Факултет техничких наука

Универзитет у Новом Саду

Рачунарски системи високих перформанси

Имплементација и упоредна анализа секвенцијалне и паралелне обраде текста применом Рабин-Карп алгоритма

Аутор:
Андрија Станишић

Индекс:
E2 105/2023

16. јануар 2024.

Сажетак

Широм света свакодневно се генеришу астрономске количине података, од којих значајан део чине текстуални подаци. Ове податке је неретко неопходно претражити и анализирати ради екстраховања вредних информација. У том контексту, развијени су разни алгоритми чији је циљ да олакшају и убрзају овај процес. Рабин-Карп алгоритам је један од њих, и одликован је изузетном ефикасношћу решавања проблема идентификације образаца у склопу текста. Међутим, када је реч о великим количинама података, секвенцијална природа Рабин-Карп-а може бити ограничавајући фактор за перформансе. Стога, ово истраживање има за циљ да првенствено идентификује делове алгоритма који су погодни за паралелизацију, и затим их унапреди коришћењем различитих приступа. Конкретно, Рабин-Карп алгоритам паралелизован је коришћењем OpenMP и OpenMPI технологија. На самом крају овог истаживачког рада пружена је упоредна анализа перформанси различитих верзија имплементација, која настоји да укаже на значајна побољшања које доноси паралелизам.

Садржај

1 Увод

Обрада текста игра кључну улогу у различитим аспектима информационих технологија, попут претраживања и анализе великих скупова података. Међутим, обим доступних текстуалних информација свакодневно расте, што директно утиче на потребу за све ефикаснијим и перформантнијим начинима за њихову обраду. У том контексту, ефикасни и брзи алгоритми за претрагу текста почињу да добијају на значају, и потенцијално могу донети велику тржишну вредност.

Секвенцијална обрада текста релативно је једноставна за имплементацију, међутим суочава се са бројним изазовима, посебно када је реч о великим количинама текста. Време одзива би био један од непремостивих изазова секвенцијалне имплементације, што је у данашњем дигиталном добу недопустиво. Поред овога, недовољно искоришћење рачунарских ресурса такође представља великуману секвенцијалног приступа. Данашње машине пројектоване су тако да пруже невероватне перформансе, и често су њихове централне процесорксе јединице изграђене од мноштва моћних језгара. Секвенцијалани алгоритми, због свог дизајна, не могу користити овај потенцијал, што резултује неефикасним коришћењем расположивих капацитета.

Управо због ових ограничења, паралелни приступ обради великих количина података постаје кључан у датом контексту. Паралелизија омогућава истовремено процесирање података на више језгара, чиме се знатно скраћује време обраде и повећава ефикасност коришћења ресурса. Због свега претходно наведеног, Рабин-Карп алгоритам (енгл. Rabin-Karp Algorithm) изабран је као централни део овог истраживачког рада због његове утврђене ефикасности у обради текстуалних података, са посебним нагласком на детекцију шаблона у оквиру обимних текстуалних скупова.

Циљ овог рада је да кроз имплементацију секвенцијалне и паралелне верзије Робин-Карп алгоритма, покаже значајне разлике у ова два приступа приликом претраживања великих текстуалних датотека. За паралелизацију алгоритма коришћена је OpenMP библиотека као и OpenMPI стандард, које представљају уобичајене парадигме у свери паралелног програмирања. Конкретно, OpenMP ће бити примењен за постизање паралелизма на нивоу нити унутар једног процесора, док ће се OpenMPI стандард користити за обраду текста помоћу више процесора. Такође, свака од поменутих имплементација биће детаљно анализирана како би се јасно разграничиле предности и мане сваке од њих. На самом крају, кроз серију тестова, рад ће представити конкретне закључке о перформансама различитих верзија истог алгоритма.

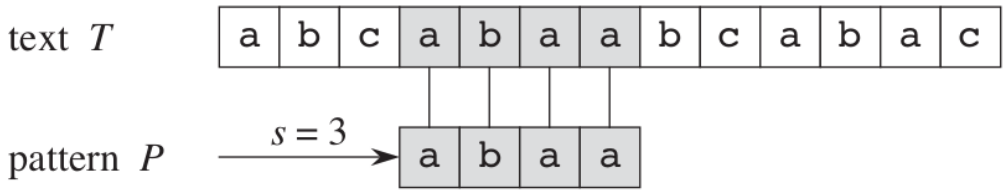
2 Формализација проблема

Програми за обраду текста често имају задатак да идентификују све инстанце одређеног обрасца унутар текста. У већини случајева, текст је документ који се уређује, а образац који се тражи представља специфичну реч коју уноси крајњи корисник. Ефикасни алгоритми за ову врсту проблема, познатији као алгоритми за претрагу текста, могу значајно допринети брзини одзива система у датом контексту.

Формализација овог проблема захтева увођење одређених термина. Текст који се анализира представљен је нотацијом $T[1..n]$, где n представља дужину улазног низа. Обрасци које је потребно идентификовати означени су као $P[1..m]$, где m представља дужину конкретног обрасца. Како би проблем имао решења, мора бити задовољен услов $m \leq n$. Такође, у основи овог проблема узета је претпоставка да сви елементи у низовима P и T потичу из исте коначне азбуке, означене помоћу Σ нотације [1]. За образац се сматра да се налази у иницијалном тексту , са неким померајем s, ако и само ако важи следећа једнакост:

$$T[s+1..s+m] = P[1..m] \text{ за } 1 \leq s \leq n-m$$

Уколико је једнакост задовољена, за s се сматра да је валидан померај, у супротном s се озачава као неважећи. Дакле, проблем претраге низова карактера своди се на проналажење свих валидних помераја са којима се дефинисани образац P појављује у одређеном тексту T. Илустративан пример проблема приказан је на слици 1. Будући да се образац P појављује једном у склопу текста T, постоји само један валидан померај s.



Слика 1: Пример проблема претраге низа карактера.

3 Рабин-Карп алгоритам

Рабин-Карп алгоритам представља софистициран метод претраге текста. Основна стратегија овог алгоритма почива на концепту хеширања (енгл. hashing), који омогућава ефикасно препознавање потенцијалних подударања у тексту. Алгоритам се састоји из две кључне фазе које укључују претпроцесирање и претрагу.

3.1 Фаза претпроцесирања

У првој фази, односно у фази претпроцесирања, алгоритам израчунава хеш вредност за образац који је неопходно идентификовати у склопу текста. Овај корак је кључан јер хеширање омогућава алгоритму да ефикасно идентификује могућа подударања делова текста и дефинисаног обрасца. Хеш функција првенствено трансформише низ карактера у одговарајућу нумеричку вредност, на пример коришћењем ASCII вредности карактера. Како би се избегли проблеми са великим бројевима, примењује се модуларна аритметика. Овај приступ обично укључује коришћење великог простог броја као модула, чиме се смањује вероватноћа да два различита низа добију исту хеш вредност.

3.2 Фаза претраге

Након претпроцесирања, алгоритам прелази у фазу претраге текста. У овој фази, алгоритам рачуна хеш вредност за сваки подниз у тексту, који је по дужини једнак обрасцу који се тражи, користећи исту методу хеширања као и у фази претпроцесирања. Ова стратегија је веома корисна јер омогућава алгоритму да ефикасно идентификује све поднизове који се сигурно не поклапају са траженим обрасцем. Наиме, уколико се њихова модуларна вредност разликује, алгоритам једноставно прескаче одређени подниз и наставља даље да претражује. Међутим, овај приступ има своје изазове, посебно када је реч о "лажним подударањима". Лажно подударање се дешава када различити поднизови текста имају исту модуларну вредност као и тражени образац, али се суштниски разликују. У таквим ситуацијама, алгоритам мора проверити сваки карактер тих поднизова појединачно како би утврдио да ли стварно постоји подударање са обрасцем. Иако овај процес осигурава прецизност у идентификацији подударања, захтева додатно време прорачунавања.

Рабин-Карп алгоритам захтева време претпроцесирања које износи $O(m)$, а његово време претраге у најгорем случају износи $O((n - m + 1)m)$, где је n дужина текста, а m дужина траженог обрасца. Овако комплексна временска сложеност обично се јавља у ситуацијама када постоји велики број лажних подударања. Иако је такав сценарио теоретски могућ, у пракси се ретко дешава, што доводи до тога да алгоритам у већини случајева постиже знатно боље перформансе. У нормалним околностима, време извршавања алгоритма може бити скоро линеарно, захваљујући употреби хеш функција у процесу претраге.

4 Имплементација

У наредним секцијама биће пружен детаљнији увид у кључне аспекте секвенцијалне имплементације Рабин-Карп алгоритма, заједно са прегледом паралелних имплементација реализованих помоћу OpenMP библиотеке и OpenMPI стандарда. Другим речима, у наредном делу овог истраживачког рада илустрован је начин на који се теоријски принципи датог алгоритма, који су описани у претходним секцијама, реализују на проблему из праксе.

4.1 Секвенцијални приступ

Секвенцијална имплементација служи као основа за детаљније разумевање алгоритма. Додатно, овај приступ је изузетно битан за идентификацију сегмената алгоритма који ефикасно могу бити паралелизовани. Детаљним испитивањем овог приступа, могуће је значајно унапредити ефикасност и перформансе. Другим речима, секвенцијална имплементација представља одличну полазну тачку за анализу, као и за развој напреднијих и перформантнијих паралелних верзија алгоритма.

Најпре је неопходно описати главну функцију (енгл. main function), која представља корен програма. У овој функцији дефинисани су важни параметри, који укључују обрасце за претрагу, модуо за хеширање, као и подаци који ће бити претраживани. Што се тиче података од интереса, генерисани су помоћу Python скрипте, њихова величина је око 500MB, и садрже секвенце које илуструју корисничку интеракцију са PS5 контролером током онлајн играња видео игара. Такође, у склопу $main$ функције имплементирано је и мерење времена одзива, чији ће резултати касније послужити за анализу и формирање конкретних закључака.

```c
int main()
{
    struct timeval start, end;

    printf("##################################\n");
    printf("Initiating␣the␣sequential␣algorithm...\n");
    char *filepath = "../scripts/controller_input.txt";
    char *buffer = read_file(filepath);
    char *patterns[] = {"R1L1X", "R1XO", "OYO"};
    int mod = 101;

    gettimeofday(&start, NULL);
    search(patterns, 3, buffer, mod);
    gettimeofday(&end, NULL);

    double elapsed_time = (end.tv_sec - start.tv_sec);
    elapsed_time += (end.tv_usec - start.tv_usec) / 1000000.0;
    printf("Execution␣time:␣%f␣seconds\n", elapsed_time);

    free(buffer);
    printf("Algorithm␣completed.\n");
    printf("##################################\n");
    return 0;
}
```

Listing 1: Полазна тачка секвенцијалне имплементације

Следећи корак је имплементација функције задужене за претрагу. Параметри ове функције су: образци које је потребно идентификовати, њихов број, текст који се претражује и модуо који се користи за хеширање.

```c
void search(char *patterns[], int num_patterns,
            char *txt, int mod){}
```

Listing 2: Дефиниција функције за претрагу

Као што је већ раније описано, Рабин-Карп алгоритам се састоји из две фазе: препроцесирање и претрага. Пратећи овај теоријски оквир, функција претраге се такође дели на два одвојена блока кода, сваки специјализован за извођење одређене фазе. Обе фазе налазе се унутар главне *for* петље, како би се омогућило претраживање за сваки образац посебно. Следећи код илуструје фазу претпроцесирања. Најпре је дефинисана главна *for* петља која изршава претраживање онолико пута колико

има образаца. Затим се иницијализују промењиве које садрже информације о обрасцу који се тренутно претражује, заједно са његовом дужином и укупном дужином прослеђеног текста. Након тога, одређује се вредност промењиве h која је задужена за ефикасно ажурирање хеш вредности приликом померања прозора претраге кроз текст. Вредност промењиве h могла је директно да се одреди као $h = d^{M-1}\%mod$, где d представља величину азбуке из које потичу улазни подаци. Међутим овај приступ може довести до проблема прекорачења опсега $integer$ типа, због чега је израчунавање тежине првог карактера у шаблону одрађено итеративно. Када се прозор претраге помери за један карактер, неопходно је ажурирати хеш вредност. Како би се овај процес ефикасно извршио, одузима се вредност старог првог карактера помноженог са h и додаје се вредност новог карактера. На овај начин постигнут је $rolling - hash$, који омогућава ефикасно ажурирање хеш вредности.

Последњи корак у фази претпроцесирања подразумева израчунавање хеш вредности образца и иницијалног прозора претраге. Хеш вредност сваког карактера добија се множењем величине улазне азбуке са хеш вредности претходног карактера и додавањем ASCII вредности наредног. Затим се ради модуо ове вредности како би се избегао рад са великим бројевима и смањила могућност поклапања хеш вредности.

```
void search(char *patterns[], int num_patterns,
char *txt, int mod)
{
    int N = strlen(txt);
    for (int k = 0; k < num_patterns; k++)
    {
        char *pat = patterns[k];
        int M = strlen(pat);
        int i, j;
        int hash_p = 0, hash_t = 0, count = 0, h = 1;

        for (i = 0; i < M - 1; i++)
            h = (h * d) % mod;

        for (i = 0; i < M; i++)
        {
            hash_p = (d * hash_p + pat[i]) % mod;
            hash_t = (d * hash_t + txt[i]) % mod;
        }
 }
```

Listing 3: Фаза претпроцесирања

Наредни блок кода представља имплементацију завршне фазе Рабин-Карп алгоритма, односно део алгоритма где се заправо реализује претраживање. Дефинисана је главна for петља која итерира кроз цео текст. Затим се пореде хеш вредности образца и тренутног прозора претраживања. Уколико су они једнаки, неопходно је извршити додатну проверу сваког појединачног карактера због потенцијалног појављивања проблема лажног поклапања. У случају да нема поклапања, ажурира се хеш вредност прозора претраживања коришћењем раније дефинисане методе, под називом $rolling - hash$. Укључена је и додатна провера која онемогућава да хеш вредност буде негативна, што је битно у контексту постизања доследности алгоритма.

```
//preprocessing phase...

{
    for (i = 0; i <= N - M; i++)
    {
        if (hash_p == hash_t)
        {
            for (j = 0; j < M; j++)
            {
                if (txt[i + j] != pat[j])
                    break;
            }

            if (j == M)
            {
                count++;
            }
        }

        if (i < N - M)
        {
            hash_t = (d * (hash_t - txt[i] * h) + txt[i + M]) % mod;
            if (hash_t < 0)
                hash_t = (hash_t + mod);
        }
    }
}
```

Listing 4: Фаза претраге

4.2 Паралелни приступ

Након темељне анализе секвенцијалног приступа, идентификовани су делови алгоритма који су подложни паралелизацији. У наредној секцији рад ће се фокусирати на имплементацију паралелне верзије Рабин-Карп алгоритма применом OpenMP и OpenMPI приступа. Ова секција представља централни део истаживања и има за циљ да објасни и прикаже како се примена ових технологија одражава на ефикасност и перформантност самог алгоритма. Претпоставка је да ће паралелни приступ значајно убрзати извршавање алгоритма, што може бити корисно приликом обраде великих скупова података на вишенитним машинама или у дистрибуираним системима.

4.2.1 OpenMP

Наредна секција посвећена је анализи паралелне имплементације Рабин-Карп алгоритма коришћењем OpenMP приступа. Ова имплементација омогућава ефикасну поделу посла између више нити једне процесорске јединице. Као и у случају секвенцијалне имплементације, основни корак је дефинисање $main$ функције. Међутим, у паралелној верзији, постоји разлика само у начину мерења одзива система. Стога, је на следећем исечку кода приказан начин мерења коришћен у паралелном приступу, док је остатак кода еквивалентан оном у секвенцијалном приступу.

```
{
    // rest of the code...
    double start = omp_get_wtime();
    search(patterns, 3, buffer, mod);
    double end = omp_get_wtime();

    printf("Execution time: %f seconds\n", end - start);
    return 0;
}
```

Listing 5: Полазна тачка OpenMP имплементације

Део кода где је извршена паралелизација налази се у $search$ функцији. Основна идеја паралелизације је дистрибуција сегмената улазног текста на више нити. Овакав приступ омогућава ефикасну паралелну обраду, јер свака нит ради са одређеним, непрекидним сегментом текста. Међутим, постоји потенцијални проблем који настаје уколико се тражени образац налази тачно на граници између сегмената који се шаљу на обраду различитим нитима. Овај проблем се решава тако што се нитима

омогући да претражују карактере ван свог сегмента, како би се осигурала доследност алгоритма. Наравно треба обратити пажњу да нит која претражује последњи сегмент не сме имплементирати ово понашање. Дакле, због претходно наведених разлога за ефикасну паралелизацију одабрана је комбинација примитива $parallel for$ и $reduction$. Другим речима, нитима се равномерно расподељују сегменти текста, након чега их оне паралелно обрађују и на крају се резултати паралелних обрада агрегирају и смештају у $count$ промењиву. Паралелна верзија $search$ функције приказана је на следећем исечку.

```
#pragma omp parallel for reduction(+ : count)
    for (int i = 0; i < N; i++)
    {
        int hash_t = 0;
        for (int j = 0; j < M; j++)
        {
            if (i + j < N)
                hash_t = (d * hash_t + txt[i + j]) % mod;
        }

        if (hash_p == hash_t && i <= N - M)
        {
            int found = 1;
            for (int j = 0; j < M; j++)
            {
                if (txt[i + j] != pat[j])
                {
                    found = 0;
                    break;
                }
            }
            count += found;
        }
    }
```

Listing 6: OpenMP верзија функције претраге

OpenMP нуди различите конструкције за паралелизацију, међу којима се налази и $task$ конструкција. Међутим, употреба овог приступа у контексту паралелизације Рабин-Карп алгоритма није посебно оптимална из неколико разлога. Првенствено, $task$ конструкција је најбоље примењива на скупове независних задатака који нису униформно распоређени по времену извршавања или сложености. Насупрот томе,

у претрази текста, рад са различитим сегментима текста представља поприлично униформне задатке. Додатно, за разлику од $task$ конструкције $parallel for$ омогућава ефикаснију синхронизацију и редукцију резултата. Ово је од суштинског значаја за тачно утврђивање укупног броја идентификованих образаца, док би употреба $task$ конструкције могла учинити овај процес доста компликованијим и мање ефикасним.

4.2.2 OpenMPI

У овој секцији представљени су најзначајнији аспекти паралелне имплементације Рабин-Карп алгоритма помоћу OpenMPI стандарда. Оваква имплементација омогућава ефикасну поделу задатака на више процеса унутар дистрибуираног система. Коришћење OpenMPI стандарда омогућава алгоритму да искористи могућност вишепроцесорске обраде, што може бити веома корисно у контексту обраде великих количина текстуалних података. Централна идеја ове имплементације је веома слична оној описаној у претходном поглављу. Улазни текстуални скуп података се сегментира и шаље свим активним процесима. Конкретан пример имплементације приказан је на следећем исечку.

```
{
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int text_size;
    if (rank == 0)
    {
        start_time = MPI_Wtime();
        text = read_file("../scripts/controller_input.txt");
        text_size = strlen(text);
    }
    MPI_Bcast(&text_size, 1, MPI_INT, 0, MPI_COMM_WORLD);
    int segment_size = text_size / size;
    char *segment = malloc(segment_size + 1);

    MPI_Scatter(text, segment_size, MPI_CHAR,
    segment, segment_size, MPI_CHAR, 0, MPI_COMM_WORLD);
    segment[segment_size] = '\0';
}
```

Listing 7: Полазна тачка OpenMPI имплементације

Сваки од процеса затим извршава функцију претраге, која је сада модификована тако да враћа вредност типа $integer$. Ова вредност представља број идентификованих образаца у сваком сегменту посебно. Дата модификација је кључна у контексту синхронизације процеса, која се постиже позивањем функције MPI_Reduce. У овој имплементацији такође постоји проблем потенцијалног пресецања образаца, међутим дати проблем није адекватно адресиран. На следећем исечку приказан је остатак $main$ функције која реализује паралелизам.

```
{
    for (int i = 0; i < num_patterns; i++)
    {
        local_counts[i] = search(patterns[i], segment, 101);
    }

    MPI_Reduce(local_counts, global_counts, num_patterns,
    MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0)
    {
        for (int i = 0; i < num_patterns; i++)
        {
            printf("Total␣occurrences␣of␣'%s':␣%d\n",
                patterns[i], global_counts[i]);
        }
        end_time = MPI_Wtime();
        printf("Total␣execution␣time:␣%f␣seconds\n",
            end_time - start_time);

        free(segment);
        free(text);
    }
    else
    {
        free(segment);
    }

    MPI_Finalize();
    return 0;
}
```

Listing 8: Синхорнизација процеса OpenMPI имплементације

5 Упоредна анализа перформанси

Основу за упоредну анализу представљају подаци добијени тестирањем претходно описаних имплементација. Ова секција има за циљ да на основу резултата мерења изведе специфичне закључке везане за разлику паралелне и секвенцијалне обраде великих скупова података, посебно у контексту текстуалних података. Најпре је потребно дефинисати методологију тестирања конкретних имплементација.

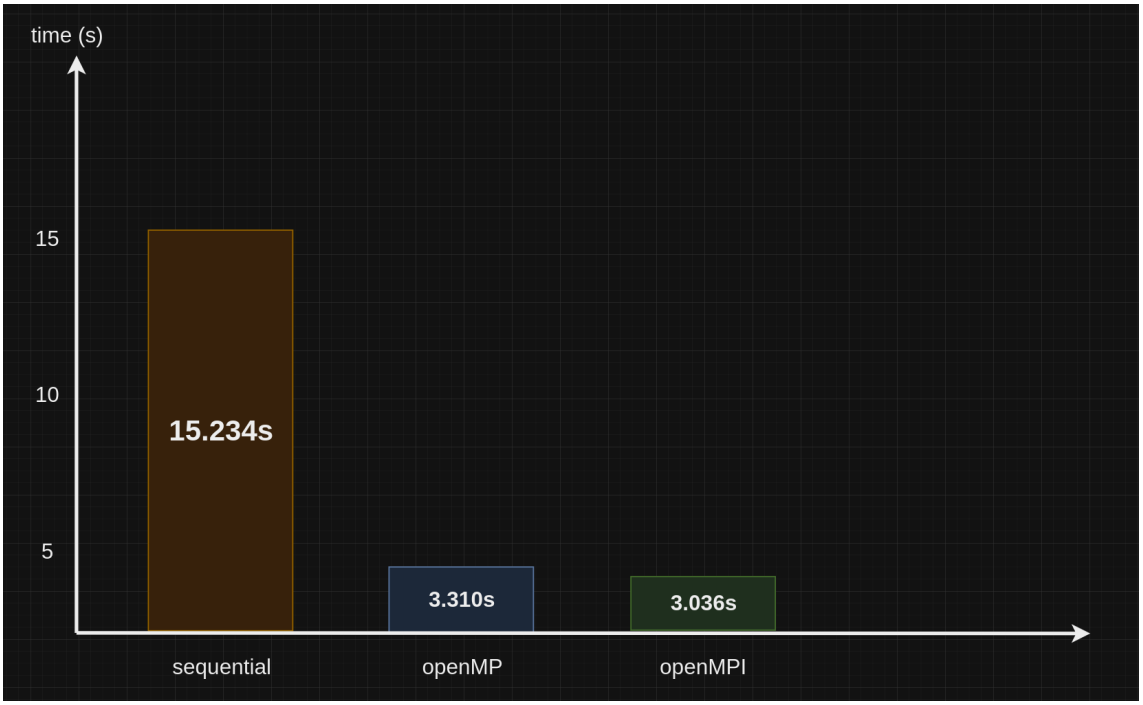
5.1 Опис експеримента

Свака од имплементација тестирана је над истим почетним сетом података. Подаци су изгенерисани покретањем Python скрипте. Пример коришћене скрипте налази се на наредном исечку кода.

```
def generate_random_sequence():
    sequences = ["R1", "L1", "X", "O", "Y"]
    sequence_length = random.randint(50, 50000)
    return ''.join(random.choice(sequences)
        for _ in range(sequence_length))

def generate_file(file_name, num_sequences, size_mb):
    target_size = size_mb * 1024 * 1024
    with open(file_name, 'w') as file:
        while os.path.getsize(file_name) < target_size:
            sequence = generate_random_sequence()
            file.write(sequence + '\n')
            file.flush()
        file.truncate(target_size)

if __name__ == "__main__":
    file_name = "controller_input.txt"
    num_sequences = 1000
    size_mb = 500

    generate_file(file_name, num_sequences, size_mb)
    print("Sequences␣generated.")
```

Listing 9: Скрипта за генерисање текста који се претражује

Генерисани подаци садрже секвенце које илуструју корисничку интеракцију са PS5 контролером током играња видео игара. Образци који се претражују су **R1L1X**, **R1XO**, **OYO** и представљају команде видео игре под називом Minecraft Dungeons. Машина коришћена за претрагу текста поседује Intel Core i7-8700 процесор са 6 језгара, радног такта на 3.20GHz, који оперише над RAM меморијом капацитета 32GB. Важно је истаћи да резултати приказани на слици 2 представљају средњу вредност добијену из пет различитих тестирања. Ово доприноси њиховој веродостојности што позитивно утиче на извођење закључака.



Слика 2: Резултати тестирања

5.2 Анализа добијених резултата

Анализом добијених података, уочава се очигледно побољшање у брзини извршавања које доноси паралелна обрада. Конкретно, време одзива за OpenMP верзију је приближно 3.3 секунде, што је више од четири пута брже у поређењу са секвенцијалном верзијом. Још веће убрзање постигнуто је са OpenMPI имплементацијом, која извршава претрагу за свега 3.03 секунде, што је пет пута брже од секвенцијалног приступа и за нијансу брже од OpenMP верзије. Ови резултати јасно истичу предности примене паралелних технологија у обради великих скупова података.

6 Закључак

У овом раду детаљно је обрађен алгоритам за ефикасну претрагу текста, који носи назив Рабин-Карп алгоритам. Првенствено овај алгоритам је имплементиран секвенцијално, како би се читалац детаљније упознао са његовим кључним аспектима. Након анализе секвенцијалне имплементације, идентификовани су делови алгоритма погодни за паралелизацију, која је затим изведена помоћу OpenMP и OpenMPI технологија.

Главни циљ истраживања је демонстрација значаја паралелног приступа у обради великих скупова података, са посебним нагласком на текстуалне податке. Експериментални резултати, приказани у петом поглављу, јасно потврђују супериорност паралелних верзија истог алгоритма у брзини и ефикасности идентификације конкретних образаца у текстовима. Као закључак овог рада може се истаћи да, иако увођење паралелизма доноси одређене изазове, његова примена значајно унапређује ефикасност система. Оваква побољшања у ефикасности могу бити кључна у пословним применама, доприносећи већој вредности производа или услуга. Имплементација паралелизма, стога, има важну улогу у технолошком напретку и може бити пресудан фактор у одржавању конкурентности на тржишту.

Списак изворних кодова

Списак слика

Библиографија

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.