

# Analiza jedne implementacije serijskog i paralelnog merge sort algoritma u programskom jeziku C++

Mladen Samardžić



## Sadržaj

<b>1</b>	<b>Uvod</b>	<b>5</b>
<b>2</b>	<b>Osnovne karakteristike serijskog merge sort-a</b>	<b>6</b>
<b>3</b>	<b>Paralelni merge sort</b>	<b>8</b>
<b>4</b>	<b>Implementacija</b>	<b>11</b>
4.1	Serijski merge i merge sort . . . . .	11
4.2	Paralelni merge i merge sort . . . . .	13
<b>5</b>	<b>Analiza performansi</b>	<b>18</b>
<b>6</b>	<b>Zaključak</b>	<b>23</b>
<b>7</b>	<b>Tabele</b>	<b>24</b>



## 1 Uvod

Sa naglim porastom paralelizacije računarskih sistema u proteklih nekoliko decenija, poraslo je i interesovanje za istraživanje i prikupljanje algoritama koji bi takve vrste arhitektura iskoristili na što efikasniji način. Sortiranje kolekcija podataka jeedna je od osnovnih operacija, kao i deo mnogih složenijih, u bilo kom dovoljno kompleksnom sistemu; zato je od ključne važnosti odabir adekvatnog algoritma u odnosu na konkretne potrebe i optimizacija njegove implementacije u odnosu na sam sistem. Zbog svoje stabilnosti, merge sort pogodan je za sortiranje podataka po više kriterijuma (na primer, sortiranje osoba po prezimenu, pa imenu), a zbog svog divide-and-conquer pristupa problemu pogodan je za paralelizaciju.

U ovom tekstu analiziraćemo performanse jedne implementacije ovog algoritma u programskom jeziku C++, poredeći ih sa performansama serijske implementacije istog algoritma i nove, paralelne verzije `std::stable_sort`, dostupne u standardnoj biblioteci od C++17 verzije jezika.<sup>1</sup> Pri paralelizaciji algoritma koristićemo apstrakciju zadataka (*tasks*), obezbeđenu od strane Intel-ove Thread Building Blocks biblioteke. Sva merenja vršićemo na Intel i7-6700HQ četvorojezgarnom (sa osam logičkih jezgara) procesoru i Ubuntu 19.04 operativnom sistemu, koristeći verziju 9.0.1 g++ kompajlera. Dodatne relevantne konfiguracije kompajlera navešćemo su u adekvatnim sekcijama.

---

<sup>1</sup>Iako je C++17 standardizovan 2017. godine, paralelne verzije STL algoritama za vreme pisanja ovog dokumenta implementirane su, što se tiče najpopularnijih kompajlera, samo u standardnim bibliotekama koje dolaze uz g++ 9+ i msvc++ 19.14+.

## 2 Osnovne karakteristike serijskog merge sort-a

Osnovna ideja merge sort algoritma je jednostavna i može se izvesti principom matematičke indukcije na sledeći način: znamo da sortiramo prazan niz i niz sa jednim elementom – svaki takav niz je sortiran. Pretpostavimo onda da znamo da sortiramo nizove dužina  $2, 3, \dots, n$ . Niz dužine  $n + 1$  sortiramo tako što ga podelimo na dva podniza dužina  $\lfloor \frac{n+1}{2} \rfloor$  i  $\lceil \frac{n+1}{2} \rceil$ , sortiramo te podnizove, pa ih prespojimo nazad u početni niz. Jednostavna Python implementacija izgledala bi ovako:

---

```
1 def merge_sort(xs, start, end):
2     if end - start < 2:
3         return
4     mid = (start + end) // 2
5     merge_sort(xs, start, mid)
6     merge_sort(xs, mid, end)
7     merge(xs, start, mid, end)
```

---

Takvu funkciju direktno bismo pozvali na sledeći način:

---

```
1 merge_sort(xs, 0, len(xs))
```

---

Ostaje nam, naravno, da definišemo proces spajanja sortiranih podnizova, što je tačno zadatak `merge` funkcije u prethodnom primeru. Podnizove ćemo spajati tako što ćemo u isto vreme šetati po oba i porediti odgovarajuće elemente. Manji element dodaćemo u originalni niz, a indeks u podniz koji odgovara tom elementu povećaćemo za jedan, pošto smo se tog elementa rešili. Kada u potpunosti prođemo kroz jedan od podnizova, sve elemente drugog dodaćemo na kraj originalnog niza, jer su po preduslovima algoritma sortirani, a po čuvanim invarijantama veći od svih elemenata koje smo prethodno dodali. Vrativši se na primer, Python implementacija `merge` izgledala bi ovako:

---

```
1 def merge(xs, start, mid, end):
2     left = xs[start:mid]
3     right = xs[mid:end]
4     i, j, k = 0, 0, 0
5     while i < len(left) and j < len(right):
6         if left[i] <= right[j]:
7             xs[k] = left[i]
8             i += 1
9             k += 1
10        else:
11            xs[k] = right[j]
12            j += 1
13            k += 1
```

---

```

14     while i < len(left):
15         xs[k] = left[i]
16         i += 1
17         k += 1
18     while j < len(right):
19         xs[k] = right[j]
20         j += 1
21         k += 1

```

---

Primetimo nekoliko stvari:

- Morali smo da izdvojimo dodatnih  $O(\text{end} - \text{start})$  memorije za podnizove `left` i `right`. To je nužno jer bismo inače izgubili neke elemente pri prepisivanju u originalni niz.
- Pošto prolazimo kroz svaki element niza `xs` tačno jednom (kroz kopije njegovih podnizova), vremenska kompleksnost je takođe  $O(\text{end} - \text{start})$ .
- Uslov na liniji 6 mogli smo napisati i kao `left[i] < right[j]`, ali bismo onda izgubili jednu od najbitnijih osobina merge sort algoritma: njegovu stabilnost. Algoritam za sortiranje je stabilan ako čuva redosled elemenata koji se u svrhe sortiranja porede kao jednaki. Ovo nema nikakav značaj ukoliko radimo sa tipovima koji se mogu porediti samo na jedan način; ako, međutim, sortiramo radnike po prezimenu, možda nam je bitno da se radnicima sa istim prezimenima održi redosled u nizu (jer smo ih pre toga sortirali po odeljenju u kom rade).

Kako sada imamo potpuni opis algoritma, možemo analizirati njegove asimptotske performanse.

Vodeći se Python implementacijom, vreme izvršavanja možemo matematički opisati kao  $T(n) = 2T(\frac{n}{2}) + O(n)$ . Jednostavnom primenom master teoreme na prethodnu diferentnu jednačinu lako se dolazi do zaključka da je vremenska kompleksnost merge sort-a  $O(n \log n)$ .

Prostorna kompleksnost u potpunosti proističe iz faze spajanja podnizova, a, kako tada alociramo količinu memorije proporcionalnu dužini niza, potrebna memorija linearno je zavisna od broja elemenata niza (tj. prostorna kompleksnost je  $O(n)$ ).

### 3 Paralelni merge sort

Vratimo se na prvi primer:

---

```
1 def merge_sort(xs, start, end):
2     if end - start < 2:
3         return
4     mid = (start + end) // 2
5     merge_sort(xs, start, mid)
6     merge_sort(xs, mid, end)
7     merge(xs, start, mid, end)
```

---

Dva rekurzivna poziva na linijama 5 i 6 međusobno su nezavisna, jer svaki od njih radi sa disjunktним polovinama niza, pa u principu mogu da se izvršavaju paralelno. Trivijalnom transformacijom u pseudo-Python kod dobijamo prvu verziju paralelnog merge sort algoritma:

---

```
1 def merge_sort_par_v1(xs, start, end):
2     if end - start < 2:
3         return
4     mid = (start + end) // 2
5     fork merge_sort_par_v1(xs, start, mid)
6     merge_sort_par_v1(xs, mid, end)
7     join
8     merge(xs, start, mid, end)
```

---

Pošto `merge` očekuje potpuno sortirane podnizove na ulazu, moramo da sačekamo da se oba podniza sortiraju (tome služi `join` na sedmoj liniji) pre nego što je pozovemo. Ovo rešenje, iako korak u pravom smeru, nije najoptimalnije u pogledu paralelizacije.

Prvo, apstrakcije za rad sa konkurentnim i paralalnim kodom (nezavisno da li su to niti, zadaci, procesi, kanali ili nešto sasvim drugo) dolaze sa određenim dodatnim zahtevima po pitanju resursa. Na primer, za niz od 4 elementa, pozivanje funkcije iz prethodnog primera stvara 3 niti, što oduzima mnogo više vremena od samog sortiranja. Generalno, stvara se onoliko niti koliko ima levih grana u binarnom stablu poziva `merge_sort_par_v1`.

Ovaj problem rešićemo tako što ćemo definisati konstantu `cutoff`, koja će kontrolisati kada će se pozivati paralelna, a kada serijska verzija funkcije. Najpogodnija vrednost `cutoff` suštinski nalazi se eksperimentalnim metodama i predstavlja balans između povećanja performansi deljenjem posla na više procesorskih jezgara i gubitka stvaranjem previše niti. Definisanje ovakve konstante nije specifično za merge sort, nego je karakteristika većine paralelnih algoritama baziranih na rekurziji. Sa uvedenom konstantom, naš pseudokod izgleda ovako:



---

```

1  # cutoff defined somewhere else
2  def merge_sort_par_v2(xs, start, end):
3      if end - start <= cutoff:
4          merge_sort(xs, start, end)
5      mid = (start + end) // 2
6      fork merge_sort_par_v2(xs, start, mid)
7      merge_sort_par_v2(xs, mid, end)
8      join
9      merge(xs, start, mid, end)

```

---

Ova promena trebalo bi da značajno poboljša performanse i za manje i za veće veličine nizova, u zavisnosti od izabrane `cutoff` vrednosti.

Druga tačka poboljšanja je paralelizacija samog procesa spajanja podnizova. Iz originalnog `merge` koda nije očigledno da je moguće uraditi to na smislen način, ali jednostavnom promenom interfejsa i inherentno sekvencijalnog pristupa spajanju možemo doći do razumne verzije algoritma na relativno jednostavan način.

Prvo, umesto da `merge` spaja podnizove u mestu, spajaće ih u novi izlazni niz. Primetimo da ovo ne zahteva dodatnu memoriju: samo smo pomerili dužnost alociranja memorije izvan funkcije. Sada `merge` algoritam prima dva podniza (u daljem tekstu: levi i desni) kao ulazne parametre i niz u koji će se spajanje izvršiti kao izlazni parametar.

Zatim, pretpostavićemo, bez gubljenja opštosti, da je dužina levog podniza veća ili jednaka dužini desnog. Ukoliko to nije slučaj, samo ćemo pozvati algoritam sa desnim podnizom kao levim i levim kao desnim. Uvešćemo sledeću notaciju:  $p_k, 0 \leq k < n$  su indeksi u levi podniz,  $q_k, 0 \leq k < m$  indeksi u desni podniz, a  $p_n$  i  $q_m$  indeksi koji označavaju krajeve odgovarajućih podnizova. Paralelni `merge` algoritam onda se može opisati na sledeći način:

- Izabraćemo srednji element levog podniza,  $\bar{a}$ . Neka je indeks tog elementa  $r_1$ . Onda je  $r_1 = \lfloor \frac{p_1 + p_n}{2} \rfloor$ .
- Neka je  $r_2$  indeks prvog elementa u desnom podnizu takvog da nije manji od  $\bar{a}$ .
- Neka je  $r_3$  indeks u izlazni niz gde će  $\bar{a}$  biti smešten. Pošto znamo da elemenata manjih (ili jednakih) od  $\bar{a}$  u levom podnizu ima  $r_1$ , a u desnom (striktno manjih)  $r_2$ , onda je  $r_3 = r_1 - p_0 + r_2 - q_0$ .
- U izlazni niz na poziciju  $r_3$  upisaćemo  $\bar{a}$ .
- Pozvaćemo algoritam rekursivno dva puta. Argumenti prvog poziva biće levi podniz u intervalu indeksa  $[p_0, r_1)$ , desni u intervalu indeksa  $[q_0, r_2)$  i izlazni niz u intervalu  $[0, r_3)$ . Argumenti drugog poziva biće levi podniz u intervalu indeksa  $[r_1 + 1, p_n)$ , desni u intervalu  $[r_2, q_m)$  i izlazni niz u intervalu  $[r_3 + 1, m + n)$ . Ova dva poziva radiće nad

disjunktним podacima, pa se bezbedno mogu izvršavati iz različitih niti.

U pseudo-Python-u, implementacija ovog algoritma imala bi sledeći oblik:

---

```
1  # merge_cutoff defined somewhere else
2  def merge_par(xs, p0, pn, q0, qm, out, oi):
3      l_size = pn - p1
4      r_size = qn - q1
5      if l_size < r_size:
6          l_size, r_size = r_size, l_size
7          p0, q0 = q0, p0
8          pn, qm = qm, pn
9      if l_size == 0:
10         return
11     if l_size + r_size <= merge_cutoff:
12         merge_into(xs, p0, pn, q0, qm, out, oi)
13     r1 = (p1 + pn) // 2
14     r2 = lower_bound(xs, q1, qm, xs[r1])
15     r3 = oi + (r1 - p0) + (r2 - q0)
16     out[r3] = xs[r1]
17     fork merge_par(xs, p0, r1, q0, r2, out, oi)
18     merge_par(xs, r1 + 1, pn, r2, qm, out, r3)
19     join
```

---

Ova implementacija dodaje parametar `oi`, koji je suštinski potreban samo da bismo znali odakle treba indeksirati izlazni niz `out`. Takođe, pretpostavićemo da su funkcije `merge_into` i `lower_bound` već implementirane; `merge_into` je jednostavna adaptacija `merge` funkcije od ranije interfejsu `merge_par` (uz minimalnu promennu semantike tako da spaja u izlazni niz umesto u mestu), dok je `lower_bound` efektivno binarna pretraga. Konačna verzija paralelnog merge sort-a onda izgleda ovako:

---

```
1  # cutoff defined somewhere else
2  def merge_sort_par_v3(xs, start, end):
3      if end - start <= cutoff:
4          merge_sort(xs, start, end)
5      mid = (start + end) // 2
6      fork merge_sort_par_v3(xs, start, mid)
7      merge_sort_par_v3(xs, mid, end)
8      join
9      out = [0] * (end - start)
10     merge_par(xs, start, mid, mid, end, out, 0)
```

---

## 4 Implementacija

Naše implementacije serijskog i paralelnog merge sort algoritma imaju cilj da budu generičke prirode: po tipu kontejnera koji se sortira, po tipu elemenata tog kontejnera i po predikatu koji se koristi radi odlučivanja da li je jedan element manji od drugog. Iz tog razloga, kao i zbog konzistentnosti sa već postojećim alatima pruženim od strane standardne biblioteke, interfejs naše implementacije baziran je na interfejsu `std::sort`.

U ovoj sekciji izdvojicemo anotirane najbitnije delove implementacije; zbog toga su većinom izostavljene `#include` direktive i `std::` i `tbb::` prefiksi. Serijske verzije algoritama nalaze se u prostoru imena `serial`, paralelne u `parallel`, a nekoliko alijasa za tipove iteratora u `common` prostoru imena.

### 4.1 Serijski merge i merge sort

Serijski merge sort direktan je prevod iz Python primera u generički C++:

---

```
1 // include/serial/sort.hpp
2 template <typename It, typename P = less<>>
3 void serial::sort(It first, It last, P&& p = {}) {
4     auto const size = static_cast<size_t>(last - first);
5     if (size < 2)
6         return;
7
8     auto const mid = first + size/2;
9     serial::sort(first, mid, p);
10    serial::sort(mid, last, p);
11    serial::merge(first, mid, last, p);
12 }
```

---

Sve funkcije (tj. šabloni za funkcije) ove implementacije očekuju da `It` zadovoljava uslov `LegacyRandomAccessIterator`, dok `P` treba da bude callable (bilo pokazivač na funkciju, funktor ili lambda) koji prima dva argumenta tipova implicitno konvertibilnih u tip elementa kontejnera čiji je iterator `It`, sa povratnim tipom implicitno konvertibilnim u `bool`. Tip elemenata kontejnera mora da bude `MoveAssignable` i `MoveConstructible`.

`merge` je podeljen u dve funkcije: `merge` i `merge_impl`:

---

```
1 // include/serial/merge.hpp
2 template <typename It, typename Out, typename P = less<>>
3 void serial::merge_impl(It lf, It ll, It rf, It rl, Out of, P&& p) {
4     while (lf != ll && rf != rl)
5         if (!p(*rf, *lf))
6             *of++ = move(*lf++);
7     else
8         *of++ = move(*rf++);
```

---

```

9     of = move(lf, ll, of);
10    move(rf, rl, of);
11 }

```

---

`merge_impl` spaja dva intervala `[lf, ll)` i `[rf, rl)` u kolekciju na koju pokazuje iterator `out`. `Out` treba da zadovoljava uslov `LegacyOutputIterator`. U suštini, kao i `sort` malopre, i ovo je skoro pa direktan prevod Python funkcije definisane ranije, s tim da utilizuje move semantiku specifičnu za C++ i umesto operatora `<` koristi komparator `P`, gde `p(x, y)` vraća `true` ukoliko je `x` manje od `y` i `false` inače. Zbog toga je uslov `!p(*rf, *lf)` ekvivalentan uslovu `left[i] <= right[j]` iz Python implementacije, sa ulogom održavanja stabilnosti algoritma. Ova funkcija u suštini ne predstavlja javni interfejs (implicirano `_impl` sufiksom), već se koristi pri implementaciji `merge` i paralelnog merge algoritma (doduše, ništa osim proizvoljnog izbora od strane autora ne sprečava interfejs da bude javan, ako uzmemo u obzir da `std::merge` semantički izgleda ekvivalentno).

`merge` se onda svodi na alociranje kontejnera u koji će se podnizovi spajaju:

---

```

1  // include/serial/merge.hpp
2  template <typename It, typename P = less<>>
3  void serial::merge(It first, It mid, It last, P&& p = {}) {
4      auto const size = static_cast<size_t>(last - first);
5      auto alloc = allocator<value_t<It>>{};
6      auto *into = alloc.allocate(size);
7      merge_impl(first, mid, mid, last, into, p);
8      move(into, into + size, first);
9      alloc.deallocate(into, size);
10 }

```

---

Ovde `value_t<It>` predstavlja tip na koji pokazuje iterator tipa `It`, a njegova se definicija nalazi u prostoru imena `common`. Jedan detalj koji će pažljivom čitaocu zapasti za oko je taj da za dinamičku alokaciju niza u koji ćemo spajati podnizove umesto `std::vector` ili operatora `new` koristimo `std::allocator` i njegove funkcije članice `allocate` i `deallocate`. To smo uradili zato što i `std::vector` i operator `new` pozivaju konstruktore odgovarajućeg tipa, dok `allocate` jednostavno alocira dovoljno prostora za taj tip, bez konstrukcije. Generalno, za proizvoljan tip `T` ne možemo da garantujemo da podržava bilo koji određeni konstruktor. Najzad, u zavisnosti od tipa konstrukcija može biti skupa, a u sklopu samog algoritma nema nikakvu ulogu: bitno nam je da imamo dovoljno prostora u memoriji gde bismo mogli da spojimo podnizove, a ne šta se u toj memoriji nalazi pre samog procesa.

## 4.2 Paralelni merge i merge sort

Za realizaciju paralelnih algoritama koristićemo Intel-ovu TBB biblioteku i apstrakciju zadataka koju ona pruža. Svaki od naših algoritama biće umotan u klasu koja nasleđuje `tbb::task` (ili neku njenu subklasu). Za `tbb::task` specifična je funkcija članica `execute`, čije telo predstavlja posao delegiran tom zadatku, a koja vraća pokazivač na sledeći zadatak koji treba izvršiti. Kompozicijom zadataka stvaramo usmeren aciklični graf (DAG) koji predstavlja ukupan algoritam koji implementiramo. U našem slučaju, DAG će biti jednostavno binarno stablo, ali dosta paralelnih algoritama zahteva komplikovaniju reprezentaciju i koordinaciju između individualnih zadataka. Kako poziv ovako realizovanog algoritma nije trivijalan, njega ćemo umotati u funkciju sa identičnim interfejsom onom u odgovarajućoj serijskoj verziji.

Na dalje, posmatraćemo samo `execute` funkcije klase, jer se isključivo u njima zapravo dešava nešto. Sve promenljive članice direktno odgovaraju ulazim parametrima `serial::sort` i `serial::merge_impl`, a svi konstruktori samo postavljaju odgovarajuće promenljive u zadato stanje.

Posmatraćemo prvo klasu `sort_impl`:

---

```
1  template <typename It, typename P>
2  class parallel::sort_impl : public task {
3  private:
4      // member variables and sort_continuation definition
5  public:
6      // constructor
7      // ...
8      auto execute() -> task* {
9          auto const size = static_cast<size_t>(last - first);
10         if (size <= sort_cutoff) {
11             serial::sort(first, last, p);
12             return nullptr;
13         }
14
15         auto const mid = first + size/2;
16
17         auto& c = *new (allocate_continuation()) sort_continuation{
18             first, mid, last, p
19         };
20         c.set_ref_count(2);
21
22         auto& left = *new (c.allocate_child()) sort_impl{first, mid, p};
23         spawn(left);
24
25         recycle_as_child_of(c);
26         first = mid;
27         return this;
28     }
29     };
```

---

Kao što smo diskutovali pri opisivanju generalnih karakteristika paralelnog merge sort-a, i ovde definišemo `cutoff` (ovde definisanu kao `sort_cutoff`) konstantu koja tera algoritam da pređe u serijski režim ukoliko je ulazni interval iteratora dovoljno mali. Mešavinom eksperimentalne metode i nezavisnog istraživanja, za ovu implementaciju definišaćemo `sort_cutoff` kao 2048. Pri analizi performansi algoritama variraćemo vrednost `cutoff` u cilju razumevanja ponašanja karakteristika performansi u odnosu na nju. Druga polovina `execute` raspoređuje dva zadatka koji odgovaraju sortiranju polovine niza, uključujući optimizaciju stvaranja što manje instanci zadataka.

Ideja je da određene zadatke koji nemaju šta da rade, a nisu uništeni jer čekaju druge zadatke recikliramo umesto da konstanto pravimo nove. Za ovo nam je koristan koncept kontinucija. Kontinucije su zadaci čija je namena da čekaju druge zadatke i eventualno odrade neki posao nakon čekanja. U `parallel::sort_impl::execute` konstruišaćemo kontinuciju `c` (linije 17–20), koja će pratiti izvršavanje sortiranje polovine niza, pa konstruisati i pokrenuti novi zadatak za sortiranje levog podniza (linije 22–23) i reciklirati, trenutni zadatak tako da sortira desni podniz (linije 25–27).

Posao `parallel::sort_impl::sort_continuation` je jednostavan:

---

```

1  template <typename It, typename P>
2  class parallel::sort_impl : public task {
3  private:
4      // sort_impl member variables
5      struct sort_continuation {
6          // sort_continuation member variables
7          auto execute() -> task* {
8              parallel::merge(first, mid, last, p);
9              return nullptr;
10         }
11     };
12 public:
13     // constructor
14     // execute function
15 };

```

---

Ona samo čeka završetak zadataka koji su joj dodeljeni na čekanje (ovo je implicitno u tome da se `sort_continuation` u `sort_impl::execute` alocira kao kontinucija i postavlja sve svoje podzadatke kao podzadatke kontinucije), i nakon toga poziva funkciju koja paralelno spaja podnizove `[first, mid)` i `[mid, last)`.

Javni interfejs za poziv ovog algoritma svodi se na kreiranje i izvršavanje korenskog zadatka:

---

```

1  template <typename It, typename P = less<>>
2  void parallel::sort(It first, It last, P&& p = {}) {
3      if (last - first <= sort_cutoff) {

```

```

4     serial::sort(first, last, p);
5     return;
6 }
7
8     auto& task = *new (task::allocate_root()) sort_impl{
9         first, last, p
10    };
11    task::spawn_root_and_wait(task);
12 }

```

---

Dodaćemo na samom početku tela funkcije još jednu proveru za prelazak u serijski režim, da ne bismo trošili procesorske cikluse na alociranje TBB zadatka, koje je u principu netrivialno skupo. Ovom transformacijom bi trebalo da smo izjednačili performanse paralelnog i serijskog algoritma za male nizove.

`parallel::merge_impl` klasa funkcioniše na sličan način:

---

```

1  template <typename It, typename Out, typename P>
2  class parallel::merge_impl : public task {
3  private:
4      // member variables and merge_continuation definition
5  public:
6      // constructor
7      // ...
8      auto execute() -> task* {
9          auto l_size = static_cast<size_t>(l_last - l_first);
10         auto r_size = static_cast<size_t>(r_last - r_first);
11
12         if (l_size < r_size) {
13             swap(l_first, r_first);
14             swap(l_last, r_last);
15             swap(l_size, r_size);
16         }
17
18         if (l_size == 0)
19             return nullptr;
20
21         if (l_size + r_size <= merge_cutoff) {
22             serial::merge_impl(
23                 l_first, l_last,
24                 r_first, r_last,
25                 o_first,
26                 p
27             );
28             return nullptr;
29         }
30
31         auto const midpt = l_first + l_size/2;
32         auto const parpt = lower_bound(r_first, r_last, *midpt, p);
33         auto const inspt = o_first + (midpt - l_first) + (parpt - r_first);
34         *inspt = *midpt;

```

```

35
36     auto& c = *new (allocate_continuation()) merge_continuation;
37     c.set_ref_count(2);
38
39     auto& right = *new (c.allocate_child()) merge_impl{
40         midpt + 1, l_last,
41         parpt, r_last,
42         inspt + 1,
43         p
44     };
45     spawn(right);
46
47     recycle_as_child_of(c);
48     l_last = midpt;
49     r_last = parpt;
50
51     return this;
52 }
53 };

```

---

Na sličan način kao i u `parallel::sort_impl`, i ovde koristimo `cutoff` za prelazak u serijski režim rada (u ovom slučaju nazvana je `merge_cutoff`), koja je takođe definisana kao 2048. U prvoj polovini `execute` radimo posao koji smo izveli u diskusiji paralelnog merge algoritma. Zatim, na sličan način kao i u `parallel::sort_impl` klasi, konstruišemo kontinuiranu `c`, konstruišemo prvi podzadatak i recikliramo trenutni zadatak kao drugi podzadatak.

U ovom slučaju, struktura koja predstavlja kontinuiranu je trivijalna, jer je spajanje podnizove pooptuno u nadležnosti ne-kontinuiranih zadataka (nasuprot sortiranju, gde nakon sortiranja podnizova moramo još i da ih spojimo).

---

```

1  template <typename It, typename Out, typename P>
2  class parallel::merge_impl : public task {
3  private:
4      // member variables
5      struct merge_continuation : empty_task { };
6  public:
7      // constructor
8      // execute function
9  };

```

---

`tbb::empty_task` je subklasa `tbb::task` čija `execute` funkcija ne radi ništa, što je savršeno za naše potrebe, iz malopre navedenih razloga (setimo se još jednom da je čekanje na zadatke jedne kontinuirane implicitno definisano pri konstrukciji te kontinuirane i dodavanja podzadataka toj kontinuiranoj).

Da bismo adaptirali interfejs poziva paralelnog merge algoritma interfejsu `serial::merge` i interfejsu koji očekuje njen poziv u `sort_impl::execute`, moraćemo malo da se pomučimo:



---

```

1  template <typename It, typename P = less<>>
2  void merge(It first, It mid, It last, P&& p = {}) {
3      auto const size = static_cast<size_t>(last - first);
4
5      if (size <= merge_cutoff) {
6          serial::merge(first, mid, last, p);
7          return;
8      }
9
10     auto alloc = allocator<value_t<It>>{};
11     auto *into = alloc.allocate(size);
12     auto& task = *new (task::allocate_root()) merge_impl{
13         first, mid,
14         mid, last,
15         into,
16         p
17     };
18     task::spawn_root_and_wait(task);
19
20     parallel_for(
21         blocked_range<size_t>{0, size},
22         [&] (blocked_range<size_t> const& range) {
23             for (auto i = range.begin(); i != range.end(); ++i) {
24                 auto pos = first + i;
25                 *pos = move(into[i]);
26             }
27         }
28     );
29     alloc.deallocate(into, size);
30 }

```

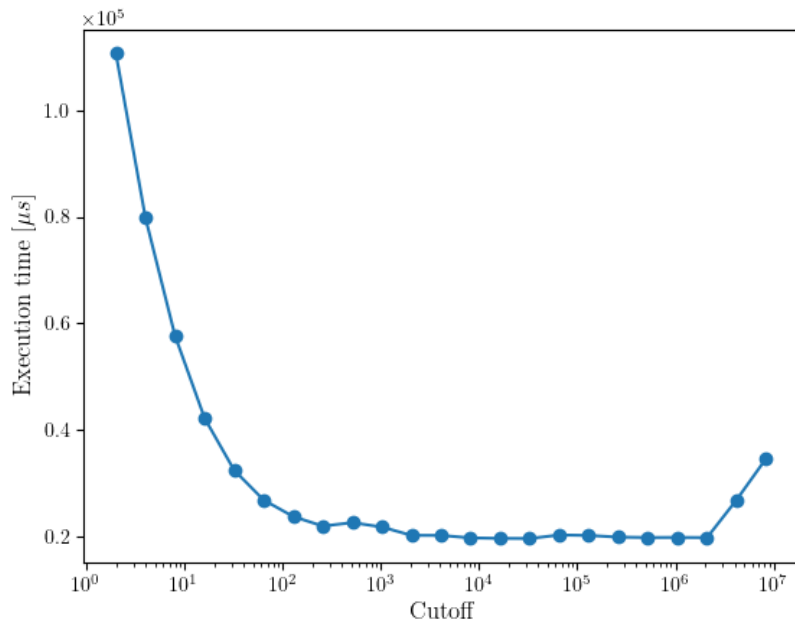
---

Slično kao i za paralelni merge sort, i ovde imamo dodatnu proveru za male intervale, dinamičku alokaciju preko `std::allocator` i konstrukciju i pozivanje korenskog zadatka. Međutim, na kraju funkcije treba da iz privremenog niza u koji smo spojili podnizove pomerimo elemente u originalni niz. U serijskoj verziji, to smo radili jednostavnim pozivom funkcije `std::move` nad relevantnim iteratorima, ali ovde imamo priliku da to paralelizujemo, što je savršena prilika da iskoristimo `tbb::parallel_for`. Ako zanemarimo detalje postavke petlje, `tbb::parallel_for` svodi se na običnu `for` petlju (ili poziv `std::for_each`), gde se na deklarativan način TBB biblioteci prepušta odgovornost raspoređivanja niti za izvršavanje iteracija petlje.

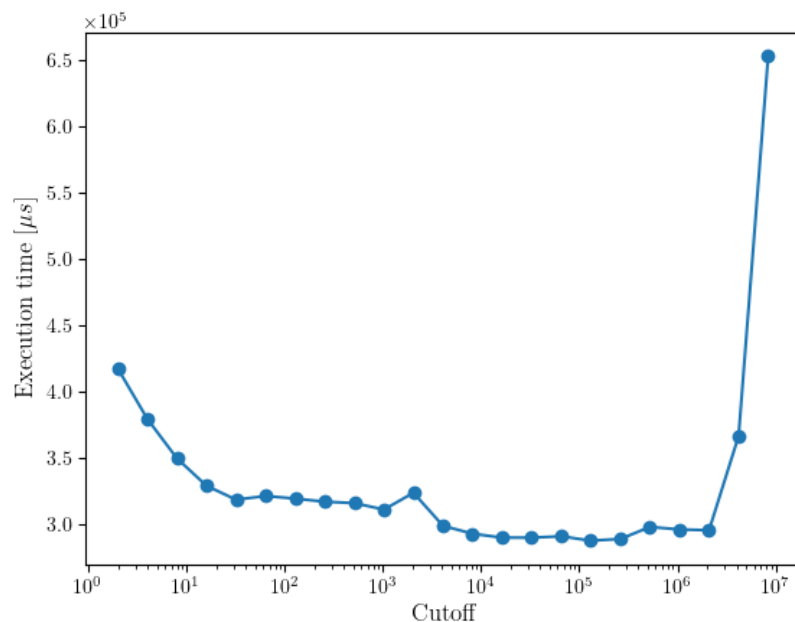
## 5 Analiza performansi

Naši algoritmi testirani su na nizovima uniformno raspoređenih celih brojeva u dve glavne sfere: prvo smo fiksirali veličinu niza na najveću koja se razmatra (što je u ovom slučaju deset miliona elemenata, ali priloženi programski kod nudi jednostavan način za modifikaciju razmatranih veličina nizova) i varirali vrednost `cutoff` – na osnovu toga doneli smo zaključak o najboljoj vrednosti za dati algoritam. Onda smo, za sve definisane veličine nizova merili vreme izvršavanja sve tri varijante svakog algoritma: serijsku verziju, paralelnu verziju i paralelnu STL verziju. Sva merenja izvršena su po deset puta, a među tih deset kao reprezentativna vrednost izabran je njihov medijan. Medijan je izabran zato što su se kod paralelnih rešenja (i našeg, ručnog, i STL) ponekad javljale anomalije, gde je vreme izvršavanja paralelne verzije bilo nekoliko puta manje od vremena izvršavanja serijske verzije. Okruženje u kom su se rešenja kompajlirala opisano je u uvodu, a optimizaciono podešavanje stavljeno je na -O3. U nastavku teksta rezultati su prikazani grafički, dok su tabele sa tačnim podacima u sekciji Tabele.

Slika 1: Vreme izvršavanja `parallel::merge` za 10000000 elemenata u zavisnosti od `cutoff` parametra



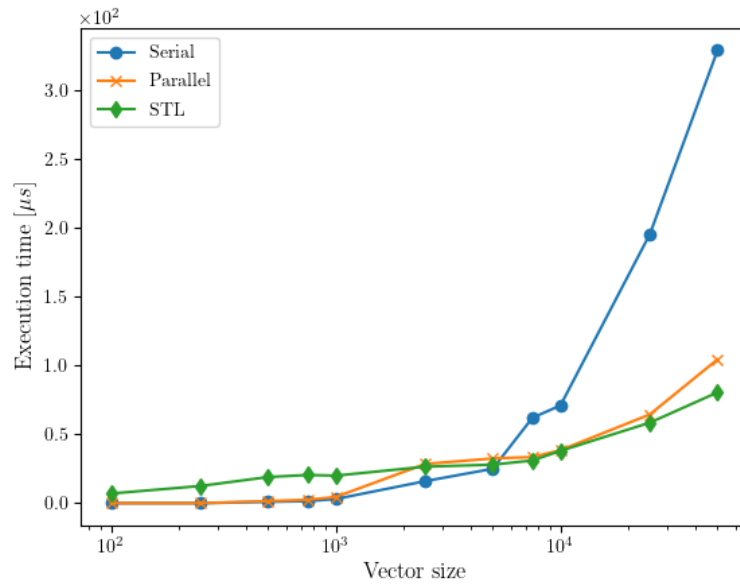
Slika 2: Vreme izvršavanja `parallel::sort` za 10000000 elemenata u zavisnosti od `cutoff` parametra



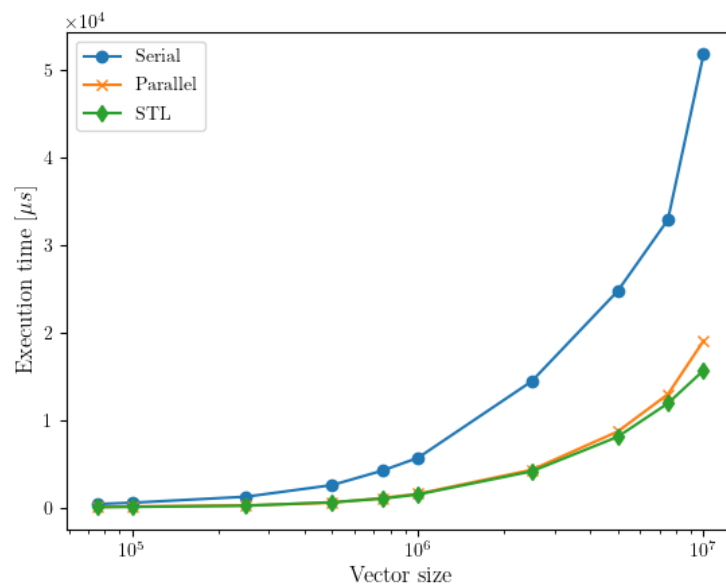
Kao što se sa prethodnih figura vidi, najbolje performanse što se tiče oba algoritma postizemo za vrednosti `cutoff` u intervalu  $10^3$ - $10^4$ , što su na prvi pogled ogromne cifre, ali se ipak slažu sa drugim objavljenim merenjima i potiču više iz same prirode algoritma i osobina procesorske arhitekture nego iz implementacionih detalja.

Za merenja performansi u odnosu na veličinu niza, za obe implementacije izabrane su `cutoff` vrednosti od 2048. U nastavku slede rezultati istih:

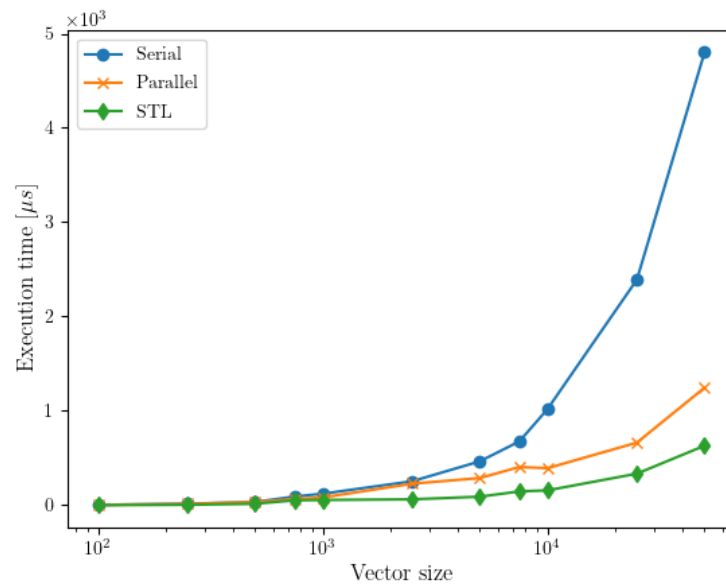
Slika 3: Vreme izvršavanja merge algoritama u odnosu na veličinu niza za nizove do 50000 elemenata



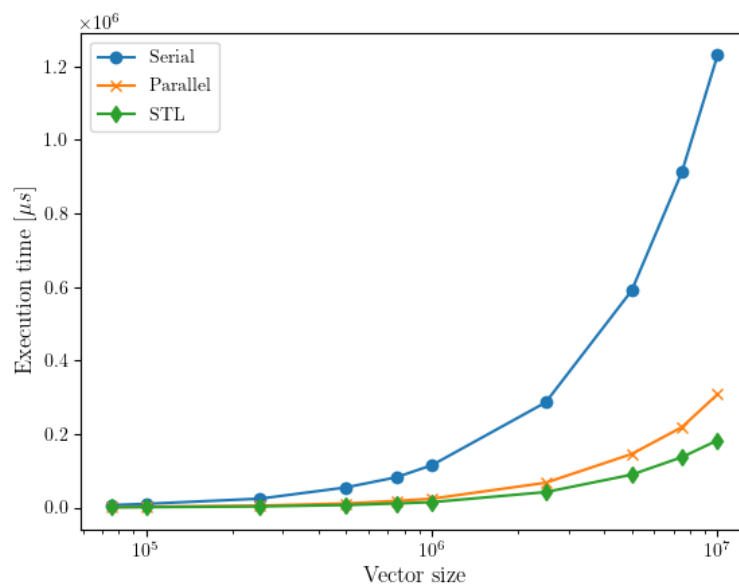
Slika 4: Vreme izvršavanja merge algoritama u odnosu na veličinu niza za nizove do 10000000 elemenata



Slika 5: Vreme izvršavanja sort algoritma u odnosu na veličinu niza za nizove do 50000 elemenata



Slika 6: Vreme izvršavanja sort algoritma u odnosu na veličinu niza za nizove do 10000000 elemenata



Kao što je i očekivano, jer smo koristili `cutoff` parametar, performanse

naših serijskih i paralelnih verzija praktično su identične za dovoljno male nizove. Ono što je interesantno da, iako ponekad neznatno sporiji, adekvatni STL algoritam takođe ima serijsko ponašanja za male nizove. Već za nizove reda veličine  $10^4$  primete se efekti paralelizacije, gde je ubrzanje paralelne verzije 4, a STL verzije 8 za sortiranje i 3, odnosno 4 za spajanje. Slična ubrzanja možemo pročitati i za velike nizove, što je u skladu sa ograničenjima odgovarajućih karakteristika procesora.

## 6 Zaključak

Paralelizacija je, u vreme kada praktično svi moderni procesori imaju određene višenitne sposobnosti, često nužna ukoliko želimo da naš softverski projekat bude zadovoljavajuće performantan. Paralelizacija je, takođe, teška. Često nije očigledno na koji način određeno računanje možemo podeliti na nezavisna podračunavanja, gde ponekad to zahteva radikalnu promenu u načinu razmišljanja. Van algoritamske strane, postoje razne za arhitekturu specifične optimizacije i jedinstvenosti koje na naše računanje mogu uticati na najrazličitije načine, samo neke od kojih možemo razumno predvideti. Zbog toga je izuzetno bitno da ne pribegavamo ručnom pisanju koda za svaki problem koji treba rešiti – često je slučaj da je neka njegova varijanta već rešena. U našem slučaju, standardna C++ biblioteka pisana je od strane eksperata u odgovarajućim poljima i ima pravo, za razliku od nas, da uđe u implementacione detalje kompajlera da bi postigla optimalne karakteristike u vidu performansi, elegantnosti interfejsa, portabilnosti i nazadne kompatibilnosti. Ukoliko je to moguće, a moguće je češće nego što se u praksi misli, skoro uvek je bolje koristiti gotove, testirane biblioteke naspram pisanja sopstenog koda.

## 7 Tabele

cutoff	Vreme izvršavanja [ $\mu s$ ]
2	110678.5
4	79825.0
8	57674.5
16	42181.0
32	32413.0
64	26793.5
128	23698.5
256	21902.0
512	22550.5
1024	21729.5
2048	20158.5
4096	20151.0
8192	19685.0
16384	19590.5
32768	19567.0
65536	20227.5
131072	20145.0
262144	19826.5
524288	19750.0
1048576	19769.0
2097152	19739.5
4194304	26843.5
8388608	34614.5

Tabela 1: Tabela za sliku 1



cutoff	Vreme izvršavanja [ $\mu s$ ]
2	417143.5
4	379141.0
8	349458.5
16	328836.0
32	318371.5
64	321162.0
128	319199.0
256	316855.5
512	315822.5
1024	310921.5
2048	323567.0
4096	298674.5
8192	292707.0
16384	289794.5
32768	289757.0
65536	290774.0
131072	287494.5
262144	288763.0
524288	297823.5
1048576	295958.5
2097152	295340.5
4194304	366401.0
8388608	652619.5

Tabela 2: Tabela za sliku 2

Dužina niza	Serijska verzija [ $\mu s$ ]	Paralelna verzija [ $\mu s$ ]	STL verzija [ $\mu s$ ]
100	0.0	0.0	7.0
250	0.0	0.0	12.5
500	1.0	1.5	19.0
750	1.5	2.5	20.5
1000	3.0	4.5	20.0
2500	16.0	28.5	26.5
5000	25.0	32.5	28.0
7500	62.0	33.5	31.0
10000	71.0	38.5	38.0
25000	195.5	64.5	58.5
50000	329.0	104.5	80.5

Tabela 3: Tabela za sliku 3

Dužina niza	Serijska verzija [ $\mu s$ ]	Paralelna verzija [ $\mu s$ ]	STL verzija [ $\mu s$ ]
75000	463.0	147.0	105.5
100000	615.0	182.5	140.5
250000	1302.5	312.5	271.5
500000	2628.5	618.0	673.5
750000	4277.5	1158.0	1074.5
1000000	5744.0	1635.0	1562.5
2500000	14468.0	4370.5	4195.5
5000000	24758.0	8730.5	8133.0
7500000	32895.5	12986.5	11950.5
10000000	51780.0	19090.5	15715.0

Tabela 4: Tabela za sliku 4

Dužina niza	Serijska verzija [ $\mu s$ ]	Paralelna verzija [ $\mu s$ ]	STL verzija [ $\mu s$ ]
100	4.0	3.5	1.0
250	16.0	15.0	5.5
500	34.0	33.5	15.5
750	92.5	65.5	52.5
1000	122.0	82.5	56.0
2500	254.5	228.5	63.0
5000	465.5	288.5	91.0
7500	677.0	405.5	147.0
10000	1019.0	393.0	158.0
25000	2391.5	664.0	335.0
50000	4801.5	1246.5	631.5

Tabela 5: Tabela za sliku 5

Dužina niza	Serijska verzija [ $\mu s$ ]	Paralelna verzija [ $\mu s$ ]	STL verzija [ $\mu s$ ]
75000	7078.5	1827.5	973.5
100000	10478.5	2682.0	1462.5
250000	24594.5	5589.5	3473.5
500000	55122.5	11541.0	7150.0
750000	82822.0	18382.5	11485.5
1000000	115874.0	24493.5	15079.0
2500000	286312.0	67821.0	42449.5
5000000	590154.0	145925.5	89771.0
7500000	912856.5	218930.0	137352.0
10000000	1230626.5	308406.5	183248.5

Tabela 6: Tabela za sliku 6