Факултет техничких наука

Универзитет у Новом Саду

Рачунарски системи високих перформанси

Паралелизација налажења најдужег заједничког подниза

Аутор:
Петар Трифуновић

Индекс:
E2 4/2021

8. фебруар 2022.

Сажетак

Динамичко програмирање представља метод којим је могуће оптимизовати извршавање решења одређених проблема у погледу утрошеног времена. Такође, и паралелно програмирање има за циљ смањење времена извршавања одређених алгоритама. Ове две методе су намењене за рад са различитим врстама проблема, али и поред те чињенице, овај рад бави се разматрањем могућности комбиновања динамичког и паралелног програмирања за решавање једног истог проблема, конкретно проблема налажења најдужег заједничког подниза (енг. *Longest Common Subsequence Problem*)

Садржај

Списак изворних кодова

Списак слика

Списак табела

1 Увод

У свету рачунарства постоји константа тежња ка оптимизацији решавања проблема и смањењу времена извршења алгоритама, и у те сврхе се данас неретко користи могућност паралелног извршавања делова алгоритама. Са друге стране, динамичко програмирање јесте концепт који такође побољшава перформансе решења, уколико проблем на који се оно односи поседује скуп тачно одређених карактеристика. Управо те карактеристике чине паралелизацију оваквих проблема тешком, али не и потпуно немогућом.

Када је динамичко програмирање у питању, постоје два приступа решавању проблема - *top-down* и *bottom-up* приступ [2]. Како је описано у [8], могуће је искористити принципе паралелизације извршавања програма како би се убрзао *top-down* приступ. Репозиторијум на овом линку[1] садржи имплементацију предложеног начина паралелизације. Имплементација је прилагођена решавању *knapsack* проблема.

Постоје други значајни радови на ову тему који користе *bottom-up* приступ. У [7] предложена је подела проблема на потпроблеме који су међусобно зависни, али је сваки од њих интерно могуће паралелизовати. Читав математички модел за решавање проблема паралелизације динамичког програмирања са *bottom-up* приступом предложен је у [6], чијим коришћењем је могуће паралелизовати извршавање тако да се добију делимично погрешни резултати, али за које се зна како их додатно обрадити да би се добили тачни.

Овај рад бави се имплементацијом управо начина паралелизације предложеног у [8], али прилагођеног решавању проблема најдужег заједничког подниза (енг. *Longest Common Subsequence*, у даљем тексту *LCS*). У поглављу 2, дат је опис принципа динамичког програмирања. Након тога следи поглавље 3 са детаљима о самом *LCS* проблему. У поглављу 4 извршен је преглед два секвенцијална начина решавања *LCS* проблема, а у поглављу 5 опис предложеног паралелизованог решења. Поглавље 6 садржи детаље о имплементацији која прати овај рад, а поглавље 7 резултате њеног тестирања. Коначно, у поглављу 8 дат је закључак о обрађеној теми.

[1] https://github.com/stivalaa/paralleldp

2 Динамичко програмирање

2.1 Особине проблема погодних за динамичко програмирање

Добар опис тога за шта се може применити динамичко програмирање дат је у [4]. Како се тамо наводи, динамичко програмирање могуће је применити за решавање проблема који су као једна целина тешко решиви, али које је могуће разложити на мање, лакше задатке. Те мање јединице дају проблему рекурзивну структуру. Конкретно, да би био погодан за динамичко програмирање, неопходно је да проблем има два својства — да се састоји од **преклапајућих потпроблема** (енг. *overlapping subproblems*) и да има **оптималну подструктуру** (енг. *optimal substructure*).

Прво својство (својство преклапајућих потпроблема) чини да мањи задаци, односно потпроблеми, од којих се састоји читав проблем, не буду потпуно независни, односно да може доћи до тога да се неки потпроблем решава више пута. На пример, уколико се рачуна Фибоначијев низ, важи да је $F_{25} = F_{24} + F_{23}$, али и $F_{24} = F_{23} + F_{22}$. Како се F_{23} користи два пута, овај алгоритам има преклапајуће потпроблеме.

Друго својство (својство оптималне подструктуре) чини могућим да се оптимално решење читавог проблема нађе коришћењем оптималних решења његових потпроблема.

Проблеми који поседују својство оптималне подструктуре јесу рекурзивни проблеми. Алгоритми динамичког програмирања најчешће имају и своју чисто рекурзивну, али спорију варијанту. Такође, уколико је проблем рекурзивне природе, али нема својсто преклапајућих потпроблема, могуће га је решити коришћењем динамичког програмирања, али то не би довело до побољшања у погледу брзине извршавања, а довело би до већег утрошка меморије због потребе коришћења додатне меморијске структуре. Због тога, динамичко програмирање треба примењивати онда када то заиста има смисла.

2.2 Идеја динамичког програмирања

Идеја динамичког програмирања јесте да, коришћењем додатних меморијских структура, убрза извршавање рекурзивних алгоритама. Тачније, сваки од преклапајућих потпроблема, односно потпроблема које би чист рекурзивни алгоритам решавао више пута, коришћењем динамичког програмирања решава се само једном.

Постоје два могућа приступа имплементацији решења проблема коришћењем динамичког програмирања - *top-down* и *bottom-up*. За овај рад, од већег значаја је

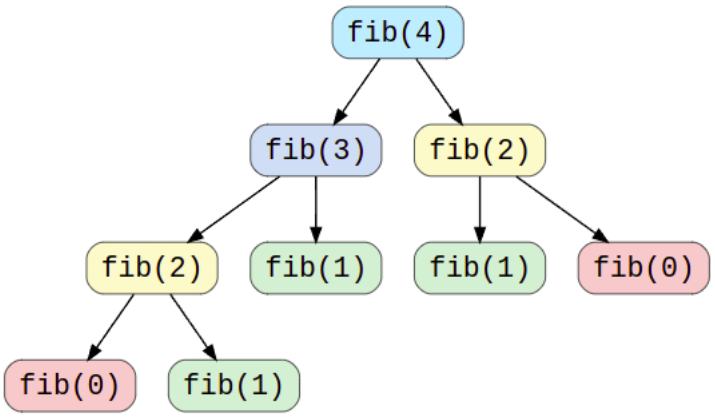
top-down приступ, и он ће детаљније бити описан у наредном поглављу. Више о *bottom-up* приступу може се прочитати у [3].

2.3 *Top-down* приступ динамичком програмирању

Код *top-down* приступа, како сам назив каже, проблем се решава почев са врха, па наниже. Овај приступ природнији је него *bottom-up*, с обзиром на то да се на овај начин приступа проблему и када се решава чистом рекурзијом, а и сам приступ користи управо рекурзивне позиве методе. Креће се од читавог проблема, односно проблема на врху рекурзивног "стабла", и он се даље рашчлањује на једноставније потпроблеме. Оно што разликује *top-down* приступ од чисте рекурзије јесте то што овај приступ у додатној меморијској структури чува резултате свих решених потпроблема. Пре обраде неког потпроблема, испита се да ли је његов резултат већ присутан у меморијској структури. Ако јесте, прочитаће се одатле и неће се вршити поновно израчунавање резултата. Ако није, резултат ће се израчунати и уписати у структуру, како се не би рачунао поново уколико се на исти потпроблем наиђе у будућности. Меморијска структура би идеално требало да има кратко време приступа, поготово када је читање у питању, с обзиром на то да се у једно поље структуре уписује само једном, а може се очекивати да читање наступи више пута. Зато, као меморијска структура може се користити низ или хеш табела.

Ситуација у којој се резултат потпроблема чита из меморијске структуре може значајно да утиче на време извршења, поготово ако више пута наступи таква ситуација. Наиме, читање једне вредности из структуре доводи до тога да се избегне рачунање резултата за читаво подстабло рекурзивних позива које би се у чистој рекурзији нашло испод потпроблема за који је резултат прочитан из структуре.

На слици 1, преузетој из [3], налази се пример стабла рекурзивних позива за рачунање четвртог члана Фибоначијевог низа. Зарад боље илустрације побољшања које доноси динамичко програмирање, рецимо да нулти и први члан нису познати на почетку и да је и за њих потребно извршити неке прорачуне. У том случају, ако се користи чиста рекурзија, нулти и други члан рачунају се по два пута, док се први члан рачуна три пута. Ако се проблем реши динамичким програмирањем, сваки од ових чланова рачунаће се само једном, и број потребних прорачуна би се са девет смањио на пет.



Слика 1: Рекурзивно стабло за рачунање члана Фибоначијевог низа

3 Проблем најдужег заједничког подниза (*LCS* проблем)

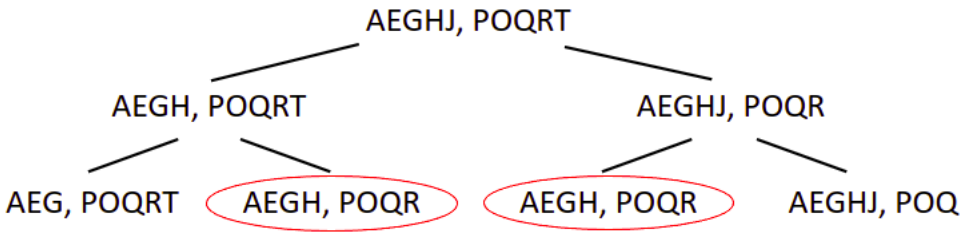
LCS проблем, представља проблем налажења најдужег подниза који се јавља у два задата низа. Низ $N2$ је подниз низа $N1$ уколико се сви елементи из $N2$ јављају и у $N1$ и уколико важи да, ако се елемент $A2$ јавља након елемента $A1$ у низу $N2$, важи да се у овом редоследу јављају и у низу $N1$. Ако се два елемента налазе на узастопним индексима у низу $N2$, није неопходно да се нађу на узастопним местима и у низу $N1$.

3.1 Погодност примене динамичког програмирања на *LCS* проблем

LCS проблем поседује оба својства описана у поглављу 2.1, која један проблем чине погодним за динамичко програмирање.

LCS се састоји од преклапајућих потпроблема. На слици 2 приказан је део рекурзивног стабла за решавање *LCS* проблема. Детаљи о решавању овог проблема дати су у поглављу 4, али за сада је важно разумети да се проблем од основног, са целим низовима (врх стабла са слике) грана на потпроблеме са мањим низовима. На слици, лево дете сваког чвора има један елемент скинут са краја првог низа у односу на први низ родитеља, и цео други низ. Десно дете има цео први низ, а један елемент мање у другом низу. Овако, постепеним скидањем елемената са краја, долази се до тога да два чвора обрађују идентичне низове, као што је случај са означеним чворовима на слици 2. Овај пример показује да *LCS* проблем садржи преклапајуће потпроблеме.

LCS такође има и оптималну подструктуру. Као што је већ поменуто, на слици 2 налази се приказ рекурзивног стабла за решавање проблема, а могућност решавања проблема рекурзијом значи да проблем управо има оптималну подструктуру.



Слика 2: Преклапајући потпроблеми *LCS* проблема

3.2 Пример *LCS*-а

Рецимо да постоје два низа карактера, $N1$="АСЛКУОБМ" и $N2$="СИОЛПУБХ". Карактери који се јављају у оба низу јесу 'С', 'Л', 'У', 'О' и 'Б'. Карактер 'С' се јавља пре свих осталих у оба низа, па се дефинитивно може укључити у решење. Како се 'О' у првом низу јавља након 'Л' и 'У', а у другом низу пре ових карактера, то значи да, уколико се 'О' укључи у решење, 'Л' и 'У' не смеју, јер решење онда неће испуњавати услов *LCS*-а. 'Б' се јавља као последњи од наведених карактера у оба низа, па се може закључити да је један могућ подниз наведених низова "СОБ". Са друге стране, ако се 'О' искључи из решења, могу се укључити 'Л' и 'У'. Овако, добија се подниз "СЛУБ", који је дужи од "СОБ", и стога представља најдужи подниз наведених низова, односно решење *LCS* проблема.

4 Решавање *LCS* проблема без употребе паралелизма

4.1 Наивно решење

У овом поглављу описан је начин решавања *LCS* проблема који је најједноставнији у погледу имплементације. Овај начин не користи никакав вид оптимизације и не узима ни на који начин у обзир чињеницу да се *LCS* састоји од преклапајућих потпроблема. Једноставно, сваки потпроблем на који се наиђе биће и решен, без обзира на то да ли је алгоритам већ прошао кроз исти такав потпроблем, или не. Како је наведено у [5], уколико имамо два низа, X_i и Y_j, дужина, редом, i и j, тада је дужина њиховог најдужег заједничког подниза дефинисана формулом са слике 3. Дакле, уколико је последњи елемент једног низа једнак последњем елементу другог низа, онда је дужина *LCS*-а тих низова за један већа од дужине *LCS*-а низова добијених уклањањем последњег елемента са оба низа. Уколико последњи елементи нису једнаки, рачуна се дужина *LCS*-а за цео први низ, и за други низ без последњег елемента, па затим за цео други и први без последњег елемента, па се као резултат узима већа вредност. Ово објашњење, као и приказана формула, јасно наговештавају да се ради о рекурзивном решењу проблема.

Као што је приказано на слици 2, у стаблу рекурзије један потпроблем може се јавити више пута. Наивни алгоритам то не узима у обзир и рачуна решење потпроблема без обзира на то колико се пута понове.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ \text{c[i-1, j-1]} + 1 & \text{if } i,j > 0 \text{ and } x_i = y_j, \\ max(\text{c[i, j-1] c[i-1, j]}) & \text{if } i,j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

Слика 3: Формула наивног решења *LCS* проблема

4.2 Решење коришћењем *top-down* приступа динамичком програмирању

Како је већ описано у поглављу 2.3, *top-down* приступ динамичком програмирању такође користи рекурзију, тако да се основа овог решења не разликује од наивног, односно такође користи формулу са слике 2. Разлика је у томе што се, пре рачунања дужине *LCS*-а за два низа у одређеном потпроблему, провери да ли се решење тог потпроблема налази у помоћној меморијској структури, па ако се налази преузме се из ње и тиме се избегне поновна обрада тих низова, односно обрада свих низова који

```
int lcs(string X, string Y, int m, int n, int dp[][maximum])
{
    if (m == 0 || n == 0) return 0;

    if (dp[m - 1][n - 1] != -1)
        return dp[m - 1][n - 1];

    if (X[m - 1] == Y[n - 1]) {
        dp[m - 1][n - 1] = 1 + lcs(X, Y, m - 1, n - 1, dp);
        return dp[m - 1][n - 1];
    }
    else {
        dp[m - 1][n - 1] = max(lcs(X, Y, m, n - 1, dp),
                               lcs(X, Y, m - 1, n, dp));
        return dp[m - 1][n - 1];
    }
}
```

Изворни код 1: *LCS* решење коришћењем *top-down* приступа и помоћног низа

би настали од њих уклањањем елемената са краја.

Како се у сваком наредном кораку рекурзије уклања последњи елемент једног, или оба низа, то је сваки потпроблем јединствено идентификован дужинама два низа са којима ради. Те дужине означавају колико елемената треба узети почев од почетка оригиналних низова. Ако имамо почетне низове $N1$="АСЛКУОБМ" и $N2$="СИОЛПУБХ", тада дужине 4 и 3 јединствено идентификују потпроблем који ради са низовима $N1_4$="АСЛК" и $N2_3$="СИО". Ово омогућава погодан избор помоћне меморијске структуре. Први избор могао би бити дводимензиони низ, односно матрица облика A_{mxn}, где су m и n дужине оригиналних низова за које се тражи дужина *LCS*-а. Провера постојања решења одређеног потпроблема би се вршила индексирањем матрице дужинама низова са којима конкретан потпроблем ради. Други избор структуре могла би бити хеш табела, код које би кључ био одређена комбинација дужина низова, која би се потом хеширала. Обе структуре имају $O(1)$ временску комплексност читања, што их чини сасвим одговарајућим за коришћење у сврхе помоћне меморијске структуре.

Код 1 преузет из [1] приказује имплементацију описаног решења у програмском језику *C++*.

5 Решавање *LCS* проблема помоћу паралелизма

5.1 Непогодност *LCS* проблема за паралелизацију

За решавање паралелним програмирањем најпогоднији су проблеми који се могу поделити у мање независне целине. У том случају могуће је покренути извршавање делова алгоритма на више нити истовремено и на тај начин смањити целокупно време извршавања. Такође, уколико паралелне линије извршења приступају истим меморијским локацијама само зарад читања, избегава се и потреба за синхронизацијом приступа меморији која би се захтевала у случају уписа на исте меморијске локације.

Нажалост, *LCS* не одговара овом опису проблема погодних за паралелизацију. *LCS* се може поделити на мање целине, односно потпроблеме, али као што је више пута наглашено, они нису независни. Ова особина чини *LCS* проблем погодним за динамичко програмирање, али зато значајно отежава паралелизацију. Такође, ако се користи динамичко програмирање и решења потпроблема памте у меморији, може доћи до проблема трке (енг. *race condition*) уколико више нити покуша да приближно истовремено изврши упис решења једног истог потпроблема у помоћну меморијску структуру.

У наредном поглављу описан је начин на који је ипак могуће искористи и паралелно програмирање за *LCS* проблем, како би се добио алгоритам од ког се може очекивати да у већини ситуација брже дође до решења од чисто секвенцијалног.

5.2 Паралелни алгоритам за решавање *LCS* проблема

У овом поглављу описан је паралелни алгоритам предложен у [8]. Предложени начин имплементације може се генерално примењивати када је динамичко програмирање у питању, конкретно за *top-down* приступ, а не само на *LCS* проблем.

5.2.1 Основна идеја алгоритма

Основна идеја поменутог алгоритма не мења секвенцијално решење. Наиме, идеја јесте да се један исти секвенцијални рекурзивни алгоритам извршава на више паралелних нити, и да се прихвати резултат оне која буде најуспешнија, односно која прва дође до коначног решења проблема. Иако суштински извршавају исти алгоритам, од нити се очекује да крену различитим путевима низ стабло рекурзије, имајући у виду да ће, ако више нити извршава исти алгоритам, неке од њих поћи путем за који ће се испоставити да брже стиже до решења. Ово се може десити јер, ако

крену различитим путевима, у почетним нивоима стабла нити ће рачунати резултате потрпоблема и уписивати их у дељену хеш табелу. Како конкретна нит силази дубље низ стабло рекурзије, тако се повећавају шансе да ће наићи на потпроблем чији је резултат нека друга нит, или она сама, уписала у хеш табелу. Дакле, када постоји само једна нит, она је та која мора да упише резултат потпроблема у меморијску структуру како би га евентуално касније прочитала. Када има више нити, једна нит може се "ослонити" и на друге да уписују резултате у меморију.

У [8] су предложена два начина за дивергенцију путева различитих нити. Први начин јесте насумично одређивање пута којим ће се поћи када се стигне до "раскрснице". Овај начин је бољи када један чвор у стаблу рекурзије има већи број деце, али може да се искористи и код чворова са двоје деце. Други начин јесте фиксирање одређеног дела пута за сваку нит, како би дивергенција на том делу пута била сигурна. Ово је најлакше урадити када је стабло бинарно (два детета за сваки чвор) и тада је могуће за n нити фиксирати избор за првих $\log_2 n$ "раскрсница".

Код *LCS* проблема, до "раскрснице" се стиже када последњи елементи два низа нису једнаки. Тада треба бирати да ли ће се прво рачунати *LCS* за цео први низ и други низ без последњег елемента, или за први низ без последњег елемента и цео други.

5.2.2 Дељена хеш табела

Још једна изузетно важна ствар за рад оваквог алгоритма јесте погодна меморијска структура. У [8] наводи се да за ове сврхе треба користити дељену хеш табелу. Оно што је проблем код дељене меморије јесу уписи, који могу да доведу до проблема уколико две нити покушају да упишу податак на исту локацију истовремено. Једно решење овог проблема јесте увођење синхронизације механизмима закључавања. Ипак, овакво решење би значајно успорило и закомпликовало рад алгоритма, с обзиром на то да треба водити рачуна о синхронизационим променљивама, чекати на откључавање и тако даље. Због тога је у [8] предложено коришћење атомичне *compare-and-swap* инструкције. Ова инструкција чита садржај меморијске локације, упоређује прочитану са неком вредношћу, и ако су исте упише нову вредност на ту локацију. Без обзира на успешност инструкције, она као резултат враћа вредност прочитану из меморије. Све ово ради се атомично, тако да не постоји шанса да друга нит модификује локацију у току извршења *compare-and-swap* инструкције. Ово се може искористити тако што би се садржај одређеног поља у хеш табели упоредио са унапред дефинисаном ознаком за празну вредност, и уколико је поље празно уписала би се нова вредност.

6 Имплементација решења *LCS* проблема у програмском језику *C*

Имплементација решења *LCS* проблема извршена је на на начин предложен у [8]. Предлог које је тамо дат не односи се конкретно на *LCS* проблем, већ се може применити генерално на *top-down* приступ динамичком програмирању, уз пратећу имплементацију предложеног на примеру *knapsack* проблема дату на овом линку[2]. Као што је описано у поглављу 5.2, главне ставке предложеног начина решавања *LCS* проблема јесу насумичан избор пута којим ће се свака нит кретати низ стабло, као и дељена хеш табела која подржава безбедан конкурентан упис, без потребе за закључавањем.

У овом поглављу описан је начин на који је извршена имплементација предложеног начина решавања *LCS* проблема.

6.1 Имплементација дељене хеш табеле

6.1.1 Структура табеле

Чланови хеш табеле дефинисани су структуром *ht_entry*, која као најважнија поља садржи шездесетчетворобитни кључ, као и целобројну вредност која се под тим кључем чува. Додатно, ова структура садржи и поље $status$, које је $enum$ типа и може имати вредност $EMPTY$ или $OCCUPIED$ у зависности од тога да ли је поље већ заузето или не. Структура под називом ht представља саму табелу. Поља ове структуре јесу динамички низ показивача на *ht_entry* инстанце, величина саме табеле, као и показивачи на примарну и секундарну хеш функцију. Показивачи на ове функције уводе одређен ниво апстракције, тако да сама инстанца хеш табеле нема знање о конкретној имплементацији функција за хеширање, већ јој се оне прослеђују при иницијализацији табеле, али се могу и касније у току рада мењати. Секундарна функција је неопходна, с обзиром на то да се ради о хеш табели са отвореним адресирањем.

У конкретној имплементацији, у хеш табелу уписују се резултати потпроблема. Сваки потпроблем јединствено је идентификован дужином низова са којима ради, како је описано у поглављу 4.2. Дужине низова су тридесетдвобитне, па се шездесетчетворобитни кључ добија шифтовањем дужине другог низа за 32 места у лево, и затим надовезивањем са десне стране дужине другог низа, коришћењем битског *ИЛИ*.

[2]https://github.com/stivalaa/paralleldp

При иницијализацији табеле, иницијализују се и сви њени чланови, односно сви чланови низа *ht_entry* инстанци. Такође, $status$ сваког члана табеле поставља се на $EMPTY$ вредност, али се и вредност кључа поставља на вредност *HT_EMPTY_KEY* константе. Ова вредност је значајна када се врши безбедан упис у табелу, без закључавања, коришћењем *compare-and-swap* инструкције.

6.1.2 Операција тражења

Тражење започиње рачунањем хеша кључа елемента који се тражи. Након тога, са хешом као индексом, приступа се низу *ht_entry* инстанци. Ако је на задатом индексу инстанца чији је статус $EMPTY$, тражени елемент не постоји. Ако статус није $EMPTY$, и кључ нађене инстанце се поклапа са задатим, инстанца је нађена. У случају непоклапања кључева, постоји могућност да је при уметању дошло до колизије, па се хеш мења секундарном хеш функцијом како би се добио нови индекс. Секундарна функција се примењује док се не нађе тражени кључ, или док се не прекорачи задати праг броја покушаја тражења, у ком случају се сматра да тражени члан не постоји у табели.

6.1.3 Операција уметања

Операција уметања јесте операција која уноси опасност од проблема трке, с обзиром на то да више нити може покушати конкурентно да упише резултат истог потпроблема на исто место у табели. Као решење овог проблема, користи се функција *__sync_val_compare_and_swap* из библиотеке *stdatomic.h*.

Уметање има следећи ток — прво се креира јединствени кључ потпроблема, коришћењем дужина низова са којима потпроблем ради; кључ се хешира и на основу њега прибави се члан табеле. За ово прибављање не користи се описана операција тражења, већ посебна функција која ће вратити елемент и ако му је статус $EMPTY$, што ће бити случај када је елемент иницијализован, али још увек није заиста уписан у табелу. Затим се провери да ли је кључ прибављеног елемента једнак константи *HT_EMPTY_KEY*, као додатна гаранција да је елемент празан. Ово се врши обичном *if* провером. Након ове провере, коју може проћи више нити, једна од њих треба атомично да промени вредност кључа са *HT_EMPTY_KEY* на реалну вредност кључа елемента који се уписује. Ту се користи *__sync_val_compare_and_swap* функција — вредност кључа елемента се овом функцијом поново упореди са *HT_EMPTY_KEY* и у случају поклапања промени на вредност кључа конкретног потпроблема. И ова провера и упис врше се атомично, тако да ће поклапање при провери добити само једна нит. Та нит ће наставити даље са уметањем елемента, постављањем његове вредности и статуса. Све нити које након ње дођу до *__sync_val_compare_and_swap* функције регистроваће непоклапање кључа са *HT_EMPTY_KEY*, и одустаће од упи-

```
1 if (entry != NULL && entry->key == HT_EMPTY_KEY)
2 {
3     if(__sync_val_compare_and_swap(&entry->key,
4                     HT_EMPTY_KEY, h_key) != HT_EMPTY_KEY)
5         return ht_insert(h_table, h_key, h_value);
6
7     entry->status = OCCUPIED;
8     entry->value = h_value;
9 }
```

Изворни код 2: Део функције уметања

са елемента у табелу.

Исечак кода 2 приказује део функције уметања. Променљива $entry$ представља елемент који је извучен из табеле на основу хеша кључа потпроблема који је потребно уписати у табелу. Након што други *if* прође неуспешно, нит позива функцију уметања још једном. У поновном позиву ће и први *if* проћи неуспешно јер кључ неће имати вредност *HT_EMPTY_KEY*, па ће нит у потпуности одустати од покушаја уписа резултата потпроблема.

6.2 Паралелизација извршавања коришћењем *OpenMP*-а

6.2.1 Избор технологије

При имплементацији, разматрано је коришћење *OpenMP*-а и *MPI*-а.

Оно што *MPI* чини погодним јесте добар начин комуникације између процеса. Ово би било корисно за имплементацију, с обзиром на то да би, након што једна линија паралелизма прва дође до решења, требало прекинути извршавање осталих. Са друге стране, велика мана *MPI*-а у овом случају јесте чињеница да ради са независним процесима који не деле директно меморију, а дељена меморија је кључна за имплементацију дељене хеш табеле.

Ни *OpenMP* није савршено решење проблема. Свакако, оно што је погодно јесте чињеница да користи дељену меморију. Са друге стране, *OpenMP* је најпогоднији када се раде итеративне обраде и када се користи *SIMD/SPMD* модел паралелизма. У овом решењу, свакако да све нити извршавају исту инструкцију, тачније исти програм, али једино где би се могло рећи да користе различите податке јесте када је у

питању променљива којом се симулира насумичност избора пута у стаблу рекурзије. Остали подаци су исти, с обзиром на то да све нити врше обраду истог, и то целог низа. Такође, проблем јесте и прекидање осталих нити након што једна заврши. *OpenMP* поседује *cancel* механизам, којим једна нит може да иницира прекидање одређеног *#omp* блока. Остале нити у обележеним тачкама проверавају да ли је иницирано прекидање, и ако јесте, престају са извршавањем. Проблем са овим механизмом јесте што тачке провере прекида не смеју да се нађу унутар функције која се позива из *#omp parallel* блока, што не одговара рекурзивној имплементацији која захтева управо постојање такве функције. Ипак, мане *OpenMP*-а имају знатно мању тежину од непостојања дељене меморије код *MPI*-а, тако да је у имплементацији коришћен управо *OpenMP*.

6.2.2 Имплементација помоћу *OpenMP task*-ова и провере услова прекида

Коначна имплементација се своди на коришћење *OpenMP task*-ова, односно задатака. Очекује се да сваки задатак извршава друга нит. У сваком задатку позива се функција за налажење дужине *LCS*-а и прослеђују јој се цели низови, па се унутар те функције насумично бира којим путем ће нит кренути низ стабло рекурзије. Додатним макроом *TEST_EXIT_CONDITION*, који се може дефинисати у току компајлирања, бира се да ли ће нити при сваком рекурзивном позиву функције проверавати тест променљиву која означава да ли треба настављати извршење, или не, уколико је нека нит већ завршила раније са извршењем. Нит која прва заврши поставиће ову глобално видљиву променљиву на 1, што ће осталим нитима наговестити да је већ пронађено решење. Када нит при провери тест променљиве наиђе на вредност 1, одмах се извршава *return* и нит постепено излази из својих рекурзивних позива. Постављање вредности ове променљиве, као и њено испитивање, такође су извршени коришћењем *__sync_val_compare_and_swap* функције. Иако је могуће користити овај механизам, он не смањује време извршења толико приметно. Наиме, када нека нит заврши са извршењем, то значи да је она попунила део табеле својим резултатима потпроблема, а остатак су попуниле друге нити. Тако, када нека нит дође до краја, осталим нитима су практично преко табеле доступни резултати већине, или чак и свих, потпроблема, што доводи до тога да и остале нити дођу до решења недуго након оне која је до њега дошла прва.

6.2.3 Насумично бирање пута у рекурзивном стаблу

Насумичност избора пута нити у рекурзивном стаблу имплементирана је помоћу *rand* функције. С обзиром на то да би, без иницијализације рандомизујуће функције, она при сваком покретању враћала исте вредност, на почетку самог програма рандомизација је иницјализована тренутним временом, позивом *srand(time(0))*. У самој рекурзивној функцији, генерише се насумична вредност, па ако је дељива са 2 кре-

```
int decider = rand();
int value;
if(decider % 2 == 0)
    value =
    max(findLcs(arrA, arrB, lenArrA - 1, lenArrB, h_table),
        findLcs(arrA, arrB, lenArrA, lenArrB - 1, h_table));
else
    value =
    max(findLcs(arrA, arrB, lenArrA, lenArrB - 1, h_table),
        findLcs(arrA, arrB, lenArrA - 1, lenArrB, h_table));
```

Изворни код 3: Насумичан избор пута у рекурзивном стаблу

ће се једним, а ако није, онда другим путем. Имплементација насумичног избора приказана је у исечку кода 3. *arrA* и *arrB* су низови који се разматрају са дужинама, редом, *lenArrA* и *lenArrB*. Као што се може видети, у једном случају се прво врши рекурзивни позив за цео други и смањен први низ, а затим се врши позив за цео први и смањен други низ. У другом случају, позиви се врше у обрнутом редоследу. Који низ ће се први смањити одређено је дељивошћу променљиве *decider* са 2.

7 Резултати тестирања

У овом поглављу приказани су резултати тестирања у погледу времена извршавања. Величина хеш табеле одређује број потпроблема које је могуће запамтити и значајно утиче на перформансе. Такође, величине низова свакако утичу на време извршавања, али ако се задрже величине низова, а промене њихови елементи, времена извршења се могу значајно разликовати. Сваки тест пример покренут је по два пута секвенцијално и два пута паралелно, где је број паралелних нити мењан при различитим покретањима.

Резултати тестова налазе се у табели 1. Прве две колоне представљају дужине низова коришћених у тесту, а трећа носи величину хеш табеле у погледу броја чланова који се у њу могу уписати. Врсте у којима су ове три колоне једнаке односе се на тестове покренуте са идентичним низовима. Четврта колона представља број коришћених нити и релевантна је само за паралелни резултат. Наредне две колоне представљају време извршења секвенцијалног и паралелног решења, у секундама. У последњој колони налази се резултат у погледу пронађене дужине *LCS*-а.

дужина1	дужина2	величина табеле	нити	секвенцијално	паралелно	резултат
171	198	2^{16}	4	15.76	10.57	78
171	198	2^{16}	6	15.99	11.13	78
66	70	2^{12}	4	21.66	6.74	29
66	70	2^{12}	8	5.66	11.27	29
152	116	2^{14}	5	3.33	1.86	48
152	116	2^{14}	7	3.19	1.80	48
248	318	2^{18}	3	6.34	5.44	124
248	318	2^{18}	8	6.89	6.35	124

Табела 1: Резултати тестирања

Може се приметити да повећан број нити не мора нужно да убрза решење. Чак, за велики број нити често долази до успорења, што може бити последица времена које је потребно утрошити за стварање и гашење нити. Такође, могуће је да, у зависности од оптерећења рачунара при покретању, али и од "среће" при избору пута, дође до великог одступања у времену између два покретања. Рецимо, секвенцијално решење (које такође користи насумичан избор пута) из треће врсте траје приметно дуже него тест из четврте, што може наговестити да се у трећој врсти кренуло знатно лошијим путем. Иако у табели нема таквих случајева, могуће је да секвенцијално решење потраје краће него паралелно, уколико се случајно деси да је једна нит у секвенцијалном решењу изабрала бољи пут него било која нит из паралелног.

8 Закључак

Тема овог рада јесте истраживање могућности комбинације паралелног и динамичког програмирања за решавање *LCS* проблема. *LCS* није проблем погодан за паралелизацију, и *top-down* алгоритам, барем за сада, није могуће паралелизовати, с обзиром на то да постоје јаке зависности између потпроблема на које га је могуће рашчланити. Ипак, у овом раду описано је како је могуће искористити принципе паралелизма како би се у основи секвенцијално решење убрзало до одређене мере.

Описана имплементација би се евентуално могла додатно оптимизовати фиксирањем различитих избора путева за различите нити. Ово би се могло урадити за првих неколико нивоа рекурзивног стабла, како би се у самом старту осигурала дивергенција путева између нити. То би до одређене мере побољшало решење, с обзиром на то да тренутна имплементација, која користи насумичан избор пута, иако не велику, ипак оставља могућност да различите нити у току читавог извршења иду истим путем.

Библиографија

[1] Longest common subsequence | dp using memoization. https://www.geeksforgeeks.org/longest-common-subsequence-dp-using-memoization/. Accessed: 2022-29-01.

[2] Tabulation vs memoization. https://www.geeksforgeeks.org/tabulation-vs-memoization/. Accessed: 2022-28-01.

[3] What is dynamic programming. https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/m2G1pAq0OO0#Bottom-up-with-Tabulation. Accessed: 2022-29-01.

[4] Nabil I Buhisi and Samy S Abu-Naser. Dynamic programming as a tool of decision supporting. 2009.

[5] Daniel Kunkle. Empirical complexities of longest common subsequence algorithms. Technical report, Citeseer, 2002.

[6] Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. Parallelizing dynamic programming through rank convergence. *ACM SIGPLAN Notices*, 49(8):219--232, 2014.

[7] Hanok Palaskar and Prof. Tausif Diwan. An optimized parallel algorithm for longest common subsequence using openmp. 2016.

[8] Alex Stivala, Peter J Stuckey, Maria Garcia de la Banda, Manuel Hermenegildo, and Anthony Wirth. Lock-free parallel dynamic programming. *Journal of Parallel and Distributed Computing*, 70(8):839--848, 2010.