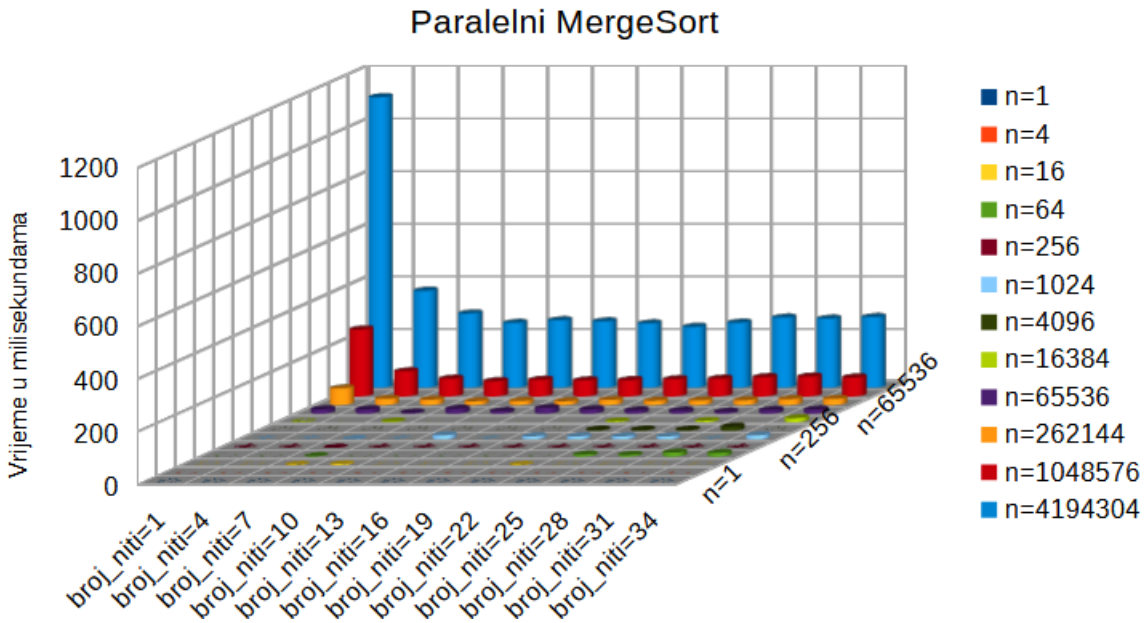
Usporedba paraleliziranog MergeSorta s paraleliziranim QuickSortom

Autor: Teo Samaržija

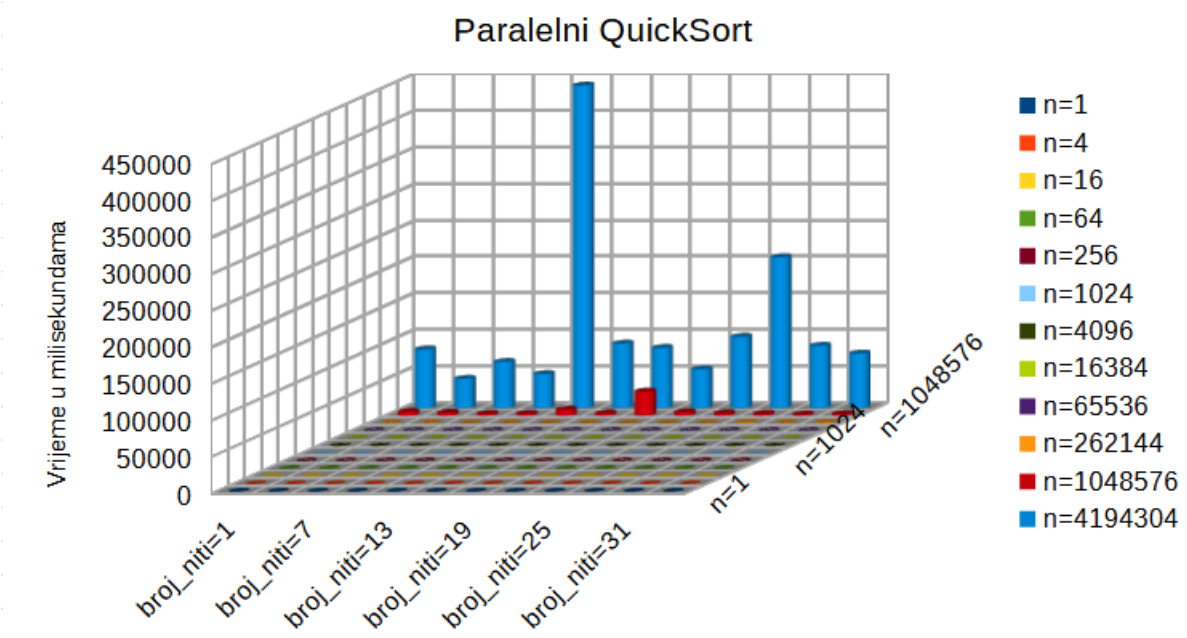
U raznorodnim programima za računala nerijetko se traži da se neki niz brojeva razvrsta po veličini (sortira): da najmanji broj bude na početku niza, a najveći na kraju niza, a svi ostali u sredinu po redu.
Najjednostavniji algoritam kojim bismo to mogli postići vjerojatno je BubbleSort (Mjehuričasto sortiranje, jer engleski *bubble* znači *mjehurić,* vjerojatno keltskog podrijetla i povezano s latinskom riječi za mjehurić *bulla.* Smatram da je moguće, ako ne i vjerojatno, da je to povezano i s ilirskom riječi *bal* u značenju *vreti*, kao u *Balissa,* antičkom nazivu za Daruvar, možda doslovno *izvor koji vrije.* Tvrditi da je postojao indoeuropski korijen sličan *bel u značenju *vreti* od kojeg dolaze sve te riječi problematično je jer je općeprihvaćeno da indoeuropski nije imao glas *b*, tako da tu srodnost trebamo objašnjavati nekako drugačije.). Ideja BubbleSorta je da prolazimo kroz niz i svaki put kad primijetimo „mjehurić" od dva broja jedan do drugoga od kojih je onaj koji dolazi prije veći od onog koji dolazi poslije, ispravimo ga, i nastavimo dalje prolaziti kroz niz. Da bismo osigurali da je niz na kraju sortiran, moramo proći kroz niz u najgorem slučaju onoliko puta koliko u nizu ima brojeva, i zato taj algoritam radi približno n^2 operacija.
Bolju ideju smislio je, i na nekom tadašnjem računalu isprogramirao, 1945. godine John Von Neumann, i njegov se algoritam zove MergeSort (sortiranje stapanjem, jer *mergere* na latinskom znači *stopiti*). Njegova je ideja da se niz rekurzivno razdvaja po pola sve dok ne dođemo do nizova veličine jedan (jer niz veličine jedan uvijek je sortiran), i zatim ih spajamo nazad dva po dva u jedan sortirani niz. Stapanje dva sortirana niza u jedan sortirani niz moguće je napraviti u broju operacija koji raste linearno s brojem elemenata u tom nizu, i broj koraka tih spajanja koje moramo napraviti raste razmjerno binarnom logaritmu broja elemenata u tom nizu, dakle, za to moramo napraviti približno $n*\log_2(n)$ operacija. Problem s MergeSortom je što za to stapanje dva poredana niza u jedan treba dodatnih mjesta u memoriji koliko ima elemenata u oba niza, što nerijetko nije opcija (ako trebamo sortirati jako velik niz, a imamo malo memorije na računalu).
Taj je problem riješio Tony Hoare 1957. godine kad je došao na ideju QuickSort algoritma (brzo sortiranje, jer je *quick* engleska riječ za *brz*, povezana preko indoeuropskog s našom rječju *živ* i latinskom rječju *vivus*). On se bazira na ideji da odaberemo određeni element u tom nizu zvan pivot (najčešće je to prvi element u nizu), i zatim preuredimo niz tako da svi elementi manji od pivota dolaze ispred pivota, a svi elementi veći od pivota dolaze nakon pivota, ali da nisu nužno poredani po veličini, što je moguće napraviti u linearnom vremenu, i zatim pokrenemo rekurziju i na elementima većima od pivota i na elementima manjima od pivota, dok ne dobijemo sortirani niz. To je, iako možda nije na prvi pogled očito kako, moguće napraviti bez dodatnog polja u memoriji računala. A, sad, koliko ćemo koraka napraviti ovisi o tome koliko sreće imamo pri odabiru pivota. Ako pivot nakon preuređivanja niza uvijek završi negdje na sredini niza, napravit ćemo $n*\log_2(n)$ operacija, i, u stvari, na gotovo svakoj današnjoj arhitekturi računala biti ćemo približno dvostruko brži nego da radimo MergeSort (jer ne moramo kopirati iz onog pomoćnog niza u memoriji nazad u originalni niz). Ali ako pivot završi na početku niza ili na kraju niza (ako je niz već približno sortiran ili obrnuto sortiran), napravit ćemo n^2 operacija i bit ćemo još sporiji nego BubbleSort.
Zadnjih nekoliko desetljeća većina računala ima više od jedne procesorske jezgre, i na prvi pogled bi se činilo da se i QuickSort i MergeSort mogu znatno ubrzati tako da se obrada različitih dijelova niza vrši na različitim procesorima, a ne slijedno na jednom procesoru. MergeSort bi mogao pokrenuti lijevu i desnu granu rekurzije na različitim procesorima, čekati da oboje završe, pa tek onda raditi operaciju stapanja dva niza, i time bismo možda mogli dobiti rezultat u vremenu koje je proporcionalno broju elemenata u nizu, a ne proporcionalno tom broju pomnoženim s njegovim logaritmom. Isto tako bi QuickSort mogao nakon što obavi to ispremještanje niza dvije drane rekurzije pokrenuti na dva različita procesora. To se radi pomoću višenitnosti, koju podržava svaki moderni operativni sustav.

Naravno, postavlja se pitanje je li za to bolji QuickSort ili MergeSort algoritam, te koliko je niti optimalno pokrenuti. Ako pokrenemo premalo niti, ne iskorištavamo procesor do kraja, a s druge strane, ako pokrenemo previše niti, gubimo na vremenu za sinkronizaciju rada među nitima (primijetite da glavna nit u MergeSortu ne smije krenuti raditi stapanje dva podniza prije no što niti koje je ona pokrenula završe s radom, inače ćemo gotovo sigurno dobiti krivi rezultat). To, naravno, ovisi o računalu na kojem radimo. Ja imam Acer Nitro 5 laptop i on ima 8 procesorskih jezgri (procesor Intel Core i5), i očekujemo da je optimalno pokrenuti približno osam niti. Napravio sam mjerenja za MergeSort:



MergeSort	n=1	n=4	n=16	n=64	n=256	n=1024	n=4096	n=16384	n=65536	n=262144	n=1048576	n=4194304
broj_niti = 1	0	0	0	0	0	0	0	1	16	63	254	1102
broj_niti = 4	0	0	0	0	0	0	0	0	16	23	95	368
broj_niti = 7	0	0	1	1	1	1	0	2	4	20	67	282
broj_niti = 10	0	0	3	0	0	0	0	0	18	14	58	246
broj_niti = 13	0	0	0	0	0	15	0	0	9	15	63	256
broj_niti = 16	0	0	0	0	0	0	0	0	21	14	61	252
broj_niti = 19	0	0	0	0	0	11	0	0	16	19	62	244
broj_niti = 22	0	0	1	0	0	11	4	3	12	16	65	231
broj_niti = 25	0	0	0	7	0	11	4	0	11	17	67	247
broj_niti = 28	0	0	0	6	0	11	4	5	7	18	73	266
broj_niti = 31	0	0	0	15	0	0	15	0	13	22	75	262
broj_niti = 34	0	0	0	13	0	15	0	15	15	23	72	268

Dakle, ispada da je optimalno pokrenuti približno 22 niti, da se tada za veliki niz sortiranje vrti 231 milisekundu, dok za više ili manje niti treba više vremena za sortiranje. Probajmo onda kako to isto izgleda s QuickSortom:



QuickSort	n=1	n=4	n=16	n=64	n=256	n=1024	n=4096	n=16384	n=65536	n=262144	n=1048576	n=4194304
broj_niti = 1	0	0	0	0	0	0	0	0	31	325	5576	80553
broj_niti = 4	0	0	0	1	2	1	0	1	28	327	4213	40384
broj_niti = 7	0	0	0	0	0	2	0	0	27	94	2759	63419
broj_niti = 10	0	0	0	0	0	0	0	14	15	172	2968	46770
broj_niti = 13	0	0	0	0	0	0	0	0	23	105	8444	441364
broj_niti = 16	0	0	0	1	1	3	3	4	25	158	3277	88454
broj_niti = 19	0	0	0	0	2	0	9	0	20	137	33499	82541
broj_niti = 22	0	0	0	0	0	0	9	0	7	118	4996	53401
broj_niti = 25	0	0	0	0	0	0	0	15	11	165	3540	97831
broj_niti = 28	0	0	0	0	0	15	0	10	15	668	2808	206656
broj_niti = 31	0	0	0	0	2	0	0	10	21	338	1899	85556
broj_niti = 34	0	0	0	0	3	0	12	0	31	902	2502	74928

Dakle, bez dileme, MergeSort se bolje paralelizira nego QuickSort. Paralelnom QuickSortu treba oko 100 puta više vremena da sortira niz iste veličine nego što treba paraleliziranom MergeSortu, i, u stvari, parelelizacija jedva da ga ubrzava.
Evo koda u programskom jeziku C++, compiliranim pomoću compilera TDM-GCC, kojim su ta mjerenja napravljena, sintaksno obojen u VIM-u:

```
1 #include <algorithm>
2 #include <chrono>
3 #include <cmath>
4 #include <fstream>
5 #include <iostream>
```

```cpp
#include <windows.h>

int broj_niti, *originalni_niz, *pomocni_niz;

struct Granice {
  int donja_granica, gornja_granica, dubina_rekurzije;
};

DWORD WINAPI MergeSort(LPVOID lpGranice) {
  Granice granice = *((Granice *)lpGranice);
  if (granice.gornja_granica - granice.donja_granica <= 1) {
    return 0;
  }
  int sredina = (granice.donja_granica +
granice.gornja_granica) / 2;
  Granice lijevi_dio = {granice.donja_granica, sredina,
                        granice.dubina_rekurzije + 1};
  Granice desni_dio = {sredina, granice.gornja_granica,
                       granice.dubina_rekurzije + 1};
  if (granice.dubina_rekurzije < std::log2((float)broj_niti)) {
    HANDLE noviThreadovi[2];
    DWORD ThreadID[2];
    noviThreadovi[0] = CreateThread(NULL, 0,
(LPTHREAD_START_ROUTINE)MergeSort,
                                    &lijevi_dio, 0,
&ThreadID[0]);
    noviThreadovi[1] = CreateThread(NULL, 0,
(LPTHREAD_START_ROUTINE)MergeSort,
                                    &desni_dio, 0, &ThreadID[1]);
    if (!noviThreadovi[0] || !noviThreadovi[1]) {
      std::cerr << "Ne mogu stvoriti novi thread: " <<
GetLastError()
                << std::endl;
      delete[] originalni_niz;
      delete[] pomocni_niz;
      ExitProcess(1);
    }
    WaitForMultipleObjects(2, noviThreadovi, TRUE, INFINITE);
    CloseHandle(noviThreadovi[0]);
    CloseHandle(noviThreadovi[1]);
  } else {
    MergeSort(&lijevi_dio);
    MergeSort(&desni_dio);
  }
  std::merge(originalni_niz + granice.donja_granica,
originalni_niz + sredina,
             originalni_niz + sredina, originalni_niz +
granice.gornja_granica,
             pomocni_niz + granice.donja_granica);
```

```
 48   std::copy(pomocni_niz + granice.donja_granica,
 49             pomocni_niz + granice.gornja_granica,
 50             originalni_niz + granice.donja_granica);
 51   return 0;
 52 }
 53
 54 DWORD WINAPI QuickSort(LPVOID lpGranice) {
 55   Granice granice = *((Granice *)lpGranice);
 56   if (granice.gornja_granica - granice.donja_granica <= 50 ||
 57       granice.dubina_rekurzije > 16) {
 58     // Napravit cemo ovdje BubbleSort, da ne bismo izazvali
StackOverflow.
 59     for (int i = granice.donja_granica; i <
granice.gornja_granica; i++)
 60       for (int j = granice.donja_granica; j <
granice.gornja_granica - 1; j++)
 61         if (originalni_niz[j] > originalni_niz[j + 1])
 62           std::iter_swap(originalni_niz + j, originalni_niz + j +
1);
 63     return 0;
 64   }
 65   int gornja_granica = granice.gornja_granica;
 66   int donja_granica = granice.donja_granica;
 67   int dubina_rekurzije = granice.dubina_rekurzije;
 68   int pivot = originalni_niz[gornja_granica - 1];
 69   int i = donja_granica - 1;
 70   int j = donja_granica;
 71   int pomocna_varijabla_za_zamjenu;
 72   while (j < gornja_granica - 1) {
 73     if (originalni_niz[j] < pivot) {
 74       i += 1;
 75       pomocna_varijabla_za_zamjenu = originalni_niz[i];
 76       originalni_niz[i] = originalni_niz[j];
 77       originalni_niz[j] = pomocna_varijabla_za_zamjenu;
 78     }
 79     j += 1;
 80   }
 81   pomocna_varijabla_za_zamjenu = originalni_niz[i + 1];
 82   originalni_niz[i + 1] = originalni_niz[gornja_granica - 1];
 83   originalni_niz[gornja_granica - 1] =
pomocna_varijabla_za_zamjenu;
 84   int gdje_je_pivot = i + 1;
 85   Granice lijevi_dio{donja_granica, gdje_je_pivot,
dubina_rekurzije + 1},
 86         desni_dio{gdje_je_pivot, gornja_granica, dubina_rekurzije +
1};
 87   if (granice.dubina_rekurzije < std::log2((float)broj_niti)) {
 88     HANDLE noviThreadovi[2];
 89     DWORD ThreadID[2];
```

```
 90      noviThreadovi[0] = CreateThread(NULL, 0,
(LPTHREAD_START_ROUTINE)QuickSort,
 91                                      &lijevi_dio, 0,
&ThreadID[0]);
 92      noviThreadovi[1] = CreateThread(NULL, 0,
(LPTHREAD_START_ROUTINE)QuickSort,
 93                                      &desni_dio, 0, &ThreadID[1]);
 94      if (!noviThreadovi[0] || !noviThreadovi[1]) {
 95        std::cerr << "Ne mogu stvoriti novi thread: " <<
GetLastError()
 96                  << std::endl;
 97        delete[] originalni_niz;
 98        delete[] pomocni_niz;
 99        ExitProcess(1);
100      }
101      WaitForMultipleObjects(2, noviThreadovi, TRUE, INFINITE);
102      CloseHandle(noviThreadovi[0]);
103      CloseHandle(noviThreadovi[1]);
104    } else {
105      QuickSort(&lijevi_dio);
106      QuickSort(&desni_dio);
107    }
108    return 0;
109  }
110
111  int main() {
112    std::ofstream MergeSort_izlaz("MergeSort.txt");
113    MergeSort_izlaz << "\t";
114    for (int n = 1; n < (1 << 24); n <<= 2)
115      MergeSort_izlaz << "n=" << n << '\t';
116    MergeSort_izlaz << std::endl;
117    for (broj_niti = 1; broj_niti < 36; broj_niti += 3) {
118      MergeSort_izlaz << "broj_niti=" << broj_niti << "\t";
119      for (int n = 1; n < (1 << 24); n <<= 2) {
120        originalni_niz = new int[n];
121        pomocni_niz = new int[n];
122        for (int i = 0; i < n; i++)
123          originalni_niz[i] = pomocni_niz[i] = std::rand();
124        auto pocetak_mjerenja =
std::chrono::high_resolution_clock::now();
125        Granice granice{0, n, 0};
126        MergeSort(&granice);
127        auto kraj_mjerenja =
std::chrono::high_resolution_clock::now();
128        if (!std::is_sorted(originalni_niz, originalni_niz + n)) {
129          std::cerr << "MergeSort nije tocno sortirao niz velicine
" << n
130                    << std::endl;
131          delete[] originalni_niz;
```

```
132         delete[] pomocni_niz;
133         MergeSort_izlaz.close();
134         return 1;
135       }
136       delete[] originalni_niz;
137       delete[] pomocni_niz;
138       MergeSort_izlaz << ((kraj_mjerenja -
pocetak_mjerenja).count() *
139
std::chrono::high_resolution_clock::period::num *
140                           1000 /
141
std::chrono::high_resolution_clock::period::den)
142                       << '\t';
143     }
144     MergeSort_izlaz << std::endl;
145   }
146   MergeSort_izlaz.close();
147
148   std::ofstream QuickSort_izlaz("QuickSort.txt");
149   QuickSort_izlaz << "\t";
150   for (int n = 1; n < (1 << 24); n <<= 2)
151     QuickSort_izlaz << "n=" << n << '\t';
152   QuickSort_izlaz << std::endl;
153   for (broj_niti = 1; broj_niti < 36; broj_niti += 3) {
154     QuickSort_izlaz << "broj_niti=" << broj_niti << "\t" <<
std::flush;
155     for (int n = 1; n < (1 << 24); n <<= 2) {
156       originalni_niz = new int[n];
157       pomocni_niz = new int[n];
158       for (int i = 0; i < n; i++)
159         originalni_niz[i] = pomocni_niz[i] = std::rand();
160       auto pocetak_mjerenja =
std::chrono::high_resolution_clock::now();
161       Granice granice{0, n, 0};
162       QuickSort(&granice);
163       auto kraj_mjerenja =
std::chrono::high_resolution_clock::now();
164       if (!std::is_sorted(originalni_niz, originalni_niz + n)) {
165         std::cerr << "QuickSort nije tocno sortirao niz velicine
" << n
166                   << std::endl;
167         delete[] originalni_niz;
168         delete[] pomocni_niz;
169         QuickSort_izlaz.close();
170         return 1;
171       }
172       delete[] originalni_niz;
173       delete[] pomocni_niz;
```

```
174       QuickSort_izlaz << ((kraj_mjerenja -
pocetak_mjerenja).count() *
175
std::chrono::high_resolution_clock::period::num *
176                             1000 /
177
std::chrono::high_resolution_clock::period::den)
178                         << '\t' << std::flush;
179     }
180     QuickSort_izlaz << std::endl;
181   }
182   QuickSort_izlaz.close();
183   return 0;
184 }
```

Raditi pomoću C++11 *thread* biblioteke nije opcija s obzirom na to da je TDM-GCC ne podržava, pa sam radio s Windows API-jem.