Generisk Programmering: Præsentation, uge 1

Jens Peter Rosenkvist[1] Kevin Hejn[2]

[1] *Datalogisk Institut, Københavns Universitet*
Universitetsparken 1, DK-2100 København Ø, Danmark
di060191@diku.dk
[2] *Datalogisk Institut, Københavns Universitet*
Universitetsparken 1, DK-2100 København Ø, Danmark
di060190@diku.dk

Resumé. Ved sammenligning af standardimplementeringen, vores egen implementering og den optimerede implementering af metoden *find_first_of()*, har vi konstateret at den langsomste metode er standardimplementeringen. Vores implementering uden optimeringen er en smule hurtigere, mens den optimerede udgave er betydelig hurtigere.

1. Implementering af metoden

Metoden *find_first_of()* skal givet to lister returnere pladsen på det første element i den første liste, der sammen med et vilkårligt element i den anden liste, opfylder en given binær funktion, f.eks. lighedsoperatoren.
I vores uoptimerede implementering anvendes to *for*-løkker til at sammenligne elementerne i den første liste en for en med elementerne i den anden liste.

Optimeringen af metoden skal ske ved at rulle løkkerne ud, hvilket skal gøres med de på hjemmesiden angivne værktøjer. For at kunne rulle løkkerne ud, skal iteratorerne understøtte *random_access_iterator*.
Da der i *find_first_of()* er to løkker, skal der laves fire forskellige templates. De fire er til, hvis ingen, en eller begge af iteratorerne understøtter dette.
Et eksempel på, hvordan sådan en template er ses på figur 1.

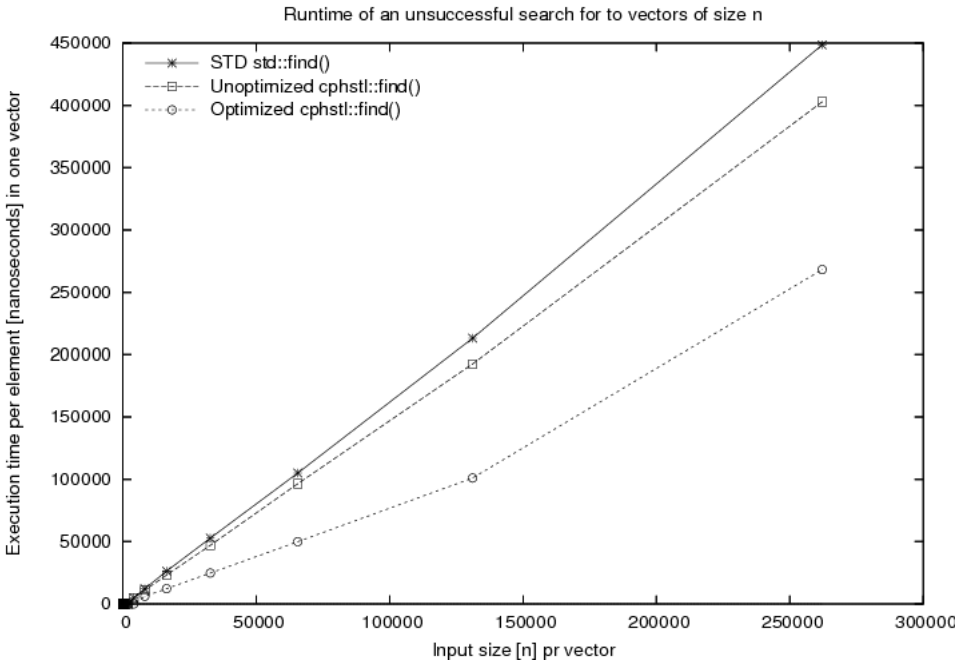
2. Sammenligning

Ved sammenligning af standard, vores uoptimerede og vores optimerede implementering er vi kommet frem til hvor effektive de tre metoder er. På figur 2 ses kørselstiderne for forskellige input størrelser. Det ses, at den optimerede kode er den klart mest effektive. Forskellen er ikke så stor på de to andre, men standard metoden er dog en smule langsommere.

Foredragsnote, april 2007

```
1  template <typename FI1, typename FI2, typename BP>
2  inline FI1 find_first_of (
3               FI1 first1, FI1 last1,
4               FI2 first2, FI2 last2, BP pred,
5               std::random_access_iterator_tag for_branching1,
6               std::random_access_iterator_tag for_branching2) {
7
8    UNROLLED_LOOP(first1,last1,
9      FI2 j = first2;
10     UNROLLED_LOOP(j, last2,
11       if (pred(*first1, *j)) {
12         return first1;
13       }
14     )
15   )
16   return last1;
17 }
```

Figur 1. Udsnit af den optimerede kode



Figur 2. Sammenligning af kørselstider for de tre metoder ved forskelligt antal elementer.