Факултет техничких наука

Универзитет у Новом Саду

Рачунарски системи високих перформанси

Паралелизација проблема најдуже заједничке подсеквенце

Аутор:
Дејан Допуђ

Индекс:
E2 2/2023

16. јануар 2024.

Сажетак

Две идеје којима се могу решавати комплексни алгоритамски проблеми су динамичко и паралелно програмирање. Проблем се дели на мање подпроблеме коришћењем динамичког програмирања, а одговори на те подпроблеме се затим спајају да би се дошло до закључка. Паралелно програмирање је процес извршавања неколико проблема истовремено како би се побољшала ефикасност и перформансе рачунарских система.

Ове две идеје су повезане јер се динамички програми могу оптимизовати коришћењем паралелног програмирања. Проблем се може поделити на мање компоненте и обрадити паралелно, што резултира бржим извршавањем.

Овај рад се бави разматрањем и имплементацијом ове две методе на проблему најдуже заједничке подсеквенце.

Садржај

Списак изворних кодова

Списак слика

Списак табела

1 Увод

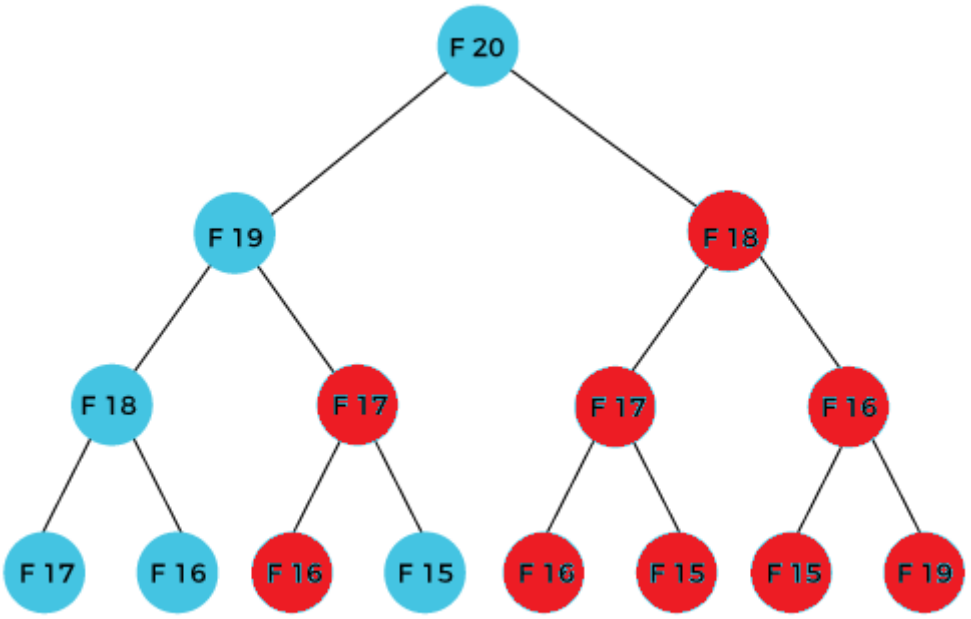
Проблем најдуже заједничке секвенце је један од најпознатијих проблема динамичког програмирања. Најједноставније се може дефинисати као тражење најдужег заједничког обрасца знакова који одржава њихов релативни редослед, чак и ако има других знакова између. Како се овај проблем решава помоћу динамичког програмирања, које процесе разлаже на подпроцесе, искаче питање да ли је могуће смањити временско извршење овог алгоритма помоћу паралелизације.

Најједноставније решење овог проблема је *brute force* решење, које не уводи никакву оптимизацију. Временска комплексност овог решења је O$(2^{n*m}))$. Како је ово очигледно неоптимално, уводи се концепт динамичког програмирања за решавање проблема. Могући приступи су *bottom up* и *top down*, где је њихова временска комплексност O$(n * m)$ [3]. Постоје два најпознатија паралелна приступа, паралелизација по дијагонали [4] и паралелизација коришћењем хеш табеле без закључавања [5].

У овом раду ће бити анализирана и имплементирана оба приступа паралелног решавања овог проблема. У првом поглављу ће бити објашњене основе динамичког програмирања, и биће представљен проблем најдуже заједничке подсеквенце. Након тога ће се прећи секвенцијална решења како би се паралелна могла лакше разумети. У поглављу 5 је дат опис и имплементација паралелних решења, и уједно објашњена додатна терминологија потребна за њихово разумевање.

2 Динамичко програмирање

Појам динамичког програмирања представља процес разлагања проблема на низ међусобно преклапајућих подпроблема и комбиновање решења мањих подпроблема како би се формирало решење за већи подпроблем. [6] Поклапајући подпроблеми престављају концепт који се односи на постојање и идентификацију мањих проблема који се појављују више пута током решавања главног проблема. Једна од кључних карактеристика динамичког програмирања, поред преклапајућих подпроблема је принцип оптималне подструктуре. Принцип оптималне подструктуре означава да оптимално решење целокупног проблема може бити састављено од оптималних решења мањих подпроблема.



Слика 1: Приказ динамичког програмирања

На слици 1 су плавом бојом представљени подпроблеми који су се решили у том кораку, док су црвеном бојом представљени подпроблеми које није имало подребе решавати, јер су већ решени у претходним подпроблемима. Ово се добија тако што када се проблем крене са извршавањем, прво провери да ли је тај проблем већ решен упитом у структуру података. У том случају се резултат добија из структуре података, најчешће хеш мапе. Уколико не постоји вредност, проблем се решава, и на крају се његово решавање уписује вредност у структуру података. Примена динамичког програмирања може значајно побољшати ефикасност алгоритама и омогућити ре-

шавање проблема који би иначе били превише захтевни. Међутим, важно је пажљиво дефинисати структуру подпроблема и правилно имплементирати механизам памћења решења ради постизања оптималних резултата.

2.1 Приступи динамичког програмирања

Top down приступ представља решавање прво надпроблема, његовим разлагањем у подпроблеме. Овај процес се извршава док се не дође до проблема који се не може даље разложити. При рекурзивном решавању подпроблема, резултати се памте (мемоизација), како се поклапајући подпроблеми не би решавали више пута и чиниле алгоритам споријим.

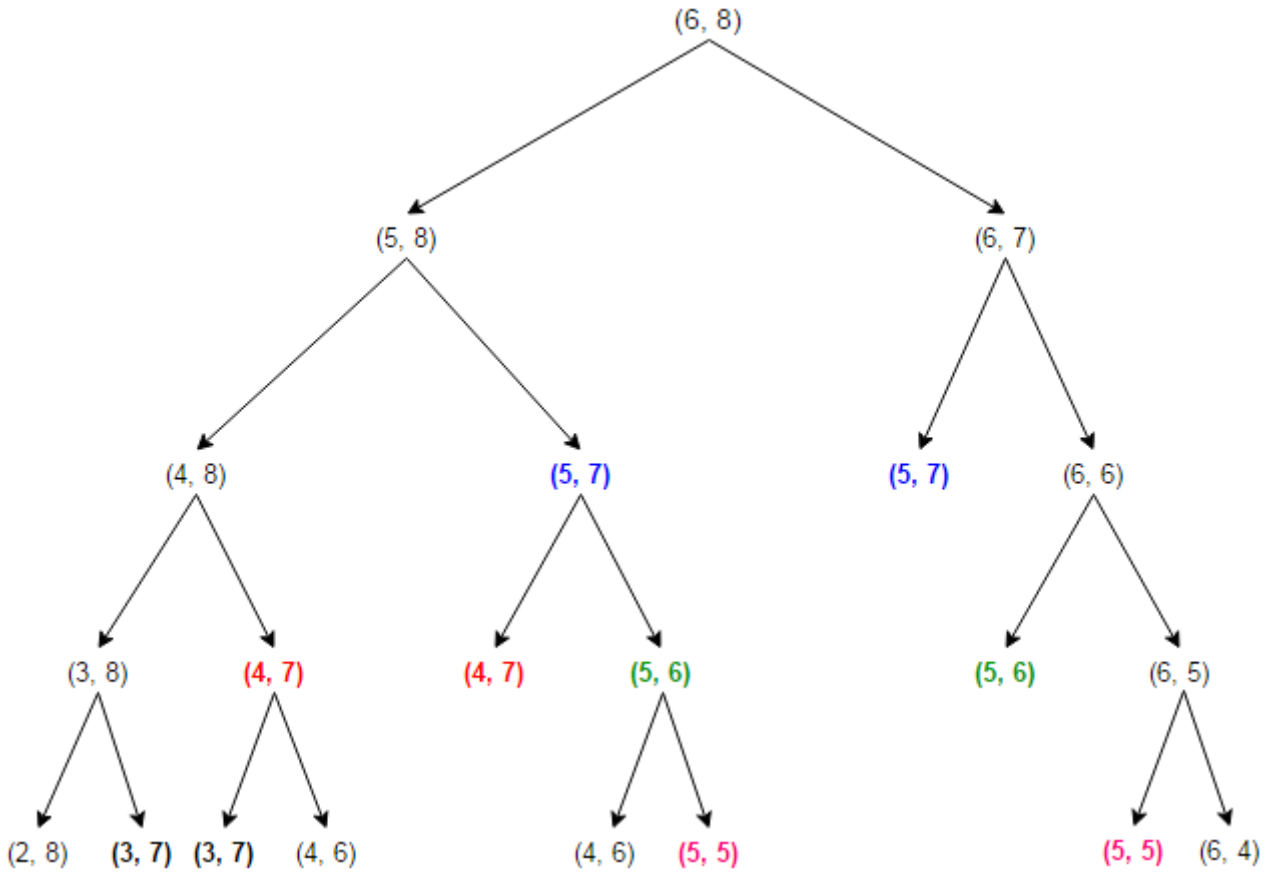
Bottom up приступ представља решавање сваког подпроблема и његово чување у структури података, све док се не дође до главног проблема. Најчешће се имплентира итеративно.

У зависности од проблема, потребно је изабрати најбољи приступ, иако су најчешће оба адекватна. *Bottom up* приступ често представља предност када је у питању ефикасност меморије, јер не користи рекурзивне позиве и омогућава директно чување резултата у табели или низу. *Top down* приступ је флексибилнији када је потребно решити само одређене подпроблеме, јер рекурзивни позиви омогућавају њихово селективно решавање.

2.2 Проблем најдуже заједничке подсеквенце

У контексту проблема најдуже заједничке подсеквенце, подсеквенца стринга је нови стринг генерисан из оригиналне секвенце са нула или више знакова обрисаних без промене релативног редоследа преосталих знакова. [2] То значи да уколико имамо секвенца1=ацдеф, секвенца2=аце, секвенца3=аец, секвенца2 јесте подсеквенце секвенце1, али секвенца3 није јер иако имају исте карактере, нису исто поређани.

Идеја за решавање овог проблема динамичким програмирањем произилиази из његових карактеристика. Овај проблем има и преклапајуће проблеме и оптималну подструктуру. Преклапајуће подпроблеме представљају подсеквенце које се могу понављати. Са слике 2 је очигледно да постоји више начина да алгоритам дође до истих подсеквенци, тј. до истих i и j параметара. Такође на слици видимо и оптималну структуру, јер је решење целокупног проблема састављено од оптималних решења подпроблема.



Слика 2: Приказ алгоритма најдуже заједничке подсеквенце (преузето са [1])

```c
int lcs(char text1[], char text2[], int i, int j) {
    if (text1[i] == '\0' || text2[j] == '\0')
        return 0;

    if (text1[i] == text2[j])
        return 1 + lcs(text1, text2, i + 1, j + 1);
    else
        return max(
            lcs(text1, text2, i + 1, j),
            lcs(text1, text2, i, j + 1)
        );
}
```

Изворни код 1: C изворни код за основну имплементацију

3 Секвенцијална решења

3.1 Без коришћења динамичког програмирања

Овај проблем се може решити секвенцијално, без коришћења динамичког програмирања, али је наравно тај начин најспорији.

Почетак је када су (i,j) ја првом карактеру у оба стринга респективно. Уколико су карактери исти, рекурзивно позивамо функцију са параметрима (i+1,j+1), тј прелазимо на наредни карактер у оба стринга и додајемо 1 на крајњи резултат након извршења позива. Уколико се карактеришу не поклапају, рекурзивно позивамо функцију са параметрима (i+1, j) и (j+1, i). То значи да ће један позив покренути функцију са наредним карактером у првом, а истим у другом стрингу, аналогно за други позив. Максимум резултата ова два позива је највећа подсеквенца.

Код 1 приказује имплементацију, и временска комплексност датог алгоритма је $O(2^{n*m})$.

```c
int lcs_top_down(char text1[], char text2[], int i, int j) {
    if (text1[i] == '\0' || text2[j] == '\0')
        return 0;

    if (dp[i][j] != -1)
        return dp[i][j];

    if (text1[i] == text2[j]) {
        dp[i][j] = 1 + lcs_top_down(text1, text2, i + 1, j + 1);
        return dp[i][j];
    } else {
        dp[i][j] = max(
            lcs_top_down(text1, text2, i + 1, j),
            lcs_top_down(text1, text2, i, j + 1)
        );
        return dp[i][j];
    }
}
```

Изворни код 2: C изворни код за *top down* имплементацију

3.2 Коришћењем динамичког програмирања

3.2.1 *Top down*

У *top down* имплементацији је искоришћено складиште *dp*, где су се чували резултати. Као што видимо у коду 2, сваки подпроблем провери да ли је исти подпроблем већ решен, уколико јесте врати решење, уколико није израчуна. Временска комплексност датог алгоритма је O$(m * n)$.

```
for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (text1[i - 1] == text2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = (dp[i - 1][j] > dp[i][j - 1]) ?
                    dp[i - 1][j] : dp[i][j - 1];
            }
        }
    }
```

Изворни код 3: C изворни код за *bottom up* имплементацију

3.2.2 *Bottom up*

У *bottom up* имплементацији је такође коришћено складиште *dp*, али као што видимо из изворног кода 3 нема рекурзије. Складиште се попуњава тако да су му висина и ширина за по један веће дужине стрингова. То омогућава попуњавање првог реда и прве колоне нулама, што олакшава даљу рачуницу. Из тог разлога, као што видимо у петљи вредности i и j почињу од јединице. Сам алгоритам за одређивање резултата је исти, и временска комплексност датог алгоритма је O$(m * n)$.

4 Паралелна решења

При решавању проблема на паралелан начин, треба издвојити независне делове кода који могу да се паралелизују. Код проблема најдуже заједничке подсеквенце, нема много независних подпроблема и из тог разлога је овај проблем нетривијалан за паралелизовати.

У разматрање за паралелину имплементацију су узете *OpenMP* и *OpenMPI* директиве. Најбитнија разлика између ових директива за дати проблем је коришћење дељене меморије. *OpenMP* је адекватнији у том смислу, јер нити у овом алгоритму деле меморију међусобно. *OpenMPI* процеси немају дељену меморију, већ сваки има свој меморијски простор. Његова предност је комуникација између процеса, али та предност није довољна да надомести непостојање дељеног меморијског простора. Иако ниједна од ових директива није идеална јер као што је речено подпроблеми нису независни, за имплементацију је одабран *OpenMP* због дељене меморије.

4.1 Паралелно рачунање по дијагоналама

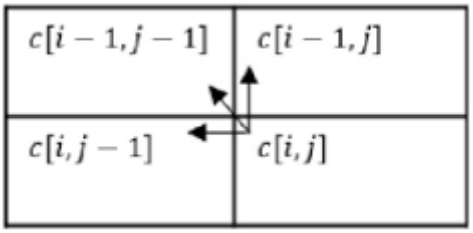
```
1     while(help < m + n ){
2         #pragma omp parallel for firstprivate(i,j)
3         for (int helpInner = 0; helpInner < my_min(i,(n-j+1));
4         helpInner++) {
5             int helpI = i-helpInner;
6             int helpJ = j+helpInner;
7             if (text1[helpI - 1] == text2[helpJ - 1]) {
8                 dp[helpI][helpJ] = dp[helpI - 1][helpJ - 1] + 1;
9             } else {
10                dp[helpI][helpJ] =
11                (dp[helpI - 1][helpJ] > dp[helpI][helpJ - 1]) ?
12                dp[helpI - 1][helpJ] : dp[helpI][helpJ - 1];
13            }
14        }
15        if (help < m){
16            help++;i++;
17        }
18        else{
19            help++;j++;
20        }
21    }
```

Изворни код 4: C изворни код за паралелну имплементацију по дијагоналама

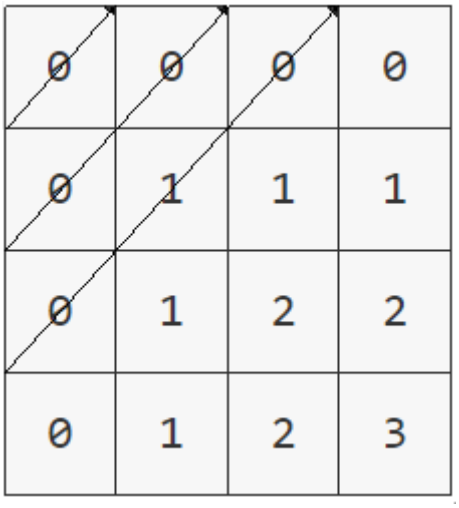
Када се детаљније проучи изворни код за *bottom up* имплементацију, примети се да је за добијање резултата у једној ћелији потребан резултат ћелије изнад(i,j-1), ћелије лево(i-1,j) и ћелије дијагонално изнад-лево (i-1,j-1), као на слици 3. Из овога се може закључити да су ћелије у дијагонали независне, и да уколико су све дијагонале лево извршене дијагонала је спремна за извршавање.

Имплементација је извршена тако што се спољашња петља се извршава $m + n$ пута, јер као што видимо на слици почетци дијагонала почињу од (1,1), настављају до (m,1), а након тога до (m,n), где m и n представљају дужине стрингова. За сваку од ових дијагонала се i смањује а j повећава како би њен правац био горе-десно, док i или j не изађу из опсега матрице, као на слици 4. За сваку дијагоналу се покреће паралелизација петље, јер је свако извршавање у дијагоналној петљи независно.

Временска комплексност датог алгоритма је O$(m + n)$, јер се спољашња петља се изврши $m + n$ пута, док је унутрашња петља паралелизована.



Слика 3: Приказ зависности имплементације по дијагонали (преузето са [2])



Слика 4: Приказ дијагонала за имплементацију по дијагонали

4.2 Паралелизација помоћу хеш табеле без закључавања

У овом решењу, свака нит решава целокупан динамички програм независно, осим што сваки пут када одреди резултат за подпроблем, смешта га у дељену хеш табелу. Такође, свака нит насумично бира подпроблеме, и сваки пут када почне израчунавање одговора за одређени подпроблем, проверава да ли резултат већ постоји у дељеној хеш табели. [5]

4.2.1 Хеш табела без закључавања

Хеш табела без закључавања је имплементирана помоћу кључне речи *struct*. Њену структуру можемо видети у изворном коду 5. Садржи величину, као и могућност чувања низа *KeyValuePair* елемената. *KeyValuePair* елемент се састоји од кључа, вредности и варијабле која нам говори да ли је место попуњено.

```
typedef struct {
    char* key;
    int value;
    int isOccupied;
} KeyValuePair;

typedef struct {
    int size;
    KeyValuePair* table;
} HashMap;
```

Изворни код 5: C изворни код структуре хеш табеле

4.2.1.1 Операција убацивања

Операција убацивања је најкомплекснији део целе имплементације, јер како се програм паралелно извршава потребно је бринути о трци до података. Почиње рачунањем јединственог кључа за потребни улаз у табели, тј. стаблу. Кључ се добија претварањем параметара i и j у стринг, и њихово спајање са карактером C. Дакле, уколико приступамо петом елементу првог стринга и седмом елементу другог стринга, кључ ће бити 7C5.

Након одређивања кључа, рачуна се његов хеш како бисмо могли приступити потребном пару кључ-вредност. За решавање дупликата хешева, имплементиран је *linear probing*. Дакле, проверава се да ли је у табели дата хеш вредност заузета, и да ли је заузета истим кључем. Уколико је заузета истим кључем, или није уопште заузета, враћа се пар кључ вредност који одговара тој хеш вредности. Уколико ниједан

од та два услова није испуњен, значи да се десило поклапање хеш вредности, и врши се *linear probing* док се не дође до првог следећег слободног места. Уколико нема више слободног места, упис је немогућ. Када добијемо пар кључ-вредност, користимо атомичну имплементацију *compare and swap*, где се проверава да ли је тренутни кључ једнак празном кључу (Код 6). Уколико јесте, уписује се нова вредност кључа и креира се пар кључ вредност који се уписује у хеш табелу. Уколико није, врши се *linear probing* док се не услов не испуни или изађе из опсега.

```
1 if ( __sync_val_compare_and_swap(&kvp->key, "-1", key)){
2     map->table[index] = createKeyValuePair(key, value);
3     return;
4 }
```

Изворни код 6: C изворни код за коришћење *compare and swap* алгоритам

4.2.1.2 Операција читања

Читање се веома једноставно имплементира, како није потребно бринути о вишеструком паралелном приступу. Уколико је место са индексом хешираног кључа попуњено, проверава се да ли се кључеви поклапају. Уколико се поклапају, успешно је нађен и прочитан пар кључ-вредност. Уколико се не поклапају, значи да је при неком од уписа дошло до *linear probing*-а, те се индекс повећава док се не поклопе или док не изађе из опсега. Уколико се поклопе у некој од наредних итерација, успешно је нађен и прочитан пар кључ-вредност, у супротном када изађе из опсега вредност није успешно прочитана.

4.2.2 Преглед и опис имплементације

Ова имплементација је паралелна надоградња *top down* имплементације. На почетку сваког позива функције, на основу креираног кључа се проверава у дељеној хеш табели уколико постоји вредност под тим кључем, тј. да ли је нека нит већ израчунала решење. Уколико јесте, вредност се врати, уколико није, функција наставља извршавање.

Када се карактери на траженим местима поклапају, обавља се рекурзивни позив врло слично као и у секвенцијалном *top down* приступу и добија резултат на основу тога. Уколико се карактери не поклапају, рачуна се насумична вредност на основу које ће се одлучити који пут ће се одабрати. Како се рачуна маскимум од рекурзивног позива са наредним карактером првог стринга и истим карактером другог и рекурзивног позива са наредним карактером другог стринга и истим карактером првог, одабир редоследа је произвољан. Уколико је насумична вредност парна, иде се

редом као што је представљено у претходној реченици, а уколико није се иде супротним редом. Када се заврше рекурзивни позиви, врати се добијена вредност.

5 Закључак

У овом раду је представљен проблем најдуже заједничке подсеквенце, као и његових разноврсних имплементација. У првом поглављу су представљене основе динамичког програмирања и проблем најдуже заједничке подсеквенце. У другом поглављу су дати опис и имплементација три секвенцијална решења, најједноставнијег, *bottom up* и *top down*. На крају су дате идеје за паралелно решавање проблема, њихове имплементације и детаљан опис.

У даљем развитку овог рада би се могло додатно оптимизовати паралелно решење одабиром правог броја нити. Дакле, наћи корелацију између улазних параметара и броја нити како би се паралелно решење оптимално извршавало.

Проблем најдуже заједничке подсеквенце није погодан за паралелизацију, највише због тога што подпроблеми нису независни. Овај рад представља енкапсулацију проблема најдуже заједничке подсеквенце, његових како паралелних тако и секвенцијалних решења, и њихово поређење.

Библиографија

[1] https://www.techiedelight.com/longest-common-subsequence/.

[2] https://cse.buffalo.edu/faculty/miller/courses/cse633/lavanya-nadanasabapathi--spring-2022.pdf.

[3] https://www.geeksforgeeks.org/longest-common-subsequence-dp-4/.

[4] Dhraief a, issaoui r, belghith a. parallel computing the longest common subsequence (lcs) on gpus: efficiency and language suitability. inthe 1st international conference on advanced communications and computation (infocomp) 2011 oct 23.

[5] Stivala a, stuckey pj, de la banda mg, hermenegildo m, wirth a. lock-free parallel dynamic programming. journal of parallel and distributed computing. 2010 aug 1;70(8):839-48.

[6] Bellman r. dynamic programming. science. 1966 jul 1;153(3731):34-7.