

# A Scalable and Efficient Prefix-based Lookup Mechanism for Large-Scale Grids

Philip Chan

David Abramson

e-Science and Grid Engineering Laboratory (MESSAGELab)

Monash University

900 Dandenong Road, Caulfield East, VIC 3145, Australia

{pchan,davida}@csse.monash.edu.au

## Abstract

*Data sources, storage, computing resources and services are entities on Grids that require mechanisms for publication and lookup. A discovery service relies on efficient lookup to locate these objects from names or attributes. And on large-scale grids, these should be scalable and be able to support range queries and multi-criteria searching. Trie-based approaches like the PHT and DPT, enable sophisticated queries while providing resilience to failures. In this paper, we propose a more efficient variant of the DPT called the Dynamic Prefix Graph (DPG). And we introduce an encoding scheme that allows DPGs to support both string partial matching and numerical range queries. A distributed algorithm for dynamic construction of DPGs is presented. We conduct simulation studies with the DPG under various data sets. And results demonstrate a reduction of up to 30% in logical hops when lookups are performed under DPGs compared to DPTs.*

## 1 Introduction

Computational Grids have become the emergent infrastructures for large-scale scientific applications. GT4 [5] and other technologies are making it possible to orchestrate experiments over wide-area resources, enabling an unprecedented level of scientific research and collaboration. And the ever-increasing scale of grids inevitably leads to the convergence of grid technologies and peer-to-peer approaches [6].

Of particular importance to grid infrastructures is object publication and discovery. In the context of grids, an object may be a data file/set, a computing resource, or more abstractly, a service. Object location is provided through a lookup mecha-

nism, which is expected to efficiently determine the address of an object given its name or some attribute/s. Names are usually location-independent strings that may identify the service provider and a short textual description of the service, while attributes are of the form:  $\langle \text{property\_name} \rangle = \langle \text{value} \rangle$ , e.g., `cputype=P4Xeon`, `ram=4096mb`, `speed=2.8GHz`. The lookup mechanism associates the address/location of the service with the attribute, allowing retrieval of this address when queried. Service discovery mechanisms [7] are built over such a lookup mechanism, providing other functionalities like contracts.

The most basic service discovery on grids will require support for exact-match queries. These simple lookups, e.g., “Find a repository with name of X”, are supported in a straightforward manner by distributed hash tables (DHTs). Second generation, structured P2P systems (e.g., Chord [10], Pastry [9] and Tapestry [11]) provide key-based routing and can efficiently support higher-level interfaces such as DHTs.

Discovery mechanisms are also expected to handle range queries and multiple criteria. Trie-based techniques like PHT [8], P-Grid [4] and DPT [3] have been demonstrated to be well-suited for this purpose. Range queries come in 2 forms: (a) auto-complete, prefix-based queries, e.g., “Find all resources with any version of the `gtk` (GIMP Toolkit) installed”, performed by searching with “`gtk`” as a prefix (matching “`gtk-1.2`” and “`gtk-2.0`”); and (b) bounded range queries, e.g., “Find all resources with the `gsl` (GNU Scientific Library) from version 1.5 to 1.8, inclusive”. For this second form of queries, current approaches uses the prefix “`gtk`” and propagates the key-lookup messages to all nodes that match this prefix, which can lead to many extraneous messages.

Finally, multiple criteria queries are useful for locating resources or services that meet several constraints,

for example: the name of the operating system (e.g., Solaris, Linux, etc.), the processor type (e.g., PowerPC G5, AMD Opteron, etc.), the minimum amount of RAM, etc. DPTs solve this by performing parallel searches on different trees, then consolidating the results at the client side.

We investigate how lookups can be made more scalable and efficient. In this paper, we propose a prefix-based lookup mechanism using a trie structure. In particular, we present two main contributions: (a) the DPG data structure which offers improved lookup performance over the Dynamic Prefix Tree (DPT) [3] by reducing the number of hops during node traversal; and (b) an lexical encoding scheme that allows numeric properties to be handled. With this encoding scheme, range queries with numeric properties require less messages as requests are sent to those nodes that are pertinent to the search.

## 2 DPGs: Dynamic Prefix Directed Acyclic Graphs

We present a trie representation called Dynamic Prefix Graph (DPG) as a scalable and efficient distributed lookup mechanism for service discovery. A variant of the Dynamic Prefix Tree (DPT) [3], the DPGs relax the constraint that requires all nodes has at most one predecessor (or parent).

We use the boolean binary operator  $\sqsubset$  (to be read as “is a proper prefix of”), which evaluates (proper) prefix for two strings. For example, “matrix”  $\sqsubset$  “matrixA” while “matrix”  $\not\sqsubset$  “matrix” and “matter”  $\not\sqsubset$  “matrix”. We also use of a *longest common prefix* function  $\text{LCP}(u, v)$ , which returns the string  $s$  that satisfies:  $s \sqsubset u$  and  $s \sqsubset v$  and  $|s|$  is maximal.

Formally, a DPG  $G = (V, E)$  is a directed-acyclic graph with a finite set  $V$  of logical nodes and  $E$  is a set of arcs. Each logical node  $v \in V$  has an attribute  $v.key$ , a non-empty string of characters from an alphabet  $\Sigma$ , i.e.,  $v.key \in \Sigma^+$ . One characteristic property<sup>1</sup> of a DPG is that for any pair of nodes  $v_i, v_j \in E$ , if  $v_i \rightarrow v_j \in E$ , then it follows that  $v_i.key \sqsubset v_j.key$ . This prefix property guarantees the DPG is rooted and acyclic.

In a DPG  $G$ , the set of arcs  $E$  is the union of two disjoint arc sets, namely:  $T$  – “vital” arcs; and  $A$  – auxiliary arcs. Formally, for every  $v_i \rightarrow v_j \in E$ ,  $v_i \rightarrow v_j \in T$  if  $\neg \exists v_k$  with  $v_k \rightarrow v_j \in E$  and  $\text{LCP}(v_i.key, v_j.key) \sqsubset \text{LCP}(v_k.key, v_j.key)$ . By this definition, it follows that given any DPG  $G = (V, T \cup A)$ , the equivalent DPT is constructed as  $G_{DPT} = (V, T)$ .

There are many variants of DPGs depending on the

auxiliary arcs in  $A$ . For example,  $A$  may contain all  $v_i \rightarrow v_j$  that satisfy  $v_i.key \sqsubset v_j.key$  and  $v_i \rightarrow v_j \notin T$ . An alternative definition to construct this DPG is to compute  $E$  as the transitive closure of  $T$ . For the purpose of implementing lookups, maintaining all these arcs on the DPG may be expensive. In this paper, we present how the DPT construction algorithm is modified to construct DPGs by retaining arcs that are normally removed during key insertion for the DPT.

### 2.1 DPG Compared to DPT

The DPG, like the DPT, is used as a lookup data structure for key/value pairs, where keys are arbitrary character strings. Keys are arranged lexicographically so that prefix-based partial matches can be performed efficiently by exploring a relative small number of logical nodes. In Figures 1 and 2, we illustrate how these two representations differ as key/value pairs are inserted.

In Figure 1, we show how a DPT is constructed as 6 keys are inserted, arranged from left to right. The keys and the order of their insertion are: “abcdefgh”, “abce”, “abcde”, “ab”, “abcdef”, “abcf”. During the insertion of the second key “abce”, a logical node (shown as shaded) representing the “virtual” key “abc” is created to serve as the immediate predecessor for both “abcdefgh” and “abce”. This logical node represents the longest common prefix of these two keys and does not hold any values.

In Figure 2, we show how a DPG changes as key/value pairs are inserted. The dotted-line arrows represent the auxiliary arcs (i.e., those in  $A$ ) of the DPG. Notice that these auxiliary arcs were at one time or another, part of the vital set of arcs  $V$ . On a DPT, these arcs will be removed to ensure the invariant property that each node has at most one predecessor.

### 2.2 A Distributed Algorithm for DPGs

We present the distributed algorithm for constructing DPGs. The detailed algorithm written in Python-like language is presented in Figure 3. As in Python, the indentation of the statements implicitly defines the block structure of the code. We selected this convention to present the algorithm concisely. And in the narrative description, we use the convention (#10) to reference specific statements on the algorithm.

The DPG construction algorithm<sup>2</sup> is similar to the one for DPT. However, since each node may several

<sup>1</sup>Note that the same property applies for a DPT.

<sup>2</sup>For brevity, we omitted the code that handles the case when the DPG does not exist yet (i.e., a new node will be created and initialised).

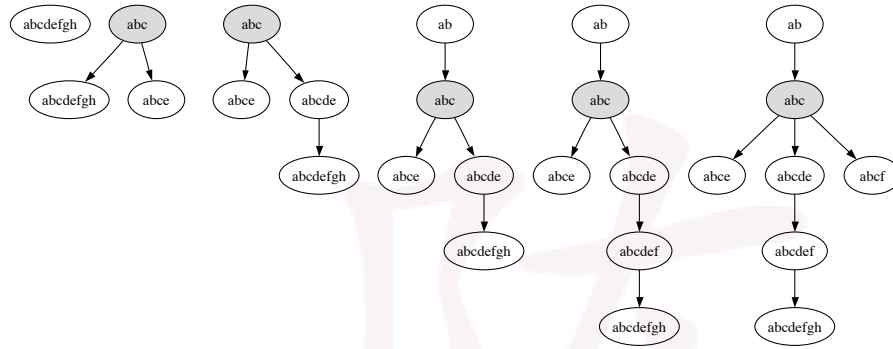


**Figure 1.** Changes to the DPT [3] as key-value pairs are inserted. Six trees are shown, arranged from left to right, as each of the keys are added. The shaded vertex represents a virtual key node.

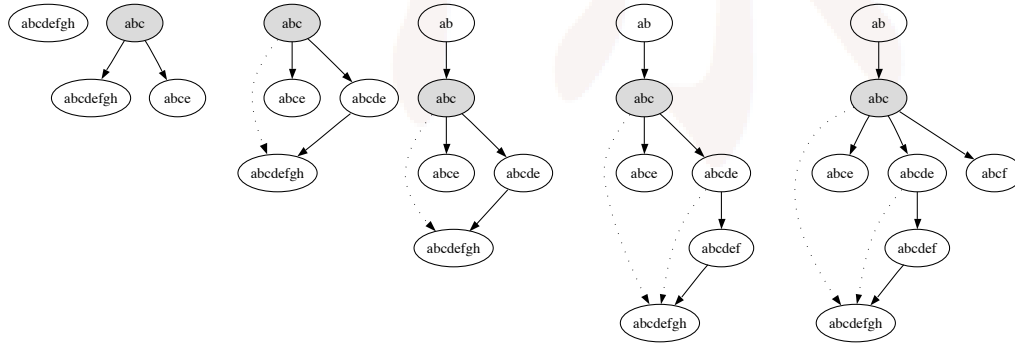


**Figure 2.** Changes to the Dynamic Prefix Graph as key-value pairs are inserted. Six DAGs are shown, arranged from left to right, as each of the keys are added. The shaded vertex represents a virtual key node. The dotted arcs are those extra arcs that differentiate DPGs from DPTs.

predecessors and successors, a more complex criteria is employed to select the predecessor or successor to forward an **add** request.

The local state on each node includes:

1. a key representing the exact string or the prefix;
2. an id which hold my own address;
3. a set **SSet** containing links/addresses to immediate successors; and
4. a set **PSet** containing links/addresses to immediate predecessors.

Upon receiving a request to add a key/value pair into the graph, the algorithm determines if this request can be handled locally or has to be forwarded to a predecessor or a successor. Informally, this algorithm works as follows (key is the new key to be added):

If  $\text{key} = \text{my.key}$ , then insert **key** locally (#2–#5);

If  $\text{my.key} \sqsubset \text{key}$ , then this key is to be inserted as a new successor. The immediate successor selection to forward the insert request involves two steps:

1. Forward the request to a successor that holds the key that is a longest prefix of **key**. (#10)
2. If none are found, then there are 3 cases. The “closest” successor **s** is determined as the one with the longest  $\text{LCP}(\text{s.key}, \text{key})$ . (#11 & #12)
  - (a) The key is a new immediate successor – happens when  $\text{LCP}(\text{my.key}, \text{key})$  is longer than  $\text{LCP}(\text{s.key}, \text{key})$  among all my current successors (this means no “closest” successor was found) (#15);
  - (b) The key is a new successor replacing the current one – happens when  $\text{my.key} \sqsubset \text{key} \sqsubset \text{s.key}$  for some closest successor **s** (#18 & #19); or

---

```

1 On message <add, key, val, src>:
2   if key = my.key:
3     store <key, val> locally
4     send my contact details to src
5     return
6   cp ← LCP(key, my.key)
7   if my.key ⊆ key:
8     find s ∈ my.SSet with s.key ⊆ key and s.key is longest
9     if s ≠ ⊥: % we found a node
10      send <add, key, val, src> to s; return
11     find nxt ∈ my.SSet with
12       LEN(LCP(key, nxt.key)) > LEN(cp)
13     new ← NEWNODE(key, val)
14     if nxt = ⊥:
15       LNK_PARENT_CHILD(my.id, new)
16     else:
17       if key ⊆ nxt.key:
18         LNK_PARENT_CHILD(my.id, new)
19         LNK_PARENT_CHILD(new, nxt)
20       else:
21         nv ← NEWNODE(LCP(key, nxt.key), ⊥)
22         LNK_PARENT_CHILD(my.id, nv)
23         LNK_PARENT_CHILD(nv, new); LNK_PARENT_CHILD(nv, nxt)
24   else:
25     select p ∈ my.PSet with p.key ⊆ key and p.key is longest
26     if p = ⊥: % we did not find a node
27       select p ∈ my.PSet with LEN(LCP(key, p.key)) > LEN(cp)
28     else:
29       send <add, key, val, src> to p; return
30     new ← NEWNODE(key, val)
31     if key ⊆ my.key:
32       LNK_PARENT_CHILD(new, my.id)
33     else:
34       nv ← NEWNODE(cp, ⊥)
35       LNK_PARENT_CHILD(nv, my.id); LNK_PARENT_CHILD(nv, new)
36   return
37 LNK_PARENT_CHILD(parent, child):
38   if parent = my.id:
39     add child to my.SSet
40   else:
41     send <add_succ, child> to parent
42   if child = my.id:
43     add parent to my.PSet
44   else:
45     send <add_pred, parent> to child
46   return
47 On message <add_pred, pred>:
48   add pred to my.PSet
49   return
50 On message <add_succ, succ>:
51   add succ to my.SSet
52   return

```

---

**Figure 3.** Distributed Algorithm for Constructing DPGs.

- (c) A new virtual key node to be created as a successor. This happens when the key and s.key of the closest successor, shares a common prefix cp that is  $\text{my.key} \sqsubseteq \text{cp}$  (#21–#23).

If none of the above, this key is either to be inserted at some predecessor or a new root. There are three steps:

1. Forward the request to a predecessor that holds the longest key that is a prefix of the new key. (#25 & #29)
2. If none are found, the “closest” predecessor p is determined as the one with the longest  $\text{LCP}(p.\text{key}, \text{key})$ . If this predecessor is found, forward the request to it. (#27 & #29)
3. If no such predecessor was found, then a new node is created and inserted at two possible locations:

- (a) key is a new immediate predecessor – which applies when  $\text{key} \sqsubseteq \text{my.key}$  (#32); or

- (b) A virtual key node is created, which becomes the predecessor of this node and the one for key (#34 & #35).

On the algorithm in Figure 3, the  $\text{LNK\_PARENT\_CHILD}(p, c)$  is a support function that performs the necessary updating of the PSet and SSet of both node predecessor p and successor c. This function sends a message to p or c to perform the update, if necessary. The add message format consists of the key, val and the src (the identity of the requesting process).

Employing DPGs for the purpose of implementing object lookup operations require addressing several other issues:

1. *Peer-to-Peer* – The  $\text{NEWNODE}()$  function creates a logical node and initialises it with the key and val. A naïve approach is to employ a DHT to locate a peer (in the case of a one-to-one mapping

between logical nodes of the DPG and peers). It is also possible to reuse known peers, allowing assignment of multiple logical nodes per peer.

2. *Concurrent Insert Requests* – A simple mutual exclusion algorithm is employed to lock logical nodes that will be modified as a result of a key insertion. Deadlock freedom in this method is ensured by locking nodes using the arcs as the direction. Since DPGs are acyclic, deadlocks are not possible.
3. *Replication Strategy* – For fault tolerance, a strategy using a fixed level of replication  $k$  is outlined below: (a) `NEWNODE()` function instantiates  $k$  peers via the underlying DHT mechanism, associating all  $k$  peers with the newly-created node; and (b) the `LNK_PARENT_CHILD()` function updates replicated copies of the node, not just one. The key is to use  $k$  versions of `PSet` and `SSet` on each node, updated by the replica-aware `LNK_PARENT_CHILD()` function. Periodic heartbeats are employed to detect failures and a replica loss is handled by identifying a replacement and restoring a copy of that node on the new peer.

### 3 Basic and Advanced Query Capability with DPGs

We present how our DPG mechanism is able to handle basic and advanced queries required for typical resource and service discovery mechanisms.

#### 3.1 Exact and Prefix-based Partial Lookups

An exact match lookup is performed by a similar algorithm like the construction one we have presented. First, the requesting process sends a `lookup` request to any peer that holds a logical node of the graph. A comparison is made between the node key and the search key, determining the appropriate node (if applicable) to forward this request to. The request is forwarded to a predecessor with the best matching key, reducing in less hops. When a node with the matching key is found, it sends a reply message containing the value is sent to the requesting peer.

Partial matching with prefixes (e.g., “*find all resources that have the gtk library*”) is performed by first locating the node that holds the key equal to the prefix. Starting at that node, a traversal of all the successors is performed by that node forwarding the lookup requests to all successors. This process takes place in parallel, allowing all subtrees rooted at that node to be explored. Each node that holds a physical key-value will send back a reply to the requesting process.

#### 3.2 Range Searches with Numeric Values

It is not uncommon for attributes of resources and services to contain numeric values. For example, a description of a computing resource may include the clock speed of the processor (e.g., 3 GHz, 2.8 GHz, etc.), or the amount of available storage (e.g., 100 GB, 500 GB, etc.). A service may specify the version number of a particular installed library (e.g., gtk-1.4, gtk-2.0, etc.). If the attribute is stored as a string without any encoding, a string-based prefix search can lead to many extraneous request messages.

To address this, we propose an encoding strategy for the numeric attributes. These attributes are typically specified as:  $\langle \text{prefix} \rangle = \langle \text{number} \rangle$ . The key is to express numeric attributes into lexicographically-ordered strings. The encoding function must satisfy the property: whenever two numbers  $n_1$  and  $n_2$  are encoded into  $s_1$  and  $s_2$ , respectively, if  $n_1 < n_2$  then  $s_1$  is lexicographically smaller than  $s_2$ . This encoding requires two user inputs: (a) the upper and lower bounds for the numeric attribute; and (b) the number of intervals to encode this attribute into. If necessary, multi-character strings may be used for when there is a large number of intervals. Two examples of encoding into single-character strings are presented in Figure 4.

During an insert operation, keys with numeric attributes are first encoded, followed by normal operation of the DPG algorithm. To perform a range query, the upper and lower bounds for this query is encoded. The search is made to find all key-value pairs with keys that fall within the encoded upper and lower bounds. For example, if some library is requested with the range requirement of 1.3 to 1.5, the request message will contain encoded keys `c` and `e` as the lower and upper bounds. During propagation of the request, only those successor nodes holding keys that fall within these two bounds need to be explored.

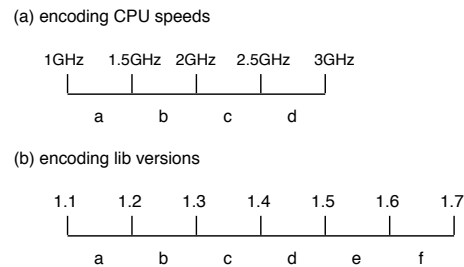


**Figure 4.** Encoding examples: (a) CPU speeds from 1GHz to 3GHz are encoded into 4 intervals, each interval is mapped to a character ‘a’ through ‘d’, respectively; (b) Version numbers are similarly mapped into 6 intervals.

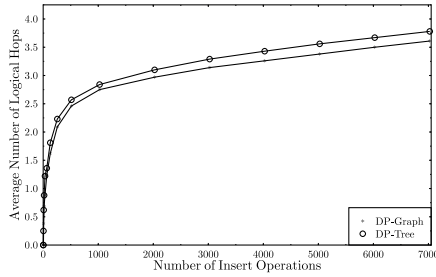


**Figure 5.** Average hops for inserting random alphabetic strings.

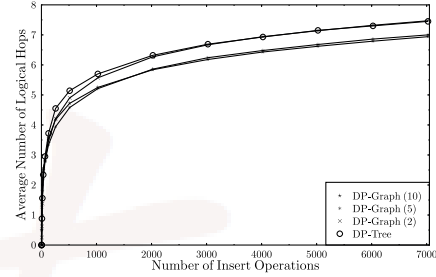


**Figure 6.** Average hops for inserting synthetic scientific data names.

### 3.3 Multiple Criteria Queries

The strategy is to employ a separate DPG for each attribute that can be used for queries. During a service/resource publication, the DPG for each attribute is updated. For example, when announcing the presence of a resource with 3 attributes: (a) `cputype=PowerPC G5`; (b) `RAM=2048GB`; and (c) `disk=500GB`, three insert operations are performed: one for each DPG assigned for the three attributes. When conducting multi-criteria queries, independent requests are issued to each of the DPGs responsible. An intersection operation is performed by the requesting process on the results to find the service or resource that matches all the specified conditions.

## 4 Simulation Results

We evaluate our DPG algorithm through a simulation. One implementation issue was the length of the vectors for SSet and PSet. The SSet has to be large enough to accommodate storage for all immediate successors. If we limit PSet to hold at most one predecessor, the algorithm reduces to a DPT. For the other cases, we considered sizes of 2, 5, and 10. This value determined how many predecessors will be “remembered” by a logical node. Since the maximum depth of the graphs we generated did not exceed 12, we found that there was no need to explore values higher than 10. We refer to variants of the DPGs for each of these as: DPG(1), DPG(5), DPG(10), respectively. On all the tests, we found no difference between DPT and DPG(2). And we found negligible differences in the average hops for DPG(5) and DPG(10). DPT results are from DPG(1) while DPG results refer to DPG(10), unless specified otherwise.

We used this simulation to evaluate the average number of logical hops under four different input sets

of key/value pairs. All these tests evaluate exact match queries highlighting how DPGs improve over DPTs. These are: (a) random strings of 20 alphabetic characters each (upper and lowercase characters); (b) randomly-generated synthetic data names typically used in scientific applications; (c) a list of 7325 full pathnames from a typical Unix installation; and (d) a list of synthetic segmented strings.

For each of the four data sets, the measured average number of hops for both DPT and DPG. The results are presented in Figure 5–8, respectively. Each graph shows the result for 7000 insert operations. For Figure 5 and Figure 6, each insert operation used a randomly-generated string, with small likelihoods of repeated keys. Each insert (add) request was sent to a randomly-selected node on the graph. The results of lookup performance were identical to the insert tests.

Our results show that the number of hops is highly dependent on the lexicographical structure of the input keys. The DPG performs slightly better than DPT on the random strings test data set. We found that the resulting graph for this input set has no more than depth of 3, since the common prefixes are organised around the first and second characters. In the Unix pathnames test, we found that DPG performed up to 20% better (in terms of number of hops). For Figure 6, we used artificial data filenames used in typical scientific applications, of the form: `<prefix>-<number>-<number>`.

In Figure 8, we used segmented strings as keys. Each segment is a substring (length 1–3), and 2 distinct substrings are created for each of the 12 segments. This resulted in  $2^{12}$  distinct keys, yielding a DPG of depth 12. Since insert operations can be started on a node that has no successors, it was necessary to route the request to the root, and then route it down another path to insert a key. The results demonstrate a reduction of up to 30% of routing hops compared to DPT.

In Figure 9, we highlight the difference between the

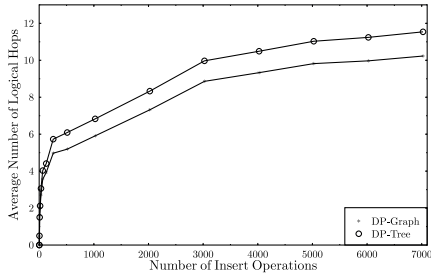


**Figure 7.** Average hops for inserting 7325 full Unix pathnames.

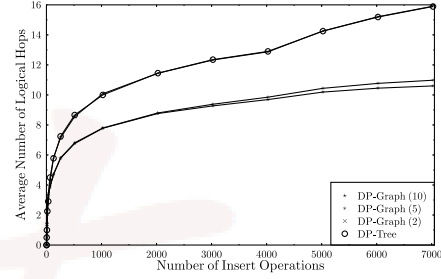


**Figure 8.** Average hops for inserting strings from a randomly-generated 12-segment substrings.

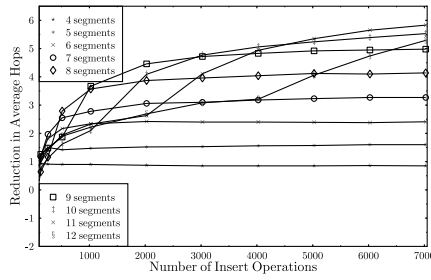


**Figure 9.** Differences in number of average hops between DPT and DPG while varying the number of segments in the segmented strings test.

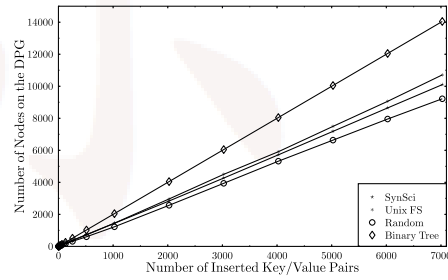


**Figure 10.** Size of the DPG is linear to the number of inserted key/value pairs.

average hops obtained through the DPT and that obtained from DPG runs. We varied the number of segments from 4 to 12. For the 12-segment case, we used the same data in Figure 8. Up to a difference of 5 hops was obtainable. Finally, in Figure 10 we show the scalability of the DPG, ensuring its size is linear to the number of keys, a property inherent to DPT. And since DPGs do not add additional nodes over that of the DPT, the same property applies.

## 5 Related Work

Our work is inspired heavily from the work of Caron, Desprez, and Tedeschi [3] concerning the Dynamic Prefix Tree (DPT/DLPT). After studying their approach, we found that the efficiency of lookups with DPT/DLPT can be improved. The main difference of DPGs from DPT is the way the arcs/links between nodes are manipulated. In our approach, previously existing arcs/links are not deleted as new nodes are inserted into the graph. In DPTs, these links are re-

moved to maintain the tree-structure, i.e., at most 1 predecessor per node. By allowing those links to be left on the graph, routing to some nodes require less hops, resulting in an improvement in performance.

The analysis (from Skip Graphs to DPT) presented on Table 1 is credited to Caron, Desprez & Tedeschi (2006) [3]. We added the last column to show how our DPG approach compares with others. For worst-case complexity, our approach does not improve over DPT, except in the messages passed during a range query ( $T_{max}$  is the max depth of the DPG,  $m$  is the number of nodes on the tree/graph, and  $m_R \leq m$  is the number of nodes on the subtrees selected during a range query  $R$ ). For the other results,  $n$  is the number of peers,  $r$  is the number of resources,  $D$  is the maximum size of keys,  $\Pi$  is size of the whole key-space, and  $\Pi_R$  is the size of the interval of a range query  $R$ . However, a future analysis of the average performance complexity may provide insights to the simulation results we obtained. For detailed discussion and analysis of the other approaches, we refer the reader to [3].

Efficient and scalable key-based lookups have been

| Operation      | Skip Graphs   | PHT                | P-Grid [4]    | DLPT (DPT)   | DPG          |
|----------------|---------------|--------------------|---------------|--------------|--------------|
| Insertion      | $O(\log n)$   | $O(D)$             | $O(\log \Pi)$ | $O(T_{max})$ | $O(T_{max})$ |
| Lookup         | $O(\log n)$   | $O(\log D \log N)$ | $O(\log \Pi)$ | $O(T_{max})$ | $O(T_{max})$ |
| Range messages | $O(r \log n)$ | $O(o)$             | $O(\Pi_R)$    | $O(m)$       | $O(m_R)$     |
| Range time     | $O(\log n)$   | $O(D)$             | $O(\log \Pi)$ | $O(T_{max})$ | $O(T_{max})$ |

**Table 1.** DPG compared to other approaches. We acknowledge [3] for the results reported here.

the subject of peer-to-peer research. There are many previous work on tori and other structures. DPGs, however, belongs to the family of trie-structured approaches. Organising keys in the form of a trie has been shown to improve lookup times by its hierarchical structure. Approaches that are similar to DPGs are Skip Graphs [1], Prefix Hash Tree [8], and NodeWiz [2]. The main difference lies in our approach of managing the connections between nodes, allowing connections to skip to successors further down the graph. This links provide additional routing options, reducing the number of hops during lookups and insert operations.

## 6 Conclusions

We have presented a scalable, prefix-based lookup mechanism suitable for service/resource discovery on large-scale grids. This mechanism supports exact match searches and more advanced range and multi-criteria queries. At the core of this mechanism is the DPG, a trie-based data structure. The DPG is an improved variant of the Dynamic Prefix Tree proposed by Caron, Desprez & Tedeschi [3]. By relaxing the “tree constraint” of DPTs, we achieved a reduction in routing hops as the graph is traversed during insert and lookup operations. We present a distributed algorithm to construct DPGs and show how replication is used for fault tolerance. Simulation results show that for a variety of input keys, DPGs perform better than the DPTs, in terms of the average number of hops to reach a node (by up to 30%).

We also propose an encoding technique to allow prefix-based lookup mechanisms to handle range queries more efficiently. By “compressing” the numeric range for these attributes into intervals and assigning a character string to each interval, it becomes possible to reduce the number of successors to consider during such queries. A detailed analysis of this encoding scheme will be considered in the future.

## References

- [1] James Aspnes and Gauri Shah. “Skip Graphs”. In *Proc. of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 384–393. ACM Press, January 2003.
- [2] Sujoy Basu, Sujata Banerjee, Puneet Sharma, and Sung-Ju Lee. “NodeWiz: Peer-to-Peer Resource Discovery for Grids”. In *Proc. of the 2005 IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2005)*, pages 213–220. IEEE Computer Society, 2005.
- [3] Eddy Caron, Frédéric Desprez, and Cédric Tedeschi. “A Dynamic Prefix Tree for Service Discovery within Large Scale Grids”. In *Proc. of the 6th IEEE International Conference on Peer-to-Peer Computing*, Cambridge, UK, 2006. IEEE Computer Society.
- [4] Anwitaman Datta, Manfred Hauswirth, Roman Schmidt, Renault John, and Karl Aberer. “Range Queries in Trie-Structured Overlays”. In *Proc. of the 5th IEEE International Conference on Peer-to-Peer Computing*. IEEE Computer Society, 2005.
- [5] Ian Foster. “Globus Toolkit Version 4: Software for Service-Oriented Systems”. In *Proc. of the IFIP International Conference on Network and Parallel Computing*, volume 3779 of *Lecture Notes in Computer Science*, pages 2–13. Springer-Verlag, 2005.
- [6] Ian Foster and Adriana Iamnitchi. “On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing”. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS 2003)*, volume 2735 of *Lecture Notes in Computer Science*, pages 118–128. Springer-Verlag, 2003.
- [7] Jaka Močnik, Marko Novak, Gregor Pipan, and Piotr Karwaczynski. “A Discovery Service for Very Large, Dynamic Grids”. In *Proc. of the Second IEEE International Conference on e-Science and Grid Computing (e-Science 2006)*, pages 213–220. IEEE Computer Society, 2006.
- [8] Sriram Ramabhadran, Sylvia Ratnasamy, Joseph M. Hellerstein, and Scott Shenker. “Prefix Hash Tree: An Indexing Data Structure over Distributed Hash Tables”. In *PODC 2003: Proc. of the 23rd ACM Symposium on Principles of Distributed Computing*, page 368. ACM Press, 2004.
- [9] Antony Rowstron and Peter Druschel. “Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems”. In *Proc. of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Heidelberg, Germany, November 2001.
- [10] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. “Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications”. *IEEE/ACM Transactions on Networking*, 11(1):17–32, February 2003.
- [11] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. “Tapestry: A Resilient Global-Scale Overlay for Service Deployment”. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.