Lazy vs. non lazy

O. Denas

12/28/2016

Contents

1 Double vs. single rank

1.1 Code

The single rank and double rank implementations

```
// RANK(idx)
const uint64_t* p = m_basic_block.data() + ((idx>>8)&0xFFFFFFFFFFFFFFFEULL);
return *p + ((*(p+1)>>(63 - 9*((idx&0x1FF)>>6)))&0x1FF) +
      (idx&0x3F ? trait_type::word_rank(m_v->data(), idx) : 0);

// DOUBLE RANK(i, j)
if((i>>8) == (j>>8)){
  const uint64_t* p = m_basic_block.data() + ((i>>8)&0xFFFFFFFFFFFFFFFEULL);
  res.first  = *p + ((*(p+1)>>(63 - 9*((i&0x1FF)>>6)))&0x1FF) +
            (i&0x3F ? trait_type::word_rank(m_v->data(), i) : 0);
  res.second = *p + ((*(p+1)>>(63 - 9*((j&0x1FF)>>6)))&0x1FF) +
            (j&0x3F ? trait_type::word_rank(m_v->data(), j) : 0);
} else {
  const uint64_t* p = m_basic_block.data() + ((i>>8)&0xFFFFFFFFFFFFFFFEULL);
  res.first  = *p + ((*(p+1)>>(63 - 9*((i&0x1FF)>>6)))&0x1FF) +
            (i&0x3F ? trait_type::word_rank(m_v->data(), i) : 0);
  p -= (((i>>8)&0xFFFFFFFFFFFFFFFEULL) - ((j>>8)&0xFFFFFFFFFFFFFFFEULL));
  res.second = *p + ((*(p+1)>>(63 - 9*((j&0x1FF)>>6)))&0x1FF) +
            (j&0x3F ? trait_type::word_rank(m_v->data(), j) : 0);
}
return res;
```

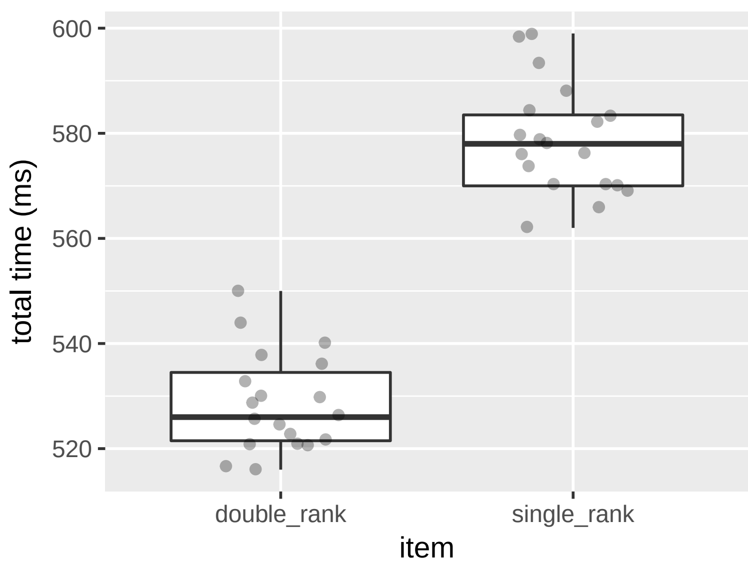
1.2 Performance

Table 1: Time (in ms) of 500K calls to `wl()` based on `single_rank()` or `double_rank()` methods on 100MB random DNA input; Mean/sd over 20 repetitions.

item	avg_time	sd_time
double_rank	524.84	9.07
single_rank	569.68	10.45

Table 2: Absolute (double_rank / single_rank) and relative (100 * |double_rank - single_rank| / single_rank) ratios of average times from the above table.

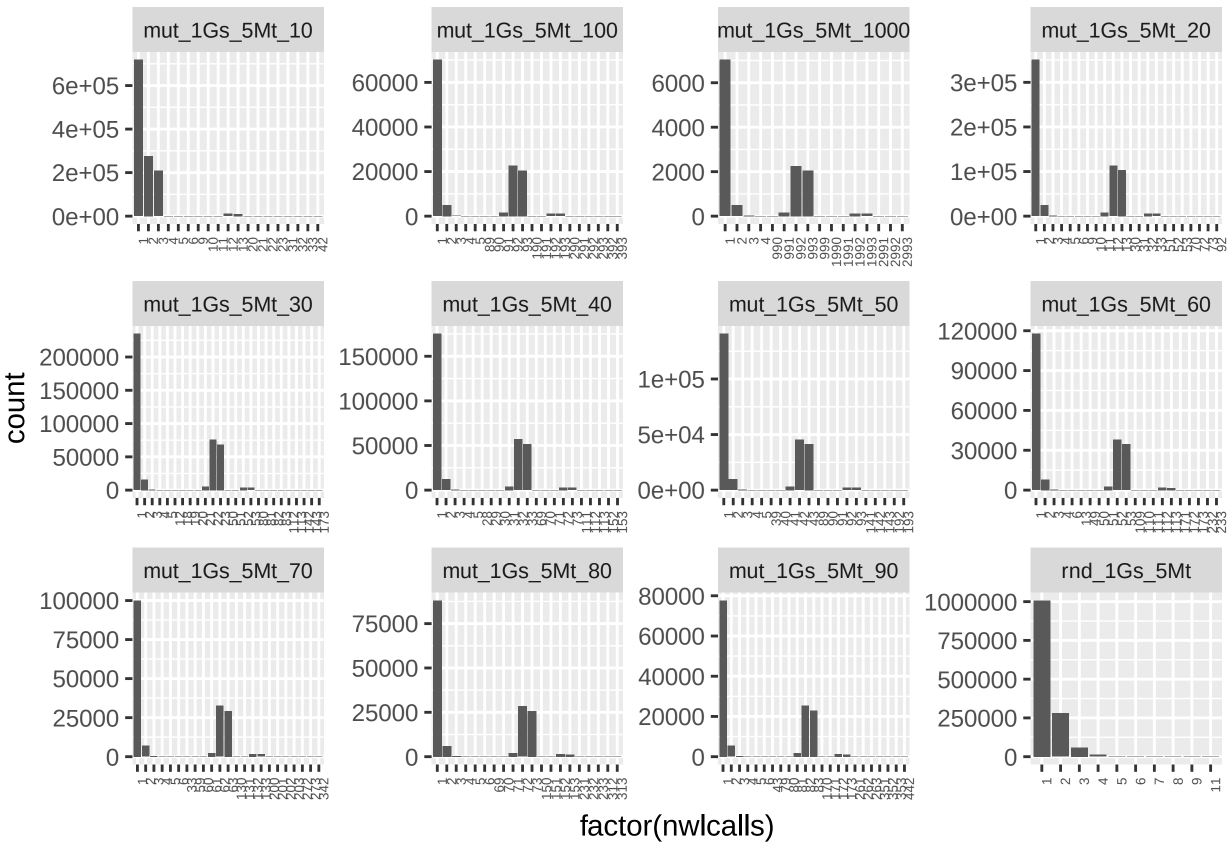
.	double_rank	single_rank	abs_ratio	rel_ratio
.	524.84	569.68	0.92	7.87



2 Lazy vs non-lazy

2.1 Input properties

For various types ("mut_XMs_YMt_Z" means `s` and `t` are random identical strings of length X, and Y million respectively with mutations inserted every Z characters. "rnd_XMs_YMt" means `s` and `t` are random strings of length X, and Y million respectively) of inputs run the MS algorithm and count the number of consecutive `lazy_wl()` calls.

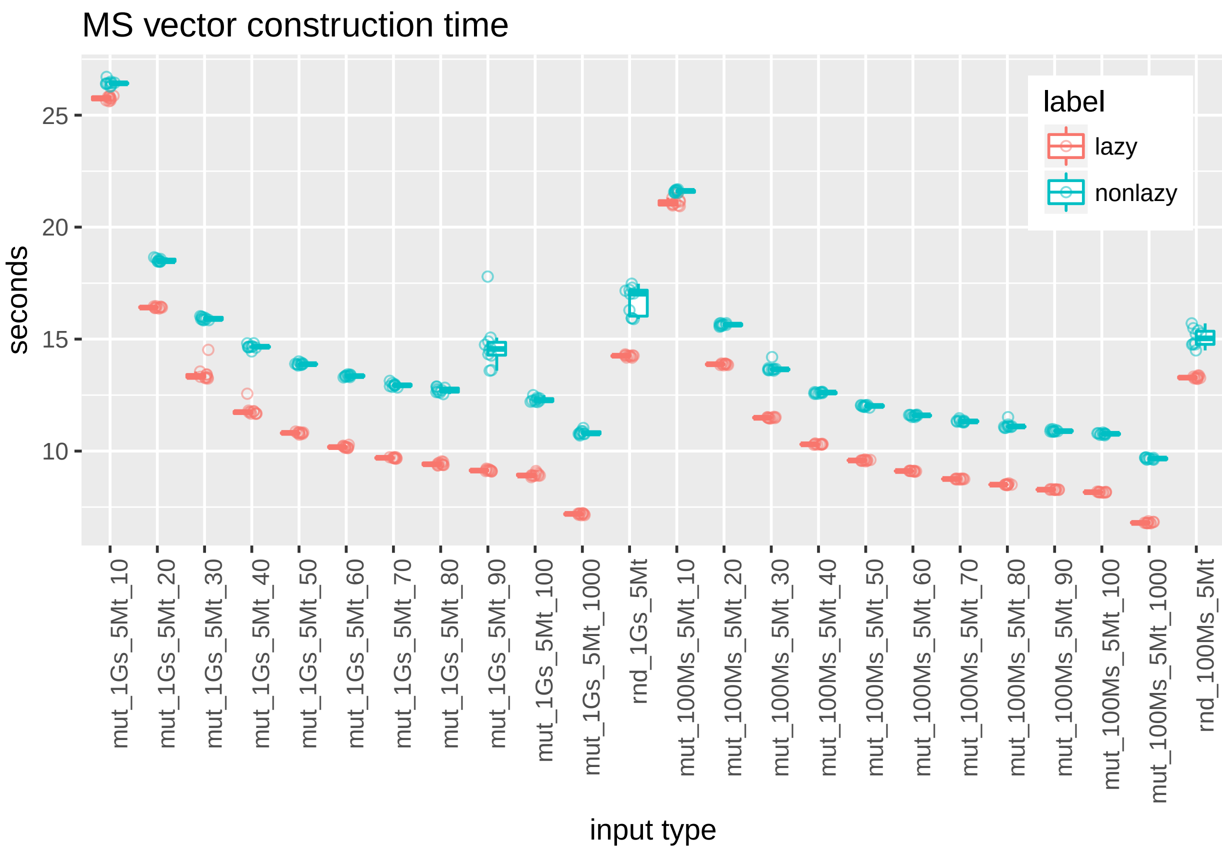


2.2 Code

The lazy and non-lazy versions differ in a couple of lines of code as follows

```
if(flags.lazy){
    for(; I.first <= I.second && h_star < ms_size; ){
        c = t[h_star];
        I = bstep_interval(st, I, c); //I.bstep(c);
        if(I.first <= I.second){
            v = st.lazy_wl(v, c);
            h_star++;
        }
    }
    if(h_star > h_star_prev) // // we must have called lazy_wl(). complete the node
        st.lazy_wl_followup(v);
} else { // non-lazy weiner links
    for(; I.first <= I.second && h_star < ms_size; ){
        c = t[h_star];
        I = bstep_interval(st, I, c); //I.bstep(c);
        if(I.first <= I.second){
            v = st.wl(v, c);
            h_star++;
        }
    }
}
```

2.3 Run time



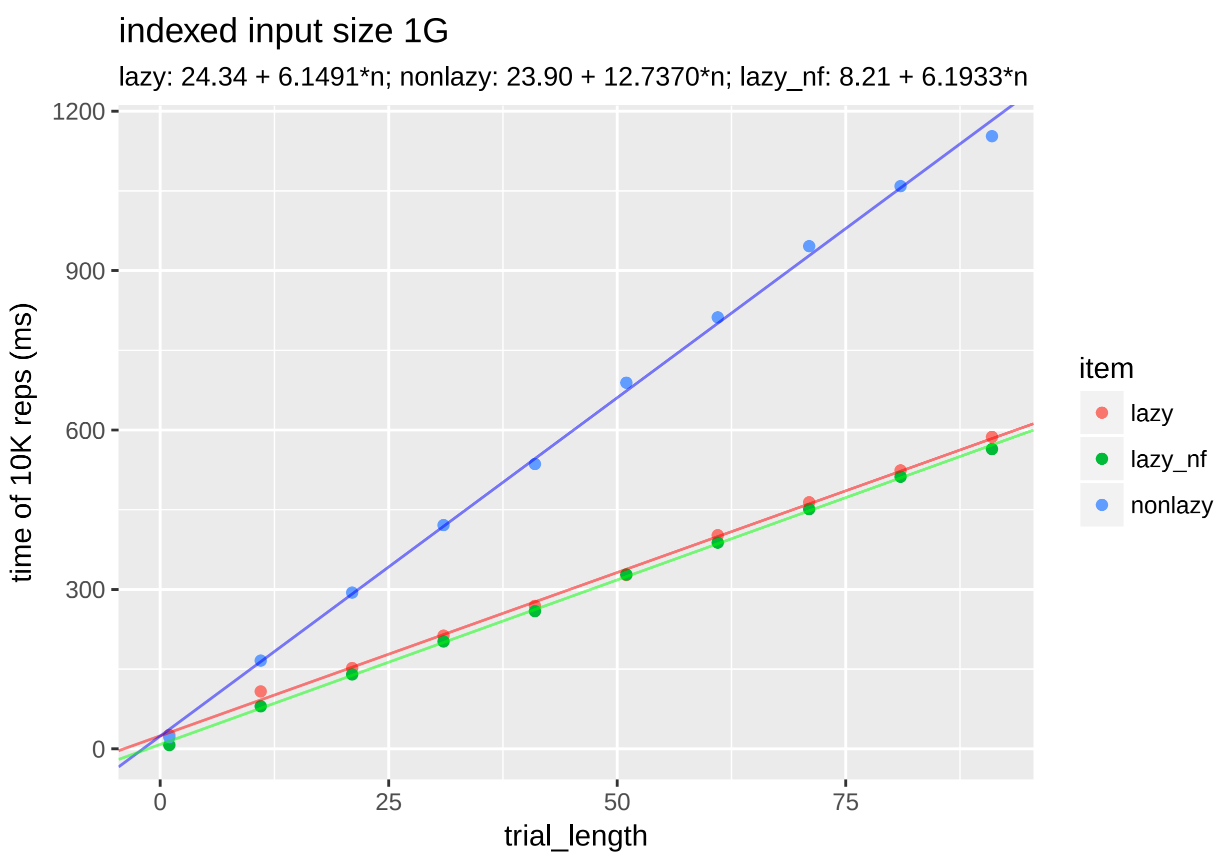
The right panel shows the time to construct the `runs` vector. This stage is the same for both versions and is shown as a control. On the left panel it can be seen that speedup correlates positively with both the size of the indexed string and the mutation period.

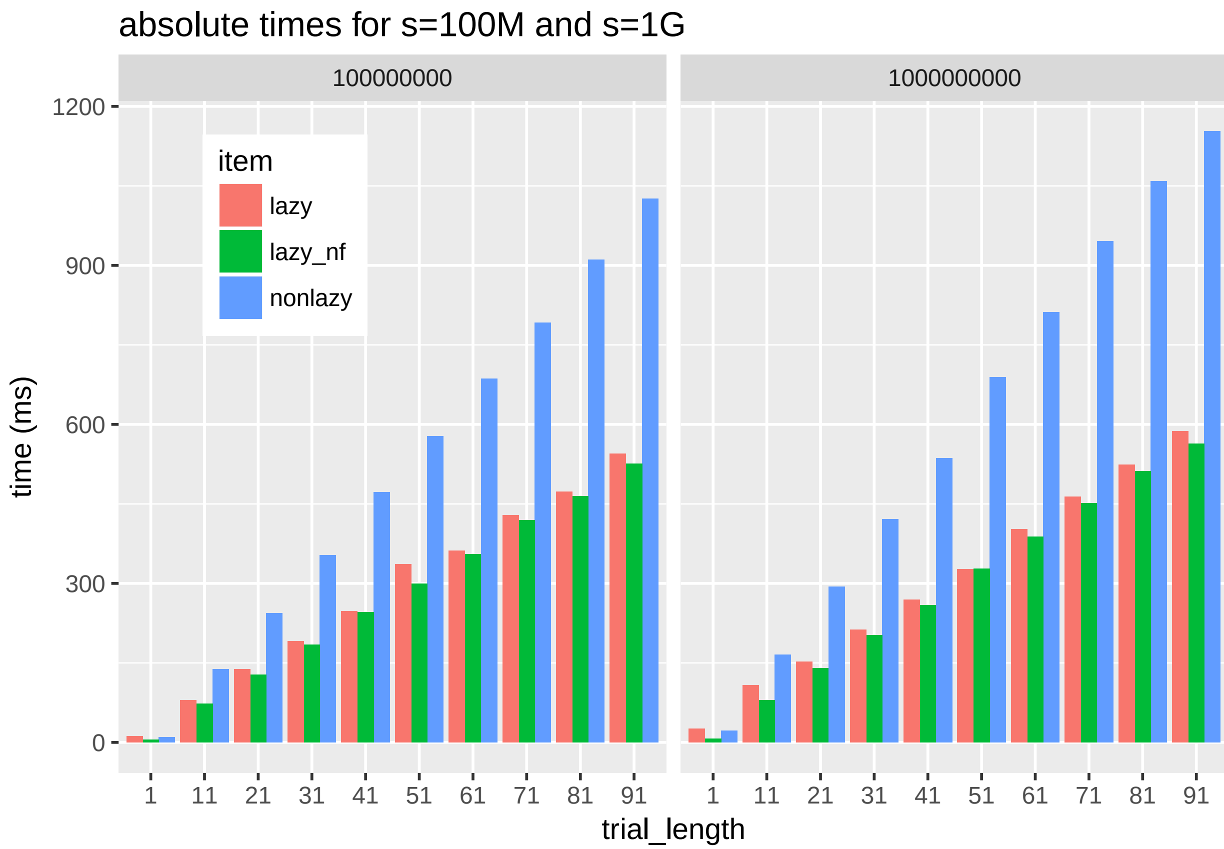
2.4 Sandbox timing

Measure the time of 10k repetitions of

- (lazy) n consecutive `lazy_wl()` calls followed by a `lazy_wl_followup()`
- (nonlazy) n consecutive `wl()` calls
- (lazy_nf) n consecutive `lazy_wl()` calls

```
// lazy
for(size_type i = 0; i < trial_length; i++)
    v = st.lazy_wl(v, s_rev[k--]);
if(h_star > h_star_prev) // // we must have called lazy_wl(). complete the node
    st.lazy_wl_followup(v);
...
// non-lazy
for(size_type i = 0; i < trial_length; i++)
    v = st.wl(v, s_rev[k--]);
...
// lazy_nf
for(size_type i = 0; i < trial_length; i++)
    v = st.lazy_wl(v, s_rev[k--]);
```





2.5 Check

In the experiments above we ran the program with the "lazy" or "non-lazy" flag and measured. The total time of each experiment can be written as $t_l = l_l + a$ and $t_n = l_n + a$ for the two versions respectively; only the ts being known. Furthermore, we have $\hat{l}_l$ and $\hat{l}_n$ estimations – computed by combining the time / wl call with the number of with the count of wl calls in each input (Section "Input Properties"). Hence we should expect

$$\delta t = t_l - t_n = l_l + a - l_n - a = l_l - l_n \approx \delta\hat{l} = \hat{l}_l - \hat{l}_n$$

b_path	t_l	t_n	l_l	l_n	delta_t	delta_l_hat
mut_100Ms_5Mt_10	21.12	21.61	4.05	3.53	-0.49	0.52
mut_100Ms_5Mt_100	8.16	10.77	2.90	4.07	-2.60	-1.16
mut_100Ms_5Mt_1000	6.80	9.67	2.79	4.12	-2.86	-1.33
mut_100Ms_5Mt_20	13.87	15.64	3.41	3.83	-1.77	-0.41
mut_100Ms_5Mt_30	11.49	13.70	3.20	3.93	-2.21	-0.73
mut_100Ms_5Mt_40	10.31	12.60	3.10	3.98	-2.30	-0.88
mut_100Ms_5Mt_50	9.58	12.01	3.03	4.01	-2.43	-0.97
mut_100Ms_5Mt_60	9.11	11.58	2.99	4.03	-2.48	-1.04
mut_100Ms_5Mt_70	8.75	11.34	2.96	4.04	-2.59	-1.08
mut_100Ms_5Mt_80	8.51	11.13	2.94	4.05	-2.63	-1.11
mut_100Ms_5Mt_90	8.28	10.90	2.92	4.06	-2.62	-1.14
mut_1Gs_5Mt_10	25.75	26.43	4.08	4.08	-0.68	-0.01
mut_1Gs_5Mt_100	8.94	12.29	3.15	4.65	-3.35	-1.50

b_path	t_l	t_n	l_l	l_n	delta_t	delta_l_hat
mut_1Gs_5Mt_1000	7.19	10.82	3.05	4.70	-3.63	-1.65
mut_1Gs_5Mt_20	16.42	18.52	3.56	4.40	-2.10	-0.84
mut_1Gs_5Mt_30	13.46	15.92	3.39	4.50	-2.46	-1.11
mut_1Gs_5Mt_40	11.81	14.66	3.30	4.55	-2.85	-1.25
mut_1Gs_5Mt_50	10.81	13.89	3.25	4.59	-3.08	-1.34
mut_1Gs_5Mt_60	10.19	13.36	3.22	4.61	-3.17	-1.39
mut_1Gs_5Mt_70	9.70	12.95	3.19	4.62	-3.26	-1.43
mut_1Gs_5Mt_80	9.43	12.72	3.17	4.63	-3.29	-1.46
mut_1Gs_5Mt_90	9.14	14.74	3.16	4.64	-5.60	-1.48
rnd_100Ms_5Mt	13.29	15.07	4.68	3.65	-1.78	1.03
rnd_1Gs_5Mt	14.25	16.72	4.15	3.94	-2.48	0.20

The numbers are not identical (process dependent factors might influence the running time of function calls), but they are correlated ($corr(\delta t, \delta\hat{l}) = 0.72$).