Fakultet tehničkih nauka
Univerzitet u Novom Sadu

Računarski sistemi visokih performansi

Komparativna analiza algoritama za sortiranje

Autor: Indeks:

Vanja Stepanoski E2 10/2021

Februar, 2022.

Sadržaj

Uvod

Ideja ovog rada jeste da čitaocu kroz primenu nad algoritmima za sortiranje prikaže šta je to paralelizacija i zašto je ona značajna.

U radu će biti reči o tome šta su algoritmi za sortiranje, kao i tome koliko je široka njihova primena. Zatim će biti malo reči i o samoj paralelizaciji, te će se u nastavku rada govoriti o tri algoritma za sortiranje (Quick sort, Merge sort i Counting sort). Ova tri algoritma će biti predstavljena i objašnjena, zajedno sa implementacijom istih, dok će pri samom kraju algoritmi biti testirani i upoređeni nad nizovima različite veličine.

Prilikom testiranja biće korišćen OpenMP.

Algoritmi za sortiranje

Iako je ideja veoma jednostavna – treba poređati elemente, problem sortiranja je zbog kompleksnosti pronalaska efikasnog rešenja, od samog početka računarskih nauka zauzimao prilično veliki udeo u istraživanjima. Prvi algoritmi za sortiranje datiraju još iz 1950-ih godina, a među autorima istih bila je i Beti Holberton (jedna od šest programera koji su radili na ENIAC-u). Analize Babl (eng. „Bubble“) sorta potiču još iz 1956. godine, štaviše većina optimalnih algoritama je poznata sa sredine 20-og veka. Naravno, još uvek dolazi do otkrivanja novih algoritama za sortiranje, pa tako imamo Timsort algoritam koji je objavljen 2002. godine, ili Library sort algoritamkoji je objavljen 2006. godine [1].

Zapravo, u računarskim naukama, algoritmi za sortiranje predstavljaju jedne od fundamentalnih algoritama, čija je uloga da vrše preraspodelu elemenata određenog skupa podataka tako da nakon izvršavanja algoritma elementi budu poređani u specifičnom poretku. U praksi se najčešće koristi numerički poredak (poredak brojeva po veličini), a odmah nakon njega je i leksikografski poredak (poredak slova, ili karaktera, po abecednom redosledu). Poredak može biti rastući (..., 1, 2, ..., 43, 51, ...) ili opadajući (..., 51, 43, ..., 2, 1, ...).

Efikasno sortiranje je veoma važan, neki bi rekli i ključan, korak u procesu optimizacije ostalih algoritama (npr. algoritam za pretragu) koji pri svom radu, kao jedan od ulaznih parametara, zahtevaju sortiranu listu elemenata. Sortiranje je takođe veoma zastupljeno kada je u pitanju kanonikalizacija podataka radi kreiranja prikaza koji je razumljiviji ljudskom oku.

Da bi se algoritam smatrao algoritmom za sortiranje, podaci koji se dobijaju na izlazu moraju da zadovolje dva kriterijuma:

- Izlazni podaci moraju biti u monotonom poretku (svaki element mora biti veći od prethodnog kada je u pitanju rastući, odnosno svaki element mora biti manji od prethodnog kada je u pitanju opadajući poredak)
- Izlazni podaci moraju biti permutacija (reraspodela, zadržavanje postojećih elemenata, bez ubacivanja novih) ulaznih podataka

Za optimalnu efikasnost najbolje bi bilo u algoritmu za sortiranje koristiti strukture podataka koje dozvoljavaju nasumičan pristup (eng. „random access“) elementima, u odnosu na one strukture podataka koje dozvoljavaju isključivo sekvencijalni pristup.

Trenutno postoji na stotine različitih algoritama za sortiranje, svaki sa svojim specifičnim karakteristikama. Oni se mogu razlikovati prema zauzeću memorije – prostorna kompleksnost (eng. „space complexity“) i prema vremenu izvršavanja – vremenska kompleksnost (eng. „time complexity“).

Ove dve vrste kompleksnosti predstavljaju se asimptotskim notacijama (eng. „asimptotic notations“ – više o ovim notacijama se može pronaći u [3]), pretežno korišćenjem simbola O, Θ, Ω („Big-O“, „Theta“ i „Omega“ notacija, respektivno), dok se u zagradama nakon simbola navodi izraz u odnosu na broj *n*, gde broj *n* predstavlja broj koji označava broj elemenata u strukturi podataka (npr. O(n), O(log(n), ...).

Većina algoritama za sortiranje spada u jednu, od sledeće dve, kategorije:

- Logaritamska kompleksnost
  Kompleksnost proporcijalna binarnom logaritmu (npr. sa osnovom 2) broja *n*. Primer algoritma

za sortiranje sa logaritamskom kompleksnošću bio bi Kvik (eng. „Quick“) sort, sa vremenskom i prostornom kompleksnošću:

$$O(n * \log n)$$

- Kvadratna kompleksnost
  Kompleksnost proporcijalna kvadratu broja *n*. Primer algoritma za sortiranje sa kvadratnom kompleksnošću bio bi Babl sort, sa vremenskom kompleksnošću:

$$O(n^2)$$

Vremenska i prostorna kompleksnost takođe dalje mogu podeljene u tri zasebna slučaja: najbolji slučaj, prosečni slučaj o najgori slučaj (eng. „best case“, „average case“, „worst case“, respektivno) [2].

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)

Slika 1 - Prikaz algoritama za sortiranje sa njihovim kompleksnostima [4]

Paralelno računarstvo i zašto je ono bitno

Jednostavno rečeno, paralelno računarstvo predstavlja korišćenje više procesora u cilju obavljanja nekog posla. Za razliku od sekvencijalnog računarstva, prilikom primene paralelne arhitekture potrebno je podeliti celokupan posao na manje celine, te identifikovati koje je od manjih celina moguće izvršiti paralelno, te ih pustiti da se paralelno i izvršavaju. Paralelno računarstvo je veoma zgodno kada je u pitanju modelovanje i simulacija svakodnevnih pojava.

Bez primene paralelizacije bi obavljanje većine digitalnog posla bilo prilično nezgodno, pa čak i neizvodljivo ako se u obzir uzme moderni („brzi“) način života. Kada bi naši „pametni“ telefoni ili računari obavljali samo jednu operaciju u trenutnku, svaki posao bi trajao značajno duže nego što danas traje. Da bi se dočarala razlika koju uvodi paralelizacija, dovoljno je spomenuti „pametne“ telefone iz 2010. godine kao što su iPhone 4 i Motorola Droid koji su koristili sekvencijalne procesore, te im je samo za otvaralje e-mail poruke bilo potrebno 30, ili više, sekundi – što je danas nezamislivo dugo.

Prednost paralelnog programiranja je u tome da računari mogu izvršavati kod efikasnije, a neki od benefita koje ono donosi su i [5]:

- Paralelno modelovanje i simulacija pojava iz stvarnog sveta (vreme, saobraćaj, finansije, zdravlje, ...)
- Ušteda vremena
- Ušteda novca
- Rešavanje komleksnijih i većih problema

Paralelizacija algoritama za sortiranje

Kao što je navedeno, algoritmi za sortiranje predstavljeju jedne od fundamentalnih algoritama, koji svoju primenu nalaze u mnogim aplikacijama. Sama činjenica da je primena algoritama za sortiranje ovoliko široka stavlja do znanja da promena brzine sortiranja može direktno uticati i na brzinu izvršavanja mnogih aplikacija, a paralelizacija nam omogućava da ovo ubrzanje i postignemo.

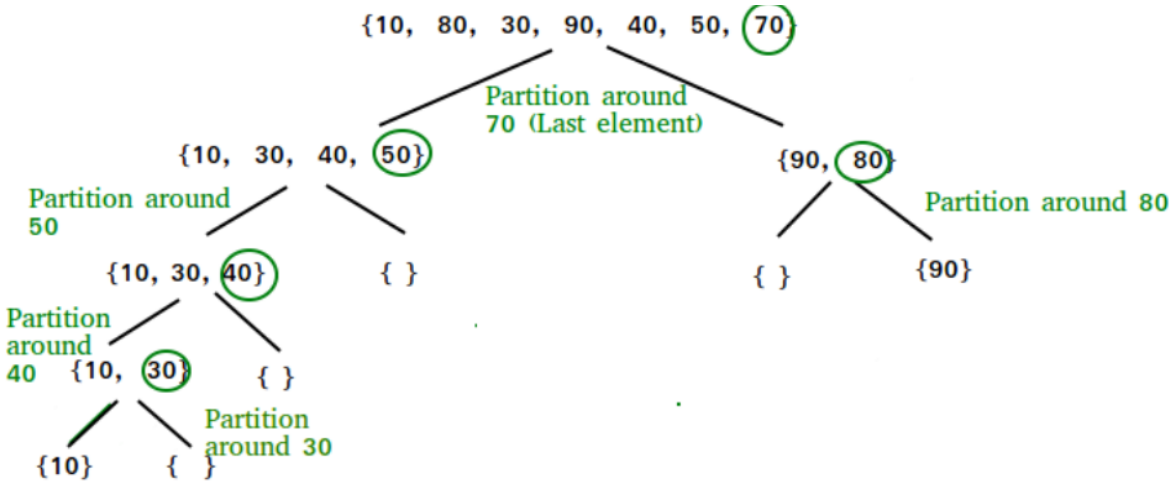
U nastavku rada će biti više reči o Quick, Merge i Counting sortu, te njihovim sekvencijalnim i paralelnim rešenjima.

Quick sort

Quick sort spada u grupu „podeli pa vladaj (eng. „Divide and Conquer“)[1]“ algoritama za sortiranje. Algoritam funkcioniše tako što odredi element koji će biti „pivot“, i u odnosu na taj pivot element vrši particionisanje niza - deli zadati niz u dva podniza. Postoji više različitih načina kako će Quick sort pronaći pivot element:

- Uvek se uzima prvi element kao pivot
- Uvek se uzima poslednji element kao pivot
- Uvek se uzima nasumični element
- Uvek se uzima medijana niza kao pivot

Ključni process u okviru Quick sort algoritmu je upravo particionisanje niza. Particionisanje niza se vrši tako što funkcija dobije niz i pivot element, te pivot element postavi na odgovarajuću poziciju u odnosu na ostale elemente niza – sve elemente niza koji su manji od pivota prebaci pre njega (levo od pivota), a sve elemente niza koji su veći od pivota prebaci nakon njega (desno od pivota), u slučaju rastućeg poretka, odnosno obrnuto u slučaju opadajućeg poretka.



Slika 2 - Prikaz načina rada Quick sort algoritma [8]

Nakon odrađenog particionisanja, funkcija za particionisanje će vratiti novu poziciju pivot elementa, te će ovaj process biti ponovljen po jednom za oba podniza – podniz levo od pivota i podniz desno od pivota.

[1] Više o Divide and Conquer algoritmima možete pronaći na [7].

Sekvencijalno rešenje

U nastavku će biti prikazani isečci implementacije Quick sort algoritma, praćeni kratkim opisima važnih delova.

```
1  void quickSort(int arr[], int low, int high){
2      if (low < high)
3      {
4          int pi = partition(arr, low, high);
5          quickSort(arr,low, pi - 1);
6          quickSort(arr, pi + 1, high);
7      }
8  }
```

Slika 3 - Prikaz quickSort funkcije

Sa slike XX se može videti princip funkcionisanja Quick sorta, o kome je bilo reči u prethodnom pasusu. Pri ulasku u funkciju quickSort se vrši ispitivanje da li je promenljiva low manja od promenljive high (u slučaju da nije, u podnizu se nalazi samo jedan element, te nema potrebe za dodatnom rekurzijom i može se nastaviti dalje), nakon što je potvrđeno da je promenljiva low manja – tj. da u nizu ima više od jednog elementa, vrši se particionisanje niza i nastavlja se rekurzivno pozivanje funkcije quickSort za oba podniza.

```
1  int partition(int arr[], int low, int high){
2      int pivot = arr[high];
3      int i = (low - 1);
4      for (int j = low; j <= high- 1; j++)
5      {
6          if (arr[j] <= pivot)
7          {
8              i++;
9              swap(&arr[i], &arr[j]);
10         }
11     }
12     swap(&arr[i + 1], &arr[high]);
13     return (i + 1);
14 }
```

Slika 4 - Prikaz partition funkcije

U funkciji za particionisanje niza se kao pivot element uzima poslednji element, te se nakon toga vrši premeštanje elemenata u odnosu na pivot. Nakon što je premeštanje završeno, funkcija vraća novu poziciju pivot elementa.

```
39 void run_quick_sequential(int quick_arr_sq[], int n){
40     quickSort(quick_arr_sq, 0, n-1);
41 }
```

Slika 5 - Prikaz poziva quickSort funkcije

U sekvencijalnom rešenju se funkcija quickSort poziva normalno, sa prosleđenim nizom koji je potrebno sortirati, pozicijom početnog elementa niza, i pozicijom krajnjeg elementa niza.

Paralelizovano rešenje

Za razliku od sekvencijalnog rešenja, kod paralelnog rešenja se za svaki rekurzivno poziv, koji sledi nakon particionisanja niza, kreira novi omp task i oni se puštaju da rade paralelno. Prilikom paralelizacije, funkcija za particionisanje je ostala nepromenjena.

```
16 void quickSort(int arr[], int low, int high){
17     if (low < high)
18     {
19         int pi = partition(arr, low, high);
20         #pragma omp task firstprivate(arr, low, pi)
21         {
22             quickSort(arr,low, pi - 1);
23         }
24         #pragma omp task firstprivate(arr, high, pi)
25         {
26             quickSort(arr, pi + 1, high);
27         }
28     }
29 }
```

Slika 6 - Prikaz quickSort funkcije - paralelno

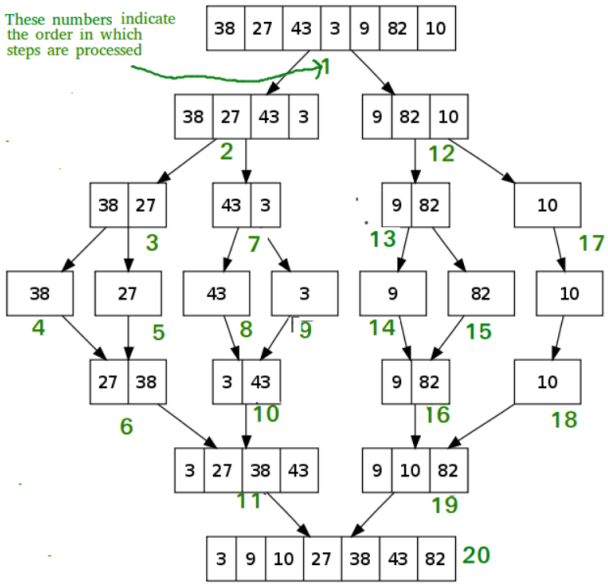
```
31 void run_quick_parallel(int quick_arr[], int n){
32     #pragma omp parallel
33     {
34         #pragma omp single nowait
35         quickSort(quick_arr, 0, n-1);
36     }
37 }
```

Slika 7 - Prikaz poziva quickSort funkcije - paralelno

U da bi se rešenje izvršilo paralelno, potrebno je početnu quickSort funkciju pozvati unutar paralelnog bloka, u suprotnom neće doći do paralelizacije, već će se sve izvršiti sekvencijalno.

Merge sort

Kao i Quick sort, i Merge sort spada u grupu „podeli pa vladaj“ algoritama za sortiranje. Ovaj algoritam deli početni niz na dva podniza, te poziva sam sebe za svaki podniz, nakon toga vrši spajanje ovih podnizova, tako što svaki element postavlja sortiran u odnosu na poredak koji se traži.



Slika 8 - Prikaz načina rada Merge sort algoritma [10]

Sa prikazanog dijagrama se može uočiti način na koji će niz [38, 27, 43, 3, 9, 82, 10] biti sortiran kroz 20 koraka (sekvencijalno rešenje).

Sekvencijalno rešenje

U nastavku će biti prikazani isečci implementacije Merge sort algoritma, praćeni kratkim opisima važnih delova.

```
1  void mergeSort(int *X, int n, int *tmp){
2      if (n < 2) return;
3      mergeSort(X, n/2, tmp);
4      mergeSort(X+(n/2), n-(n/2), tmp + n/2);
5      mergeSortAux(X, n, tmp);
6  }
```

Slika 9 - Prikaz mergeSort funkcije

Funkcija mergeSort vrši podelu niza i rekurzivne pozive sve dok u nizu ne ostane samo jedan element, nakon toga počinje vraćanje i izvršavanje glavne funkcije – mergeSortAux.

```
1  void mergeSortAux(int *X, int n, int *tmp) {
2      int i = 0;
3      int j = n/2;
4      int ti = 0;
5
6      while (i<n/2 && j<n) {
7          if (X[i] < X[j]) {
8              tmp[ti] = X[i];
9              ti++; i++;
10         } else {
11             tmp[ti] = X[j];
12             ti++; j++;
13         }
14     }
15     while (i<n/2) {
16         tmp[ti] = X[i];
17         ti++; i++;
18     }
19     while (j<n) {
20         tmp[ti] = X[j];
21         ti++; j++;
22     }
23     memcpy(X, tmp, n*sizeof(int));
24 }
```

Slika 10 - Prikaz mergeSortAux funkcije

Funkcija mergeSortAux vrši spajanje dva podniza, tako da novodobijeni niz bude sortiran u traženom poretku.

```
48 void run_merge_sequential(int merge_arr_sq[], int n){
49     int tmp[ARRAY_MAX_SIZE];
50     mergeSort(merge_arr, n, tmp);
51 }
```

Slika 11 - Prikaz poziva mergeSort funkcije

Prilikom poziva funkcije mergeSort, potrebno je proslediti niz koji se sortira, broj elemenata u njemu, kao i pomoćni niz koji će koristiti funkcija mergeSortAux.

Paralelizovano rešenje

U slučaju paralelnog rešenja, za svaki od rekurzivnih poziva metode mergeSort se kreira poseban task koji će se izvršavati u paraleli. Nakon što je glavni niz podeljen na male podnizove, pristupa se spajanju u funkciji mergeSortAux. Funkcija mergeSortAux se označava kao taskwait, te predstavlja tačku sinhronizacije za taskove koji se izvršavaju unutar funkcije mergeSort. Metoda mergeSortAux je prilikom paralelizacije ostala nepromenjena.

```
26  void mergeSort(int *X, int n, int *tmp){
27      if (n < 2) return;
28
29      #pragma omp task shared(X) if (n > 100)
30      mergeSort(X, n/2, tmp);
31
32      #pragma omp task shared(X) if (n > 100)
33      mergeSort(X+(n/2), n-(n/2), tmp + n/2);
34
35      #pragma omp taskwait
36      mergeSortAux(X, n, tmp);
37  }
```

Slika 12 - Prikaz mergeSort funkcije - paralelno

```
39  void run_merge_parallel(int merge_arr[], int n){
40      int tmp[ARRAY_MAX_SIZE];
41      #pragma omp parallel
42      {
43          #pragma omp single
44          mergeSort(merge_arr, n, tmp);
45      }
46  }
```

Slika 13 - Prikaz poziva mergeSort funkcije - paralelno

U da bi se rešenje izvršilo paralelno, kao što je slučaj i kod Quick sorta, potrebno je početnu quickSort funkciju pozvati unutar paralelnog bloka, u suprotnom neće doći do paralelizacije, već će se sve izvršiti sekvencijalno.

Counting sort

Counting sort predstavlja algoritam za sortiranje baziran na korišćenju ključeva koji označavaju pozicije i broj elemenata koji se nalaze počev od te pozicije. Funkcioniše tako što broji pojavljivanje svakog elementa unutar niza, te nakon toga primenjuje određenu artitmetiku da bi odredio pozicije na kojima se elementi nalaze.

Sekvencijalno i paralelno rešenje

Prilikom pokušaja implementacije sekvencijalnog i paralelnog rešenja došlo je do delimičnog uspeha, naime ovakva implementacija Counting sort algoritma ne daje svaki put ispravno sortiran niz, već dolazi do grešaka prilikom sortiranja.

```
1  void countingSort(int arr[], int arr_sorted[], int n) {
2      int i, j, count;
3      #pragma omp for private(i, j, count)
4      for (i = 0; i < n; i++) {
5          count = 0;
6          for (j = 0; j < n; j++) {
7              if (arr[i] > arr[j])
8                  count++;
9          }
10         while (arr_sorted[count] != 0)
11             count++;
12         arr_sorted[count] = arr[i];
13     }
14 }
```

Slika 14 - Prikaz countingSort funkcije

```
16 void run_counting_parallel(int counting_arr[], int counting_arr_sorted[], int n){
17     #pragma omp parallel
18     {
19         countingSort(counting_arr, counting_arr_sorted, n);
20     }
21 }
22
23 void run_counting_sequential(int counting_arr[], int counting_arr_sorted[], int n){
24     countingSort(counting_arr, counting_arr_sorted, n);
25 }
```

Slika 15 - Prikaz poziva countingSort funkcije

Komparativna analiza rezultata

Prilikom pokretanja i testiranja gorenavedenih implementacija korišćen je računar koji ima Intel Core i7-7600HQ 2.60GHz procesor sa 4 fizička, odnosno 8 logičkih jezgara i 16GB ram memorije. Svaki od algoritama je dobijao identičan nesortirani niz sa N elemenata, gde je svaki element nasumično izabran broj u rasponu 1-50. Za testiranje je korišćeno N za svaki broj iz [50, 500, 5000, 50000, 500000, 1000000, 2000000, 2500000]. Redosled pokretanja algoritama je pri svakom pokretanju bio sledeći:

1. Quick sort – paralelno
2. Quick sort – sekvencijalno
3. Merge sort – paralelno
4. Merge sort – sekvencijalno
5. Counting sort – paralelno
6. Counting sort – sekvencijalno

Dobijeni rezultati:

```
Sorts with array of 50 elements:
Quick sort parallel execution time in seconds: 0.002865
Quick sort sequential execution time in seconds: 0.000025
Merge sort parallel execution time in seconds: 0.000056
Merge sort sequential execution time in seconds: 0.000032
Counting sort parallel execution time in seconds: 0.000025
Counting sort sequential execution time in seconds: 0.000068
```

Slika 16 - Test sa N = 50

```
Sorts with array of 500 elements:
Quick sort parallel execution time in seconds: 0.004744
Quick sort sequential execution time in seconds: 0.000329
Merge sort parallel execution time in seconds: 0.000183
Merge sort sequential execution time in seconds: 0.000315
Counting sort parallel execution time in seconds: 0.005053
Counting sort sequential execution time in seconds: 0.006950
```

Slika 17 - Test sa N = 500

```
Sorts with array of 5000 elements:
Quick sort parallel execution time in seconds: 0.035146
Quick sort sequential execution time in seconds: 0.008882
Merge sort parallel execution time in seconds: 0.003411
Merge sort sequential execution time in seconds: 0.003543
Counting sort parallel execution time in seconds: 0.166162
Counting sort sequential execution time in seconds: 0.542001
```

Slika 18 - Test sa N = 5000

```
Sorts with array of 50000 elements:
Quick sort parallel execution time in seconds: 0.288265
Quick sort sequential execution time in seconds: 0.650823
Merge sort parallel execution time in seconds: 0.019735
Merge sort sequential execution time in seconds: 0.041908
Counting sort parallel execution time in seconds: 10.767967
Counting sort sequential execution time in seconds: 51.556963
```

Slika 19 - Test sa N = 50000

```
Sorts with array of 500000 elements:
Quick sort parallel execution time in seconds: 16.208704
Quick sort sequential execution time in seconds: 59.908868
Merge sort parallel execution time in seconds: 0.136919
Merge sort sequential execution time in seconds: 0.266140
Prekinuto nakon 7 minuta
Nije isprobano
```

Slika 20 - Test sa N = 500000

```
Sorts with array of 1000000 elements:
Quick sort parallel execution time in seconds: 63.001017
Segmentation fault
Merge sort parallel execution time in seconds: 0.257721
Merge sort sequential execution time in seconds: 0.531599
Nije isprobano
Nije isprobano
```

Slika 21 - Test sa N = 1000000

```
Sorts with array of 2000000 elements:
Nije isprobano
Nije isprobano
Merge sort parallel execution time in seconds: 0.589470
Merge sort sequential execution time in seconds: 1.103728
Nije isprobano
Nije isprobano
```

Slika 22 - Test sa N = 2000000

```
Sorts with array of 2500000+ elements:
Nije isprobano
Nije isprobano
Segmentation fault
Segmentation fault
Nije isprobano
Nije isprobano
```

Slika 23 - Test sa N = 2500000

Zaključak

Prilikom izrade ovog rada, zaključeno je da prilikom sortiranja nizova malih dimenzija nema potrebe za primenom paralelizacije, jer samo kreiranje taskova oduzima previše vremena i izvršavanje paralelnog programa bude sporije od izvršavanja sekvencijalnog programa (videti sliku 16). Dok je u slučajevima sortiranja nizova velikih dimenzija poželjno primeniti paralelizaciju jer se postiže ubrzanje od oko 50%.

Implementacije koje su date u radu nisu pogodne za sve vrste nizova, naime kada su u pitanju veliki nizovi date implementacije nemaju efekta (implementacija paralelnog Quick sorta pri radu sa nizovima od milion elemenata ima vreme izvršavanja od oko 60 sekundi, dok kada su u pitanju nizovi sa preko 2.5 miliona elemenata ni jedan od sortova ne radi), te je potrebno istražiti i druge implementacije ili algoritme.

Literatura

[1] Wikipedia, Sorting algorithm – https://en.wikipedia.org/wiki/Sorting_algorithm
[2] – https://sortvisualizer.com/
[3] Programiz, Asymptotic notations– https://www.programiz.com/dsa/asymptotic-notations
[4] Leonardo Galler and Matteo Kimura, Sorting algorithms – https://lamfo-unb.github.io/2019/04/21/Sorting-algorithms/
[5] HP, Parallel computing and its modern uses – https://www.hp.com/us-en/shop/tech-takes/parallel-computing-and-its-modern-uses
[6] Ricardo Rocha and Fernando Silva, Parallel computing – https://www.dcc.fc.up.pt/~ricroc/aulas/1516/cp/apontamentos/slides_sorting.pdf
[7] Geeks for Geeks, Divide and Conquer– https://www.geeksforgeeks.org/divide-and-conquer-algorithm-introduction/
[8] Geeks for Geeks, Quick sort– https://www.geeksforgeeks.org/quick-sort/
[9] Geeks for Geeks, Merge sort– https://www.geeksforgeeks.org/merge-sort/
[10] Geeks for Geeks, Counting sort – https://www.geeksforgeeks.org/counting-sort/

Spisak slika