Report of experiments

O. Denas

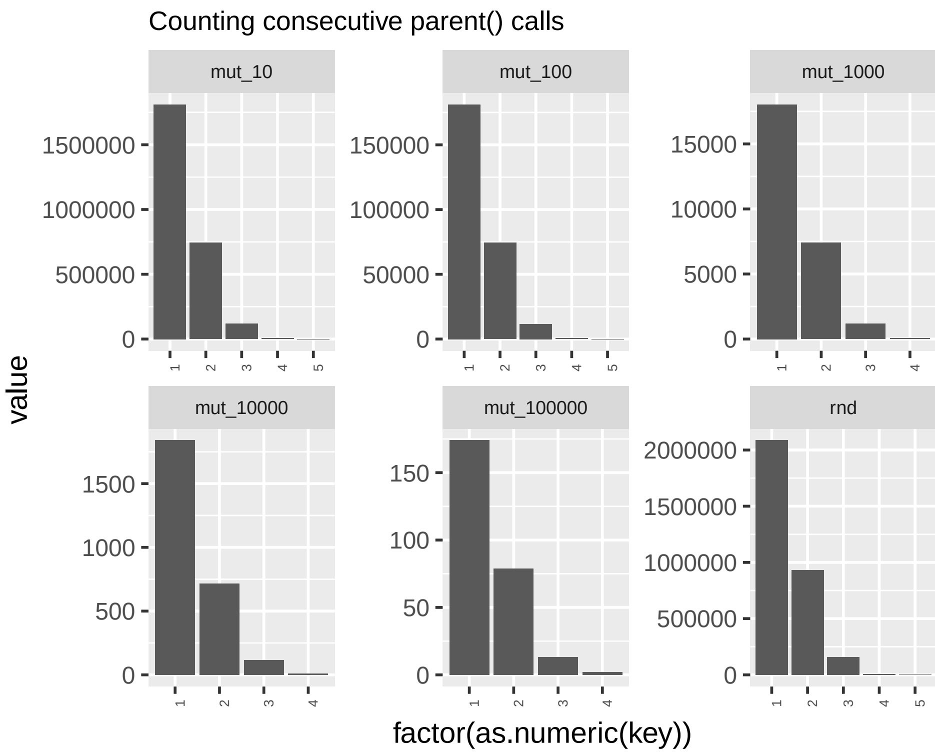
12/28/2016

Contents

1 Input properties

For various types of inputs ("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) run the MS algorithm and count the number of

- consecutive `parent()` calls during the `runs` construction.
- consecutive `wl()` calls during the `ms` construction.
- the number of 1s in the `runs` bit vector
- double rank calls that fail (i.e the search down the WT is interrupted prior to reaching a leaf)
- the number of maximal repeats



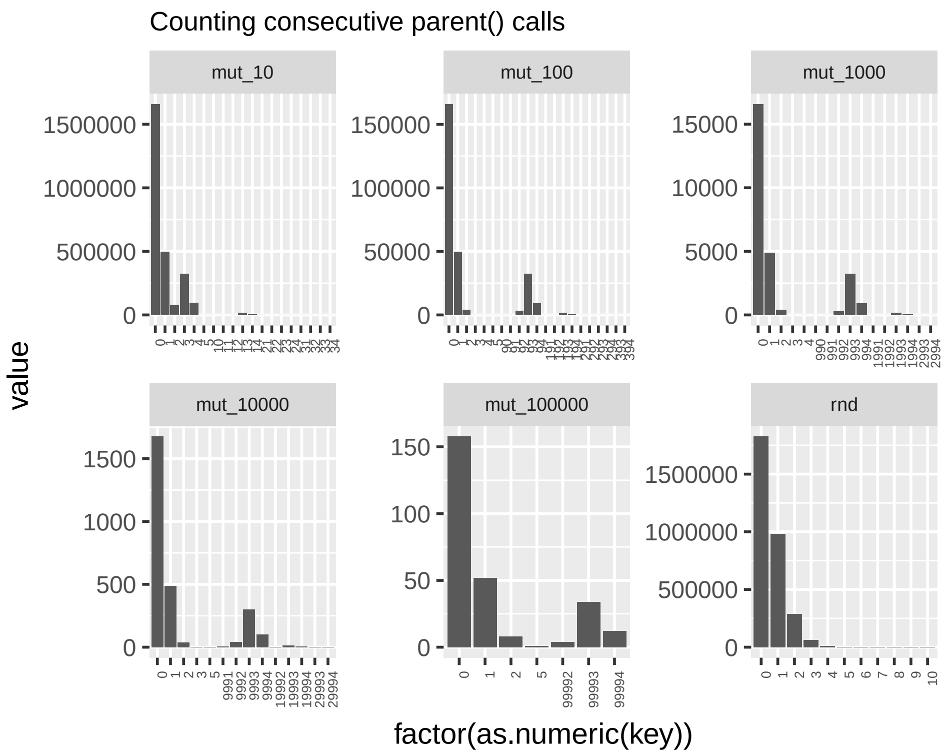


Table 1: Double rank iterations that fail for various input types.

b_path	fail	nofail	perc
rnd_100Ms_5Mt	690911	1818974	37.98
mut_100Ms_5Mt_10	1494544	2323907	64.31
mut_100Ms_5Mt_100	4649474	4732340	98.25
mut_100Ms_5Mt_1000	4965082	4973326	99.83
mut_100Ms_5Mt_10000	4996520	4997320	99.98
mut_100Ms_5Mt_100000	4999634	4999731	100.00

Table 2: Composition of the `runs` vector for various input types.

inp_type	one	zero	zero_perc
mut_10	2323907	2676093	115.15
mut_100	4732340	267660	5.66
mut_1000	4973326	26674	0.54
mut_10000	4997320	2680	0.05
mut_100000	4999731	269	0.01
rnd	1818974	3181026	174.88

Table 3: Composition of the `B` vector (containing ends of maximal repeats) for various input types.

inp_type	maximal	non_maximal	non_maximal_perc
mut_10	34894073	982212	2.81
mut_100	34893276	981720	2.81

inp_type	maximal	non_maximal	non_maximal_perc
mut_1000	34891214	980977	2.81
mut_10000	34894098	982309	2.82
mut_100000	34897894	980817	2.81
rnd	50053548	5525912	11.04

2 Current performance

Table 4: Run time in seconds, on random input with |s| = 1MB, |t| = 5MB

lazy	fail	maxrep	total_s	maxrep_s	tot_minus_maxrep_s
0	1	0	90.351	NA	90.351
1	1	0	90.817	NA	90.817
1	1	1	92.478	2.024	90.454
1	0	0	92.844	NA	92.844
0	0	0	92.850	NA	92.850
0	1	1	93.020	2.020	91.000
0	0	1	94.044	1.987	92.057
1	0	1	94.092	1.992	92.100

3 Double vs. single rank

3.1 Code

The single rank and double rank implementations in sdsl: rank_support_v.hpp link

```
// 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 OD(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

// DOUBLE RANK FC(i, j)
const uint64_t* b = m_basic_block.data();
const uint64_t* pi = b + ((i>>8)&0xFFFFFFFFFFFFFFFEULL);
const uint64_t* pj = b + ((j>>8)&0xFFFFFFFFFFFFFFFEULL);

return (*pi + ((*(pi+1)>>(63 - 9*((i&0x1FF)>>6)))&0x1FF) +
              (i&0x3F ? trait_type::word_rank(m_v->data(), i) : 0),
      *pj + ((*(pj+1)>>(63 - 9*((j&0x1FF)>>6)))&0x1FF) +
              (j&0x3F ? trait_type::word_rank(m_v->data(), j) : 0));
```

The FC implementation worked better. Data are saved in `rank_timing.csv.fc` and `rank_timing.csv.od` but not shown below.

3.2 Performance

item	avg_time	sd_time
double	551.5556	4.666667
double_fail	559.0000	0.000000
single	583.7778	8.613620

4 Maxrep

4.1 Maxrep construction

Applying the first optimization we get 8% improvement on a (ran of a 1MB input string).

```
# EXISTING CODE
denas@denas-osx:$ for i in 1 2 3 4 5; \
do compute_maxrep -answer 0 -load_cst 0 -s_path datasets/synthetic/rnd_1Ms_5Mt.s; \
done  2>&1 | grep mill
 * computing MAXREP DONE ( 1098 milliseconds)
 * computing MAXREP DONE ( 1116 milliseconds)
 * computing MAXREP DONE ( 1120 milliseconds)
 * computing MAXREP DONE ( 1094 milliseconds)
 * computing MAXREP DONE ( 1100 milliseconds)
denas@denas-osx:$

# OPTIMIZED CODE
denas@denas-osx:$ for i in 1 2 3 4 5; \
do compute_maxrep -answer 0 -load_cst 0 -s_path datasets/synthetic/rnd_1Ms_5Mt.s; \
done  2>&1 | grep mill
 * computing MAXREP DONE ( 1020 milliseconds)
 * computing MAXREP DONE ( 1023 milliseconds)
 * computing MAXREP DONE ( 999 milliseconds)
 * computing MAXREP DONE ( 1020 milliseconds)
 * computing MAXREP DONE ( 1015 milliseconds)
```

4.2 Performance

The figure below shows 8 runs of the program with and without the use of the `maxrep` (or `B`) vector. The plot shows times (in seconds) for the construction of the `ms` bitvector. The table below that, shows the time (in seconds) to construct the `maxrep` vector. The input data is random and has |s|=100MB and |t|=5MB.

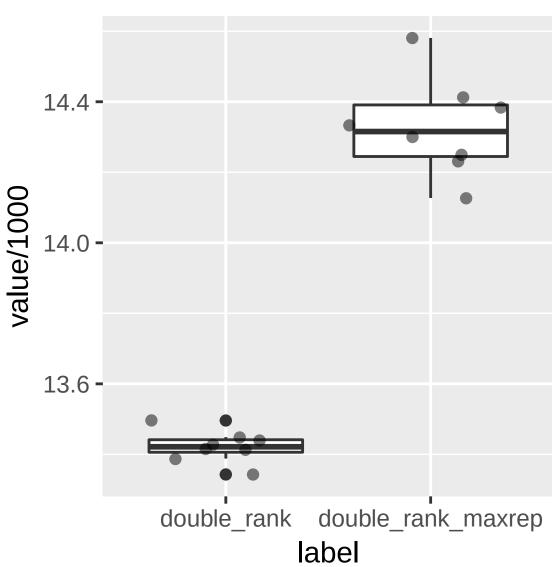


Table 6: Time to build the maxrep bit vector

label	avg	se
double_rank_maxrep	113	8

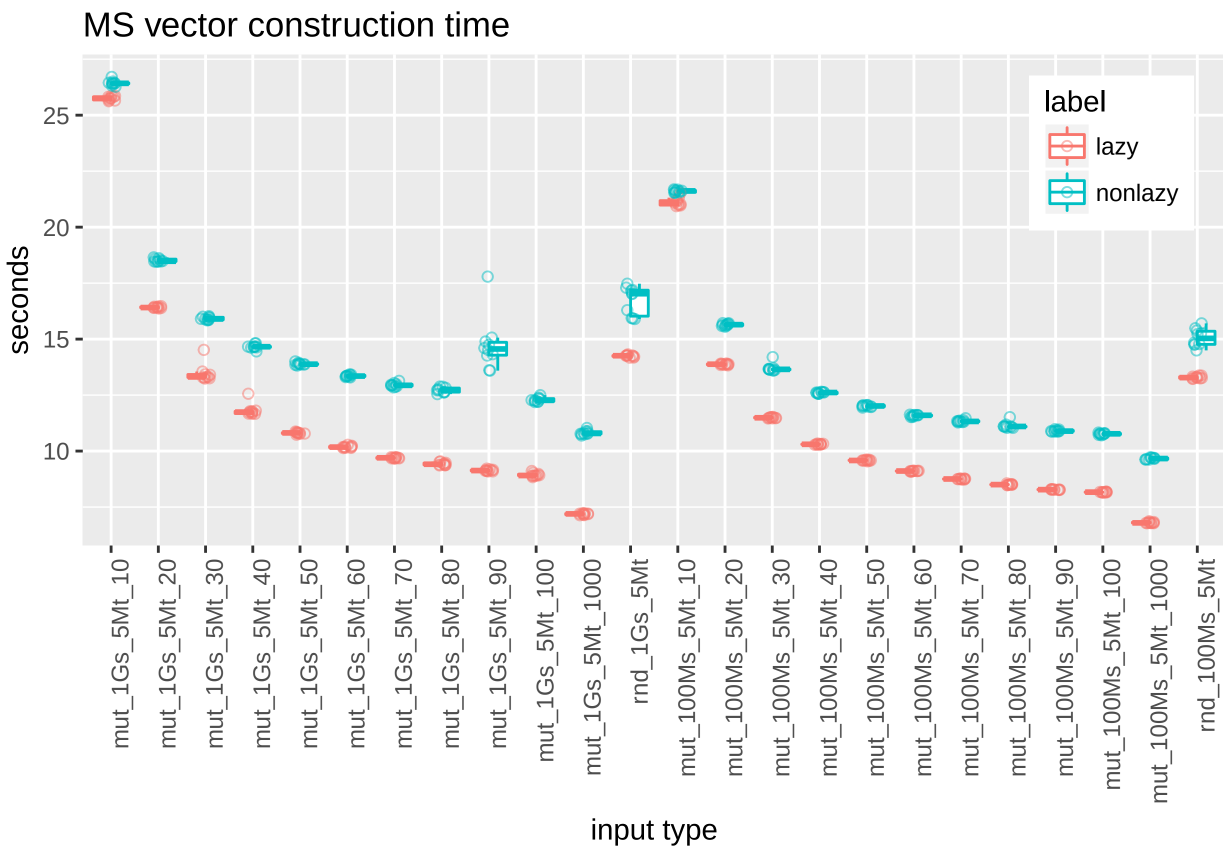
5 Lazy vs non-lazy

5.1 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++;
        }
    }
}
```

5.2 Performance



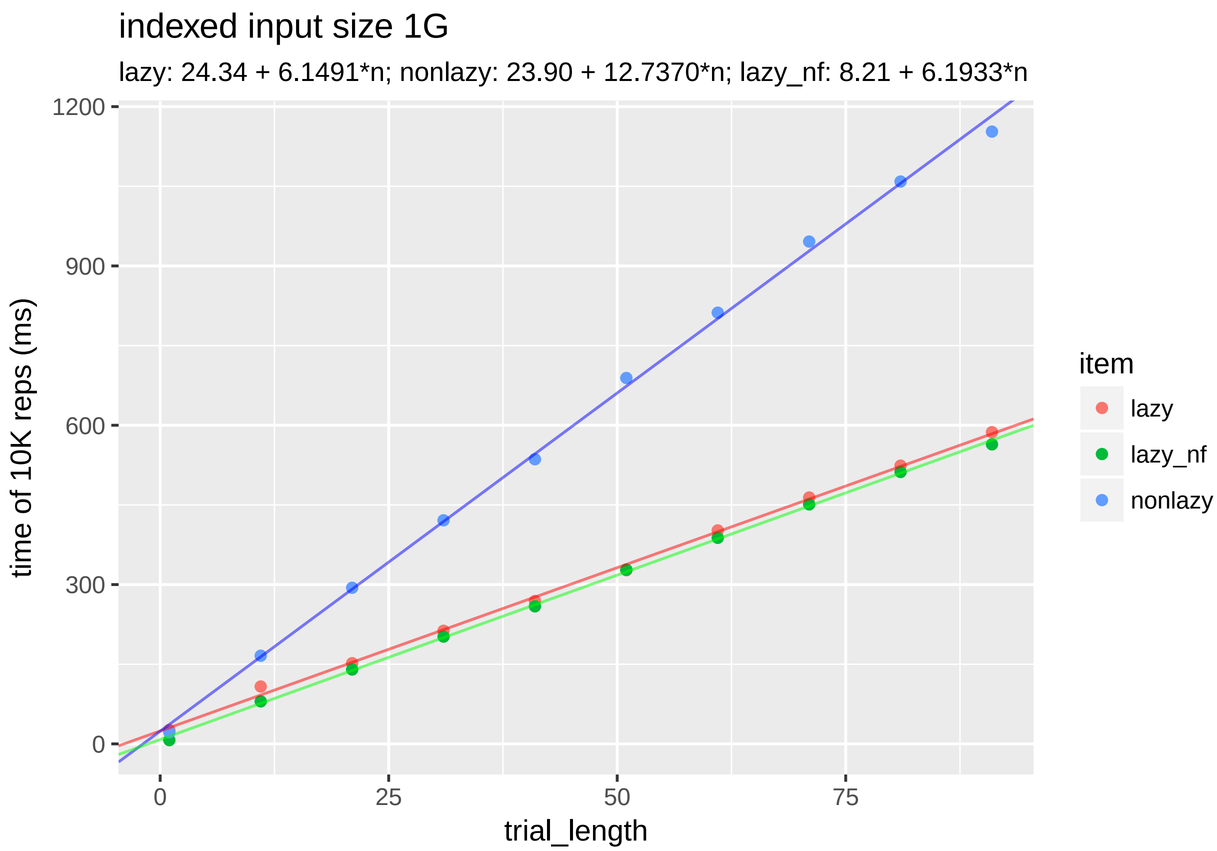
The right panel shows the time to construct the `runs` vector. This stage is the same for both versions and is shown as a control. On the left panel it can be seen that speedup correlates positively with both the size of the indexed string and the mutation period.

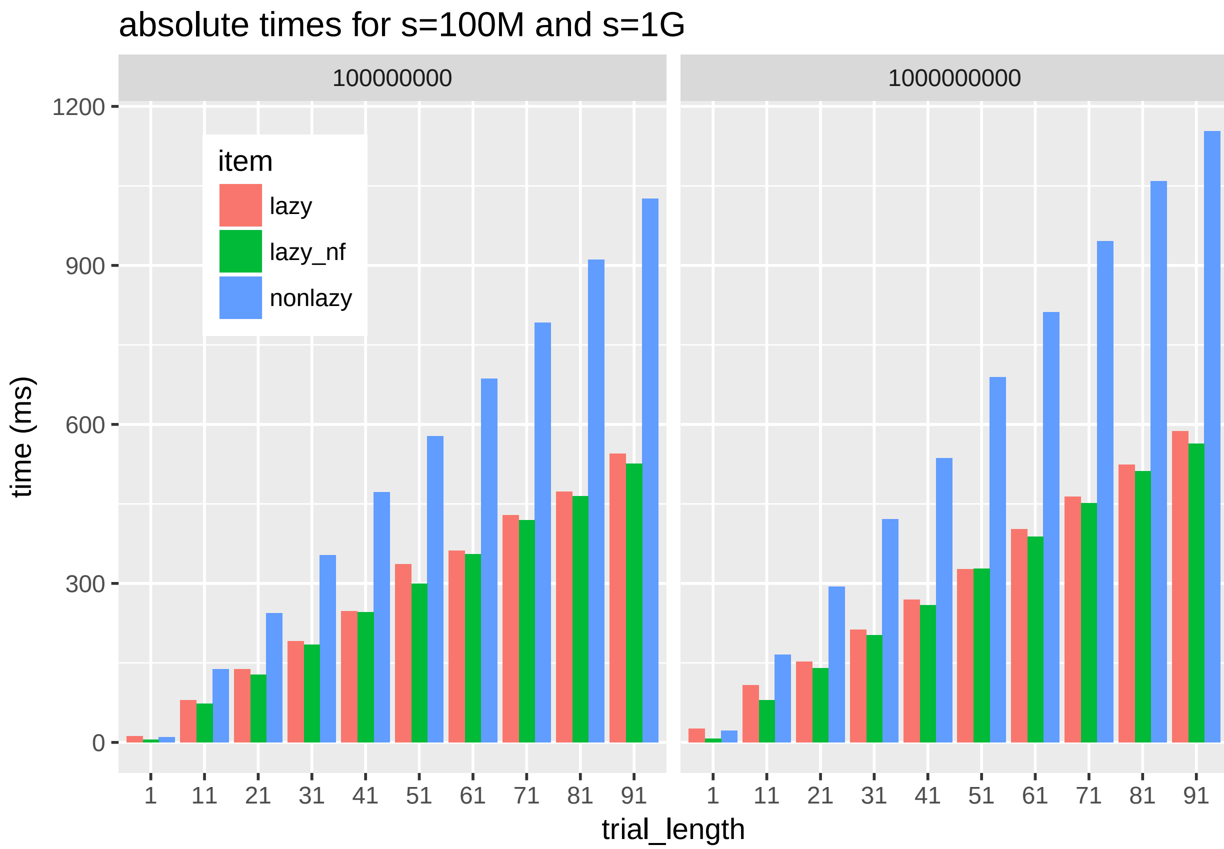
5.3 Sandbox timing

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--]);
```





5.4 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	8.56	6.16	-0.49	2.39
mut_100Ms_5Mt_100	8.16	10.77	3.36	4.33	-2.60	-0.97
mut_100Ms_5Mt_1000	6.80	9.67	2.84	4.15	-2.86	-1.31
mut_100Ms_5Mt_20	13.87	15.64	5.66	5.14	-1.77	0.52
mut_100Ms_5Mt_30	11.49	13.70	4.71	4.81	-2.21	-0.10
mut_100Ms_5Mt_40	10.31	12.60	4.22	4.64	-2.30	-0.41
mut_100Ms_5Mt_50	9.58	12.01	3.93	4.53	-2.43	-0.60
mut_100Ms_5Mt_60	9.11	11.58	3.74	4.47	-2.48	-0.72
mut_100Ms_5Mt_70	8.75	11.34	3.60	4.42	-2.59	-0.81
mut_100Ms_5Mt_80	8.51	11.13	3.50	4.38	-2.63	-0.88
mut_100Ms_5Mt_90	8.28	10.90	3.42	4.35	-2.62	-0.93
mut_1Gs_5Mt_10	25.75	26.43	7.57	6.65	-0.68	0.92
mut_1Gs_5Mt_100	8.94	12.29	3.49	4.90	-3.35	-1.41

b_path	t_l	t_n	l_l	l_n	delta_t	delta_l_hat
mut_1Gs_5Mt_1000	7.19	10.82	3.08	4.72	-3.63	-1.64
mut_1Gs_5Mt_20	16.42	18.52	5.30	5.68	-2.10	-0.37
mut_1Gs_5Mt_30	13.46	15.92	4.55	5.36	-2.46	-0.81
mut_1Gs_5Mt_40	11.81	14.66	4.17	5.20	-2.85	-1.02
mut_1Gs_5Mt_50	10.81	13.89	3.95	5.10	-3.08	-1.15
mut_1Gs_5Mt_60	10.19	13.36	3.80	5.03	-3.17	-1.24
mut_1Gs_5Mt_70	9.70	12.95	3.69	4.99	-3.26	-1.30
mut_1Gs_5Mt_80	9.43	12.72	3.61	4.95	-3.29	-1.35
mut_1Gs_5Mt_90	9.14	14.74	3.55	4.93	-5.60	-1.38
rnd_100Ms_5Mt	13.29	15.07	9.65	6.55	-1.78	3.10
rnd_1Gs_5Mt	14.25	16.72	8.20	6.92	-2.48	1.28

The numbers are not identical (process dependent factors might influence the running time of function calls), but they are correlated ($corr(\delta t, \delta \hat{l}) = 0.71$).

6 Double rank and fail

6.1 Code

```
// Given subtree_double_rank(v, i, j) -> (a.first, a.second) -- to simplify code

// DOUBLE RANK: int i, int j, char c
p = bit_path(c)
result_i, result_j = i, j;
node_type v = m_tree.root();
for (l = 0; l < path_len; ++l, p >>= 1) {
  a = subtree_double_rank(v, m_tree.bv_pos(v) + result_i, m_tree.bv_pos(v) + result_j);

  if(p&1){ // left child
      if(result_i > 0) result_i = a.first;
      if(result_j > 0) result_j = a.second;
  } else { // right child
      if(result_i > 0) result_i -= a.first;
      if(result_j > 0) result_j -= a.second;
  }
  v = m_tree.child(v, p&1); // goto child
}
return(result_i, result_j)

// DOUBLE RANK AND FAIL
p = bit_path(c)
result_i, result_j = i, j;
node_type v = m_tree.root();
for (l = 0; l < path_len; ++l, p >>= 1) {
  a = subtree_double_rank(v, m_tree.bv_pos(v) + result_i, m_tree.bv_pos(v) + result_j);

  if(p&1){ // left child
      if(result_i > 0) result_i = a.first;
      if(result_j > 0) result_j = a.second;
  } else { // right child
      if(result_i > 0) result_i -= a.first;
      if(result_j > 0) result_j -= a.second;
  }
  if(result_i == result_j) // Weiner Link call will fail
    return(0, 0)
  v = m_tree.child(v, p&1); // goto child
}
return(result_i, result_j)
```

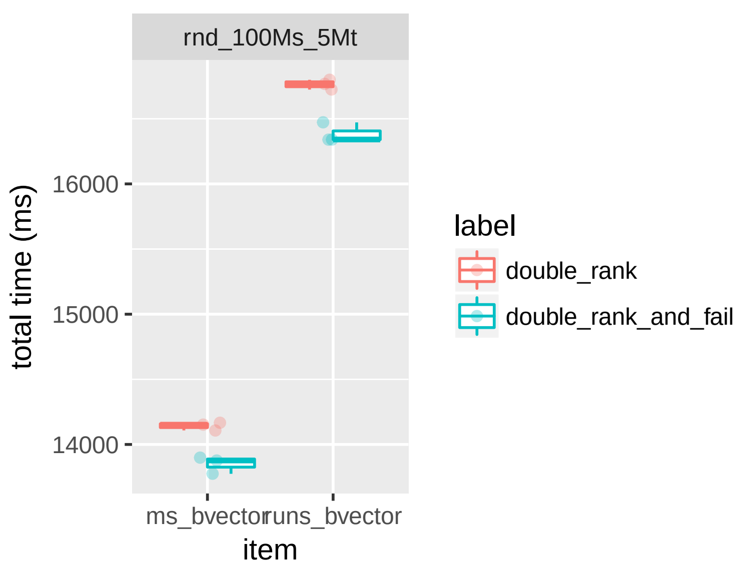
6.2 Performance

Table 8: 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	label	b_path	avg_time	sd_time
ms_bvector	double_rank	rnd_100Ms_5Mt	14142.00	30.27
ms_bvector	double_rank_and_fail	rnd_100Ms_5Mt	13850.33	66.16
runs_bvector	double_rank	rnd_100Ms_5Mt	16763.67	37.69
runs_bvector	double_rank_and_fail	rnd_100Ms_5Mt	16384.00	76.22

Table 9: Single vs. double rank. Absolute (double / single) and relative (100 * |double - single| / single) ratios of average times.

item	double_rank	double_rank_and_fail	abs_ratio	rel_ratio
ms_bvector	14142.00	13850.33	0.98	2.06
runs_bvector	16763.67	16384.00	0.98	2.26



7 Parallelization

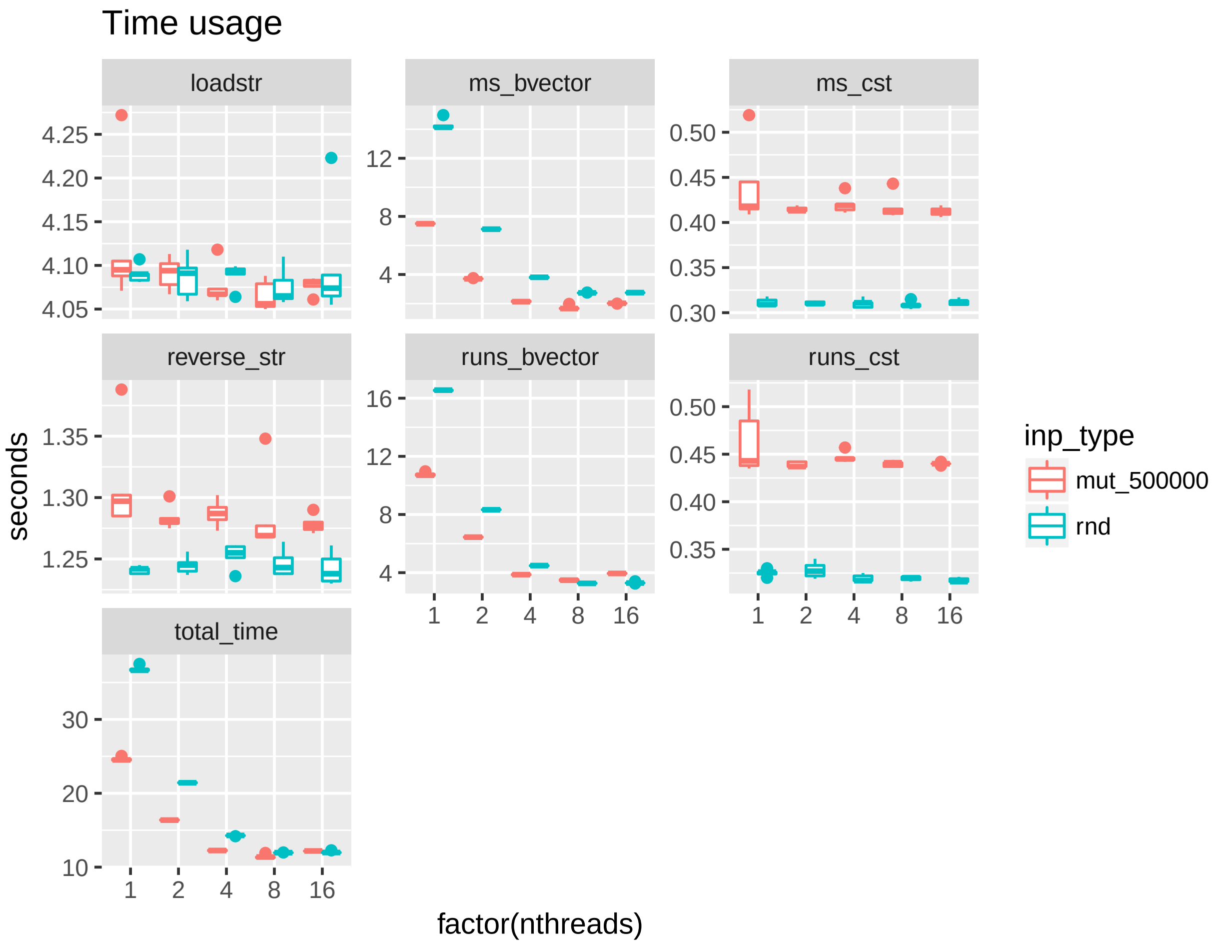
7.1 Code

See the pseudo-code in the repo (link)

7.2 Performance

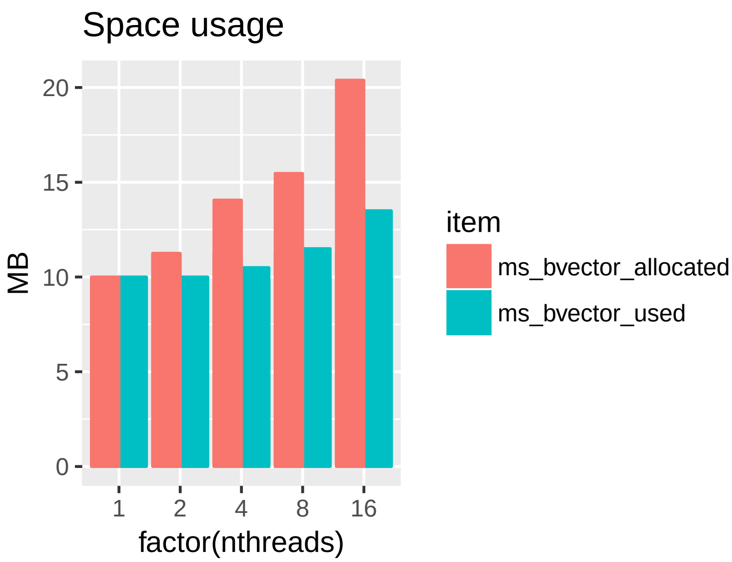
Run the MS construction program on the same input (random strings s of length 100M and t of length 5M) with varying parallelzation degree (nthreads = number of threads).

The time is reported over 5 runs for each fixed number of threads.



Space in MB for the same settings as above.

Each thread allocates its own ms vector with initial size $|t|/nthreads$ then it resizes by a factor of 1.5 each time it needs to. Resizing will always result in a vector smaller than $2|t|$ elements.


Space usage
20
15
10
5
0
MB
1
2
4
8
16
factor(nthreads)
item
ms_bvector_allocated
ms_bvector_used