CS33: Data Structures 20 marks Other CIE component Term: Nov 2023- March 2024

Parallel Name Lookup for Named Data Networking

TECHNICAL CODETHAN REPORT

Submitted By

YASH SURESH GAVAS	**1MS22CS163**
SHARANYA M S	**1MS22CS130**
RASHMITHA M	**1MS22CS113**

Under the guidance of
Dr. J.Sangeetha
Associate Professor

M. S. Ramaiah Institute of Technology
(Autonomous Institute, Affiliated to VTU)
Feb 2024

CERTIFICATE

This is to certify that **YASH SURESH GAVAS: 1MS22CS163**, **SHARANYA MS : 1MS22CS130,RASHMITHA M: 1MS22CS113** have completed the **Parallel Name Lookup for Named Data Networking**

Technical Codethan as part of the course **CS33: Data Structures 20 marks Other CIE component**.

DECLARATION

We hereby, declare that the entire content embodied in this B.E. 3rd Semester Codethan report carried out by us for the course **CS33: Data Structures 20 marks Other CIE component**

are not copied

Name : YASH SURESH GAVAS

USN : 1MS22CS163

Name : SHARANYA M S

USN : 1MS22CS130

Name : RASHMITHA M

USN : 1MS22CS113

Evaluation Sheet

Sl. No	USN	Name	Research Content understanding and Coding (10)	Demo & Report submission (10)	Total Marks (20)
1.	1MS22CS163	YASH SURESH GAVAS			
2.	1MS22CS130	SHARANYA M S			
3.	1MS22CS113	RASHMITHA M			

Evaluated By

Signature:
Name: Dr. J Sangeetha
Designation: Associate Professor
Department: Computer Science & Engineering, RIT

TABLE OF CONTENTS

1.Abstract :

The given research introduces a paradigm-shifting approach to address the intricate challenge of name-based route lookup in Named Data Networking (NDN). As NDN names exhibit hierarchical characteristics and variable, unbounded lengths, surpassing those of IPv4/6 addresses, expediting the name lookup process becomes a formidable task. Our novel contribution, Parallel Name Lookup (PNL), introduces a parallel architecture leveraging hardware parallelism to achieve substantial speedup in lookup operations while maintaining low and controllable memory redundancy. At the heart of PNL lies an innovative allocation algorithm that adeptly maps the logically tree-based structure of NDN names to physically parallel modules, minimizing computational complexity. Through a rigorous performance evaluation, we showcase PNL's exceptional acceleration capabilities, effectively addressing the challenges posed by lengthy and hierarchical names within the NDN framework. Furthermore, our exploration into integrating prior probability knowledge into PNL underscores the potential for significant speedup improvements, highlighting the adaptability and versatility of this proposed parallel architecture. In conclusion, PNL emerges as a robust and efficient solution, laying the groundwork for advancing content-centric networking and overcoming the intricacies of NDN name lookup. In this paper, we propose, analyse and implement Parallel Name Lookup (PNL), a fast name lookup architecture using parallel memory resources to obtain high speedup for NDN. PNL is based on an analogue of IP prefix tree, called Name Prefix Tree (NPT) where each node represents a set of components. The key observation is that millions of nodes of NPT can be selectively grouped and allocated into parallel memories. When any input name comes, it will traverse and be matched in some memories so that when multiple input names come concurrently, almost all memories will be visited and work simultaneously, achieving a high overall speedup. Despite the novelty of NDN, NDN operations can be grounded in current practice, routing and forwarding of NDN are semantically the same as that of IP network. What differs is that, every piece of content in NDN is assigned a name, and NDN routes and forwards packets by these names, rather than IP addresses. Indeed, the allocation requires some nodes to be duplicated, causing potential memory explosion. But we show that NPL fully exploits parallelism with strictly bounded redundancy. The application of PNL is beyond NDN name lookup as the methods used are independent to how name or component is written. We develop NPL to be a generic architecture to accelerate name lookup as long as such name causes any of the challenges

2.Introduction :

Named Data Networking (NDN) introduces a revolutionary content-centric network architecture that shifts the focus from "where" information is located to "what" information is needed. In this model, every piece of content is identified by a name instead of traditional IP addresses for devices in an IP network. However, this transition presents several challenges. First, the variable lengths of NDN names, unlike fixed-length IP addresses, lead to unbounded processing times and low throughput for name lookup, especially when dealing with arbitrarily long names. Second, NDN's hierarchical and coarser granularity structure poses inefficiencies in prefix aggregation, deviating from the classless IP addresses' model. This distinction hampers the effectiveness of traditional prefix aggregation methods in NDN name lookup.

Another significant challenge arises from the higher update rate experienced by NDN compared to IP routers' Forwarding Information Base (FIB). The dynamic nature of NDN name lookup involves frequent updates triggered by content storage and replacement, requiring a mechanism that supports fast insertion and deletion with minimal overhead. As NDN aims to store this information alongside FIB in routers, the name lookup process must be optimized to accommodate the dynamic and evolving nature of the content-centric networking paradigm. Addressing these challenges is paramount to realizing the full potential of NDN and ensuring efficient and scalable name lookup in this innovative network architecture. PNL that performs accelerated NDN name lookup and give worst case analysis. The core of PNL is an allocation algorithm that allocates the logically tree-based structure to parallel physical modules with low and controlled storage redundancy. To handle highly frequent update, the allocation algorithm is designed to be performed with small computational complexity. We implement and test the prototype, the evaluation results indicate that PNL could achieve remarkable average speedup for name lookup. The speedup significantly augments as the number of modules k increases.

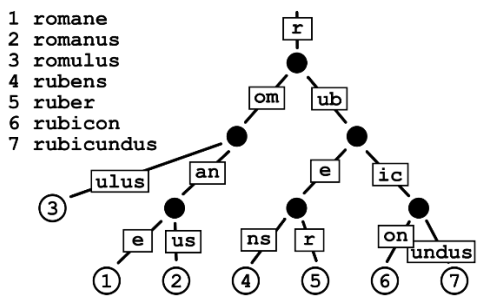
3. Literature Survey :

An NDN name is hierarchically structured and composed of explicitly delimited components, while the delimiters, usually slashes or dots, are not part of the name. The hierarchical structure, like that of IP address, enables name aggregation and allows fast name lookup by longest prefix match, and aggregation in turn is essential to the scalability of routing and forwarding systems. Though NDN routes and forwards packets by names, names are opaque to the network, which means routers are not aware of what a name means, but only know the boundaries between components. The naming strategy may include designing name structure, name discovery and namespace navigation schemes, and is not limited to one solution but now an open research question. For the purpose of early exploring the name-based route lookup mechanism before the naming specification finally goes to standard, in this paper, we temporarily use hierarchically reversed domains as NDN names, and apply our proposed PNL scheme instead. For example, maps.google.com is hierarchically re□versed to com/google/maps, and com, google, maps are three components of the name.

Name Prefix Tree (NPT)

NDN names are composed of explicitly delimited components, hence they can be represented by NPT. NPT is of component granularity, rather than character or bit granularity, since the longest name prefix lookup of NDN names can only match a complete component at once, i.e., no match happens in the middle of a component. Each edge of NPT stands for a name component and each node stands for a lookup state1. Name prefix lookups always begin at the root. When the lookup for a given name starts, it firstly checks if first component matches one of the edges originated from the root, i.e., the level-1 edge. If so, the transfer condition holds and then the lookup state transfers from the root to the pointed level□2 node. The subsequent lookup process proceeds iteratively. When the transfer condition fails to hold or the lookup state reaches one of the leaf nodes, the lookup process terminates and outputs the ports' number that the last state corresponds to. For example given name /com/google/maps, the first component com matches a level-1 edge and the lookup state transfers from root (state 1) to state 2; the level-2 transfer condition also holds and the lookup state continues to transfer to state 5; the third name component maps does not satisfy the level-3 transfer condition and the lookup process terminates, with the lookup state remaining to be state 5. NPT possesses faster lookup speed and less storage overhead when compared to the traditional character tree. However the lookup process still needs to search from the root to lower level nodes in serial, without taking full advantage of the router hardware parallelism.

Example of npt:



4. Abstract Data Type :

1. Prefix Trie (NPT) ADT:

Objects:

A finite set of nodes forming a hierarchical tree structure.

Each node has a character, a pointer to its children (an array of nodes), and an optional associated name.

Functions:

create_npt() ::= Creates and returns an empty prefix trie.

insert_npt(Node, name) ::= Inserts a name into the prefix trie.

lookup_npt(Node, name) ::= Looks up a name in the prefix trie.

delete_npt(Node, name) ::=Deletes a name from the prefix trie.

print_npt(Node) ::= Prints the entire prefix trie.

2. Hash Table ADT :

Objects:

An array to store access probabilities for each character.

Functions:

create_hash_table() ::=Creates and returns an empty hash table.

update_probability(HashTable, character)::= Updates the access probability for a character.

print_probabilities(HashTable) ::=Prints the access probabilities.

calculate_average_probability(HashTable)::= Calculates and returns the average access probability.

PSEUDOCODE:

Prefix Trie(NPT) Functions :

//create_npt():

```
root = allocate_memory_for_node()
root.character = NULL
root.children = allocate_memory_for_children_array()
for each child in root.children:
  child = NULL
return root
```

//insert_npt(root, name):

```
current = root
for each character in name:
  if current has no child with that character:
    create new node with character
    add the new node as a child of current
  move to the child node
set current's name to name
```

//lookup_npt(root, name):

```
current = root
for each character in name:
  if current has no child with that character:
    return 0  // Name not found
  move to the child node
return 1 if current has a name, else 0
```

//delete_npt(root, name):

```
current = root
parent = NULL
for each character in name:
  index = get_index_from_character(character)
  if current.children[index] is NULL:
    display_error("Node not found for deletion.")
    return
  parent = current
  current = current.children[index]
if current has children:
  free(current.name)
  current.name = NULL
else:
  free(current.name)
  free(current)
  parent.children[index] = NULL
display_message("Node deleted successfully.")
```

//print_npt(root):

```
recursively_print(root, 0)
recursively_print(node, level):
for each child in node.children:
  if child is not NULL:
    print(level * 2 spaces + child.character)
    if child.name is not NULL:
      print(" (" + child.name + ")")
    recursively_print(child, level + 1)
```

Hash Table Function:

```
//create_hash_table():
  ht = allocate_memory_for_hash_table()
  ht.access_probabilities = allocate_memory_for_probabilities_array()
  for each probability in ht.access_probabilities:
    probability = 0.5
  return ht

//update_probability(ht, character):
  index = get_index_from_character(character)
  ht.access_probabilities[index] += 0.1

//print_probabilities(ht):
  for each probability in ht.access_probabilities:
    print(probability)

//calculate_average_probability(ht):
  sum = 0
  for each probability in ht.access_probabilities:
    sum += probability
  average = sum / ALPHABET_SIZE
  return average
```

Main Function:

```
root = create_npt()
ht = create_hash_table()
while true:
  display_menu()
  choice = get_user_choice()
  switch choice:
    case 1:name = get_user_input("Enter name to insert: ")
              insert_npt(root, name)
    case 2:name = get_user_input("Enter name to lookup: ")
             result = lookup_npt(root, name)
             display_result(result)
              if result == 1:
              update_probability(ht, last_character_of(name))
    case 3:name = get_user_input("Enter name to delete: ")
             delete_npt(root, name)
    case 4: print_npt(root)
    case 5: print_probabilities(ht)
    case 6:average_probability = calculate_average_probability(ht)
            display_average_probability(average_probability)
    case 7: save_npt_to_file(root, "npt_output.txt")
    case 8:
      free_npt(root)
      free_hash_table(ht)
      exit_program()
    default:
      display_error("Invalid choice. Please try again.")
```

IMPLEMENTATION :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define ALPHABET_SIZE 26

struct npt_node {
    char *name;
    struct npt_node **children;
};

struct hash_table {
    float *access_probabilities;
};

int char_to_index(char c) {
    if (c >= 'a' && c <= 'z') {
        return c - 'a';
    } else if (c >= 'A' && c <= 'Z') {
        return c - 'A';
    }
    return -1;
}

struct npt_node *npt_create() {
    struct npt_node *root = (struct npt_node *)malloc(sizeof(struct
npt_node));
    if (root) {
        root->name = NULL;
        root->children = (struct npt_node **)malloc(ALPHABET_SIZE *
sizeof(struct npt_node *));
        for (int i = 0; i < ALPHABET_SIZE; i++) {
            root->children[i] = NULL;
        }
    }
    return root;
}

struct hash_table *create_hash_table() {
    struct hash_table *ht = (struct hash_table *)malloc(sizeof(struct
hash_table));
    if (ht) {
```

```
        ht->access_probabilities = (float *)malloc(ALPHABET_SIZE *
sizeof(float));
        for (int i = 0; i < ALPHABET_SIZE; i++) {
            ht->access_probabilities[i] = 0.5;
        }
    }
    return ht;
}

void free_npt(struct npt_node *root) {
    if (root) {
        free(root->name);
        for (int i = 0; i < ALPHABET_SIZE; i++) {
            free_npt(root->children[i]);
        }
        free(root->children);
        free(root);
    }
}
void free_hash_table(struct hash_table *ht) {
    if (ht) {
        free(ht->access_probabilities);
        free(ht);
    }
}
void npt_insert(struct npt_node *root, const char *name) {
    struct npt_node *current = root;
    for (int i = 0; i < strlen(name); i++) {
        char lowercase = tolower(name[i]);
        int index = char_to_index(lowercase);
        if (index == -1) {
            printf("Invalid character in name: %c\n", name[i]);
            return;
        }
        if (!current->children[index]) {
            current->children[index] = npt_create();
        }
        current = current->children[index];
    }
    current->name = (char *)malloc((strlen(name) + 1) * sizeof(char));
    strcpy(current->name, name);
}

int npt_lookup(struct npt_node *root, const char *name, struct hash_table
*ht) {
    struct npt_node *current = root;
```

```
    for (int i = 0; i < strlen(name); i++) {
        char lowercase = tolower(name[i]);
        int index = char_to_index(lowercase);
        if (index == -1 || !current->children[index]) {
            return 0;
        }
        current = current->children[index];
        if (ht) {
            ht->access_probabilities[index] += 0.1;
        }
    }
    return current->name != NULL;
}

void npt_delete(struct npt_node *root, const char *name) {
    struct npt_node *current = root;
    struct npt_node *parent = NULL;
    int index;

    for (int i = 0; i < strlen(name); i++) {
        char lowercase = tolower(name[i]);
        index = char_to_index(lowercase);
        if (index == -1 || !current->children[index]) {
            printf("Node '%s' not found for deletion.\n", name);
            return;
        }
        parent = current;
        current = current->children[index];
    }

    int hasChildren = 0;
    for (int i = 0; i < ALPHABET_SIZE; i++) {
        if (current->children[i] != NULL) {
            hasChildren = 1;
            break;
        }
    }

    if (hasChildren) {
        free(current->name);
        current->name = NULL;
    } else {
        free(current->name);
        free(current);
        parent->children[index] = NULL;
    }
```

```
    printf("Node '%s' deleted successfully.\n", name);
}

void print_npt_recursive(struct npt_node *node, int level) {
    if (node) {
        for (int i = 0; i < ALPHABET_SIZE; i++) {
            if (node->children[i]) {
                printf("%*s%c", level * 2, "", i + 'a');
                if (node->children[i]->name) {
                    printf(" (%s)\n", node->children[i]->name);
                } else {
                    printf("\n");
                }
                print_npt_recursive(node->children[i], level + 1);
            }
        }
    }
}

void print_npt(struct npt_node *root) {
    printf("\nName Prefix Tree (NPT):\n");
    print_npt_recursive(root, 0);
}

void print_access_probabilities(struct hash_table *ht) {
    printf("\nAccess Probabilities:\n");
    for (int i = 0; i < ALPHABET_SIZE; i++) {
        printf("%c: %.2f\n", i + 'a', ht->access_probabilities[i]);
    }
}

void print_average_access_probability(struct hash_table *ht) {
    float sum = 0;
    for (int i = 0; i < ALPHABET_SIZE; i++) {
        sum += ht->access_probabilities[i];
    }
    float average = sum / ALPHABET_SIZE;
    printf("\nAverage Access Probability: %.2f\n", average);
}

void load_npt_from_file(struct npt_node *root, const char *input_filename) {
    FILE *input_file = fopen(input_filename, "r");
    if (input_file) {
        char buffer[100];
        while (fscanf(input_file, "%s", buffer) != EOF) {
```

```
            npt_insert(root, buffer);
        }
        fclose(input_file);
        printf("NPT loaded successfully.\n");
    } else {
        perror("Error opening file for NPT loading");
    }
}


void save_npt_to_file(struct npt_node *root, const char *output_filename) {
    FILE *output_file = fopen(output_filename, "w");
    if (output_file) {
        print_npt_recursive(root, 0);
        fclose(output_file);
    } else {
        printf("Error opening file for NPT saving.\n");
    }
}

int main() {
    struct npt_node *root = npt_create();
    struct hash_table *ht = create_hash_table();

    char input_name[100];
    int choice;
    const char *output_filename = "npt_output.txt";

    load_npt_from_file(root, output_filename);

    while (1) {
        printf("\nMenu:\n");
        printf("1. Insert Name\n");
        printf("2. Lookup Name\n");
        printf("3. Delete Name\n");
        printf("4. Print NPT\n");
        printf("5. Print Access Probabilities\n");
        printf("6. Print Average Access Probability\n");
        printf("7. Save NPT to File\n");
        printf("8. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter name to insert: ");
```

```
                scanf("%s", input_name);
                npt_insert(root, input_name);
                break;

            case 2:
                printf("Enter name to lookup: ");
                scanf("%s", input_name);
                printf("Lookup result for '%s': %s\n", input_name,
npt_lookup(root, input_name, ht) ? "Found" : "Not Found");
                break;

            case 3:
                printf("Enter name to delete: ");
                scanf("%s", input_name);
                npt_delete(root, input_name);
                break;

            case 4:
                print_npt(root);
                break;

            case 5:
                print_access_probabilities(ht);
                break;

            case 6:
                print_average_access_probability(ht);
                break;

            case 7:
                save_npt_to_file(root, output_filename);
                break;

            case 8:
                free_npt(root);
                free_hash_table(ht);
                exit(0);

            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

RESULTS AND DISCUSSION:

Insert Name:

```
Menu:
1. Insert Name
2. Lookup Name
3. Delete Name
4. Print NPT
5. Print Access Probabilities
6. Print Average Access Probability
7. Save NPT to File
8. Exit
Enter your choice: 1
Enter name to insert: yash
```

User Input: Enter a name to insert.

Output: Inserts the specified name into the Prefix Trie (NPT). If the name contains invalid characters, it prints an error message.

Lookup Name:

```
Menu:
1. Insert Name
2. Lookup Name
3. Delete Name
4. Print NPT
5. Print Access Probabilities
6. Print Average Access Probability
7. Save NPT to File
8. Exit
Enter your choice: 2
Enter name to lookup: yash
Lookup result for 'yash': Found

Menu:
1. Insert Name
2. Lookup Name
3. Delete Name
4. Print NPT
5. Print Access Probabilities
6. Print Average Access Probability
7. Save NPT to File
8. Exit
Enter your choice: sharanya
Enter name to lookup: Lookup result for 'sharanya': Not Found
```

User Input: Enter a name to lookup.

Output: Searches for the specified name in the Prefix Trie (NPT). Prints "Found" if the name exists, "Not Found" otherwise.

Delete Name:

```
Menu:
1. Insert Name
2. Lookup Name
3. Delete Name
4. Print NPT
5. Print Access Probabilities
6. Print Average Access Probability
7. Save NPT to File
8. Exit
Enter your choice: 3
Enter name to delete: yash
Node 'yash' deleted successfully.
```

User Input: Enter a name to delete.

Output: Deletes the specified name from the Prefix Trie (NPT). If the name is not found, it prints an error message.

Print NPT:

```
Menu:
1. Insert Name
2. Lookup Name
3. Delete Name
4. Print NPT
5. Print Access Probabilities
6. Print Average Access Probability
7. Save NPT to File
8. Exit
Enter your choice: 4

Name Prefix Tree (NPT):
s
  h
    a
      r
        a
          n (sharan)
            y
              a (sharanya)
          t
            h (sharath)
y
  a
    s
      h (yash)
```

Output: Displays the hierarchical structure of the Name Prefix Tree (NPT), including characters and names stored at each level.

Print Access Probabilities:

```
Menu:
1. Insert Name
2. Lookup Name
3. Delete Name
4. Print NPT
5. Print Access Probabilities
6. Print Average Access Probability
7. Save NPT to File
8. Exit
Enter your choice: 5

Access Probabilities:
a: 0.60
b: 0.50
c: 0.50
d: 0.50
e: 0.50
f: 0.50
g: 0.50
h: 0.60
i: 0.50
j: 0.50
k: 0.50
l: 0.50
m: 0.50
n: 0.50
o: 0.50
p: 0.50
q: 0.50
r: 0.50
s: 0.60
t: 0.50
u: 0.50
v: 0.50
w: 0.50
x: 0.50
y: 0.60
z: 0.50
```

Output: Displays the access probabilities for each character in the alphabet. These probabilities are updated during the lookup operation.

Print Average Access Probability:

```
Menu:
1. Insert Name
2. Lookup Name
3. Delete Name
4. Print NPT
5. Print Access Probabilities
6. Print Average Access Probability
7. Save NPT to File
8. Exit
Enter your choice: 6

Average Access Probability: 0.52
```

Output: Calculates and prints the average access probability across all characters in the alphabet.

Save NPT to File:

```
Menu:
1. Insert Name
2. Lookup Name
3. Delete Name
4. Print NPT
5. Print Access Probabilities
6. Print Average Access Probability
7. Save NPT to File
8. Exit
Enter your choice: 7
r
  a
    s
      h
        m
          i
            t
              h
                a (rashmitha)
s
  h
    a
      r
        a
          n (sharan)
            y
              a (sharanya)
          t
            h (sharath)
y
  a
    s
      h (yash)
```

Output: Saves the current state of the Prefix Trie (NPT) to a file named "npt_output.txt".

Exit:

```
Menu:
1. Insert Name
2. Lookup Name
3. Delete Name
4. Print NPT
5. Print Access Probabilities
6. Print Average Access Probability
7. Save NPT to File
8. Exit
Enter your choice: 8
Exiting..
```

Output: Frees memory allocated for the Prefix Trie (NPT) and the Hash Table, then exits the program.

CONCLUSION :

NDN's hierarchical and variable-length names pose challenges in processing time, prefix aggregation, and handling frequent updates, making fast name lookup difficult.PNL introduces a parallel architecture, utilizing hardware parallelism to enhance name lookup speed. The core of PNL involves an allocation algorithm mapping logical tree-based structures to physically parallel modules.PNL aims to keep memory redundancy low and controllable, addressing concerns related to memory usage and efficiency in the name lookup process.The research evaluates PNL's performance, demonstrating a significant acceleration in the name lookup process compared to traditional methods.PNL's speedup can be further improved with prior knowledge of probability, indicating a potential avenue for optimization.

The research contributes a parallel architecture, PNL, as a solution to enhance the efficiency of name-based route lookup in NDN. The proposed approach addresses the unique challenges posed by NDN's naming structure and demonstrates notable improvements in the name lookup process, offering a promising direction for the advancement of NDN technologies.

REFERENCES :

Title :

Parallel Name Lookup for Named Data Networking.

Author :

Yi Wang, Huichen Dai, Junchen Jiang, Keqiang He, Wei Meng, Bin Liu Tsinghua National Laboratory for Information Science and Technology Department of Computer Science and Technology, Tsinghua University, 100084, China.

Year of publication :

2011