .132 & 0.0 & .159 & .159 \\ .176 & .176 & .159 & .159 & 0.0 & .132 \\ .176 & .175 & .159 & .156 & .132 & 0.0 \end{bmatrix}$$

Figure 6. A round-trip-time matrix

Data clustering

As can be seen from Figure 6, the raw RTTs collected in the first step have noises. For example, in Figure 6, *RawRTT[0][2] = 0.157ms* and *RawRTT[0][3] = 0.156ms*. As discussed earlier, the difference in RTT values with different hop-count distances is at least $16\mu s = 0.016ms$ when the packet size is 1000B. Hence, the difference in the two measurements is due to measurement noises. The data clustering step tries to clean up measurement noises and group similar measurements into the same class, with each class covering a range of data values that correspond to a fixed hop-count distance.

The technique to locate these classes is traditionally known as *data clustering* [14]. In the tool, we use a simple data clustering algorithm shown in Figure 7. In this algorithm, the data are sorted first. After that, data clustering is done in two phases. The first phase (lines (4) to (15)), raw clustering, puts measurements that differ by less than a threshold, ($measurementnoise$) into the same class. After raw clustering, in the second phase, the algorithm further merges adjacent classes whose inter-class distance is not sufficiently larger than the maximum intra-class distances. The inter-class distance is the difference between the centers of two classes. The maximum intra-class distance is the maximum difference between two data points in a class. In the tool, the inter-class distance is at least *INTER_THRESHOLD* times larger than the maximum intra-class distances (otherwise, the two adjacent classes are merged into one class). *INTER_THRESHOLD* has a default value of 4. Note that standard data clustering algorithms such as the k-means algorithm [14] can also be used. We choose the simple algorithm because it is efficient and effective in handling data in our tool.

Input: RawRTT Matrix; Output: RTT classes

```
(1) Sort all RTTs in an array, sorted[..]. The sorted
        array has N*(N-1) elements
(2) measurementnoise = NUMRTT * 0.001
(3) numclass = 0;
        /* initial clustering */
(4) class[numclass].lowerbound = sorted[0];
(5) prev = sorted[0];
(6) for (i=0; $i < N * (N - 1)$; i++)
(7)   if ($sorted[i] - prev < measurementnoise$)
(8)     prev = sorted[i]; /* expand the class */
(9)   else
(10)    class[numclass].upperbound = prev;
(11)    numclass++;
(12)    class[numclass].lowerbound = sorted[i];
(13)    prev = sorted[i]
(14)  end if
(15) end for
        /* further clustering */
(16) do
(17)  for each class[i]
(18)    if ((intra-distance(i) is less than
        inter-distance(i, i+1)/INTER_THRESHOLD)
(19)      Merge class[i] and class[i+1]
(20)    end if
(21)  end for
(22) until no more merge can happen
```

Figure 7. Data clustering algorithm

For the round-trip-time matrix in Figure 6, the clustering algorithm will group the data into three classes: Class 0 covers range [0.131, 0.134]; Class 1 covers range [0.155, 0.160]; and Class 2 covers range [0.173, 0.177]. Finally, the RTT measurements in a class are replaced with the center of the class so they are numerically the same.

Inferring hop-count distances

There are two ways that the hop-count distances can be inferred in the tool. When the hop-count distances between some machines in the cluster are known, the hop-count distances can be correlated with RTT times off-line. The RTT times (that correspond to hop-count distances) can be merged into the RawRTT times during data clustering. In this case, the hop-count distance for a class can be assigned to the known hop-count distance that corresponds to the RTT time inside the class. The hop-count distances are determined when data clustering is done. This approach can be applied when the hop-count distances of some machines in the cluster are known, which is common in practice.

When the tool does not have any additional information, it uses a heuristic to infer the hop-count distance automatically. First, the heuristic computes the gaps between the center of adjacent classes. For the example from the previous section, the gap between Class 1 and Class 0 is $0.157 - 0.132 = 0.025$; and the gap between Class 2 and Class 1 is $0.175 - 0.157 = 0.018$. The tool then assumes that the smallest gap, 0.018 in this example, is the time introduced by one switch. The tool further assumes that the smallest time measured corresponds to the hop-count of one switch (two machines connecting to one switch): $class[0].hops = 1$. The formula to compute the hop-count for other classes is
$class[i].\ hops = class[i-1].hops +$

$$\left\lfloor \frac{class[i].center - class[i-1].center + \frac{smallest\_gap}{2}}{smallest\_gap} \right\rfloor$$

In the example, we have $smallest\_gap = 0.018$. $class[1].hops = 1 + \lfloor \frac{0.157-0.132+0.018/2}{0.018} \rfloor = 2$, and $class[2].hops = 2 + \lfloor \frac{0.175-0.157+0.018/2}{0.018} \rfloor = 3$. Hence, the hop-count matrix for this example is shown in Figure 8.

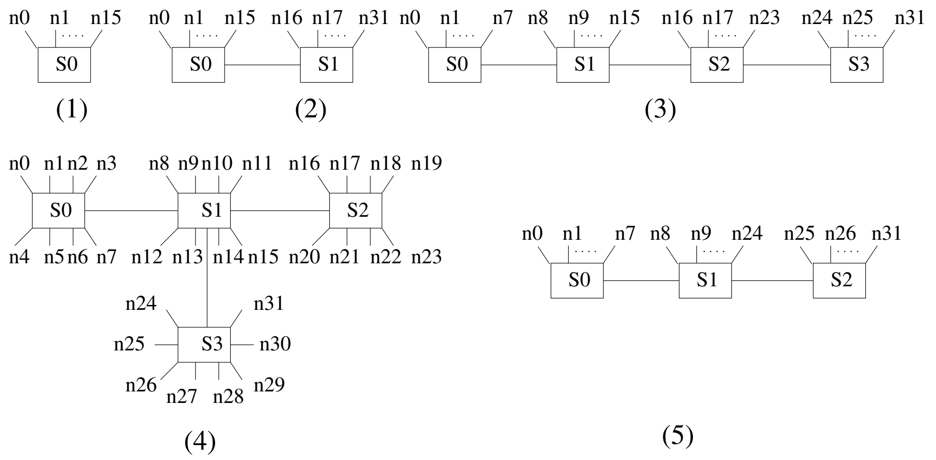
$$HC = \begin{bmatrix} 0 & 1 & 2 & 2 & 3 & 3 \\ 1 & 0 & 2 & 2 & 3 & 3 \\ 2 & 2 & 0 & 1 & 2 & 2 \\ 2 & 2 & 1 & 0 & 2 & 2 \\ 3 & 3 & 2 & 2 & 0 & 1 \\ 3 & 3 & 2 & 2 & 1 & 0 \end{bmatrix}$$

Figure 8. A Hop Matrix

This heuristic does not work in all cases. It works when (1) there exists one switch connecting more than one machine; and (2) there exist machines that are two switches apart. Most practical systems consist of a small number of switches and satisfy these two conditions.

4 Evaluation

We evaluate the topology discovery tool on homogeneous clusters with various configurations. Figure 9 shows these topologies. We will refer to the topologies in Figure 9 as topologies (1), (2), (3), (4), and (5). Topology (1) contains 16 machines connected by a single switch. Topologies



n0 n1 n15
S0
(1)
n0 n1 n15 n16 n17 n31
S0
S1
(2)
n0 n1 n7 n8 n9 n15 n16 n17 n23 n24 n25 n31
S0
S1
S2
S3
(3)
n0 n1 n2 n3 n8 n9 n10 n11 n16 n17 n18 n19
S0
S1
S2
n4 n5 n6 n7 n12 n13 n14 n15 n20 n21 n22 n23
n24 n31
n25 S3 n30
n26 n27 n28 n29
(4)
n0 n1 n7 n8 n9 n24 n25 n26 n31
S0
S1
S2
(5)

Figure 9. Topologies for evaluation

(2), (3), (4), and (5) are 32-machine clusters with different network connectivity. The machines are Dell Dimension 2400 with a 2.8 GHz P4 processor, 640MB of memory, and 40GB of disk space. All machines run Linux (Fedora) with 2.6.5-1.358 kernel. The Ethernet card in each machine is Broadcom BCM 5705 1Gbps/100Mbps/10Mbps card with the driver from Broadcom. Both Giga-bit Ethernet (1Gbps) and fast Ethernet (100Mbps) switches are used in the test (each configuration contains one type of switch). The Giga-bit switches are Dell Powerconnect 2624 (24-port 1Gbps Ethernet switches) and the fast Ethernet switches are Dell Powerconnect 2224 (24-port 100Mbps Ethernet switches).

In the evaluation, we have defined the following parameters. The RTT measurement scheme has a packet size of 1400 Bytes; *NUMRTT= 5*; *INITPINGS= 26*; the 95% confidence interval *THRESHOLD* is 3% of the average. In the data clustering scheme, the noise threshold is *measurement-noise=0.001*NUMRTT* (NUMRTT micro-seconds) and the inter-cluster distance is *INTER_THRESHOLD=4* times the maximum intra-cluster distance.

The tool successfully recognizes all the topologies in Figure 9 with both 1Gbps and 100Mbps switches, which demonstrates the tool is robust in discovering switch level topologies. In the following, we will present timing results for the tool. Each result is an average of five executions. All times are in the unit of second.

Table 1 shows the total execution time of the tool for each configuration in Figure 9. The second column contains the results with 100Mbps Ethernet switches; and the third column are the results with 1Gbps switches. As can be seen from the table, it takes about 120 seconds to discover the topology with 32 nodes using 100Mbps switches and less than 35 seconds for 1Gbps switches. We further breakdown the execution time of each phase in the tool in Tables 2 and 3, which show the times for raw RTT measurement (RTT), data clustering and hop count inferring (DCHC), and topology calculation (TC). As can be seen from the table, for all cases, more than 99.9% of the total time is spent on the RTT measurements phase. This is due to the $O(N^2)$ entries in the RTT matrix that need to be measured and the extensive statistical techniques that are incorporated in the tool to ensure the accuracy of the measurement results. If there is a need to improve the execution time, one can focus on optimizing the RTT measurement methods in this tool. In the experiments, most ($> 99\%$) of the *RTT* measurements take 26 samples to reach the confidence interval threshold. Few measurements require 52 samples. No measurements require more than 104 samples. The DCHC and TC phases run on machine 0 and do not involve communication. When N is small, the time for these two phases is insignificant.

Topology	100Mbps	1Gbps
(1)	20.200s	6.935s
(2)	101.812s	31.368s
(3)	126.741s	34.940s
(4)	122.377s	33.436s
(5)	110.007s	32.248s

Table 1. Total run time for configurations

Topology	RTT	DCHC	TC
(1)	20.188s	.007s	.005s
(2)	101.792s	.013s	.008s
(3)	126.716s	.056s	.010s
(4)	122.351s	.016s	.010s
(5)	109.987s	.013s	.008s

Table 2. The execution time for each phase (100Mbps switches)

5 Related work

Determining the switch level topology in an Ethernet cluster is a challenging and important problem. Most existing methods [3, 1, 8, 6] and commercial tools [7, 16]

Topology	RTT	DCHC	TC
(1)	6.926s	.005s	.004s
(2)	31.350s	.010s	.007s
(3)	33.900s	.022s	.020s
(4)	33.406s	.020s	.011s
(5)	32.231s	.010s	.007s

Table 3. The execution time for each phase (1Gbps switches)

rely on the Simple Network Management Protocol (SNMP) to query the Management Information Base (MIB) such as the address forwarding tables in the switches in order to obtain the topology information. These techniques are mainly designed for system management and assume the system administrator privilege. Moreover, they also require the switches to support SNMP, which is a layer-3 protocol and is not supported by simple layer-2 switches. In [2], the authors discussed the problems with the SNMP based approaches in general and advocated an end-system based approach. The technique proposed in [2], however, is also designed with the assumption that the system administrative privilege is granted. Moreover, the techniques to derive the topology in [2] are totally different from ours. Due to various difficulties in topology discovery, none of the existing methods can reliably discover topologies in all cases. In this paper, we present a new topology discovery tool that allows a regular user to discover the system topology for homogeneous clusters where MPI applications typically run.

6 Conclusion

We introduce an MPI topology discovery tool for Ethernet switched clusters. The tool infers the topology from end-to-end measurements. We describe the theoretical foundation and the algorithms used in the tool. This tool is proved to be effective in discovering the topology and can be used with automatic MPI routine generators to produce efficient topology specific MPI collective routines.

Acknowledgement

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

References

[1] Y. Bejerano, Y. Breitbart, M. GaroFalakis, R. Rastogi, “Physical Topology Discovery for Large Multi-subnet Networks,” *IEEE INFOCOM*, pp.342-352, April 2003.

[2] R. Black, A. Donnelly, C. Fournet, “Ethernet Topology Discovery without Network Assistance.” *IEEE ICNP*, 2004.

[3] Yuri Breitbart, Minos Garofalakis, Ben Jai, Cliff Martin, Rajeev Rastogi, and Avi Silberschatz, “Topology Discovery in Heterogenous IP Networks: The *Net-Inventory* System” *IEEE/ACM Transactions on Networking*, 12(3):401-414, June 2004.

[4] A. Faraj, X. Yuan, P. Patarasuk, “A Message Scheduling Scheme for All-to-all Personalized Communication on Ethernet Switched Clusters,” *IEEE Transactions on Parallel and Distributed Systems*, 18(2):264-276, 2007.

[5] A. Faraj, P. Patarasuk, and X. Yuan, “Bandwidth Efficient All-to-all Broadcast on Switched Clusters,” *International Journal of Parallel Programming*, Accepted.

[6] Hasan Gobjuka, Yuri Breitbart, “Ethernet Topology Discovery for Networks with Incomplete Information,” *IEEE ICCCN*, pp.631-638, Aug. 2007

[7] HP OpenView. Web page at http://www.openview.hp.com/products/nnm/index.asp, 2003.

[8] Bruce Lowekamp, David O’Hallaron, Thomas Gross, “Topology Discovery for Large Ethernet Networks,” *ACM SIGCOMM Computer Communication Review*, 31(4):237-248, 2001.

[9] The MPI Forum, “The MPI-2: Extensions to the Message Passing Interface,” Available at http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[10] MPICH - A Portable Implementation of MPI. http://www.mcs.anl.gov/mpi/mpich.

[11] Open MPI: Open Source High Performance Computing, http://www.open-mpi.org/.

[12] P. Patarasuk, A. Faraj, and X. Yuan, ”Pipelined Broadcast on Ethernet Switched Clusters,” *IEEE IPDPS*, April 2006.

[13] P. Patarasuk and X. Yuan, “Bandwidth Efficient All-reduce Operation on Tree Topologies,” *IEEE IPDPS Workshop on High-Level Parallel Programming Models and Supportive Environments*, March 2007.

[14] P. Tan, M. Steinbach, and V. Kumar, “Introduction to Data Mining”, Addison-Wesley, 2006.

[15] Andrew Tanenbaum, “Computer Networks”, 4th Edition, 2004.

[16] IBM Tivoli. Web page at http://www.ibm.com/software/tivoli/products/netview, Feb. 2003.