社団法人　情報処理学会　研究報告
IPSJ SIG Technical Report

2005－DPS－122（34）
2005－CSEC－28（34）
2005／3／23

# Distributed Data Management Based on Extensible Hashing in Grid Computing

**Yasutaka Nishimura, Yuichi Ayusawa, Tomoya Enokido, and Makoto Takizawa**

*Dept. of Computers and Systems Engineering*
*Tokyo Denki University, Japan*
{*yasu, ayu, eno, taki*}@*takilab.k.dendai.ac.jp*

**Abstract**

A huge volume of data are created, stored, and used in ubiquitous networks like stream data in sensor networks. We discuss how to dynamically distribute and efficiently locate huge volume of these data in computers of a Grid environment. We adopt a type of extensible hashing to distribute data. The index to locate data is tree-structured with sibling chain. However, since every access is sent to the root node, the root is a performance bottleneck and a single point of failure. In order to resolve performance bottleneck and improve the reliability, an access request is first issued to a leaf node which is local or nearest to an application. Then, the request is forwarded up and them down to the destination node. We evaluate the algorithm compared with traditional top-down searching one in terms of access time, computation overhead, and number of messages.

## グリッドコンピューティング環境における拡張ハッシングを用いた分散データベースシステム

西村 康孝　鮎沢 祐一　榎戸 智也　滝沢 誠
東京電機大学 理工学部 情報システム工学科

グリッドコンピューティング環境における分散データベースシステムでは、膨大なデータが広範囲に分散している。データの動的な分散配置方法として拡張ハッシングを用い、ノードを 2 分木に配置する。2 分木では、根ノードを中心に上位ノードの障害により、システム全体の障害となってしまう。また、これらのノードに検索要求が集中することによってアクセス時間が増大し、システム全体の性能に大きな影響を与えてしまう。本論文では、単一障害点を解決するための最適なデータの分散配置方法、データへのアクセス時間を減少させるために葉ノードから検索を開始する方法を提案し、アクセス時間、耐障害性についての評価も行う。

## 1. Introduction

In peer-to-peer overlay networks [10] and Grid computing systems [6], huge number and various types of computers are interconnected with various types of networks like the Internet. In addition, various types of computers are interconnected with networks. The Grid computing is defined as infrastructure for sharing coordinated resources and solving problem in dynamic among a set of individuals and/or institutions to achieve a shared goal [5]. In various types of applications, the Grid computing systems are now widely used [6].

In ubiquitous societies [2], huge volume of data like stream data [7] is dynamically created though sensor networks [13]. Here, data is not only generated but also accessed in various areas. In this paper, we discuss how to distribute and make an access to these data in the Grid. A Grid computing system is composed of various types of computers, ranging from high-performance and high-reliable computers like database servers to less-reliable computers like personal computers. Each computer does not provide an above large volume of data storage. Different computers provide different sizes and performance of data storages. A collection of data is distributed to a large number of computers, each of which is equipped with a smaller data storage. A unit of data storage is referred to as *bucket*. A bucket is realized in a computer. Records are dynamically hashed to buckets in computers by using the extensible hashing scheme [11]. The number of buckets is dynamically changed so that buckets neither overflow nor underflow. Using the index tree in main memory, the complexity for accessing to buckets through the index tree is kept in $O(1)$. Since huge number of records are stored in a large number of computers, the height of the index is getting deeper. In addition, huge number of applications make access to records through the index. The index tree is searched from the root to a leaf, i.e. in the top-down manner. The root node may be performance bottleneck and single point of failure. In order to resolve the difficulties, we newly adopt the hybrid searching algorithm. First, an application makes an access to a leaf node, i.e. bucket which is in a local computer or in a computer nearest to the application. The leaf node is referred to as an initial node of the request. If the record is not found, a request is forwarded up to the parent node. Here, if the record can be located through some subtree, the request is forwarded down to the subtree. Otherwise, the request is furthermore forwarded up to the parent node. We evaluate the hybrid searching algorithm compared with the traditional top-down one in terms of best and average access time and overhead of each node. Furthermore, if a bucket satisfying an access request is found, a pointer to the bucket is stored in the initial node. If a same access request is sent to the initial node, the target node is found through the pointer in the buffer.

Finally, we evaluate how many records each application can locate in presence of index node fault.

In section 2, we overview the extensible hashing scheme. In section 3, we discuss the modified index tree and the hybrid searching mechanism. In section 4, we evaluate the hybrid searching mechanism.

## 2. Extensible Hashing Scheme

We take a type of dynamic hashing algorithm [4, 8, 9] to distribute and locate records in buckets of computers. The maximum number of records which can be stored in a bucket $B$ is the size of the bucket $B$ denoted by $||B||$. A bucket *B* is a unit of data storage. The sizes of buckets are not necessarily the same. Each bucket is realized in a secondary storage of a computer. For example, a bucket is one file and another bucket is one database. A unit of data is a *record*. A record *r* is composed of attribute values. A collection of records with same attributes is referred to as *dataset*. A scheme of a dataset is a collection of attributes $a_0, \cdots, a_m$. A record *r* is a tuple of attribute values $<v_1, \cdots, v_m>$ where $v_i$ is a value of an attribute $a_i (i = 1, \cdots, m)$. The value $v_i$ in a record *r* is denoted by $r.A_i$. $|B|$ shows the number of records stored in a bucket $B$. The utilization ratio of a bucket $B$ is defined to be $|B|$ / $||B||$. Suppose a record is to be stored in a bucket *B*. Here, the bucket *B* is referred to as *overflow* if $|B| > ||B||$. If $|B| \leq c$ $(\leq ||B||)$ for a constant $c$, the bucket underflows.

First, one bucket $B_0$ in a computer $C_0$ is allocated [Figure1]. Every record generated is stored into the bucket $B_0$. In the meantime, the bucket $B_0$ overflows. Here, a new bucket $B_1$ is allocated in a computer $C_1$. Then, records in $B_0$ are redistributed to a pair of the buckets $B_0$ and $B_1$, i.e. $B_0$ is *split* into $B_0$ and $B_1$. There is a hash function $h(x)$ which gives a sequence of bits $b_1 b_2 \cdots b_l$ for a value $x$. A key *K* is a subset of the attributes, $K \subseteq \{a_1, \cdots, a_m\}$. For each record *r*, $h(r.K)$ is calculated and a sequence of bits $b_1 b_2 \cdots b_l$ is obtained. Here, if $b_1 = 0$, the record is stored in the bucket $B_0$, else in the other bucket $B_1$. Thus, every record *r* is stored in either $B_0$ or $B_1$ by using the hash function $h$. In the meantime, suppose the bucket $B_1$ overflows. Here, a new bucket $B_{11}$ is allocated in a computer $C_{11}$ and the bucket $B_1$ is renamed with $B_{10}$. Records in the bucket $B_{10}$ are rehashed to $B_{10}$ and $B_{11}$. For every record *r* in $B_{10}$, $h(r.K) = b_1 b_2 \cdots b_l$. Here, if the second bit $b_2$ is 0, the record *r* is stored in $B_{10}$, else in $B_{11}$. Each edge from a parent node to a child node is labeled 0 or 1 as shown in Figure 1. A label $label(N)$ of a node *N* is a sequence of labels for the root to the node $N$. For a root node *r*, $label(r) = \phi$. For example, $label(B_{10}) = 10$, $label(B_{11}) = 11$, and $label(b) = 1$ [Figure 1].

Let $b$ show a sequence $b$ of bits $b_1 b_2, \cdots, b_l$. Here, $b[i]$ shows $b_i$ and $b\langle i]$ stands for a prefix $b_1 b_2 \cdots b_i$, $b[i\rangle$ indicates a postfix $b_i b_{i+1} \cdots b_{l-1}$ of the bit sequence b $(i = 0, \cdots, l-1)$. $b[0\rangle = \phi$. $b\langle i] = b$ and $b[i\rangle = b$ if $i \geq l$.

For a node $N$ in the index tree, $parent(N)$ denotes a parent node of $N$. $child(N, 0)$ and $child(N, 1)$ show left and right child nodes of $N$, respectively. For example, $parent(b) = a$, $child(b, 0) = B_{10}$, and $child(b, 1) = B_{11}$ in Figure 1. $level(N)$ shows the level of a node $N$ in an index tree. $level(r)$ is 0 for a root node $r$. For example, $level(d) = 0$ and $level(B_{10}) = 2$. In this paper, we assume that the length $l$ of sequence obtained by the hash function $h$ in larger than the maximum depth of the index tree.

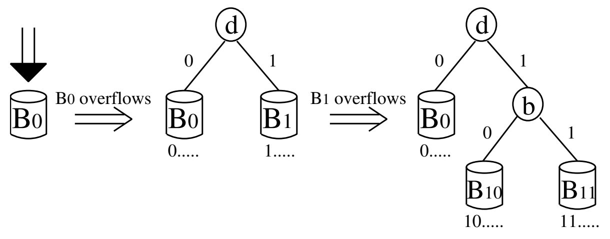

**Figure 1. Extensible hashing.**

In the paper [3], the index to locate buckets is realized in a linear table. The size of table depends or the deepest depth of the tree. In this paper, the index is realized in a $2^h$-ary tree. For simplicity, the index is assumed to be binary in this paper. Each node in the index tree is stored in one computer. Suppose an application would like to find a record whose key *K* is $v$. First, the application sends an access request $access(v)$ of the key value *v* to a root node $N$. The following procedure **TDsearch**$(0, v, N)$ is executed in a top-down manner:

```
TDsearch$(i, v, N)$ {
    if $label(N) = v\langle i]$,
        if $N$ is a leaf node, {
            $N$ is searched;
            if a record $r$ satisfying the key $v$ is found, return($r$);
            else return(NULL);
        }
    $h := v[i+1]$;
    return(TDsearch$(i+1, v, child(N, h))$);
}
```

Let us consider Figure 2. In the traditional index tree, every access request $access(011)$ is issued to the root node $r$. The access request sent to the child node $a$ since the first bit of the key 011 is 0. Then, the access request is forwarded to the node $c$ and finally the leaf node buket $B_3$. Thus, every access request is initially is sent to the root and then the request is forwarded down to the leaf node in the traditional top-down search..

Suppose a pair of buckets $B_{10}$ and $B_{11}$ have a parent index node $N$, i.e. $N = parent(B_{10}) = parent(B_{11})$. Let $\sigma$ be the size of a bucket. If $|B_{10}| + |B_{11}| < \alpha\sigma(0 < \alpha \leq 1)$, the buckets $B_{10}$ and $B_{11}$ are merged into $B_{10}$. Here, the parent node of $B_{10}$ is a parent of $N$.

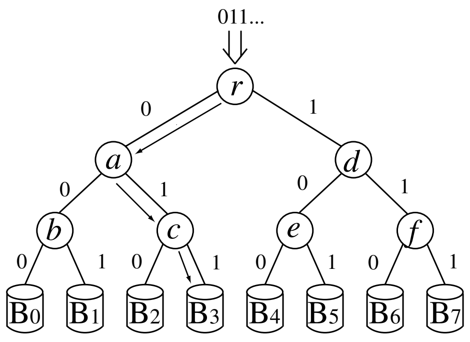

**Figure 2. Top-down search (TD).**

## 3. Index in a Grid Computing System

### 3.1. Hybrid search

In the Grid environment, data is not only distributed in a large number of computers but also accessed by a large number of applications. Since every application first makes an access to the root node in the index tree to locate the bucket node as presented in the top-down search algorithm **TDsearch**, the root node may be a performance bottleneck. That is, at the higher layer a node is in the index tree, the lager overheads the node implies. In addition, if the root node is faulty due to the fault of the computer, no data can be accessed. That is, the root node may be a single point of failure. In order to overcome the difficulties, we propose a hybrid searching approach where every application initially issues an access request to a leaf bucket node, not the root. The hybrid searching (HB) algorithm is shown as follows, where an access request with key value $v$ $access(v)$ is issued to a node whose level is $i$,

```
**HBsearch**$(i, v, N)$ {
    if $label(N) = v\langle i]$, {
        $h := v[i+1]$;
        return(**TDsearch**($i+1$, $v$, $child(N, h)$));
    } else return(**HBsearch**($i-1$, $v$, $parent(N)$));
}
```

An application first issues an access request $access(v)$ with key value $v$, to a bucket $N$ in some computer *C*. The node *N* is referred to as *initial* node of the access request $access(v)$. An application takes a local computer *C* of the local computer has a bucket $N$, otherwise a computer *C* with a bucket $N$ which is nearest to the application. Here, **HBsearch**$(i, v, N)$ is performed where $i$ is the level of the node $N$, $i = level(N)$

In Figure 2, an access request with hash key value 011 $access(011)$ is sent to a leaf bucket node, e.g. $B_1$ where an application exists. $label(B_1) = 001$ as shown in Figure 3. Since there is no record whose key value is 011 and $v\langle 2](= 0) \neq label(2)(= 1)$. Here, the access request $access(011)$ is forwarded up to the parent node $b$ where $label(b) = 00$. The access request is forwarded up to the parent node $a$ via a node $c$. Then, the access request is forwarded down to the leaf node $B_3$. Records are searched in the bucket $B_3$.

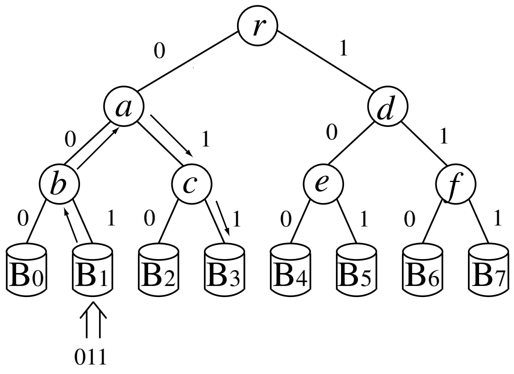

**Figure 3. Hybrid search (HB).**

### 3.2. Hybrid search with sibling chain

Next, we consider an index tree which is composed of same nodes as Figure 3 except that there is no root node, as shown in Figure 4. A pair of sibling nodes of a same parent are interconnected with a sibling link. $sibling(N)$ shows another sibling node of a node $N$. For example, $sibling(b) = c$, $sibling(f) = e$, and $sibling(B_0) = B_1$. For an access request $access(v)$ issued to a node $N$ where $level(N) = i$, the index tree is searched in the following procedure **HBCsearch**$(i, v, N)$;

```
**HBCsearch**$(i, v, N)$ {
    if $label(N) = v\langle i]$,
        return(**TDsearch**($i$, $v$, $N$));
    else if $label(N)\langle i-1] = v\langle i-1]$,
        return(**TDsearch**($i$, $v$, $sibling(N)$));
    else
        return(**HBCsearch**($i-1$, $v$, $parent(N)$));
}
```

An access request $access(010)$ is first is issued to the initial leaf node $B_1$ in Figure 4. Here, **HBCsearch**(3, 011, $B_1$) is performed. Then the access request $access(011)$ is forwarded up to the parent node $b$ in a same way as the HB algorithm since $label(B_1) \neq v\langle 3]$. In the HB algorithm, the access request is further forwarded up to the parent node $a$ of the node $b$. Since $label(b\rangle 1] = v\langle 1] = 0$, the request $access(011)$ is forwarded down to the child node $c$. In the HBC algorithm, the request $access(011)$ is directly forwarded to the sibling node $c$ from the node $b$ without passing the parent node $a$ as shown in Figure 4. Then, the access request $access(011)$ is sent to the target leaf node $B_3$.

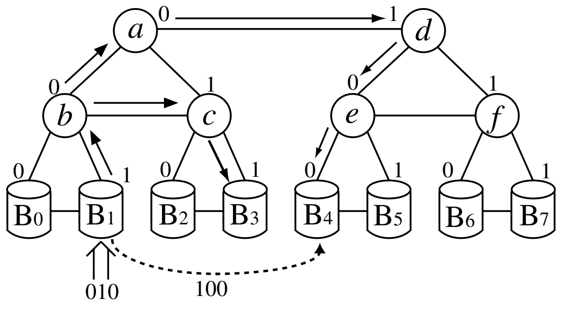

**Figure 4. Hybrid search with chain (HBC).**

We discuss the cost **cost**$(v, N)$ which shows how many nodes an access request $access(v)$ passes to the target node from a node $N$. For example, in Figure 3, an access request $access(011)$ is initially issued to the leaf node $B_{10}$. Then, the access request $access(011)$ is issued to the nodes $b$, $a$, $c$, and finally to the target leaf node $B_1$. Thus, the access request $access(011)$ passes over four nodes in the HB algorithm. On the other hand, since three nodes are passed in the HBC algorithm, **cost**$cost(011, B_1) = 3$. The cost **cost**$(v, N)$ in the HBC algorithm is shown as follows:

```
cost($v, N$) {
    for $i$ such that $label(N)\langle i] = v\langle i]$
    and $label(N)[i+1] \neq v[i+1]$,
        return($2(level(N) - (i+1)) + 1$); }

cost($v, N_1, N_2$) {
    if $N_1 = N_2$, return(cost($v, N_1$));
    else return(cost($v, N_2$) + 1); }
```

### 3.3. Hybrid algorithm with chain and link

In order to improve the response time in the HBC algorithm, we take the following approach:

1. On receipt of an access request $access(N)$ with key value $v$, an initial node *N* forwards a pair $\langle label(N), N\rangle$ of request $v$ and its identifier of *N* to the parent node of $N$ in the index tree as explained in the HB and HBC algorithms.
2. If the target leaf node *N'* is found, the target leaf node *N'* sends a response $\langle v, N'\rangle$ with a target record to the initial node *N*.
3. On receipt of the response $\langle label(N'), N'\rangle$, the response $\langle label(N'), N'\rangle$ is stored in the buffer of the initial node *N*.

Let *R* be a set of responses stored in the buffer. The size of the buffer in each node is limited. Suppose a target node *N'* is sent back the a response $\langle label(N'), N'\rangle$ of an access request $access(v)$ to the initial node *N*. If three is space to store the response $\langle label(N'), N'\rangle$ is added to the buffer, i.e. $R := R \cup \{\langle label(N'), N'\rangle\}'$. Otherwise, the buffer overflows. In this paper, a response which is least recently used is discussed and then the respance $\langle v, N'\rangle$ in the stored in the buffer, by using the LRU buffer replacement algorithm [12].

An access request $access(v)$ is issued to a leaf node *N*. Here, the following procedure **HBCLsearch**$(i, v, N)$ is performed where the node *N* is at level $i$ in the index tree:

**HBCLsearch**$(i, v, N)$ {

1. A response $\langle v', N'\rangle$ whose **Cost**$(v, N_1)$ is the minimal in $R \cup \{\langle label(N), N\rangle\}$ is selected.
2. An access request $access(v)$ is issued to the node $N'$. i.e. **HBCLsearch**$(i, v, N)$ is executed where *N'* is at level $i$ if $N \neq N'$.
3. If $N' = N$, **HBCsearch**$(i, v, N)$ is executed.

}

Figure 4 shows the hybrid search with sibling chain (HBC) algorithm, for the same bucket allocation as Figures 2 and 3. Suppose a bucket $B_4$ is detected through the initial node $B_1$ as explained. Here, the link to $B_4$ is stored in the bucket $B_1$. The link is identified by the label of the node $B_1$, i.e. 100. If an access request 100 is issued to the bucket $B_1$, the access request is directly sent to the node $B_4$ through the link 100.

Next, suppose an access request $access(110)$ is issued to the initial bucket $B_1$. Then are two ways to find the bucket $B_6$ whose label is 110. In one way, the request is forwarded to the nodes $b$, $a$, $d$, $f$, and finally $B_6$. In another way, the access request $access(110)$ is issued to the node $B_4$ through the link 100. Then, the request $access(110)$ is forwarded to the nodes $e$, $f$, and lastly the node $B_6$. **cost**$(110, B_1) = 5$, **cost**$(110, B_4) = 3$. Since **cost**$(110, B_1) >$ **cost**$(110, B_4) + 1$, the access request $access(110)$ is forwarded to the leaf node $B_4$.

## 4. Evaluation

In the evaluation, we consider the following three models discussed in :

1. Top-down (TD) model.
2. Hybrid model without sibling chain (HB) model.
3. Hybrid model with sibling chain (HBC) model.

In the top-down (TD) model, every access request is first sent to the root node. Then, the request is forwarded down to a child node. This is the traditional tree search model, for example, which is used in B+tree [1]. In the hybrid HB and HBC models, a request is first sent to a leaf node. Then, the request is forwarded up to a parent node and down to a child node. In the HBC model, a pair of child nodes with a same parent node are interconnected in a sibling chain as shown in Figure 4. Here, there is no root node. On the other hand, in the HB model, the index tree is the same as Figure 1, where the access request is forwarded to the sibling node only via the parent node.

Each access request is randomly assigned with a key value in the evaluation. In the HB and HBC models, each access request is randomly issued to one leaf node. Then, the request is forwarded to the target node according to the hybrid search algorithm. On the other hand, every request is first sent to the root node in the top-down (TD) model. Then, the access request is forwarded down to the target leaf node.

We evaluate the hybrid searching HB and HBC algorithms compared with the traditional top-down (TD) searching algorithm in terms of access time and overhead of each index node. We make the following assumptions:

1. The size of each bucket is 128k [bytes].
2. The number of buckets is 128.
3. The index tree is binary and height-balanced, where $h$ is the height of the index tree.

First, suppose that an access request to find a target leaf node is issued. We obtain access probability $P_\alpha(i)$ that a request is issued to a node at each level $i$ of the index tree in the algorithm $\alpha$ ($\in \{TD, HB, HBC\}$). In the TD algorithm, every access request is issued to the root node, i.e. the access probability $P_{TD}(0) = 1$. At the second level, the probability $P_{TD}(1)$ is 1/2. Thus, $P_{TD}(i) = 2^{-i}$ for each level $i$.

In the HB and HBC models, an access request is first issued to a leaf node. Each node receives access requests from the parent and the child nodes in the HB model. In the HBC model, each node receives access requests from the sibling node in addition to the parent and child nodes. The access probabilities $P_{HB}(i)$ and $P_{HBC}(i)$ are given for each level $i$ as follows:

$$P_{HB}(0) = 1/2$$

$$P_{HB}(i) = \frac{1}{2^i}\prod_{k=1}^{h-i}\frac{2^h - 2^{h-i-1}}{2^h - \lfloor 2^{h-2-i}\rfloor} + \frac{1}{2^{i+1}} \ (0 < i < h)$$

$$P_{HB}(h) = 2^{h-1} + 1/4^h$$

$$P_{HBC}(1) = 1/2$$

$$\begin{aligned} P_{HBC}(i) &= \frac{1}{2^i}\prod_{k=1}^{h-i}\frac{2^h - 2^{h-i}}{2^h - \lfloor 2^{h-1-i}\rfloor + \lfloor 2^{1-h+i}\rfloor} \\ &\times \left(1 + \frac{1}{2^h - 2^{h-i}}\right) + \frac{1}{2^i} \ (1 < i < h) \end{aligned}$$

$$P_{HBC}(h) = 2^h + 2/4^h$$

The access probabilities $P_{TD}(i)$, $P_{HB}(i)$ and $P_{HBC}(i)$ shows how many percentages of the total number of access requests are issued to each node at level $i$ in the index tree. The access probability indicates overhead of each node to process an access request. Figure 5 shows the access probability ratios for the TD, HB, and HBC algorithms. In the HB and HBC algorithms, the overhead of the root node can be reduced. However, the overhead of nodes at lower level are increased than the TD algorithm.

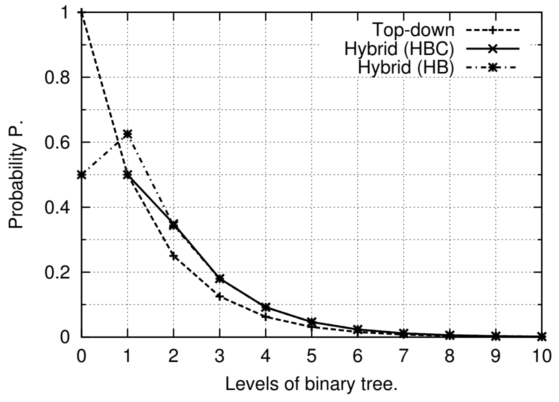

**Figure 5. Ratio of request messages.**

A node may be faulty. In the TD algorithm, if the root node is faulty, all the nodes cannot be reached. However, nodes are still able to be accessed in the HB and HBC algorithms even if the root node is faulty. Figure 6 shows the fault ratios of the TD, HB, and HBC algorithms for each level in the index tree. In the TD algorithm, $S_{TD}(0) = 1$ means that all the nodes in the index tree cannot be accessed if the root node is faulty. In the HB algorithm, the half of the nodes are reachable even if the root node is faulty. Following the figure, if a node in the index tree is faulty, more number of nodes still accessible in the HB and HBC algorithms. Let $NS_\alpha(i)$ be the number of nodes which can be accessed due to the fault of a node at level $i$ in an algorithm $\alpha$. $S_\alpha(i)$ shows the ratio of $NS_\alpha(i)$ to the total number of nodes in the index tree. i.e. how many nodes can be accessed in the index tree. $S_\alpha(i)$ is obtained as follows:

$$S_{TD}(i) = 1 - 2^{-i}$$

$$S_{HB}(0) = 1/2$$

$$S_{HB}(i) = \frac{4^{i+1} - 2^{i+3} + 5}{4^{i+1}} \ (0 < i < h)$$

$$S_{HB}(h) = 1 - 2^{-h}$$

$$S_{HBC}(1) = 1/2$$

$$S_{HBC}(i) = \frac{4^i + 4^{i-1} - 2^{i+1} + 1}{4^i} \quad (1 < i < h)$$

$$S_{HBC}(h) = 1 - 2^{-h}$$

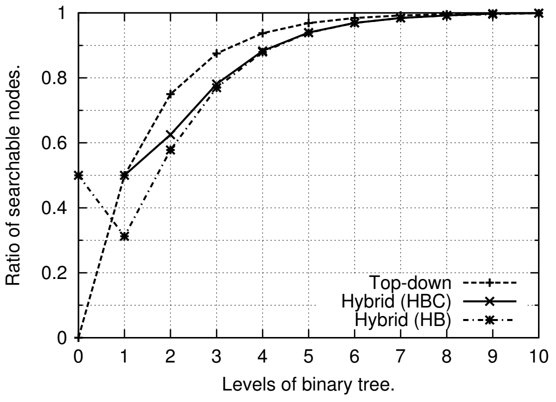

**Figure 6. Ratio of searchable nodes.**

## 5. Concluding Remarks

We discussed the extensible hashing scheme to distribute records in the Grid computing environment and the newly proposed hybrid searching algorithm HB and HBC to locate records distributed in the network. In addition, we discuss another algorithm HBCL where the target index information obtained from each leaf node is buffered in the leaf node. We evaluated the hybrid searching algorithms compared with the traditional top-down (TD) searching one in terms of performance and survivally.

## References

[1] R. Bayer and E. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acts Informatica*, 1:173–189, 1972.

[2] E. Einemann and M. Paradiso. Digital Cities and Urban Life: a Framework for International Benchmarking. *Proc. of the Winter International Synposium on Information and Communication Technologies (WISICT'04)*, pages 1–6, 2004.

[3] R. J. Enbody and H. C. Du. Dynamic Hashing Schemes. *ACM Computing Surveys*, 20(2):85–113, 1988.

[4] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible Hashing - a Fast Access Method for Dynamic Files. *ACM Transactions on Database Systems (TODS)*, 4(3):315–344, 1979.

[5] I. Foster. *What is the Grid? A Three Point Checklist*. GRIDToday, pages http://www–fp.mcs.anl.gov/ foster/Articles/WhatIsTheGrid.pdf, 2002.

[6] I. Foster and C. Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2003.

[7] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2001.

[8] P.-Å. Larson. Dynamic Hashing. *BIT*, 18(2):184–201, 1978.

[9] W. Litwin. Linear Hashing: A New Tool for File and Table Addressing. *Proc. 6th International Conf. on Very Large Data Bases (VLDB)*, pages 212–223, 1980.

[10] R. Schollmeier. A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications. *Proc. of the First International Conf. on Peer-to-Peer Computing (P2P'01)*, pages 101–102, 2001.

[11] O. Shalev and N. Shavit. Split-Ordered Lists: Lock-Free Extensible Hash Tables. *Proc. of the 22nd Annual Symposium on Principles of Distributed Computing*, pages 102–111, 2003.

[12] P. Silberschatz and A. W. Galvin. *Operating System Concepts (4th Edition)*. John Wiley & Sons.Inc., 2001.

[13] F. Zhao and L. J. Guibas. *Wireless Sensor Networks: An Information Processing Approach*. Morgan Kaufmann, 2004.