Fakultet tehničkih nauka
Univerzitet u Novom Sadu

Seminarski rad

Paralelizacija 0-1 problema ruksaka

Student:

Milovan Milovanović, E2-119/2022

Predmet:

Računarski sistemi visokih performansi

Sažetak

Paralelnim programiranjem se na današnjim računarima može postići višestruko poboljšanje performansi određenih algoritama. Sa druge strane, tehnika koja optimizuje rešavanje određenih problema, a ujedno je često netrivijalna u kontekstu paralelnog programiranja, jeste dinamičko programiranje. Cilj ovog rada je da na što bolji način paralelizuje rešenje jednog problema koji koristi pristup dinamičkog programiranja: 0-1 problem ruksaka.

Sadržaj

Spisak isečaka koda

Spisak slika

Spisak tabela

1 Uvod

Kako je u računarstvu trend povećavanja broja procesora odn. jezgara u računarima dominantniji i izvodljiviji od poboljšanja performansi pojedinačnih jezgara, teži se ka postizanju optimalnog iskorišćenja tih jezgara paralelizacijom klasičnih, sekvencijalnih algoritama. Neke klase algoritama su daleko problematičnije za paralelizaciju od drugih. Jedna od problematičnijih klasa jesu algoritmi dinamičkog programiranja, jer zavise od rezultata potproblema, koji treba prethodno da se izračunaju. Ovo znači da nisu dovoljne samo proste nezavisne kalkulacije u paraleli.

Cilj ovog rada je paralelizovati jedan takav algoritam koji koristi pristup dinamičkog programiranja: 0-1 problem ruksaka (engl. *0-1 knapsack problem*).

U poglavlju 2 date su teorijske osnove dinamičkog programiranja. Nakon toga, u poglavlju 3 opisan je 0-1 problem ruksaka i njegovo matematičko rešenje. U poglavlju 4 opisane su implementacije sekvencijalnog algoritma, paralelizacije unutar reda i paralelizacije na nivou blokova. Rezultati testiranja su dati u poglavlju 5. Najzad, poglavlje 6 predstavlja zaključak ovog rada.

2 Dinamičko programiranje

Dinamičko programiranje je tehnika računarskog programiranja gde se problem koji algoritam rešava prvo razlaže na potprobleme, rezultati se čuvaju, a potom se potproblemi optimizuju radi pronalaska konačnog rešenja. Ono se primenjuje na probleme koji mogu imati svojstva **preklapajućih potproblema** i **optimalne podstrukture** [4].

Kada se obimniji skup jednačina razloži na manji skup, preklapajućim potproblemima se nazivaju jednačine u kojima se više puta upotrebljava deo istih manjih jednačina kako bi se došlo do rešenja [4].

Sa druge strane, pod optimalnom podstrukturom se podrazumeva da se rešenje potproblema koristi kako bi se pronašlo rešenje čitavog problema [4].

Ova tehnika rešava probleme razlaganjem problema na manje, preklapajuće potprobleme. Rezultati rešavanja tih potproblema se čuvaju u nekoj memorijskoj strukturi (najčešće tabeli) kako bi se mogli upotrebiti kad zatrebaju u okviru rešavanja natproblema, odnosno kako bi se izbegla ponovna izračunavanja [4].

Dinamičko programiranje može imati *top-down* ili *bottom-up* pristup.

Top-down pristup najčešće podrazumeva implementaciju rešenja modifikacijom obične rekurzije tako da se rešenje svakog potproblema čuva u memorijskoj strukturi. Prvo se proverava u okviru memorijske strukture (tabele) da li je problem već rešen. Ako jeste, vraća se sačuvana vrednost i time izbegava ponovno izračunavanje. Ako nije, problem se rešava na uobičajen način [5].

Sa druge strane, *bottom-up* pristup je obrnut, odnosno najčešće predstavlja iterativnu verziju *top-down* pristupa. Zavisi od prirodne ideje da rešenje bilo kog problema može zavisiti samo od rešenja manjih potproblema. Naime, problemi se sortiraju od najmanjeg do najvećeg (konačnog) i iterativno rešavaju u tom redosledu. Drugim rečima, kada se rešava određeni potproblem, *bottom-up* pristup prvo reši sve manje potprobleme od kojih može zavisiti i čuva vrednosti rešenja u tabeli [5].

3 0-1 problem ruksaka

3.1 Definicija

Problem ruksaka je problem kombinatorne optimizacije: ako je dat skup nekih predmeta, svaki sa svojom vrednošću i težinom, potrebno je odrediti koje predmete treba staviti u ruksak tako da je ukupna zbirna težina tih predmeta manja od kapaciteta ruksaka, a ukupna vrednost predmeta u njemu maksimalna, odnosno treba odrediti takvu optimalnu ukupnu vrednost [1].

Ukoliko je u pitanju 0-1 problem ruksaka, to znači da nije dozvoljeno staviti frakciju nekog predmeta u ruksak, kao ni više istog predmeta. Tačnije, predmet se ili stavlja u ruksak ili se ne stavlja.

3.2 Rešenje pristupom dinamičkog programiranja

Neka je dat skup n predmeta numerisanih od 1 do n, svaki sa težinom w_i i vrednošću v_i, kao i ukupan kapacitet ruksaka W. Neka su $w_1, w_2, \dots, w_n, W$ striktno pozitivni celi brojevi. Definišimo $m[i, w]$ kao maksimalnu vrednost koja se može dobiti sa težinom jednakom ili manjom od w koristeći predmete do i (prvih i predmeta) [1].

Možemo definisati $m[i, w]$ rekurzivno na sledeći način [1]:

- $m[0, w] = 0$
- $m[i, w] = m[i-1, w]$ ako $w_i > w$ (ako je posmatrani predmet teži od trenutnog kapaciteta)
- $m[i, w] = \max(m[i-1, w], m[i-1, w-w_i] + v_i)$ ako $w_i \leq w$.

Rešenje u vidu maksimalne ukupne vrednosti se tada može dobiti izračunavanjem $m[n, W]$. Kako bismo povećali efikasnost ovog algoritma, možemo upotrebiti tabelu da sačuvamo prethodna izračunavanja, što je tipično za algoritam dinamičkog programiranja.

4 Implementacija rešenja

4.1 Sekvencijalno rešenje

```
int knapsack_bottom_up_seq(int capacity, int weights[], int values[], int
num_items)
{
    int i, w;

    int (*table) [capacity+1] = malloc(sizeof(int[num_items+1][capacity+1]));

    for (i = 0; i <= num_items; i++) {
        for (w = 0; w <= capacity; w++) {
            if (i == 0 || w == 0)
                table[i][w] = 0;
            else if (weights[i-1] <= w)
                table[i][w] = max(table[i-1][w], table[i-1][w-weights[i-1]] +
values[i-1]);
            else
                table[i][w] = table[i-1][w];
        }
    }

    int ret_val = table[num_items][capacity];

    free(table);

    return ret_val;
}
```

Isečak koda 1: Sekvencijalna bottom-up implementacija

Na isečku koda 1 prikazana je sekvencijalna implementacija algoritma dinamičkog programiranja za rešavanje 0-1 problema ruksaka. Algoritam se oslanja na način rešavanja objašnjen u prethodnom poglavlju, s tim što je implementiran *bottom-up* pristup, gde se iterativno popunjava prethodno pomenuta tabela izračunavanja od baznih slučajeva ka „natproblemima“.

Ulaz u funkciju je celobrojna vrednost kapaciteta ruksaka, celobrojni nizovi težina i vrednosti predmeta i ukupan broj predmeta. Spoljašnja *for* petlja služi za iteriranje kroz redove tabele, koji predstavljaju redne brojeve predmeta koji se razmatraju, dok unutrašnja *for* petlja služi za iteriranje kroz kolone tabele, koje predstavljaju kapacitete ruksaka koji se razmatraju. Izlaz iz funkcije je poslednja ćelija tabele, što je maksimalna vrednost koja se može dobiti za ruksak predstavljen datim ulaznim parametrima.

Vremenska (a i prostorna) kompleksnost ovog algoritma je $O(nW)$ [2].

4.2 Paralelizacija unutar reda

S obzirom na to da izračunavanje elemenata u tabeli zavisi od prethodno izračunatih elemenata, ovo sekvencijalno rešenje nije moguće trivijalno paralelizovati na nivou cele tabele tako da svaka jedinica izvršavanja odn. nit radi nezavisne operacije. Konkretno, kad je u pitanju red (indeks i), on u većini operacija zavisi od prethodnog reda ($i-1$), te nije moguće paralelizovati spoljašnju petlju (koja je zadužena za red), jer se ne može garantovati takav redosled izračunavanja. Zamena petlji takva da spoljašnja bude zadužena za kolonu takođe ne bi bila skladna za paralelizaciju, jer se u jednoj od operacija indeks kolone takođe menja.

Paralelizacija koja se može primeniti (upotrebom *OpenMP*-a) na ovakvo sekvencijalno rešenje data je na isečku koda 2.

```
int knapsack_bottom_up_parallel(int capacity, int weights[], int values[],
int num_items)
{
    int i, w;

    int (*table) [capacity+1] = malloc(sizeof(int[num_items+1][capacity+1]));

    for (i = 0; i <= num_items; i++) {
        #pragma omp parallel for shared(i) private(w) schedule(static)
        for (w = 0; w <= capacity; w++) {
            if (i == 0 || w == 0)
                table[i][w] = 0;
            else if (weights[i-1] <= w)
                table[i][w] = max(table[i-1][w], table[i-1][w-weights[i-1]] +
values[i-1]);
            else
                table[i][w] = table[i-1][w];
        }
    }

    int ret_val = table[num_items][capacity];

    free(table);

    return ret_val;
}
```

Isečak koda 2: Paralelizacija unutar reda

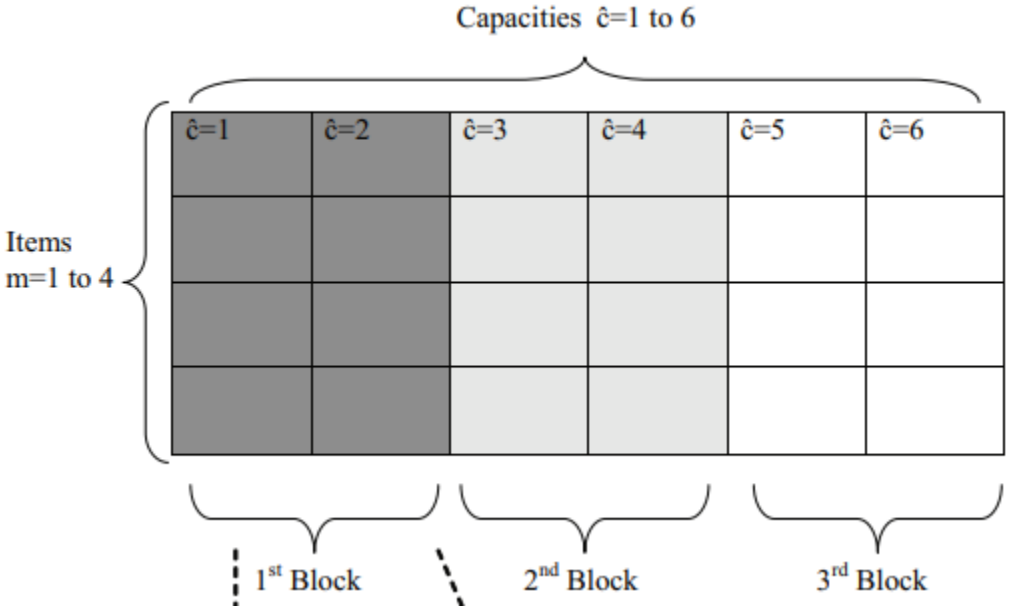
Naime, dodavanjem *OpenMP* direktive za paralelizaciju *for* petlje ideja je da se u svakoj iteraciji reda iznova paralelizuju izračunavanja unutar tog reda. Ovo je moguće postići zato što su izračunavanja unutar reda nezavisna od ostalih elemenata u tom redu, a sekvencijalnim

izvršavanjem spoljašnje petlje garantujemo da će prethodni redovi uvek u potpunosti biti izračunati pre nego što niti započnu svoj paralelni rad.

Načelno, mana ovog pristupa je to što će potencijalno biti skupo kreiranje niti za svaku iteraciju reda, te će biti najpogodnije u slučajevima kad je broj kolona znatno veći od broja redova, odnosno kad je kapacitet ruksaka znatno veći od broja predmeta u opticaju. U slučaju kad je kapacitet ranca mali, a razmatra se velik broj predmeta, ova paralelizacija će sigurno više štete doneti nego koristi kada su u pitanju performanse.

4.3 Paralelizacija na nivou blokova

Rešenje inspirisano pristupom u [3] se oslanja na prethodni pristup, ali nastoji da ublaži njegove mane. Naime, ideja je da se uvedu blokovi (fiksne particije tabele) koji se namapiraju na jedinice izvršavanja. Jedan blok je u okviru svake iteracije reda zadužen za jedan te isti interval kolona.



Slika 1: Primer podele na blokove [3]

Na slici 1 je dat primer podele tabele na blokove. U okviru svakog reda prvi blok je zadužen za kapacitete 1 i 2, drugi blok za kapacitete 3 i 4 i treći blok za kapacitete 5 i 6. Opštije, prvi blok pokriva kolone $[1, veličina\_bloka]$, drugi kolone $[veličina\_bloka + 1, 2 * veličina\_bloka]$ itd.

Na isečku koda 3 se može videti konkretna implementacija ovog rešenja. Može se primetiti da je primenjena *OpenMP* direktiva za stvaranje paralelnog regiona na nivou koji obuhvata obe petlje (ovim se, takođe, niti jedan-na-jedan mapiraju na blokove). Kako bi ovaj način paralelizacije bio pravilno omogućen, unutrašnja *for* petlja, zadužena za iteriranje kroz kapacitete, modifikovana je tako da blok (nit) iterira samo kroz prostor reda za koji je zadužen, odnosno iterira se od rednog broja bloka (broj *OpenMP* niti) pomnoženog sa veličinom bloka do

rednog broja bloka uvećanog za 1 pomnoženog sa veličinom bloka. Takođe, stavljena je eksplicitna barijera pred kraj spoljašnje petlje kako bi svaka nit završila izračunavanje u trenutnoj iteraciji reda pre nego što se nastavi izračunavanje za sledeći red.

U smislu performansi, cilj ovog pristupa je da se minimizuje cena kreiranja niti tako što se samo jednom kreiraju pre izvršavanja algoritma. Preferencija za slučajeve kada je kapacitet ranca velik nepromenjena je u odnosu na prethodni pristup.

```
int knapsack_bottom_up_parallel_blocks(int capacity, int weights[],
int values[], int num_items)
{
    int i, w;

    int (*table) [capacity+1] = malloc(sizeof(int[num_items+1][capacity+1]));

    int num_blocks = omp_get_max_threads();
    printf("%d\n", num_blocks);

    #pragma omp parallel private(i, w) num_threads(num_blocks)
    {
        for (i = 0; i <= num_items; i++) {
            int block = omp_get_thread_num();
            for (w = block * capacity/num_blocks; w < block+1 *
capacity/num_blocks, w <= capacity; w++) {
                if (i == 0 || w == 0)
                    table[i][w] = 0;
                else if (weights[i-1] <= w)
                    table[i][w] = max(table[i-1][w], table[i-1][w-weights[i-
1]] + values[i-1]);
                else
                    table[i][w] = table[i-1][w];
            }

            #pragma omp barrier
        }
    }

    int ret_val = table[num_items][capacity];

    free(table);

    return ret_val;
}
```

Isečak koda 3: Paralelizacija na nivou blokova

5 Rezultati testiranja

U ovom poglavlju prikazani su rezultati testiranja implementiranih rešenja i upoređene su performanse različitih pristupa. Svi rezultati su dobijeni testiranjem na Ubuntu 22.04.1 LTS virtuelnoj mašini sa 4 jezgra Intel i7-8700K procesora i 4GB 3200MHz DDR4 radne memorije.

	Sekvencijalno	Paralelizacija unutar reda	Blokovi
800 x 100	0,00037	0,10986	0,00214
800 x 10000	0,06540	0,05808	0,0676
800 x 25000	0,12852	0,14513	0,15099
800 x 100000	0,53191	0,17385	0,46654
800 x 250000	1,25627	0,33595	1,21396

Tabela 1: Rezultati testiranja

U tabeli 1 su dati rezultati testiranja izraženi u sekundama. Kolone predstavljaju sekvencijalno rešenje, paralelizaciju unutar reda i paralelizaciju na nivou blokova, respektivno. Redovi predstavljaju veličinu tabele (gde je levo broj redova, a desno broj kolona), koja se dobija brojem predmeta koji se razmatraju za stavljanje u ruksak (redovi) i kapacitetom ruksaka (kolone). Npr. 800 x 100 znači da se razmatra 800 predmeta za ruksak kapaciteta 100. Paralelni algoritmi su pokretani sa 4 niti svaki put, što odgovara broju jezgara na virtuelnoj mašini.

Prikazani rezultati potvrđuju pretpostavke u prethodnom poglavlju kada je u pitanju paralelizacija u slučaju malog kapaciteta ruksaka. Naime, sve do kapaciteta 100000, kad se dobija ubrzanje od 3,06 puta u odnosu na sekvencijalni algoritam, ne primećuje se poboljšanje performansi prilikom paralelizacije. Razlog je što je cena kreiranja i upravljanja nitima veća nego njihova korist. Kada se kapacitet poveća na 250000, paralelizacijom unutar reda dobija se ubrzanje od 3,74 puta u odnosu na sekvencijalni algoritam, što je relativno blizu idealnom ubrzanju od 4 puta za taj broj niti.

Ono što iznenađuje u rezultatima jesu performanse paralelizacije na nivou blokova, koja se nije pokazala kao poboljšanje paralelizacije unutar reda koje je nastojala da bude. Naime, performanse paralelizacije na nivou blokova su prilikom testiranja uvek komparabilne performansama sekvencijalnog algoritma, čak i kad je broj kolona veoma velik. Po svoj prilici, *OpenMP* konstrukcija za deljenje posla na nivou petlje je optimalnija nego ručno definisanje paralelnog regiona i upotreba eksplicitne barijere, čak i kad se ta konstrukcija koristi unutar petlje.

6 Zaključak

U ovom radu istražena je mogućnost paralelizacije algoritma dinamičkog programiranja za rešavanja 0-1 problema ruksaka. Slično većini drugih problema dinamičkog programiranja, paralelizacija nije idealna. Ipak, s obzirom na to da red u tabeli u *bottom-up* pristupu zavisi samo od prethodnog reda, moguće je paralelizovati izračunavanja unutar jednog reda upotrebom *OpenMP* direktive za paralelizaciju (unutrašnje) petlje, čime se dobijaju dobre performanse ukoliko je kapacitet ruksaka, odnosno broj kolona, dovoljno velik.

Pokušan je i pristup baziran na blokovima koji je nastojao da poboljša mane prethodne paralelizacije, gde bi se izbeglo ponovno kreiranje niti nakon svake iteracije reda. Međutim, nije se pokazao kao poboljšanje u smislu performansi.

7 Bibliografija

[1] Knapsack problem. [Online]. https://en.wikipedia.org/wiki/Knapsack_problem. Accessed: 2023-01-11

[2] 0-1 Knapsack Problem | DP-10. [Online]. https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/. Accessed: 2023-01-11

[3] Hammad Rashida, Clara Novoab, Mark McKenneya, Apan Qasema. Efficient Parallel Solutions to the Integral Knapsack Problem on Current Chip-multiprocessor Systems. *The International Journal of Parallel, Emergent and Distributed Systems*, 2010

[4] What is Dynamic Programming? Working, Algorithms, and Examples. [Online]. https://www.spiceworks.com/tech/devops/articles/what-is-dynamic-programming/. Accessed: 2023-01-18

[5] Top-down vs Bottom-up approach in Dynamic Programming. [Online]. https://www.enjoyalgorithms.com/blog/top-down-memoization-vs-bottom-up-tabulation. Accessed: 2023-01-18