Alternative Exercise Report

Team: Stefanska Anastasiia (1 member)

Table of Contents

Summary **1**
Testing **1**
How to run the code **2**
To run the task 1 2
To run the task 2 2
To run the task 3 2
Task 1 details **2**
Functions and elements description 3
test_file.cpp 3
my_core.cpp 3
Task 2 details **4**
Main improvements 4
Functions and elements description 5
test_file.cpp 5
my_core.cpp 5
Task 3 details **6**
Functions and elements description 7
class Query 7
class RequestDict 7
File processing 7
hamming_distance(word1,word2) 7
check_match(result) 7

Summary

Language: C++
Task 1 result: all tests passed.
Task 2 result: all tests passed, **17.4** times speedup.
Task 3 result: tests passed.

Testing

All local tests are done on the original test file (small_test.txt) on Apple M1 16 GB MacBook Air in plugged state (unplugged slower results included for reference). Large test file was not used for testing, as it was running over 30 minutes in reference implementation and did not finish, so I could not use it for effective benchmarking.

How to run the code

The folder for Task 1 and Task 2 is a copy of the original sigmod-contest-2013 git folder.
To run the tasks - same mechanics as original contest - see below.

To run the task 1

1. Check the Makefile - line 33 is uncommented.
2. Check the Makefile - line 34 is commented.
3. Add small_test.txt to test_data folder. Test data file is same as in git of sigmod contest, and it is not included due to submission file size limitations.
4. In terminal:
    a. cd to sigmod-contest-2013 folder
    b. run make command
    c. run ./testdriver command

To run the task 2

1. Check the Makefile - line 34 is uncommented.
2. Check the Makefile - line 33 is commented.
3. Add small_test.txt to test_data folder. Test data file is same as in git of sigmod contest, and it is not included due to submission file size limitations.
4.
5. In terminal:
    a. cd to sigmod-contest-2013 folder
    b. run make command
    c. run ./testdriver command

To run the task 3

1. Go to Colab Notebook.
2. Add small_test_spark.txt to the same folder. Test data file is same as in git of sigmod contest, and it is not included due to submission file size limitations.
3. Run cells.

Task 1 details

Task main focus is to achieve accurate search per document. There is no focus on performance at this stage (see Task 2 details). As such, the task's main logic is stored in 2 files in subfolder my_impl_task1:
test_file.cpp - functions for search
my_core.cpp - wrapping my functions into the required functions

Test results: all tests passed in 4:04 mins (6:27 mins unplugged state).

```
(base) resheras@Anastasiias-Air sigmod-contest-2013 % make
g++ -O3 -fPIC -Wall -g -I. -I./include -std=c++11 -shared -o libcore.so my_impl_task1/my_core.o my_impl_task1/test_file.o
g++ -O3 -fPIC -Wall -g -I. -I./include -std=c++11 -o testdriver test_driver/test.o  ./libcore.so
(base) resheras@Anastasiias-Air sigmod-contest-2013 % ./testdriver
Start Test ...
Your program has successfully passed all tests.
Time=244962[4m:4s:962ms]
```

Functions and elements description

test_file.cpp

```
list<string> split_doc(const char *doc_str);
```

This function is used to transform the input pointer to C++ list container. We are going to use it on both document and query inputs.

```
int edit_distance(string word1, string word2, int threshold, int
current_dist);
```

Recursive calculation of the edit distance between 2 words.

```
bool edit_matching(list<string> query, list<string> doc, int edit_dist);
```

Loop of the edit distances between 2 lists (query and document).

```
int hamm_recursive_distance(string word1, string word2, int threshold, int
current_dist)
```

Modified edit function to only work for words of the same length.

```
bool hamming_rec_matching(list<string> query, list<string> doc, int
edit_dist);
```

Loop of the hamming distances between 2 lists (query and document). Will be used for both hamming and exact matching.

my_core.cpp

```
struct Query
```

Kept as is from the reference implementation.

```
struct Document
```

Kept as is from the reference implementation.

```
list<Query> queries;
```

List of all active queries.

```
list<Document> docs;
```

List of all documents results.

```
ErrorCode StartQuery(QueryID query_id, const char* query_str, MatchType
match_type, unsigned int match_dist)
```

Adds a new query to the queries list.

```
ErrorCode EndQuery(QueryID query_id)
```

Removes a query from the queries list.

```
bool one_query_check(Query* quer, list<string> document_list,int * query_ids)
```

Given one query and one document, check if the document has the query. Returns true on match found. Writes matching query id to a global list of results by pointer.

```
ErrorCode MatchDocument(DocID doc_id, const char* doc_str)
```

Iterates on all active queries to compare them with this new document. Uses naive loops. Pushes the value of document structure (with its matches) into a global list of documents.

```
ErrorCode GetNextAvailRes(DocID* p_doc_id, unsigned int* p_num_res, QueryID**
p_query_ids)
```

Delivers a document result. Only change to ref_impl - handling of list.

Task 2 details

The main goal of this task is to improve speed performance.The task's main logic is stored in 2 files in subfolder my_impl_task2:
test_file.cpp - functions for search
my_core.cpp - wrapping my functions into the required functions

Main improvements

1. Change of the queries **container** from list to **unordered_map** by query_id. This allows for faster deletion of a query by its id.
2. Change all the rest of list **containers** to **vector**. Vector container is the most performant for the required task, as it is a contiguous memory storage container optimized for faster consecutive reads and faster appends via pushback.
    a. Other containers explored were deque and unordered_set - both produced slower performance for consecutive reads required for non-exact matches.
    b. Additionally explored creating 2 structures in parallel: unordered_set in addition to vector to use in different matches, but creating multiple containers was also slower than one structure.
    c. Additionally explored access via char * and processing word-by-word without storing in an intermediate container. Results are also slower than vector.
3. **Sorted** vector of docs after creating it for every doc. This optimizes find and any_of functions.
4. **Optimized loops** with early exit used: all_of and any_of.
5. **Specialized** non-recursive **function** for **exact** match.
6. **Optimized recursion** for edit match.
7. **Optimized** non-recursive **function** for **hamming** match.
8. String access - **direct index access** with **[i]** instead of .at(i)
9. **Parallel threads** within MatchDocument - each thread is corresponding to a query within queries.
    a. Additionally explored controlling the number of threads with the hardware limit or custom manual configuration - no improvement achieved (Mac thread controller manages queuing of threads efficiently).
10. **Minor functions improvements** - reduced new variables declaration in edit and hamming functions in favor of direct calls.
11. *Did not work*: pre-allocation of memory. Tested pre-allocating for both vector of query size MAX_QUERY_WORDS and document size (MAX_DOC_LENGTH+1)/(MIN_WORD_LENGTH+1). No difference in results of testing times.

Test results: all tests passed in 0:14 mins (0:23 mins unplugged state). That is **17.4 times** faster than task 1.

```
(base) resheras@Anastasiias-Air sigmod-contest-2013 % make
g++ -O3 -fPIC -Wall -g -I. -I./include -std=c++11 -shared -o libcore.so my_impl_task2/my_core.o my_impl_task2/test_file.o
g++ -O3 -fPIC -Wall -g -I. -I./include -std=c++11 -o testdriver test_driver/test.o  ./libcore.so
(base) resheras@Anastasiias-Air sigmod-contest-2013 % ./testdriver
Start Test ...
Your program has successfully passed all tests.
Time=14522[14s:522ms]
```

Functions and elements description

test_file.cpp

```
vector<string> split_doc(const char *doc_str);
```

This function is used to transform the input pointer to C++ list container. We are going to use it on both document and query inputs.

```
bool exact_matching(vector<string> query, vector<string> doc);
```

Optimized new function: Exact non-recursive comparison of 2 vectors of strings - all values of query must be present in doc.

```
bool hamming_matching(vector<string> query, vector<string> doc, int
distance);
```

Optimized new function: Hamming non-recursive comparison of 2 vectors of strings - all values of query must be present in doc.

```
int edit_distance(string word1, string word2, int threshold, int
current_dist);
```

Optimized function: Recursive calculation of the edit distance between 2 words.

```
bool edit_matching(vector<string> query, vector<string> doc, int edit_dist);
```

Optimized function: Loop of the edit distances between 2 vectors of words (query and document).

my_core.cpp

```
struct Query
```

Kept as is from the reference implementation.

```
struct Document
```

Kept as is from the reference implementation.

```
unordered_map<QueryID, Query> queries;
```

Optimized container: unordered map with query id as key.

```
vector<Document> docs;
```

Optimized container: vector of the documents with matched queries.

```
ErrorCode StartQuery(QueryID query_id, const char* query_str, MatchType
match_type, unsigned int match_dist)
```

Optimized function: adds a new query to the queries vector.

```
ErrorCode EndQuery(QueryID query_id)
```

Optimized function: removes a query from the queries vector.

```
bool one_query_check(Query* quer, vector<string> document_vector,int *
query_ids)
```

Optimized function: given one query and one document, check if the document has the query. Returns true on match found. Writes matching query id to a global list of results by pointer.

```
ErrorCode MatchDocument(DocID doc_id, const char* doc_str)
```

Optimized function: Iterates on all active queries to compare them with this new document. Uses naive loops. Pushes the value of document structure (with its matches) into a global list of documents.

```
ErrorCode GetNextAvailRes(DocID* p_doc_id, unsigned int* p_num_res, QueryID**
p_query_ids)
```

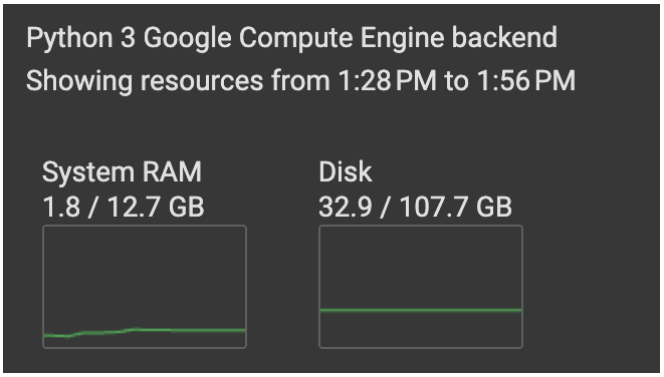
Same as ref_impl. Delivers a document result.

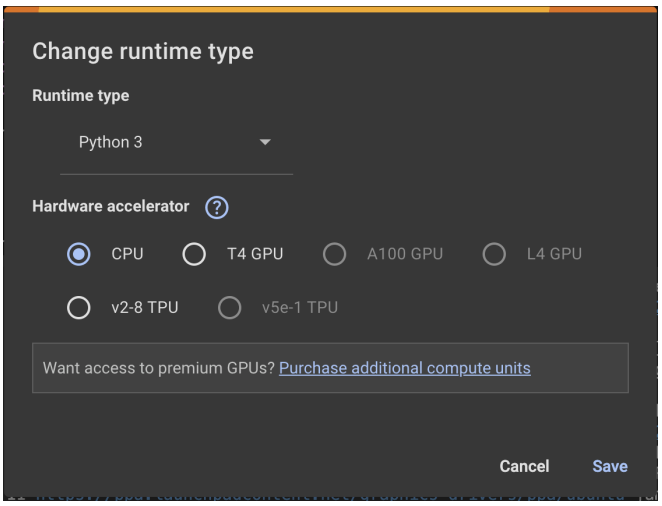
Task 3 details

Link to Colab Notebook

The task is to run the same program on Spark.
To achieve this, I use pyspark and jupyter notebook.

I use 2 environments: the same local machine (or virtual environment on Google Colab.
Virtual environment of Colab:





Functions and elements description

class Query

Each query is an element of this class. It contains all info about the query: id, type, distance, length, and text terms.

class RequestDict

Each request is defined as `RequestDict`. It contains one document, current list of queries for this document, and results for comparison.

File processing

File is read line by line. Each line is processed and added to the respective list (queries, document, results). This is the same file as in Task 1 and 2 (small_test.txt) - just renamed for backup.

hamming_distance(word1,word2)

Letter pair-wise comparison of 2 words.

check_match(result)

Main function to perform checks over documents.
Parallezation of this function is achieved via:

- RDD of the elements of `RequestDict`
- map(), collect() functions over RDD

For hamming distance, above function is used.
For Levenstein distance, python-Levenstein package is used.

To estimate performance, for local run I use:

```
import time
b= time.time()
…
a= time.time()
print(a-b)
```

For Colab run, the cell execution time is recorded automatically on the bottom of the page.

Performance of this function locally over small_test set is **15 seconds**:

```
[25]: import time
      b= time.time()
      my_output = distributed_requests.map(check_match)
      collected_my_output = my_output.collect()
      a= time.time()
      print(a-b)

      25/01/09 21:18:40 WARN TaskSetManager: Stage 3 contains a task of very large size (1102 KiB). The maximum recommended task size is 1000 KiB.
      [Stage 3:=====================================================>       (7 + 1) / 8]
      15.303361892700195
```

Performance of this function over small_test set in Google Collab- **2m:26s**.

I do not consider the comparison between runtimes possible, as it is based on very different systems and code. Yet, local is comparable with task 2 result, and virtual in Colab is almost 2x improvement compared to task1 solution.

Test of accuracy
All comparisons between defined and obtained results are collected in `collected_my_output` variable in `check_match`.
All tests pass (check in last cell).