GRUPA: 2
ZADATAK: 5
NAZIV: Merge sort

IME: Aleksandar
PREZIME: Tulic
INDEKS: 1179/18

Ukoliko zelite da pokrenete neku od aplikacija preporucujem da prvo procitate sekciju ovog dokumenta **NAPOEMENE.**

Postoji nekoliko nacina da odradimo paralelizam programa, pri čemu sam ja koristio dva pristupa:

- **Shared Memory Programming - openMP**
- **Message Passing Interface**

SHARED MEMORY PROGRAMMING - openMP

Smatram da nema potrebe da objasnjavam kako funkcionise sami openMP jer se to radilo na vjezbama, ili ukoliko treba da se to radi na odbrani projektnog zadatka.

Kada govorimo o merge sort algoritmu glavni dio programa je funkcija(bez paralelizma):

```
void mergeSort(int *a, int size, int *p){
        if ( size > 1  ){
                mergeSort(a, size / 2, p);
                mergeSort(a + size / 2, size - size / 2, p);
                merge(a, size, p);
        }
}
```

Ovde dolazi dijeljenja na dva dijela pri čemu svaki treba da se sortira i to tako ide sve dok velicina niza koji treba da se sortira ne bude 1 tada je sortiran(bez paralelizacije = standardno).

Ali posto ovi pozivi se obaziru na različite regione tj. radice operacije sortiranja na razlicitim dijelovima niza **a** onda ovde mozemo da implementiramo paralelizam.

Koristio sam #pragma omp task firstprivate(a, size, p)

- firstprivate - napominje da elementi koji se koriste pri pozivu su inicijalizovani sa onim prije vrijednostima koje su imali prije paralelnog regiona

Zasto sam koristio #pragma omp taskwait?

- specifikuje(definise) čekanje završetka child procesa trenutnom zadatku

Takodje jedan od najvaznijih detalja kod implementacije i **MPI-A I openMP** jeste da, umjestoda kreiramo neku promjenljivu svaki put kada pozovemo merge da bi mogli privremeno skladistiti podatke mi proslijedimo jedan ili dva pokazivaca na neki niz koji ce biti iste dimenzije kao i niz koji se sortira.

Možda se pitate zasto je to toliko bitno?

- Pa zato sto ta funkcija se takodje veliki broj puta poziva i sad zamislite da za duzina samo dva elementa i milion pta poziva merge da moramo alocirati memoriju i onda da se kasnije destruktor poziva. To uništava vrijeme izvršavanja algoritma
- Npr. da smo tako implementirali tako tj prethodno navedeno onda bi sa dva thread-a imali bolje vrijeme izvršavanja ali u poredjenju sa tablicom dole navedenom za openMP merge sort imalo bi losije rezultate
- i kada bi povecavali broj thread-ova koji bi se koristio imali bi slicne rezultate kao sam dva thread-a jer vrijeme ce se najvecim dijelom odredjivati sa tom alokacijom memorije i dealokoacijom.

Tako da prethodno navedena funkcija bi sada bila:

```
void mergeSort(int *a, int size, int *p){
        if ( size > 1 ){
                #pragma omp task firstprivate(a, size, p)
                mergeSort(a, size / 2, p);
                #pragma omp task firstprivate(a, size, p)
                mergeSort(a + size / 2, size - size / 2, p);
                #pragma omp taskwait
                merge(a, size, p);
        }
}
```

Zasto sam radio size - size / 2?

- pa ako je size = 8 tj paran onda je sljedece logicno:
    - nova vrijednost = 8 - 8 / 2 = 8 - 4 = 4 tj isto kao i size / 2
- ali ako je size = 7 tj neparan broj onda je
    - nova vrijednost = 7 - 7 / 2 = 7 - 3 = 4 dok size / 2 = 3

Zasto sam radio a + size / 2?

- pa zato sto zelim da radim nad različitim dijelovima niza tj da se ne isprepliću ti dijelovi i onda bi se vjerojatno i izgubili neki podaci tj jedni bi druge prepisali
- a i samom tom dijelu nikad necu pristupiti jer drugi dio je zadužen za sortiranje tog dijela niza

Zasto sam radio sljedece:

- **#pragma omp parallel**
  - ovo sam postavio jer trebaju zadaci da se izvrše paralelno tj. kreiramo(napravimo dostupnim) osam thread-ova (niti, zavisi koliko je korisniku dostupno)
- **#pragma omp single**
  - dok sa ovim mi pozivamo mergeSort da se izvršava samo sa jednim thread-om
  - kada se sada nadjemo u funkciji mergeSort dodjeljuju se dva thread-a on njih osam za izvršavanje dva task-a(posla)
  - ne može se znati koji ce po redu kojem poslu se dodijeliti

MASSAGE PASSING INTERFACE - MPI

- eksplicitna komunikacija između procesa kao da šaljemo i primamo neke poruke

MPI_Init(&argc, &argv)

- sa ovim inicijalizujemo MPI okruženje

MPI_Comm_rank(MPI_COMM_RANK, &rank)

- pronalazimo **rank** procesa iz zadatog komunikatora(tj. MPI_COMM_WORLD), tj. iz grupe procesa

Vjerojatno se pitate šta je MPI_COMM_WORLD:

- mpi procesi se mogu grupisati u grupe
- svaka grupa može da ima vise boja, nekad se nazivaju i konteksti
- grupa i boja određuju komunikator tj. naziv grupe procesa
- kada se MPI aplikacija starta grupi svih procesa daje se zajednicko ime MPI_COMM_WORLD

MPI_Comm_size(MPI_COMM_WORLD, &size)

- pronalazimo veličinu grupe procesa koji pripadaju MPI_COMM_WORLD

MPI_Finalize()

- unistavamo okruženje izvršavanja MPI procesa

MPI_Scatter(inputArr, n / size, MPI_INT, processArr, n / size, MPI_INT, 0, MPI_COMM_WORLD)

- ideja je da razlomimo ulazni niz brojeva na zadani broj procesa da se paralelno izvrsava ali sada se nemoramo brinuti o prepisu podataka sto je dobro jer svaki ima svoju memoriju ali se medjusobno salju podaci tako da na kraju kada svi dijelovi zavrse sa sortiranjem salju procesu sa rank = 0 i on ce to da objedini

- ali ukoliko broj procesa koji je na ulazu zadan nije djeljiva sa brojem podataka onda kada zavrsimo sa sortirajem pojedinacnih dijelova kao sto sam rekao prethodno, oni se objedine u jedan niz nazvna sa **res** i sada podatke koji su ostali an kraju(zato broj elemenata nije djeljiv sa procesom) se dodjeljuje na kraj res-a i ponovo vrsimo sort ali serijski tj. ne koristimo paralelizam
- inputArr - niz koji treba da sortiramo
- n / size - po koliko elemenata se uzima iz inputArr
- MPI_INT - podaci koji se šalju drugim procesima su tipa INT
- processArr - gdje se skladište podaci koji se razlamaju iz inputArr
- 0 - sa kog procesa se salju ti elementi
- MPI_COMM_WORLD - koji komunikator se koristi

MPI_Gather(processArr, n / size, MPI_INT, res, n / size, MPI_INT, 0, MPI_COMM_WORLD)

- slično kao prethodno samo sto se sada skupljaju podaci u jedan niz

MPI_Barrier(MPI_COMM_WORLD)

- dolazi do blokade dok svi procesi ne dođu do ovog mjesta

Očekivano je da openMP ima najbolje rezultate, malo se javljaju problemi sa MPI da kada se poveca broj procesa koji se koristi da nema bolje rezultate kao sa manjim brojem procesa ali kada o tome pricamo moramo obratiti paznju i na to da svaki poziv kosta vremena kao i vrijeme na cekanje sa MPI_Barrier(MPI_COMM_WORLD) i sl.

Mozemo primjetiti da kada radimo bez paralelizma da imamo nekad bolje vrijednosti nego sa MPI-om pogotovo ako su male vrijednosti dok kada su vece onda su bolje vrijednosti u MPI ili priblizno isto.

Koristeci obadva pristupa dali pojedinačno ili odvojeno u većini slučajeva ostvaruju bolje rezultate.

- **NAPOMENE:**
  - preporucujem da ne koristite u vidual studio-u jer on podrzava samo verziju openMP 2.5 dok #pragma omp task je implementiran tek od openMP 3.0
  - instalacija na linux ubuntu **mpi**
    - sudo apt-get install libopenmpi-dev
    - sudo apt-get install openmpi-bin
  - kompajliranje i izvršavanje mog programa sa mpi.cpp:
    - mpic++ mpi.cpp Time.cpp -o mpi
    - **mpirun -n [vasa_vr] ./mpi [velicina_niza]**

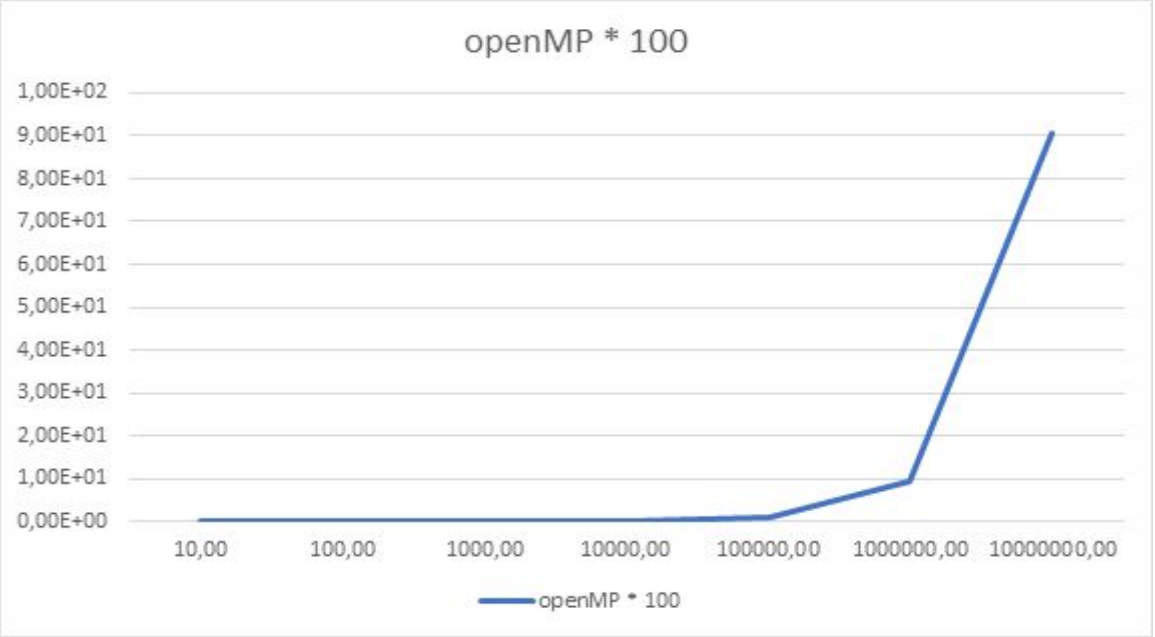
- **NISAM SE TOLIKO OBAZIRAO NA GREŠKE KAO STO JE VELICINA NIZA > 0 I SL. JER SMATRAM DA JE VAŽNIJA IMPLEMENTACIJA SAMOG ALGORTMA KORISTECI MPI I openMP**
- kompajliranje i izvršavanje mog programa sa openMP.cpp:
    - g++ -fopenmp openMP.cpp -o openMP
    - ./openMP pa onda unosite veličinu niza
- kompajliranje i izvršavanje mog programa sa openMP I MPI hybrid.cpp:
    - mpic++ -fopenmp hybrid.cpp -o hybrid
    - mpirun -n [vasa_vr] ./hybrid [velicina_niza]

Informacije o procesoru:

- Intel(R) Core(TM) i7-4770 CPU @ 3.4GHz
- Broj jezgara: 4
- Broj logičkih procesora: 8

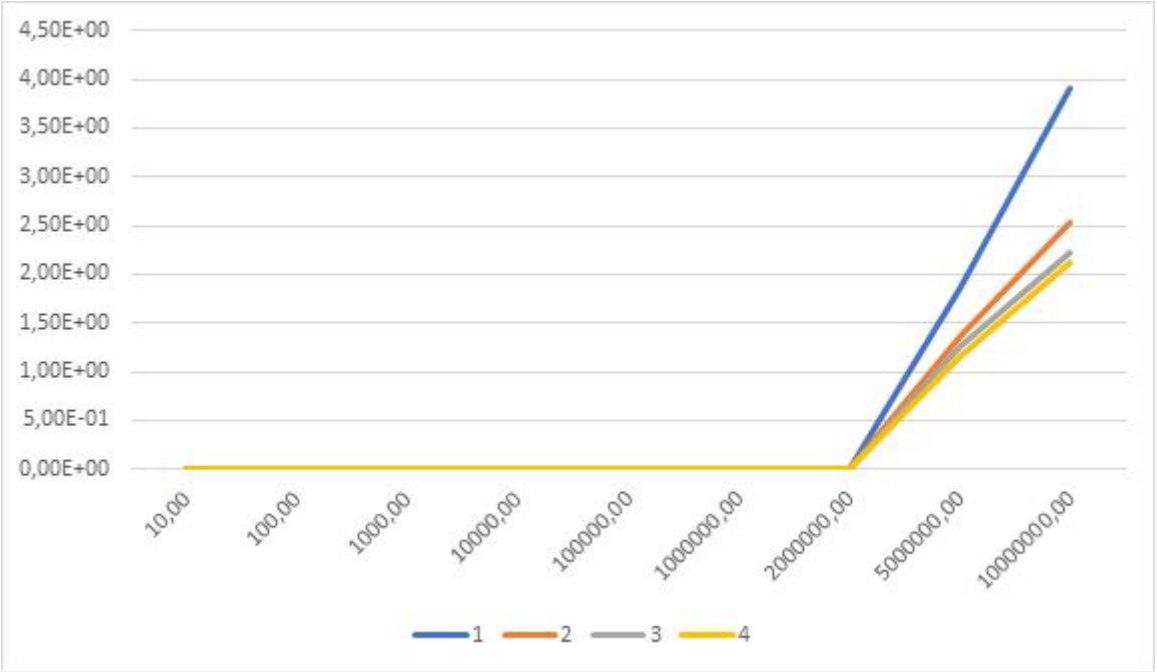
- **openMP merge sort**

Elemenata	Time[s] * 100
10,00	0.0372157
100,00	0.0511461
1000,00	0.0659816
10000,00	0.568284
100000,00	1,12E+00
1000000,00	9,49E+00
2000000,00	1,74E+01
5000000,00	4,22E+01
10000000,00	9,05E+01

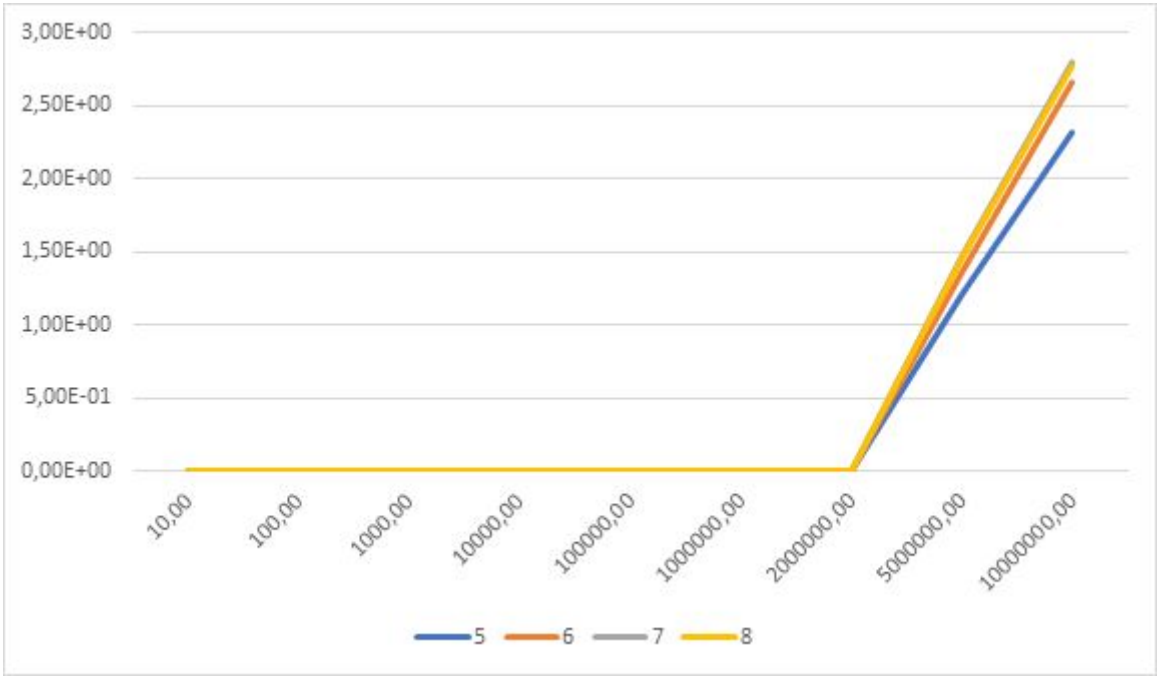


- **MPI merge sort**

Elemenata	1 Time[s]	2 Time[s]	3 Time[s]	4 Time[s]
10,00	0.237160	0.234729	0.251925	0.245493
100,00	0.237361	0.23319	0.256873	0.252583
1000,00	0.231640	0.249784	0.247632	0.250513
10000,00	0.243019	0.24493	0.250472	0.248173
100000,00	0.278937	0.279301	0.293074	0.273825
1000000,00	0.533216	0.450166	0.438214	0.442995
2000000,00	0.858652	0.672805	0.618739	0.619931
5000000,00	1,86E+00	1,36E+00	1,26E+00	1,16E+00
10000000,00	3,91E+00	2,53E+00	2,23E+00	2,12E+00

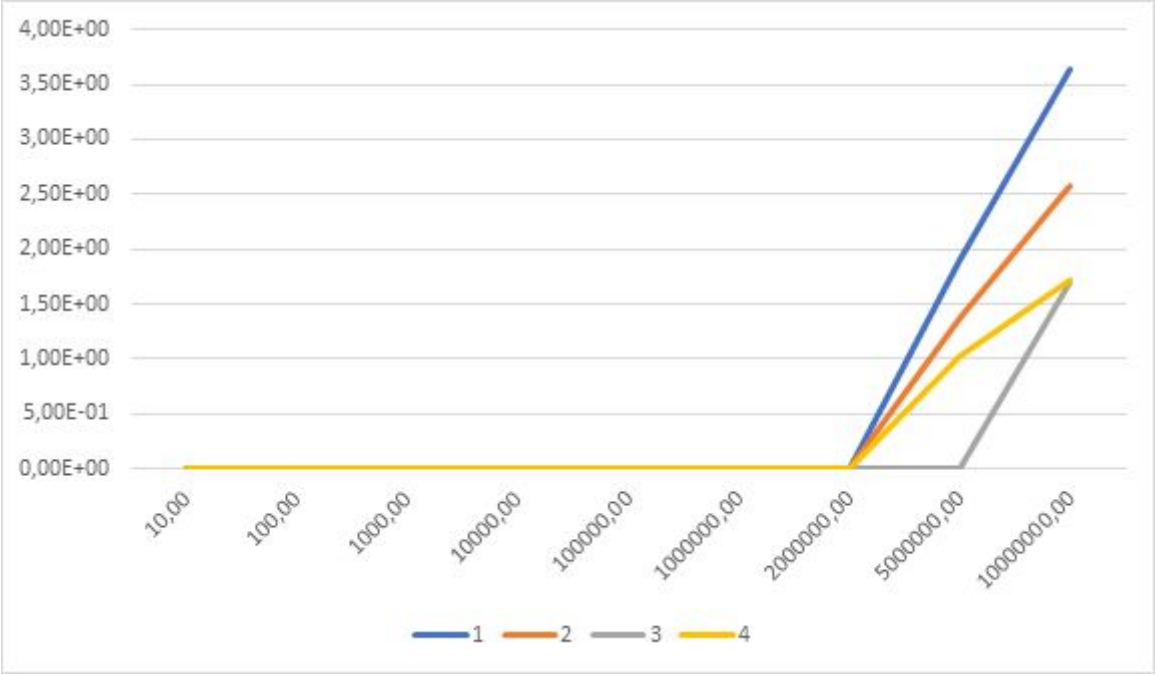


Elemenata	5 time[s]	6 time[s]	7 time[s]	8 time[s]
10,00	0.25292	0.275063	0.267386	0.27698
100,00	0.26251	0.275513	0.262727	0.269295
1000,00	0.251464	0.279105	0.302216	0.281907
10000,00	0.250226	0.270029	0.27675	0.278181
100000,00	0.288871	0.288509	0.313849	0.348775
1000000,00	0.502443	0.50039	0.498506	0.525626
2000000,00	0.642457	0.687943	0.676877	0.733802
5000000,00	1,21E+00	1,36E+00	1,47E+00	1,46E+00
10000000,00	2,32E+00	2,67E+00	2,80E+00	2,77E+00

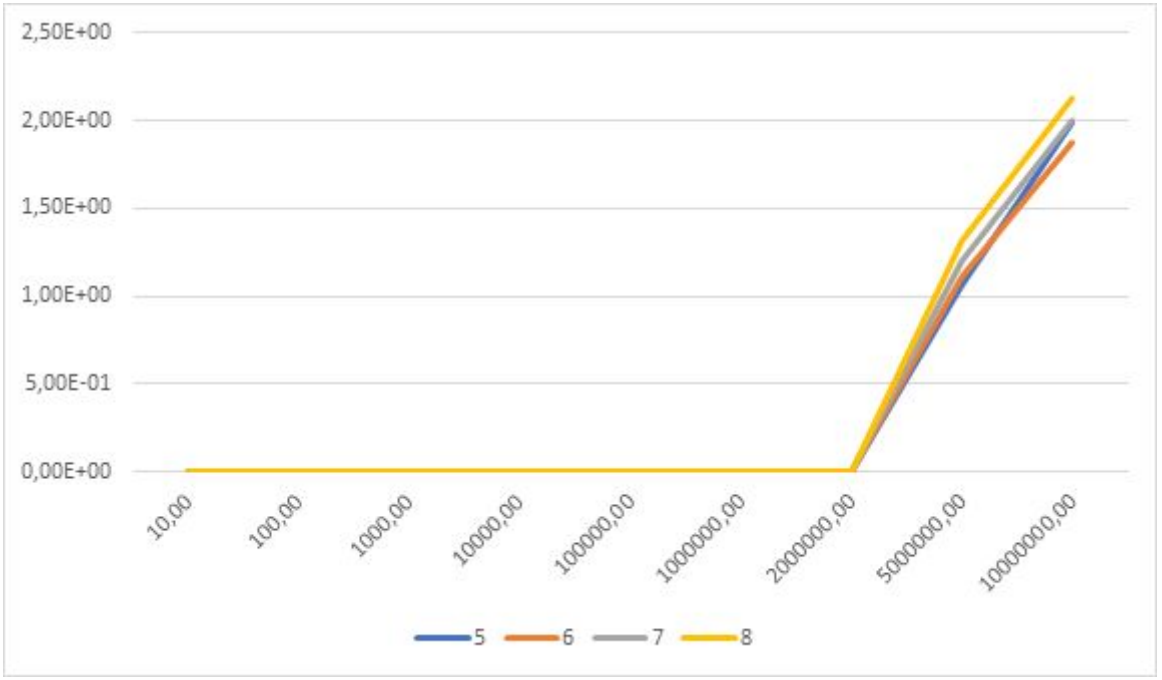

3,00E+00
2,50E+00
2,00E+00
1,50E+00
1,00E+00
5,00E-01
0,00E+00
10,00
100,00
1000,00
10000,00
100000,00
1000000,00
2000000,00
5000000,00
10000000,00
5
6
7
8

- **openMP i MPI merge sort**

Elemenata	1 Time[s]	2 Time[s]	3 Time[s]	4 Time[s]
10,00	0.232804	0.234652	0.297525	0.302539
100,00	0.234671	0.2353	0.29048	0.314295
1000,00	0.23453	0.241268	0.278655	0.307338
10000,00	0.243297	0.257244	0.282007	0.29278
100000,00	0.284367	0.278353	0.294764	0.302475
1000000,00	0.558452	0.465298	0.450449	0.445882
2000000,00	0.886936	0.690408	0.599203	0.612229
5000000,00	1,91E+00	1,37E+00	0.986005	1,02E+00
10000000,00	3,64E+00	2,58E+00	1,70E+00	1,72E+00



Elemenata	5 Time[s]	6 Time[s]	7 Time[s]	8 Time[s]
10,00	0.312103	0.324378	0.317166	0.334568
100,00	0.294898	0.321918	0.315455	0.317037
1000,00	0.296275	0.330923	0.352057	0.362451
10000,00	0.29986	0.326421	0.349581	0.342578
100000,00	0.330057	0.330847	0.367196	0.379514
1000000,00	0.450327	0.528018	0.493675	0.543219
2000000,00	0.627905	0.658694	0.731644	0.805437
5000000,00	1,06E+00	1,11E+00	1,20E+00	1,31E+00
10000000,00	1,99E+00	1,87E+00	2,00E+00	2,13E+00



- **Serial merge sort**

Elemenata	Serial Time[s]
10	2,00E-06
100	1,60E-05
1000	4,12E-04
10000	1,99E-03
100000	2,15E-02
1000000	2,64E-01
2000000	5,15E-01
5000000	1,24E+00
10000000	2,44E+00