CrossTree: A New HTC Architecture with High Reliability and Scalability

Xiao-guang Liu, Meng Yu, Gang Wang, Jing Liu
College of Information Science, Nankai University, Tianjin, China, 300071)
E-mail: liuxg74@yahoo.com.cn

Abstract

HTC (High Throughput Computing) is an environment that can provide large amounts of processing capacity over long periods of time. To HTC, users are more concerned about how many jobs can be completed in a long period, but not how fast can a single job be finished. Condor, an implementation of HTC, is constructed by commodity CPUs and memory. As long as the Condor nodes are controlled by the Central Management Node, its reliability and scalability had been restricted. Based on the concept of DHT (Distributed Hash Table), this paper presents a new distributed HTC architecture, named CrossTree, which has no central parts, and its metadata is distributed across all nodes in the system. Theoretical analysis and the simulation results proved CrossTree to be an efficient architecture with high scalability and reliability.

1. Introduction

Floating point operations per second (FLOPS) has been used by most High Performance Computing (HPC) efforts to evaluate their systems. In many applications, most researchers are concerned with how many jobs per month or per year they can finish rather than FLOPS. We refer to such environments as High Throughput Computing (HTC) environments.

A common solution of HTC is using the spare time of many PCs. As a result, the more computers the system contains the more computing resources it can provide. However, if nodes in the system are organized in traditional centralized mode, its reliability and scalability will be restricted. DHT[1] (Distributed Hash Table) is a resources organization and location method in structured overlay network. It breaks the hash table held by central node into many pieces, and scatters them to all the nodes in the system. In such scheme, each node contains information about a certain part of whole system. When a resource location request arrived, it will be routed to its target through a series of nodes. DHT has good scalability and reliability.

Based on the idea of DHT, a new HTC architecture with no central parts, named CrossTree, is presented in this paper. In CrossTree, the metadata is distributed across nodes in the system. CrossTree is an efficient architecture with high scalability and reliability.

2. Related work

SETI@home is a distributed computing project using Internet-connected computers, hosted by the Space Sciences Laboratory in Berkeley[2]. Users who want to participate can download and install SETI@home client software, which repeatedly downloads jobs from a central server and reports any signals found in radio telescope data. SETI@home has over 500,000 active users and provides computing power over 20 TeraFlop/sec cumulatively. The cost of setting up and maintaining SETI@home is only $500 thousand. The overwhelming success of SETI@home project has demonstrated the tremendous capabilities of Internet-connected commodity resources.

Condor[3] is a typical HTC. It organizes many PCs together to form a Condor Pool. Nodes in Condor Pool can run tasks from other users or submit its tasks to spare nodes. Each Condor Pool has a single Central Management Node. It collects all the information about resources and tasks in the Condor Pool, and it carries out all the match and dispatch actions. Condor can make effective use of spare resources in the system. However, as long as the Condor nodes are controlled by the Central Management Node, its reliability and scalability is restricted.

3. Design of CrossTree

3.1. CrossTree’s Node ID and Architecture
Node ID in CrossTree is formed based on the resources it provides. For example, if Node A provides the following resources,

- **OpSys** = WINNT
- **Architecture** = INTEL
- **Memory** = 256 MB
- **DiskSpace** = 4096 MB

Then A’s Node ID is, 0001 0001 0256 4096

The first 0001 represents for WINNT operating system, the following 0001 stands for INTEL architecture, ‘0256’ means this node provides 256M memory space for sharing, and 4G disk space as well. It’s only a conceptual example. In practice, memory and disk space will also be encoded, and the number of resources is more than four listed above.

Using this method, the Node’s ID just corresponds to the resources it provides. The system architecture is organized as follows,

1. Nodes have same Node ID are collected together and built as trees, as shown in figure 1.
2. All the trees’ roots are connected with each other under full connection mode, called Cross part.

In CrossTree, nodes in the same tree have same Node ID and provide same resources. Each tree represents a kind of resource and corresponding jobs can be run on any spare nodes in the tree.

![Figure 1. CrossTree’s Architecture](image)

### 3.2. Route table

Nodes in CrossTree can be divided into two classes: the full-connection nodes (trees’ roots) and the tree nodes (excluding roots). For full-connection nodes, route table has two parts,

1. **Full-connection route table.**
   - It records the other roots’ <ID, IP Address> pair.
2. **Tree route table.**
   - There are some fields in the records. Such as, P1 (Parent’s IP), P2 (Grandparent’s IP), RP (Root’s IP), and < IP Address, Free, Level, FC >, which is collection of its child nodes. Here, ‘IP Address’ is the IP address of this child node. ‘Free’ indicates whether this child node is spare or not. ‘Level’ is a yardstick of this child’s height. It is used to maintain tree height balance when new nodes join the tree. ‘FC (Free Count)’ records the number of free nodes in all children of current node. When a request arrives at current node, if it is too busy, it will choose a node from its children whose FC field greater than zero, and forwards this request.

   For tree nodes, the route table includes only the tree route table. In order to improve system’s reliability, besides the tree route table, the direct children of full-connection nodes also maintain a copy of full-connection route table from their parents (the roots).

### 3.3. New node Joining

Firstly, the joining node needs to know a full-connection node, called K, which exists in the system. It’s similar to the DHT system. When new node N joins, it acts as follows,

1. Based on the resources it provides, node N get its Node ID, noted as NID.
2. N sends JOIN request to node K with its own NID and IP Address.
3. K looks up NID in its full-connection route table, if found then go to (5), else go to (4).
4. K adds <NID, IP Address> pair to its own full-connection route table, and sends <NID, IP Address> to all the other full-connection nodes to update their route tables. Meanwhile, K sends its full-connection route table to N for N’s initialization. P1 and P2 fields of N’s Tree route table are set to NULL, RP field is set to its own IP. Go to (*).
5. K forwards the JOIN message to the found node F, and F will become N’s root node.
6. F checks its tree route table, if the child list is not full, then <IP, true, 0, 0> is added to F’s tree route table, and F sends a copy of its full-connection route table to N, go to (&). If the child list is full, go to (7).
7. Select a node M with the minimum Level value in the child list, and forward the JOIN message to it.
8. M checks its tree part route table, if the child list neither empty nor full, then adds <IP, true, 0, 0> to it, go to (&). If the child list is full, go to (7). If the child list is empty, go to (9).
9. Insert < IP, true, 0, 0 > into tree route table, set N’s P1 field to be the IP address of current node, set N’s P2 field to be P1 value of current node, set N’s RP field to be RP value of current node. Traveling up step by step through P1 field until reach the root node, all the nodes on the way should increase its corresponding
FC field by 1, and Level field increased by 1 as well. Go to (*).

(&) Set N’s P1 field to be IP address of current node, set N’s P2 field to be P1 value of current node, set N’s RP field to be RP value of current node. Traveling up step by step through P1 field until reach the root node, all the nodes on the way should increase its corresponding FC field by 1.

(*) Node N joining procedure is over. The procedure is shown in figure 2.

3.4. Normal node leaving

When Node E decides to leave, it acts as follows:
(1) The tree is searched until one of E’s deepest leaf node, named P, is gotten.
(2) Move P from leaf to E’s position, copy E’s tree route table to P. If E is the root node or root’s child node, full-connection route table is also copied to P.
(3) Update the P1 of all E’s children, make P1 point to their new parent P. If E is the root node, P need broadcast its IP address to all the other full-connection nodes. The procedure is shown in figure 3.

3.5. System Structure Maintenance

When a full-connection node is doing the routing job, it may find that the target full-connection node fails to respond. At this time, it only sends this request to another tree which can provide enough resources, marks the leaving node as ‘disabled’ and start a timer for it. If this node receives response from the disabled node before the timer expires, then the disabled node returns to normal state again, else it would be deleted from the route table as soon as the timer expires. In fact, as long as the tree where the disabled node lies is not empty, due to the tree’s self-repair mechanism described in the next sector, a leaf node will soon replace the leaving node and update the route table of all the other full-connection nodes. Only when the leaving node is the last node of the tree, its failure will cause the tree vanish from the system.

Alive detection between tree nodes is the main part of CrossTree’s structure maintenance. In each tree, every node periodically sends keep-alive packets to its parent node to ensure its haleness.

The reasons that alive detection is used include:
(1) Maintain the integrity of the tree structure
When node realized that its parent node has been disabled, it will launch a corresponding process, as described in the next section, to deal with this case. Finally, a leaf node will be lifted up to replace the leaving node.
(2) Ensure all the status information up to date
Another important task that alive detection responsible for is to ensure all the status information of the tree is up to date. Along with the keep-alive packets, the sending node’s FC and Level fields are also sent to its parent node, thus during a single detection period, the parent node collects all its children’s FC field and Level field and then update itself to keep them up to date.

3.6. Node failure

When node E turns to be unreachable because of some unexpected reasons, E’s direct children (noted as collection S) will sense this failure depended on detection mechanism. These nodes act as follows:
(1) S send their own <Level, Free, IP> to their grandparent node Q (P2 pointes), or send to the first node Q in its copy of the full-connection route table if the failure node E is the root node.
When node Q receives these requests, it will perform arbitration and select node T which has the largest Level value in S. Q requires T for its deepest leaf node. T forwards this request to its deepest leaf node Z. Z sends its own IP address to node Q.

Move Z from leaf to E’s position, and set Z as new parent of all nodes in S. If E is root node, Z need to broadcast its IP address to all other full-connection nodes also. As shown in figure 4,

![Figure 4. Node failure in CrossTree.](image)

### 3.7. Job matching and dispatching

The job matching procedure acts as follows:

1. Based on the resources that the job needs, its Job ID, which noted as TID, is calculated.
2. If TID equals to current node’s NID, it means this job can be satisfied in the submitting tree, go to step 3. Else, go to step 5.
3. Checking the child list of current node to find a free node. If found, select it and go to step *. Else, select a child whose FC field greater than zero and forward this job request to it. This step is repeated on that node. If the children’s FC fields are all zero, go to step 4.
4. Forwarding this job request to parent node Q (P1 points). If Q is the root node, then go to step 7, else go back to step 3.
5. The job request is sent to the root node of the tree (RP points).
6. Looking up TID in R’s full-connection route table, if found, forward this job request to the found node Q. Else, go to step 0.
7. Checking the child list of current node to find a free node. If found, select it and go to step *, else select a child whose FC field greater than zero and forward this job request to it. This step is repeated on that node. If the children’s FC fields are all zero, go to step 0.

(*) Setting free field of selected node to be false, traveling up step by step according to P1 until reach the root node. All the nodes on the way should decrease its corresponding FC field by 1.

(X) Spare node A has been found, match success.

(0) There is no suitable node for this task. As shown in figure 5,

![Figure 5. Job Match and Dispatch](image)

### 4. Optimization to CrossTree

#### 4.1. Integration of physical and overlay network

Overlay network is a logical topology constructed over the physical network. Adjacent nodes in the overlay network may be far away in the physical network. It means the high efficiency in the overlay network does not mean it’s really in physical network. A solution is adding geographic information into the Node ID’s representation. For example, if A’s original ID is:

0001 0001 0256 4096

Then its new ID with geographic information is:

86 0001 0001 0256 4096

Here, ‘86’ means that this node resides in China. By this way, nodes in the same area will get together spontaneously. It’s similar that other information based on physical network also can be added into Node ID.

The new method is actually a more detailed division on the original ones. Compared with original grouping results, new method divides each group into smaller pieces according to different geographic zones. This detailed division makes trees smaller, tree’s height lowers and the hop count decreases in the logic level. Meanwhile, new method may lead to the growth of the full-connection route table.

#### 4.2. Cache Mechanism
In CrossTree, full-connection nodes not only need record more information than tree nodes, but also must work during submitting jobs to other group or receiving jobs from other group. It makes full-connection nodes as bottle-neck of the system.

Adding cache mechanism to submitting nodes can greatly alleviate this problem. New match process with cache tables can be described as follows:

1. When node S submits a job T, T’s ID is calculated, noted as TID.
2. If TID equals to S’s ID, then the job can be satisfied in local tree. Using original match algorithm to find a free node, go to step 0. If the job needs to be transferred to other group, go to step 3.
3. Looking up TID in S’s cache table, if found, then forward this job request according to the cache record. Original algorithm is used to continue the matching procedure. If TID is not found, original algorithm is used to find a remote free node.
4. After the target node A is found, the submitting node, S, inserts or updates < TID, A’s IP Address > in its cache table.

0. Match is finished. As shown in figure 6,

\[\text{Figure 6. Route with cache}\]

5. Theoretical analysis in CrossTree

5.1. Route Cost

If each tree node contains 100 children, then a tree which height is 4 can have 1 million nodes, a tree which height is 3 can contain 10,000 nodes. If jobs can be satisfied in local tree, the worst route hop is 6 in one-million-nodes tree, or 4 in 10,000-nodes tree. As shown in figure 7,

\[\text{Figure 7. Job matched in the local tree}\]

If a job need to be transferred to remote trees, it need one hop to the root node of local group, one hop to the target group root, and travel down. It means the worst route hop is 5 in the one-million-nodes tree, or 4 in the 10,000-nodes tree.

Compared with classic DHT networks, CrossTree provides nearly constant hop count. Figure 8 shows the worst route hop’s trends of Chord, CAN and CrossTree with the number of nodes increasing. Here, horizontal axis is the number of nodes and vertical axis is the hop count. We note that the hop count of Chord [4] and CAN [5] increases rapidly with the growth of the system scale. But the hop count of CrossTree remains low.

\[\text{Figure 8. Hop trends for three systems}\]

5.2. Maintenance Cost

1. Node joining

When a new node joins the system, the node will travels in the tree until it becomes a new leaf. This procedure is similar to the worst case of the remote match process, because they both modify the FC field of the nodes on the way. The only difference is that the joining node leads to the tree’s height increasing. It only happens 201 times in 10,000-nodes tree.

2. Normal node leaving

When a node decides to leave the system, it will travel down to one of the leaf nodes and go up to the root node to update FC field along the way. During this process, the number of nodes, whose information
need to be updated, equals to the tree’s height. Meanwhile, all the children of the leaving node will update their P1 field which point to their new parent. As a result, when a normal node leaves the system, the number of nodes, whose information needs to be updated, equals to height plus number of branch of the tree. So the total cost is nearly constant.

(3) Node failure

Compared with leaving process, node failure takes additional cost to send messages from the failure node’s children to their grandparent for arbitration.

5.3. Reliability Analysis

In this section, dual failure will be considered. Not all dual failure will cause breakdown. Only a node and its parent fail simultaneously, its children will be lost. As shown in figure 9,

\[ M = \begin{bmatrix}
-100\lambda & \rho \\
100\lambda & -\lambda - \rho
\end{bmatrix} \]

\[ \text{MTTF} = \left[ 1 \right]^T M^{-1} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{101\lambda + \rho}{100\lambda^2} \]

\[ = 2299.1 \hspace{1cm} (2) \]

It means this tree’s MTTF in CrossTree can reach 2299 hours, about 96 days.

Some parameters are used, \( \alpha \), which is number of branch in the tree and its value is 100. \( \lambda \) is failure probability of node and its value is 0.0083. It means each node can work without failure for 120 hours in expectation. \( \rho \) is recovery probability. Its value is 15. It means a node will take 4 minutes from failure to back alive again in expectation.

Considering a 10,000-nodes tree, its MTTF (mean time to failure) can be calculated as follows. Only the root and one of its child failed in same time can cause lost of deeper nodes. According to Markov chain shown in figure 10, status 0 is the initial state, status 1 is the state when one of the level 2 nodes fails, status B is the absorb state. The transition matrix \( M \) is

\[ M = \begin{bmatrix}
-100\lambda & \rho \\
100\lambda & -101\lambda - \rho
\end{bmatrix} \hspace{1cm} (3) \]

\[ \text{MTTF} = \left[ 1 \right]^T M^{-1} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{201\lambda + \rho}{10100\lambda^2} \]

\[ = 23.96 \hspace{1cm} (4) \]

In one-million-nodes tree, MTTF decreases to about 1 day. It’s noted that the loss nodes make little influence to whole system. Furthermore, when a node realizes it has been ‘lost’ by the system, it can contact the root node to join again.

6. Simulations and results

6.1. Simulation for node joining cost

In simulation experiments, one million nodes are added into an empty CrossTree and the average join hop count is calculated. Here, uniform mode means
each node provides same resources. Non-uniform mode means the resources owned by some nodes are much different to others. As shown in table 1, the average hop count is acceptable to one-million-node trees.

Table 1: Simulation result for node join cost

<table>
<thead>
<tr>
<th>Mode</th>
<th>Scale</th>
<th>Full-con number</th>
<th>Max. tree scale</th>
<th>Min. tree scale</th>
<th>Avg. hop</th>
</tr>
</thead>
<tbody>
<tr>
<td>uniform</td>
<td>1 M</td>
<td>400</td>
<td>2648</td>
<td>2357</td>
<td>3.026</td>
</tr>
<tr>
<td>n-uniform</td>
<td>1 M</td>
<td>400</td>
<td>19547</td>
<td>13</td>
<td>3.663</td>
</tr>
</tbody>
</table>

6.2. Simulation for route cost

It’s obvious that route cost relies on whether the trees’ height is balanced or not. The simulated experiment performs on a one-million-node CrossTree. The number of full-connection nodes is 400 and each tree has balanced height. In the experiment, submitting nodes are selected randomly. They generate 100,000 requests totally. There are two cases are considered in the experiment. Firstly, jobs are matched in local tree mostly. The probability that one job can be matched in local tree is 75%. Secondly, jobs are generated randomly. It means most of the jobs are matched in remote trees. As shown in table 2,

Table 2 Simulation result for route cost

<table>
<thead>
<tr>
<th>Mode</th>
<th>Scale</th>
<th>Task Num</th>
<th>Avg. hop (no cache)</th>
<th>Avg. hop (with cache)</th>
</tr>
</thead>
<tbody>
<tr>
<td>local</td>
<td>1 M</td>
<td>100,000</td>
<td>1.832</td>
<td>1.643</td>
</tr>
<tr>
<td>random</td>
<td>1 M</td>
<td>100,000</td>
<td>3.902</td>
<td>2.734</td>
</tr>
</tbody>
</table>

In random mode, the average hop count decreases remarkably when cache mechanism is used. However, the optimization is not obvious in local mode because matches in local tree are not recorded in the cache table. The results show that CrossTree has high route efficiency in both modes.

7. Conclusions

CrossTree, a new HTC architecture, is represented in this paper. In CrossTree, nodes are organized based on trees and ring. The metadata is scattered across all the nodes in the system like DHT. Distributed scheme and fault recovery mechanism make CrossTree to be an efficient architecture with high scalability and high reliability.

Acknowledgements

This paper is sponsored by NSF of China (90612001), Science and Technology Development Plan of Tianjin, (043185111-14), Nankai university R&D innovation fund and ISC.

References


