CS F422

PARALLEL COMPUTING

ASSIGNMENT – 1

REPORT

Anirudh Kumar Bansal	2014A7PS081P
Lakshit Bhutani	2014A7PS095P

March 5, 2017

Problem – 1

Longest Common Subsequence of Documents

Table of Contents

Abstract

The problem involves computing the pair-wise length of LCS (longest common-sub-sequence) of N documents by considering each line as a sequence. Top K pair of documents with longest LCS lengths are selected and pairwise LCS of these documents computed. Finally intersection of the content in the K' documents is calculated. The problem has been done using OpenMP (*Open Multi-Processing*).

Usage Instructions

The source code is in ‘lcs_documents.c’. To compile the code -
gcc -fopenmp lcs_documents.c

[Make sure ‘test1’, ‘test2’, ‘test3’ , ... files are present in the same directory before running. The maximum number of documents is 100 but user can work with any number of documents by changing the MAX_N parameter in the code.*]*

*T*o run the code -
./a.out

Then enter the value of ‘N’ and ‘K’ as prompted. Do ensure that value of K is less than (N * (N + 1)) / 2.

The result on the terminal displays the time taken by the code. The output file of the intersection of documents will be created with name ‘out_lcs’.

Make sure that the documents are named as *text1 , text2, text3, ... , textn*. The documents used for testing are provided in ‘*test_cases_1*’ folder.

Program Structure

The program follows shared memory model. In this specific case the shared memory is the hash values of the lines in all documents stored in *all_hash* the pair wise LCS lengths stored in *lcslen.*

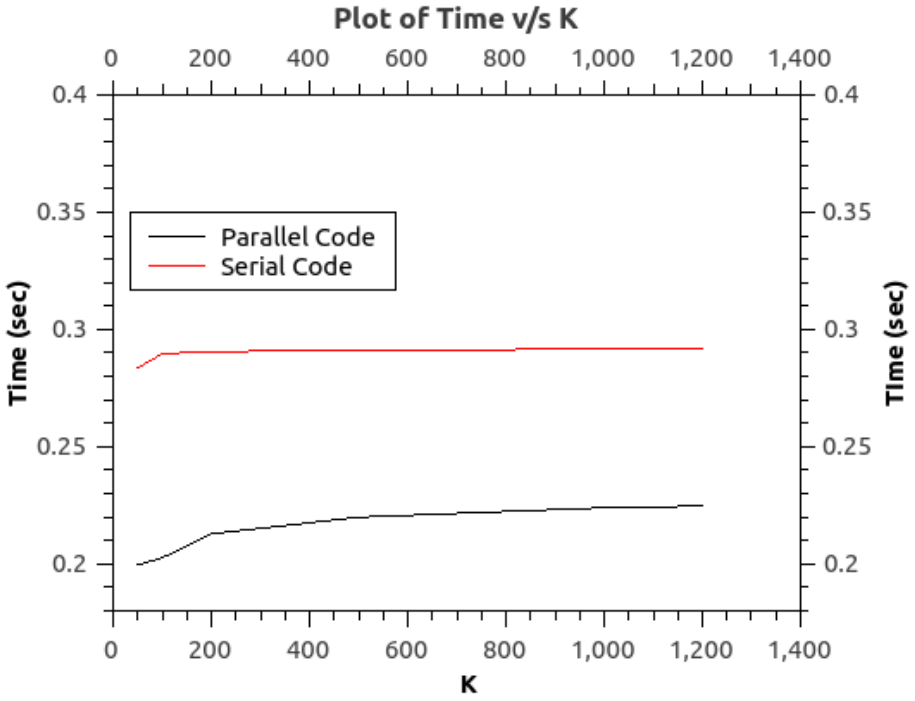
- The documents are read by individual threads and hash value of individual strings stored in *all_hash*.
- Then individual threads calculate the pair-wise LCS of the documents. And store it in *solution* array.The *solution* array also stores the indices of the two documents whose LCS is being computed.
- The solution array is sorted using quick sort to find all the documents involved in the top k pairwise LCS values.The indices of these documents are stored in *topk* array.
- The document in *topk* with the least number of lines is found and its lines are matched with all the lines of the remaining documents.Two lines are compared using their hash values stored in *all_hash* array.The lines present in the intersection of all these documents is written to the file *out_lcs.*

Experimental Results

Test document generation : A random paragraph of 100 lines with line size less than 550 characters was saved as *test* file. Test files *test1, test2, ... , test100* was generated by picking random lines from *test* file. It was ensured that there are some lines common to all test files for checking purposes. The document also contains duplicate lines. Although the length of all documents is the same *i.e.* 100 lines each, but the code is generic to handle variable size files.

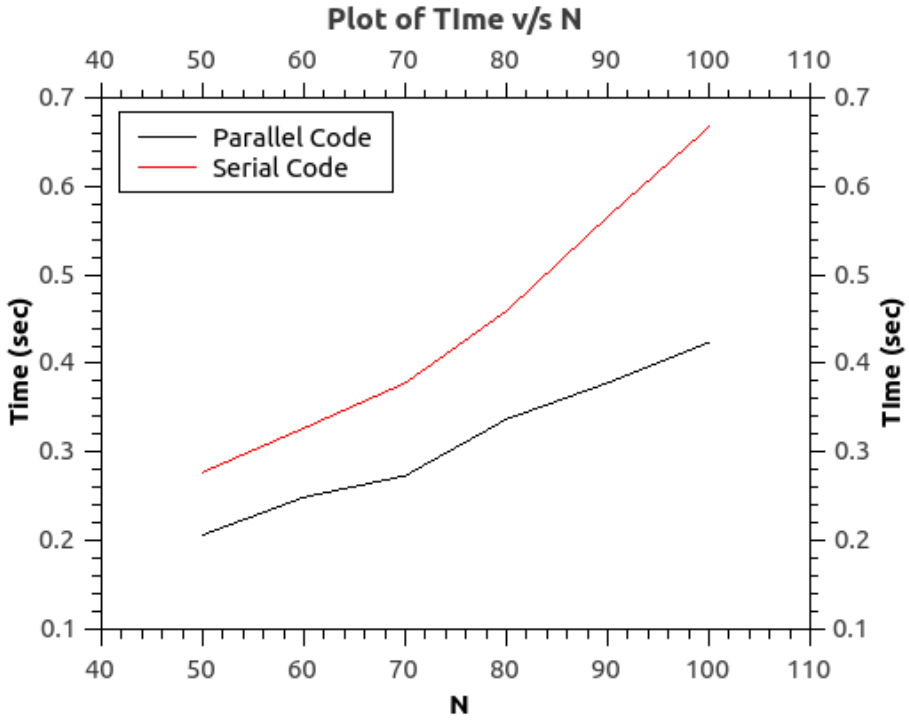
1. Variation of K keeping N constant

N	K	Serial code time (sec)	Parallel code time (sec)	Speedup
50	50	0.283591	0.199634	1.42
50	100	0.289639	0.202635	1.43
50	200	0.290595	0.212811	1.36
50	500	0.291091	0.219922	1.32
50	1000	0.291662	0.223968	1.30



2. Variation of N keeping K constant

N	K	Serial code time (sec)	Parallel code time (sec)	Speedup
50	500	0.277310	0.206526	1.34
60	500	0.327013	0.248946	1.31
70	500	0.377949	0.273272	1.38
80	500	0.459250	0.337102	1.36
90	500	0.565748	0.378108	1.50
100	500	0.666870	0.423691	1.57



Best parallelism and speedup

With N constant at 50 , the best speedup obtained is 1.43 when K is 100.

With K constant at 500, the best speedup obtained is 1.57 when N is 100.

Problem – 2

Open addressed hashtable with concurrent access

Table of Contents

Abstract

The problem involves implementing a concurrent open-addressed hashtable with quadratic probing that supports operations to initialize the hash table, add an element, delete an element, find an element and rehash the hash table. Rehashing is invoked internally if the load factor exceeds a given threshold L and it must be run in the background to ensure that other operations are locked out for the smallest time possible. The problem has been done using Pthreads(*POSIX Threads*).

Usage Instructions

The source code is in 'hash_table_parallel.c'. To compile the code -
gcc -w -fopenmp -pthread hash_table_parallel.c

[Make sure the 'query' file containing the input queries is present in the same directory before running. The total number of queries in the 'query' file provided in the test_cases_2 are 10^5 but user can work with any number of queries by changing the NUM_QUERIES parameter in the code.]

Then enter the value of 'C' and 'L' as prompted. Do ensure that value of L is between 0 and 1.

The result on the terminal displays the time taken by the code. The output file containing the result of queries executed by the threads will be created with name '*query_out_parallel*'. The actual queries (in the order executed by the threads) will be created in the file *'query_actual'*.

The file used for testing is '*query*' , which was generated randomly is available in folder '*test_cases_2*'. The format of the queries is :-
1. 0 key - find element with given key.
2. 1 key value - add element with given key and value.
3. 2 key - delete element with given key.

Program Structure
The program follows a multithreaded model using POSIX threads. It uses 4 threads as specified in the NUM_THREADS parameter in the program.

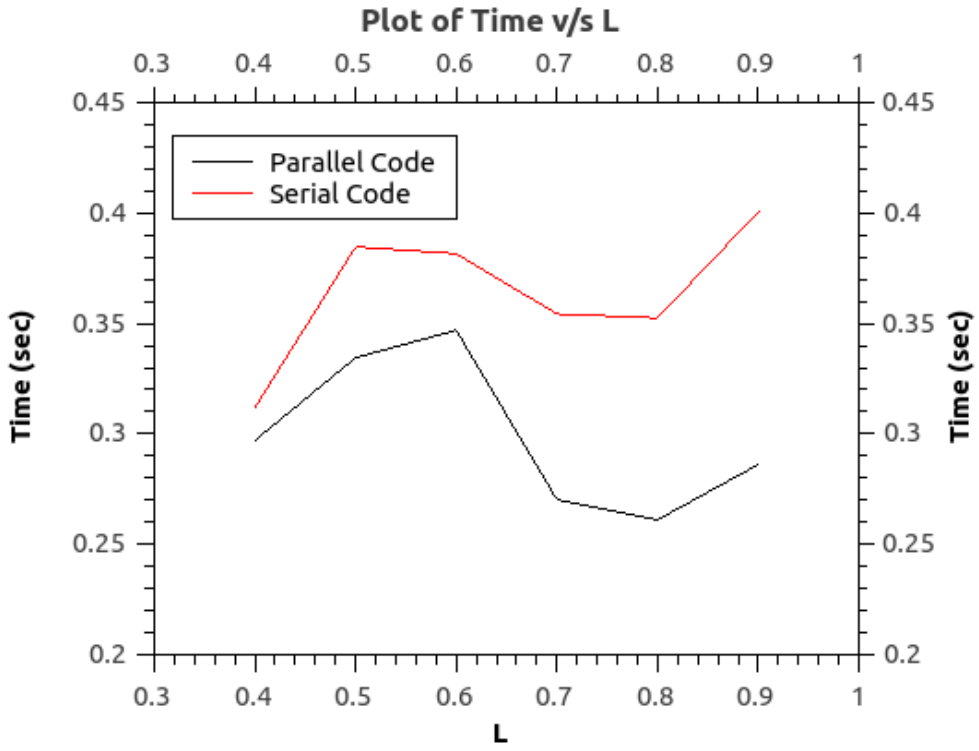
- Queries are randomly distributed to different threads taking care to ensure that no two threads get the same query.Each thread executes one of add (with rehash if necessary) , delete or find operation.
- During add,insert and delete, only those locations in the hash table that are being probed are locked instead of the entire hash table to increase concurrency. Once the location is probed , the location is unlocked.
- During rehashing, first all queries that are curently active are completed and then further queries are blocked only for the time that the hash table is being doubled in capacity. This keeps locking to a minimum.
- The output is printed in '*query_out_parallel*' file. A separate lock is used for the printing process.

Experimental Results

Test data generation : A set of 10^5 random queries containing roughly an equal number of find, add and delete operations was generated and saved in *query* file. The key and/or value paramters for each query are also generated randomly.

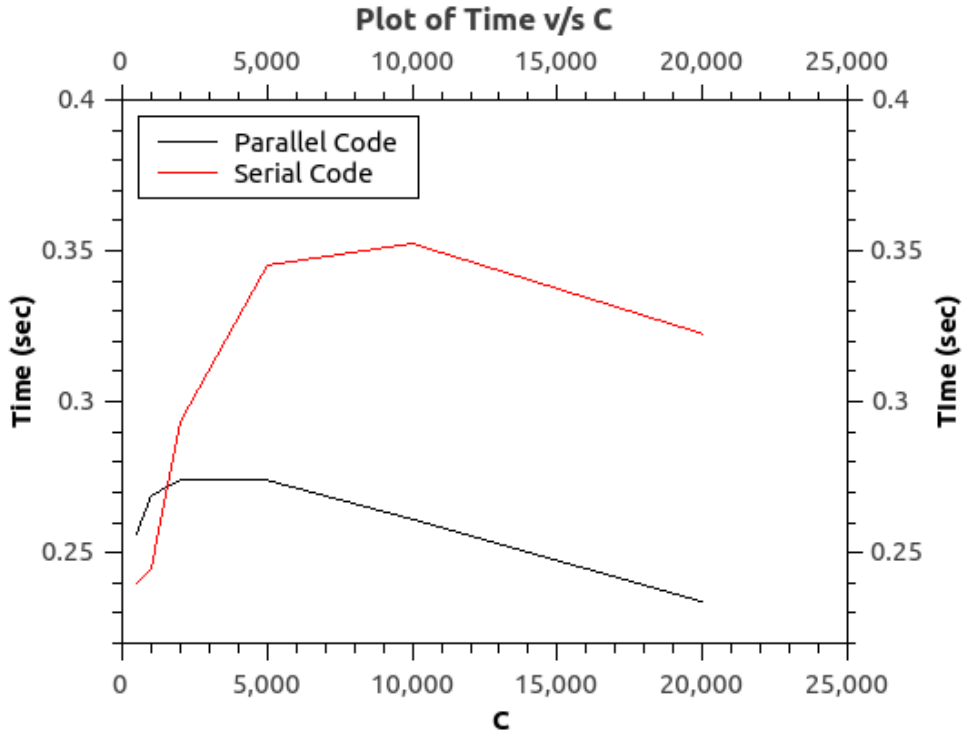
1. Variation of L keeping C constant

C	L	Serial code time (sec)	Parallel code time (sec)	Speedup
10000	0.9	0.400543	0.286105	1.40
10000	0.8	0.352421	0.261056	1.35
10000	0.7	0.354302	0.270232	1.31
10000	0.6	0.381605	0.346915	1.09
10000	0.5	0.384639	0.334467	1.15
10000	0.4	0.311860	0.297009	1.05



2. Variation of C keeping L constant

C	L	Serial code time (sec)	Parallel code time (sec)	Speedup
500	0.8	0.239671	0.256352	0.93
1000	0.8	0.244691	0.268918	0.91
2000	0.8	0.293326	0.274135	1.07
5000	0.8	0.345228	0.273990	1.26
10000	0.8	0.352421	0.261056	1.35
20000	0.8	0.322532	0.233720	1.38



Best parallelism and speedup

With C constant at 10000 , the best speedup obtained is 1.40 when L is 0.9.

With L constant at 0.8 , the best speedup obtained is 1.38 when C is 20000.
