ID1217 Concurrent Programming

Programming Project – Concurrent Prefix Trees

First Name: Yi Heng

Last Name: Lee

Project Overview

A *prefix tree*, or a *trie*, is a tree data structure used to store and locate a collection of sequences, most commonly strings. Each node represents a character in a string.

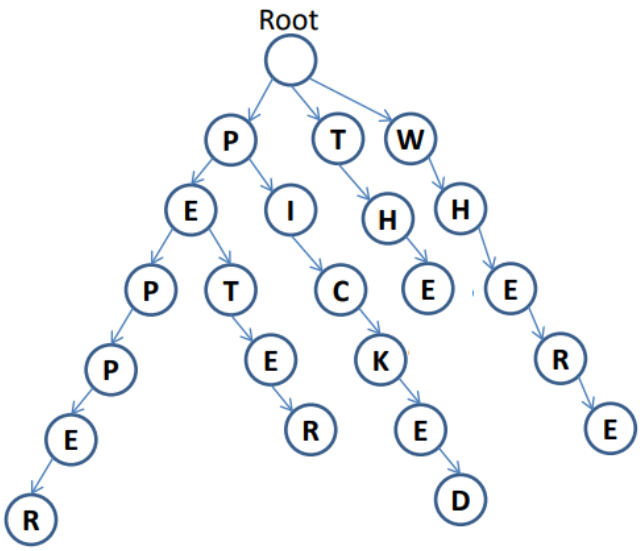

Figure 1: A visualization of a prefix tree, containing the strings "PEPPER", "PETER", "PICKED", "THE", and "WHERE"

Example usages of prefix trees include dictionary lookups or autocomplete. Such a data structure allows for operations that involve using common prefixes to be fast. For instance, given a particular prefix, returning all strings in the set that contain such a prefix is a fast operation. Sorting the strings in a prefix tree is also simple: by performing a pre-order traversal of the tree.

Parallelism Opportunities

Insertion, deletion and searching of strings involve traversing down references of nodes, akin to linked lists. Referring to the figure above, for instance, checking for the existence of the string "PEPPER" and the string "WHERE" involve traversing down two different sets of nodes.

If the user were to query for the existence of multiple strings, these operations could be performed in parallel, without affecting each other. Referring to the previous example, checking for the existence of "PEPPER" and the existence of "WHERE" involves traversing a different set of nodes.

The insertion of these two strings will also involve the creation of two different sets of nodes. Thus, we can exploit the structure of tries to perform operations concurrently.

Implementation

Nodes

Each node contains an ASCII character, and is represented as a class in C++, with references to its parent and children node. References to its children nodes are stored in an array. For instance, if we were to only allow alphabetical letters, 'a' can be mapped to index 0, 'b' to 1, 'z' to 26, 'A' to 27 etc.

Each node also has a boolean which indicates whether there is a string that ends at this particular character (without this boolean, adding the string "HOPE" will imply that the strings "H", "HO", and "HOP" are also present in the set).

More importantly, each node object also contains a mutex, which locks and unlocks the attributes of the node. Thus, only one thread can modify a node at one time, for instance to add children nodes.

Example: Insertion

Below is pseudocode for inserting a string into the prefix tree, in a sequential implementation.

```
void insert(string s) {
  Node cur = root;
  for (char c : s) {
    int index = toIndex(c);
    if (cur.children[index]== NULL) cur.children[index] = Node();
    cur = cur.children[index];
  }
  cur.isEndOfString = true;
}
```

Starting from the root, for each character in the string, we find the respective child node, creating it if it did not exist yet. Finally, when we are at the node representing the last character in the string, we set the boolean flag to true, to signify that there is a string in the set that ends here.

In the actual implementation, there are other attributes in a node that we manipulate, such as a pointer to the parent node and the number of children nodes that it has, however those are additional information to aid in implementing other pieces of logic, and the basic logic detailed above works.

In a concurrent implementation, however, we cannot directly have two threads modifying the same node, for instance to add a "P" child and a "Q" child node, since both would have to modify the array stored in this node.

To prevent race conditions when multiple threads attempt to visit this node to possibly modify it, we add a mutex.

```
void insert(string s) {
  Node cur = root;
  for (char c : s) {
    int index = toIndex(c);
    cur.mutex.lock()
    if (cur.children[index]== NULL) cur.children[index] = Node();
    cur.mutex.unlock()
    cur = cur.children[index];
  }
  cur.isEndOfString = true;
}
```

Working with multiple threads: A Readers-Writers Problem

The addition of mutexes allow multiple threads to call method simultaneously, since we are assured that the threads will modify attributed sequentially only after obtaining the mutex. However, we potentially have both readers and writers attempting to access our prefix tree.

The classic Readers-Writers Problem is a synchronization problem, where:

1) Readers and writers cannot access a shared object at the same time.
2) Multiple readers can access the shared object at the same time.
3) Only one writer can access the shared object at the same time.

In our case, we can relax (3) to allow multiple writers concurrent access, because multiple writers will not lead to race conditions due to our use of mutexes. This problem is similar to that of the Unisex Bathroom Problem, introduced in Homework 3 and 4. To solve this, we encapsulate this logic into another class, called *ReadersWriters*, which provides the methods *startRead(), endRead(), startWrite(), endWrite()*. This class ensures that the rules above are satisfied, and ensures fairness such that neither writers or readers are starved. We can cleanly include this as follows.

```
void insert(string s) {
  readersWriters.startWrite();
  // rest of logic
  readerWriters.endWrite();
}
```

```
bool contains(string s) {
  readersWriters.startRead();
  // rest of logic
  readerWriters.endRead();
}
```

Parallelism with OpenMP

Given a collection of strings, on average, the addition of each string should take roughly similar amounts of time as the work done is likely about the same, assuming the strings are random. This gives us an opportunity to easily parallelize it using OpenMP.

```
void insert(vector<string> strings) {
  #pragma omp parallel for
  for (int i = 0; i < strings.size(); i++) {
    insert(strings[i]);
  }
}
```

Performance Evaluation

To evaluate the performance, we compare the concurrent implementation, hereby referred to as ConcurrentTrie, against a sequential implementation, hereby referred to as SequentialTrie, as well as the most common implementation of a set - a hashset, or an *unordered_set* in C++.

A wordlist was obtained from the Internet, consisting of 1,000,000 common words in the English language. For the evaluations below, random subsets of this set of words are used.

Initialize

	SequentialTrie	ConcurrentTrie	Hashset
Average initialization time / s	$4.99 * 10^{-6}$	$5.90 * 10^{-6}$	$3.75 * 10^{-8}$

For initialization time, the average of **100,000** readings was taken. The ConcurrentTrie had the longest initialization time, even longer than the SequentialTrie likely due to the need to initialize the synchronization constructs. While the ConcurrentTrie was slower than the hashset, the initialization time was still in the order of microseconds, and such a difference is not likely to make a huge impact.

Insert

Inserting a single string

Average execution time / s	ConcurrentTrie	Hashset
30,000 strings	$4.71 * 10^{-5}$	$3.88 * 10^{-7}$
300,000 strings	$3.49 * 10^{-5}$	$4.28 * 10^{-7}$
500,000 strings	$3.25 * 10^{-5}$	$4.51 * 10^{-7}$

Strings were sampled at random and inserted into the sets sequentially. The insertion of more strings only improved the average time to insert a single string, because it was more likely that the nodes had already existed in the set.

While the ConcurrentTrie was slower than the hashset, the execution time was still in the order of microseconds, and such a difference is not likely to make a huge impact. For inserting multiple strings, we provide other methods.

Inserting a vector of strings

Average execution time / s	10,000 strings	50,000 strings	100,000 strings	300,000 strings
SequentialTrie	0.46	1.64	3.23	10.06
ConcurrentTrie (3 threads)	0.26	0.88	1.46	4.27
ConcurrentTrie (4 threads)	0.42	1.15	2.70	3.18
ConcurrentTrie (5 threads)	0.49	1.26	3.30	3.12
ConcurrentTrie (6 threads)	0.46	2.06	2.37	6.35
Hashset	0.06	0.63	0.43	2.37
ConcurrentTrie (insertAsync)	$8.8 * 10^{-5}$	$1.28 * 10^{-4}$	$7.6 * 10^{-5}$	$3.9 * 10^{-3}$

Strings were sampled at random, and the decision to compare against 3-6 threads were due to preliminary testing which showed that these numbers of threads generally yielded the best performance. From these readings, a few conclusions can be drawn.

1) For a large number of strings, parallelizing the work definitely leads to faster execution compared to the sequential trie. This is reasonable because the benefit obtained from parallelization would outweigh the overhead of thread creation and management.
2) For a large number of strings, the optimal number of threads depends. It is not always optimal to use more threads.
3) For insertion, it is hard to beat the hashset in terms of timing, for smaller number of strings. However, as the number of strings increase, the difference in execution time decreases.

For all intents and purposes, we can use the async version of *insert()*, since it is likely that in practice, we will perform insert and proceed with other operations, and not instantly read from the data structure. Even if we do attempt to read immediately, using *contains()*, the method will block to wait for the insertion to finish, so it would not make a difference since using *insert()* would have resulted in waiting as well.

Contains

Checking for existence of a single string

Average execution time / s	ConcurrentTrie	Hashset
100,000 strings	$6.94 * 10^{-7}$	$3.55 * 10^{-7}$
300,000 strings	$7.80 * 10^{-7}$	$6.54 * 10^{-7}$
600,000 strings	$9.79 * 10^{-7}$	$7.51 * 10^{-7}$

For each observation for **x** number of strings, **x/2** strings were sampled at random and inserted into the sets, before testing for **x** strings. The ConcurrentTrie is slower than the hashset, however the difference is extremely small as well.

Checking for existence of a vector of strings

Average execution time / s	SequentialTrie	ConcurrentTrie	Hashset
100,000 strings	0.039	0.041	0.011
300,000 strings	0.173	0.133	0.056
600,000 strings	0.521	0.273	0.154
1,000,000 strings	0.942	0.569	0.329

For each observation for **x** number of strings, **x/2** strings were sampled at random and inserted into the sets, before testing for **x** strings. As the number of strings increased, adding concurrency led to better execution timings. While the ConcurrentTrie is still slower than the hashset, the difference in execution time is less than half a second, even for 1 million strings.

The above ConcurrentTrie implementation uses 4 threads, which seems to yield the best performance on average.

Remove

Removing a single string

Average execution time / s	ConcurrentTrie	Hashset
100,000 strings	$3.15 * 10^{-6}$	$2.71 * 10^{-7}$
300,000 strings	$3.56 * 10^{-6}$	$2.99 * 10^{-7}$
600,000 strings	$4.55 * 10^{-6}$	$3.68 * 10^{-7}$

Strings were sampled at random and inserted into the sets, before being removed. While the ConcurrentTrie was slower than the hashset, the execution time was still in the order of microseconds, and such a difference is not likely to make a huge impact. For removing multiple strings, we provide other methods.

Removing a vector of strings

Average execution time / s	50,000 strings	100,000 strings	300,000 strings
SequentialTrie	0.1803	0.3166	1.031
ConcurrentTrie (4 threads)	0.0616	0.1122	0.395
ConcurrentTrie (6 threads)	0.0540	0.0973	0.344
ConcurrentTrie (8 threads)	0.0551	0.0929	0.306
ConcurrentTrie (10 threads)	0.0480	0.0957	0.342
ConcurrentTrie (12 threads)	0.0535	0.0999	0.305
Hashset	0.0145	0.0323	0.145
ConcurrentTrie (removeAsync)	$1.17 * 10^{-4}$	$8.9 * 10^{-5}$	$9.3 * 10^{-5}$

Strings were sampled at random, and the decision to compare against 4-12 threads were due to preliminary testing which showed that these numbers of threads generally yielded the best performance. From these readings, a few conclusions can be drawn.

1) Parallelizing the work definitely leads to faster execution compared to the sequential trie. The benefits obtained from parallelization outweighed the overhead of thread creation and management.
2) For a large number of strings, the optimal number of threads depends. It is not always optimal to use more threads.
3) For removal, it is hard to beat the hashset in terms of timing, for smaller number of strings. However, as the number of strings increase, the difference in execution time decreases.

For all intents and purposes, similar to *insert()*, we can use the async version of *remove()*, since it is likely that in practice, we will not instantly wish to read from the data structure. Even if we do attempt to read immediately, using *contains()*, the method will block to wait for the remove operation to finish, so it would not make a difference since using *remove()* would have resulted in waiting as well.

Retrieving strings in sorted order

Average execution time / s	ConcurrentTrie	Hashset
10,000 strings	0.0359	0.00722
50,000 strings	0.140	0.025
100,000 strings	0.250	0.045

For retrieving all the strings in sorted order, the execution time of ConcurrentTrie cannot beat the naïve implementation of retrieving all strings from the hashset, before sorting them.

Retrieving strings with a specific prefix, in sorted order

Average execution time / s	ConcurrentTrie	Hashset
Prefix with 2 letters	0.01322	0.053093
Prefix with 3 letters	0.00172	0.047188
Prefix with 4 letters	0.000829	0.052364

For a ConcurrentTrie, as well as a hashset, with **500,000** strings, I attempt to retrieve all strings with a given prefix, in sorted order. The execution time of ConcurrentTrie will constantly beat the hashset, especially given more letters as the prefix. This is because this query plays to the advantage of the ConcurrentTrie, given its structure, whereas the hashset will always have to perform a O(n) filter operation and an O(n log n) sorting operation.

Conclusion

The prefix tree is commonly used as a set data structure for strings. Its only advantage is when there are a large number of strings with the same prefix stored in the set, and when the query takes advantage of this structure, because the prefix tree is optimized for strings with the same prefix.

When using a prefix tree for normal operations such as insertion, search, and removal, we cannot beat the timings of a hashset, since a hashset's insertion, search and removal operations are amortized O(1). However, by the nature of the structure of a prefix tree, we can easily employ parallelism to speed up these operations. Since we ensure thread-safety using synchronization primitives, we can also implement "asynchronous" non-blocking versions of these methods, where we spawn another thread to complete the operation in the background for us instead.

For operations that utilize the advantage of prefix trees being space-optimized regarding strings with the same prefix, the performance of a prefix tree will outshine the hashset.

In conclusion, we can utilize parallelism to improve the execution times of common operations where it loses out to a hashmap, and retain the advantages of a prefix tree when it comes to strings with a common prefix. For instance, when storing a set of hyperlinks, all of which start with "https://www.", since this prefix is sufficiently long, there could be advantages in using a prefix tree instead.

Regarding the project itself, I feel accomplished knowing that I got to utilize what I have learnt in a real setting. I had an idea and I was not limited by technical ability: I could implement what I wanted to implement, based on what I learnt from the course.

Appendix

Development platform

The compilation environment is Ubuntu 22.04.1 LTS, compiled using g++ 11.3.0.

Compilation

Included in the directory are three wordlist files containing strings.

wordlist_1m.txt is a text file containing 1 million strings.

wordlist_100k.txt is a text file containing 100,000 strings.

wordlist_10k.txt is a text file containing 10,000 strings.

A Makefile is provided, with a few options.

To run an example script that utilizes the ConcurrentTrie:

1) Run *make sample*
2) Run *./SampleUsage*

To run a file that tests the correctness of SequentialTree and ConcurrentTrie:

1) Run *make test*
2) Run *./TrieTest {wordListFilePath} {maxStringsToRead}*
   For instance, ***./TrieTest wordlist_100k.txt 99995*** will use 99995 strings from *wordlist_100k.txt*

To run a file that benchmarks the execution time of SequentialTree and ConcurrentTrie:

1) Run *make benchmark*
2) Run *./benchmark {wordListFilePath} {maxStringsToRead} {prefixToTest}*
   For instance, ***./benchmark wordlist_100k.txt 99995 abc*** will use 99995 strings from *wordlist_100k.txt*. Since *prefixToTest* is provided, if the string retrieval benchmark is chosen, it will time the execution of retrieving all strings with the prefix **abc**

Full list of methods

`void insert(std::string word)`

Inserts a single string into the set.

`void insert(std::vector<std::string>* words)`

Inserts a vector of strings into the set. Implemented by parallelizing the for-loop that inserts a string one at a time, using OpenMP.

`void insertAsync(std::vector<std::string>* words)`

Inserts a vector of strings into the set asynchronously and returns immediately. Implemented by spawning a new thread that calls *insert*.

`void remove(std::string word)`

Removes a single string from the set.

`void remove(std::vector<std::string>* words)`

Removes a vector of strings into the set. Implemented by parallelizing the for-loop that removes a string one at a time, using OpenMP.

`void removeAsync(std::vector<std::string>* words)`

Removes a vector of strings into the set asynchronously and returns immediately. Implemented by spawning a new thread that calls *remove*.

`bool contains(std::string word)`

Returns true if the string is in the set, and false otherwise.

`std::vector<bool> contains(std::vector<std::string> words)`

Returns a vector of Booleans, where the *i*-th boolean represents whether the *i*-th string is in the set.

`int size()`

Returns the number of strings in the set.

`std::vector<std::string> getAllStringsSorted()`

Return all strings in the set, in sorted order.

`std::vector<std::string> getStringsWithPrefix(std::string prefix)`

Given a string prefix, return all strings that have this prefix, in sorted order.