Pitfalls of Algorithm Comparison

Waltteri Pakalén[1], Hannu Peltola[1], Jorma Tarhio[1], and Bruce W. Watson[2]

[1] Department of Computer Science
Aalto University, Finland
[2] Information Science, Centre for AI Research
School for Data-Science & Computational Thinking
Stellenbosch University, South Africa

Abstract. Why is Algorithm A faster than Algorithm B in one comparison, and vice versa in another? In this paper, we review some reasons for such differences in experimental comparisons of exact string matching algorithms. We address issues related to timing, memory management, compilers, tuning/tune-up, validation, and technology development. In addition, we consider limitations of the widely used testing environments, Hume & Sunday and SMART. A part of our observations likely apply to comparisons of other types of algorithms.

Keywords: exact string matching, experimental comparison of algorithms

1 Introduction

Developing new algorithms is a common research objective in Computer Science, and new solutions are often experimentally compared with older ones. This is especially the case for string matching algorithms. We will consider aspects which may lead to incorrect conclusions while comparing string matching algorithms. We concentrate on running times of exact matching of a single pattern, but many of our considerations likely apply to other variations of string matching or even to other types of algorithms.

Formally, the *exact string matching problem* is defined as follows: given a pattern $P = p_0 \cdots p_{m-1}$ and a text $T = t_0 \cdots t_{n-1}$ both in an alphabet Σ, find all the occurrences (including overlapping ones) of P in T. So far, dozens of algorithms have been developed for this problem — see e.g. Faro and Lecroq [6].

Many experimental comparisons of string matching algorithms employ one of two testing environments, Hume & Sunday [11] (HS for short) and SMART [7], for measuring running times. The HS environment consists of a main program and shell scripts. Each algorithm is compiled to a separate executable. Most comparisons applying the HS environment actually use some variation of it, see e.g. Hirvola [9].

SMART [5] is an environment developed by Faro et al. [7], which includes implementations for more than a hundred algorithms for exact string matching. SMART tries to make comparisons easy for a user by offering a user interface and tools for various subtasks of a comparison. The SMART application controls all the runs of algorithms. SMART offers options to present the results in various forms.

The rest of the paper is organized as follows: Section 2 presents general aspects of algorithm comparison; Section 3 reviews issues related to the testing environments HS and SMART; Section 4 studies how running time should be measured; Sections 5 and 6 analyze how cache and shared memory affect running times; Section 7 lists miscellaneous observations; and the discussion of Section 8 concludes the article.

Waltteri Pakalén, Hannu Peltola, Jorma Tarhio, Bruce W. Watson: Pitfalls of Algorithm Comparison, pp. 16–29.
Proceedings of PSC 2021, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06869-4 © Czech Technical University in Prague, Czech Republic

2 General

Comparing the running times of algorithms may seem easy: store the clock time in the beginning, run the algorithm, store the clock time in the end, and calculate the difference. However, the results are valid only for the combination of implementation, input, compiler and hardware used in the same workload.

Comparison of algorithms should be done according to *good measurement practice*. In the case of exact string matching algorithms, it means several things. One should verify that algorithms work properly: whether all the matches are found, and whether the search always stops properly at the end of the text; additionally, it can happen that a match in the beginning or in the end of the text is not correctly recognized. When measuring is focused on performance, it is standard to use at least some level of optimization in the compilation. The measurement should not disturb the work of algorithms. For example, printing of matches during time measurement is questionable because printing produces also additional overhead, which is partly unsynchronized. More generally, one should investigate all possibly measurement disturbances and rule them out, if possible. We think that the preprocessing of a pattern should not be included in the search time because the speed of an algorithm may otherwise depend on the length of the text.

Use of averages easily hides some details. Averages in general or calculations on speed-ups may lead to biased conclusions. One should be especially careful with arithmetic mean [8]. Median would be a better measure than arithmetic mean for many purposes because the effect of outliers is smaller, but computing median requires storing all the individual numbers.

The choice of an implementation language for algorithms usually limits available features: the number of different data types and the exactness of requirements given to them varies. The programming language Java is defined precisely, but it lacks the unsigned integer data type, which is useful for implementing bit-vectors. String type should not be used for serious string matching comparisons. On the other hand, the Java virtual machine adds an additional layer on the top of hardware. The programming language C is flexible, but its standard states quite loose requirements for the precision of integers. With assembly language, it would be possible to produce the most efficient machine code, while losing portability to different hardware. Nevertheless, the programming language C is currently the de facto language for implementing efficient exact string matching algorithms.

The C language standard from 1999 introduced the header file `stdint.h`, which is included via the header `inttypes.h`. The fastest minimum-width integer types designate integer types that are usually fastest to operate with among all the types that have at least the specified width. However, footnote 216 in the standard states 'The designated type is not guaranteed to be fastest for all purposes, if the implementation has no clear grounds for choosing one type over another, it will simply pick some integer type satisfying the signedness and width requirements'. For example, the choice of certain data types may cause implicit type conversions that may ruin the otherwise fast operation.

The exact-width integer types are ideal for use as bit-vectors. The typedef name `uint`N`_t` designates an unsigned integer type with a width of N bits. These types are optional. However, if the implementation (of a C compiler) provides integer types with widths 8, 16, 32, and 64 bits, it shall define the corresponding typedef names.

Therefore, the exact-width integer types should be available in all the C compilers conforming to the C99 standard.

The change of the running process from one core to another empties cache memories with various degree. Often caches are shared by several cores, slowing down reads from memory and induce annoying variation to the timing of test runs. To avoid it we recommend to use the Linux function `sched_setaffinity` to bind the process to only one processor or core. The use of this function reduced substantially variation in time measurements in our experiments.

3 Aspects on Testing Environments

SMART includes a wide selection of corpora with 15 different types of texts. In addition, it contains implementations over 100 string matching algorithms. Both texts and algorithm implementations serve as valuable reference material for any comparison of exact string matching, regardless of the testing approach used.

Both HS and SMART allow comparison of algorithms compiled with different parameters, like optimization level and buffer size, or with different compilers; unfortunately, this is occasionally also a source of incorrect results.

An advantage of SMART is that it verifies the correctness of a new algorithm by checking the output of the algorithm (the number of matches), but not that the implementation actually matches the target algorithm. In other words, the implementation might be incorrect despite producing correct output. For example, the SMART implementation[1] of the bndm algorithm [15] finds correct matches but it shifts incorrectly to the last found factor of the pattern instead of the last found prefix.

Moreover, unit tests, like the verification aspect in SMART, often fail to capture all erroneous cases. In cases where the verification fails, SMART does not directly support debugging code, therefore one may need at least a separate main program for debugging.

Another means of verifying correctness is to manually inspect the count of pattern occurrences, which is employed in HS. SMART reports the occurrences as average occurrences over a pattern set. These averages are based on integer division which may hide edge problems common in developing string matching algorithms.

The generating of pattern files is delegated to the user in HS. SMART dynamically generates patterns for each experiment, and this has some drawbacks. The pattern sets vary from run to run, which causes unpredictable variation between otherwise identical runs, as well as making debugging cumbersome.

Lastly, SMART cannot be used in the Unix-like subsystems of Microsoft Windows because they do not support shared memory.

4 Measuring Running Time

Background. The C standard library offers only the `clock` function for watching the CPU time usage of processes. To determine the time in seconds, the value returned by the `clock` function should be divided by the value of the macro `CLOCKS_PER_SEC`. Additionally the POSIX standard[2] declares that `CLOCKS_PER_SEC` is defined to be one million in `<time.h>`, and also that 'the resolution on any particular system need not

[1] Release 13.04

[2] IEEE Std 1003.1-2008

be to the microsecond accuracy'. Essentially, time is an internal counter in Linux that is incremented periodically. A periodic interrupt invokes an interrupt handler that increments the counter by one. At a common 100 Hz frequency, the counter has a granularity of 10 ms as it is incremented every 10 ms.

The POSIX standard offers the `times` function for getting process (and waited-for child process) times. The number of clock ticks per second can be obtained by a call `sysconf(_SC_CLK_TCK)`.

The POSIX function `clock_gettime` returns the current value *tp* for the specified clock, *clock_id*. The struct *tp* is given as a parameter. If `_POSIX_CPUTIME` is defined, implementations shall support the special *clock_id* value `CLOCK_PROCESS_CPUTIME_ID`, which represents the CPU-time clock of the calling process. The resolution of the given clock at the `clock_gettime` function is provided with the `clock_getres` function.

The POSIX function `gettimeofday` returns the current time, expressed as seconds and microseconds since the Epoch. Applications should use the `clock_gettime()` function instead of the obsolescent `gettimeofday()` function.

Algorithm timings in HS and SMART. The algorithm timings are affected in at least two ways. First, the functions to read time directly determine the timings. Second, the testing environments alter the internal state of the computer system which introduces interference on the algorithm execution and timing.

HS and SMART use two different functions, `times` and `clock_gettime` respectively, to read time. Their implementations are platform specific, but on x86/Linux they work somewhat similarly. The running times of algorithms in HS are given as user time (`tms_utime`) fetched with the `times` function. So the system time is excluded. Occasionally we have checked that in HS there is not any hidden use of the system time, but have never noticed such use. The function used in SMART, `clock_gettime` additionally improves the granularity to nanoseconds through other means such as interpolating a time stamp counter (TSC) that counts core cycles [2]. Furthermore, SMART includes time spent in user space and kernel space, whereas HS includes only user space time.

The rest of this section deals with time measuring in HS.

Precision of individual search. When the digital clock moves evenly, it is safe to assume that its value is incremented at regular fixed intervals. These time intervals are called *clock ticks*, and they are typically so long that during individual tick several instructions are executed. It this section, the term *clock tick* refers to the precision of time measurements, and it is assumed that the processor time increases one tick at a time. When the measuring of a time interval starts, we fetch the last updated value of the clock, but a part of the current clock tick may be already spent. This time follows the continuous uniform distribution $[0, 1]$. So its mean is 0.5 and variance $1/12$. Respectively when the measuring of a time interval ends, possibly a part of the current clock tick may be unspent. This slice follows the continuous uniform distribution $[-1, 0]$.

Thus the time measurement with clock ticks has an inaccuracy which is the sum of two error terms following the above mentioned distributions. When the length of the measured interval is at least one clock tick, the probability density function of

the sum is

$$f(x) = \begin{cases} 1 + x \text{ if } -1 \leq x \leq 0, \\ 1 - x \text{ if } 0 \leq x \leq 1, \\ 0 \qquad \text{otherwise.} \end{cases}$$

The mean of the inaccuracy caused by clock ticks is 0 and the variance 1/6.

The variance of the inaccuracy caused by clock ticks becomes relatively smaller, when the count of clock ticks increases. An easy way to achieve this is to use a longer text, which is at the same time more representative (statistically). However, it is not advisable to use concatenated multiples of a short text of a few kilobytes, because it is probable that the shifts of patterns start to follow similar sequences in such a case. This happens surely, if the pattern matches the text, assuming that the pattern is moved from left to right, and the shifting logic does not have any random behavior. Therefore the text produced with concatenation will with a high probability show the same statistical peculiarities as the original text element.

There are also other causes for inaccuracy. Generally, all context switches of a process produce some delay, which is very difficult to minimize on a single CPU system. We have noticed that on modern multicore processors it is possible to get a more accurate measurement of used CPU time than with singlecore processors spending similar number of clock ticks. The variance caused by other processes becomes relatively smaller, when the measured time intervals get longer. Then the results are more accurate.

Precision of search with a pattern set. The search with a pattern set brings yet another source of variance to the time measurements. Search for some patterns is more laborious, while others are highly efficient with algorithms tuned for them. This joint impact of the patterns and the algorithms can be seen as samples of the all possible cases between the worst and the best cases of a given algorithm. This kind of variance is minimal with the well known shift-or algorithm [1] where, in practice, a large number of occurrences cause small variation. One may also argue that if certain algorithms have a similar search time, the algorithm with smallest variance can be regarded the best.

When several successive time measurements are done within a relatively short period, it is possible that the unused time slice (before the next clock tick) is utilized in the next time measurement. Thus the time measurements may not be completely independent. In HS and SMART, preprocessing and search are alternating. If the measured time intervals are at least a few clock ticks, there is always sufficient variance that it is unlike that these surplus times cumulate more to either preprocessing or search.

Let us consider the variance of the mean. If the measurements are (statistically) independent, then the variance of the sum of times is the sum of the variances of individual time measurement. If the measurements have the same variance, and if they are independent of each other, the variance V of the mean of r measurements is

$$V\left(\frac{X_1 + X_2 + \cdots + X_r}{r}\right) = \frac{1}{r^2} V(X_1 + X_2 + \cdots + X_r) = \frac{r \cdot V(X_1)}{r^2} = \frac{V(X_1)}{r}$$

If the measured time is zero within the given accuracy of measurements, this could cause a bias in other measurements.

5 Cache Effects

CPU Cache memory is typically divided into three levels: L1, L2, and L3, which are accessed in this order. Caches that distinguish instruction data from other data are identified with suffixes i and d, respectively (e.g. L1i and L1d).

The relationship between subsequent runs and their memory accesses is clear in HS: all the runs of an algorithm are performed in a relatively tight loop. There is little other execution, beside proceeding from one loop iteration to another, between `-p` runs of `prep`, `-e` runs of `exec`[3], and each pattern in the patterns.

In the case of SMART, the relationship between subsequent runs and their memory accesses is more muddied. All the runs of `alg`[4] are executed in their own process. Quantifying execution in between the runs is not straightforward; a lot of it is performed by the operating system when creating and running processes. Moreover, SMART performs its own bookkeeping, such as reading the `search` times and storing them, in between the runs. It is quite possible that some of the residual data is replaced in the cache.

Faro et al. [7] claim that SMART is free of such residual data altogether. However, SMART takes no measures to prevent it and there is no reason why the data could not be accessed in the cache between different runs. In principle, this could be solved if the caches were logically addressed, but caches are often physically addressed. The physical addresses of the shared memory segments remain unchanged throughout the runs. That is, all the runs of a given algorithm reference the same physical addresses over and over.

One case where `alg` is less likely to access residual data in the cache is if all the caches are private, as SMART lets its processes run on any core without pinning them. As a result, if a run and a subsequent run are executed on different cores, the runs fail to look-up the data from their respective caches. That said, shared last level caches are common.

The lack of thread pinning in SMART comes with additional nondeterminism. Any of the processes might be migrated from one core to another during execution, which disrupts multiple aspects of the execution including the caches, branch prediction, and prefetching. Moreover, cache line states may affect cache latencies. A cache line in a shared state may have a different access latency than the cache line in a different state [14], and cache line states are currently unpredictable in SMART.

Lastly, caches also cache instructions. Similar reasoning (as above) applies to reading instructions from the cache between runs. However, string matching algorithms often compile to only a few hundred instructions. Thus, the space they occupy is small. Whether they load from the cache or main memory on the first accesses likely has little overall effect. The same few hundred instructions are referenced continuously, which should always hit the cache after the first accesses. The first accesses are few compared to the overall accesses during string matching.

Experiments. We performed extensive experiments with HS and SMART in order to find out how cache memory affects running times of algorithms. The tests were run in two core Intel Core i7-6500U CPU (Skylake microarchitecture) with 16 GiB

[3] `prep` is the subprogram for preprocessing and `exec` is the subprogram for searching. Their repeats are given with options `-p` and `-e`.

[4] `alg` is the subprogram inside which a particular algorithm is embedded.

DDR3-1600 SDRAM memory. Each core has 32 KiB L1d cache, and 256 KiB exclusive L2 cache, the 4 MiB inclusive L3 cache is shared. The operation system was Ubuntu 16.04 LTS. We used a widely used interface, PAPI [3], for accessing the hardware performance counters and to interpret phenomena during runs. All the running times shown were obtained without PAPI, although the overhead of PAPI was minimal.

To interpret the resulting cache metrics during string matching, the values must be compared to some reference value. String matching algorithms behave very predictably in that the pattern is continuously shifted from left to right over the text without ever backtracking. At each alignment window, i.e. an alignment of the pattern in the text, at least one text character is inspected before possibly shifting the pattern again. Accessing the text character unavoidably fills the corresponding cache line in the highest level cache (L1d). Now, under the assumption that the text does not reside in the cache, the access results in a cache miss across the whole cache hierarchy all the way to main memory. Thus, given maximal shifts of m, the lower bound for cache misses is

$$\lfloor n/max(m, \text{cache line size}) \rfloor \tag{1}$$

where n is the text size. Clearly, multiple accesses to the same cache line in short succession are not going to fill the cache line into the cache over and over again. Hence, a pattern incapable of shifting past whole cache lines only fills a cache line once despite possibly referencing it on multiple alignment windows.

In general, with any shift length, it is fair to assume that a cache line is never filled into the cache more than once. A text character remains in alignment windows spanning at most $2m - 1$ characters. After the alignment windows have passed the text character, it is never referenced again. Modern caches are several times the size of even the larger patterns. Only a bad cache line replacement policy or a bad hardware prefetcher would evict the corresponding cache line during processing the $2m - 1$ window.

The above lower bound ignores memory accesses to the pattern and its preprocessed data structures. However, these realistically cause very few cache misses. As stated above, patterns are relatively small compared to cache sizes. A very rarely occurring pattern might drop cache lines towards its one end but the dropped data is refilled as rarely as the pattern occurs. All in all, the lower bound is the expected number of cache misses during string matching.

The following cache metrics results and other measurements have been collected over a static set of 500 random patterns. All of the experiments used concatenated multiples of 1 MiB prefix of King James Version (KJV) as the text to keep searches over different text sizes as comparable as possible.

In the experiment with HS, each pattern is searched `-e` times such that the effective text size is roughly 100 MiB. A pattern search is repeated until roughly 100 MiB of text has been covered. E.g., `-e` is 100 for 1 MiB text, 50 for 2 MiB text, etc. Such a sliding scale is necessary because short running times involve a large margin of error from the low granularity of the `times` function, while constant `-e` repetitions suitable for small texts cause too long an experiment for larger texts. Ideally, the effective text size would be always 100 MiB but 100 is not divisible by all text sizes (e.g. 6 MiB). So the point is to minimize error introduced by `times`, which we deemed to be minimal at around 100 MiB of text. In the case of SMART, a static pattern

set of 500 patterns was multiplied[5] to effectively search 100 MiB of text. For HS, we ported the SMART implementations of algorithms. The algorithms were compiled with `-O3` optimization level. The reported results excluded preprocessing and they are arithmetic means over all of the executions.

Table 1 shows results from running the brute force algorithm ($m = 16$) against KJV on both testing environments. The table also includes the expected cache misses explained above. If the cache misses fall below the expected value on any of the caches, the cache contained residual data.

Table 1. Measured average cache metrics during string matching with the brute force algorithm ($m = 16$).

	Text size (MiB)	1	2	4	8	16	32
	Expected misses	16 384	32 768	65 536	131 072	262 144	524 288
HS	L3 requests	34 303	68 695	139 158	280 662	564 696	1 131 615
	L3 misses	452	3 922	28 459	164 434	389 601	809 080
	L2 requests	37 713	75 630	151 011	301 570	603 800	1 206 983
	L2 misses	21 718	43 453	86 954	174 312	348 535	696 914
	L1d misses	16 434	32 829	65 622	131 502	262 926	525 782
SMART	L3 requests	36 085	72 196	143 657	287 142	574 129	1 146 851
	L3 misses	26 542	53 456	106 510	213 023	426 182	852 296
	L2 requests	37 735	75 478	150 871	301 502	602 771	1 204 804
	L2 misses	21 728	43 616	87 153	174 071	347 920	695 519
	L1d misses	16 559	32 933	65 828	131 599	263 075	525 832

The L1d misses are approximately 16k for every 1 MiB increase in text size, which matches the expected value. That is, the L1d contains no residual data between multiple runs. Additionally, the cache misses strongly suggest that a cache line is only ever loaded into the cache once throughout string matching, as reasoned above. Hypothetically, the L2 prefetchers could thrash L2 and L3, but this seems unrealistic[6].

The most important metric, L3 misses, clearly indicates the existence of residual data in HS for short texts. The L3 misses are too few for the smaller text sizes. That is, many requests to L3 hit instead of miss. The trend is also such that the misses grow at a changing rate. The expectation is a constant increase of the L3 misses since the caches should behave similarly from one text size to another. At minimum, a doubling of the text size causes a doubling of the L3 misses. This only happens towards the largest text sizes, but not the smaller ones.

Figure 1 shows relative increases in running times for the brute force algorithm (bf), Horspool's algorithm [10] (hor), and the sbndmq2 algorithm [4] in HS as a function of text size. The y values are relative changes to the respective running times for the text of 1 MiB (thus y is zero at $x = 1$). In other words, the inverses of search speeds (s/MiB) are compared.

For HS in Figure 1 there is a modest increase between -1% and 14% in the relative running times as the text size grows until it is roughly 1.5 times the size of the last level cache L3. Longer patterns exhibit larger increase because the same amount of cache misses divide over shorter running times. Moreover, the hardware prefetchers have less time to perform their function, which possibly leads to a bigger overlap between demand and prefetch request. Similarly, bf exhibits very little increase because the

[5] This required small changes to the code of SMART.

[6] Note that prefetching to cache may go to a different cache level in some other processors.

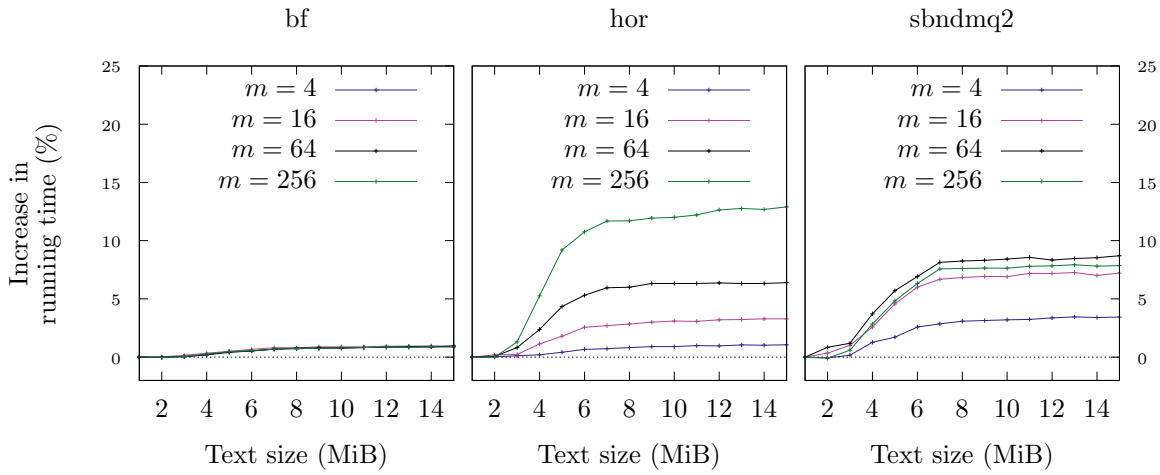

Figure 1. Relative increases in running times during string matching in HS when adjusting text size for three algorithms and four pattern lengths.

running times are already large from its many memory accesses and instructions executed with little variation between pattern sizes.

As Figure 1 shows, the increase of running time depends on the length of the text and the algorithm for a fixed pattern size. So the text should be long enough, 1.5 times the size of L3, that the experiment would be close to a steady state.

We ran a similar experiment to Figure 1 in SMART and repeated it on another computer. The results were incoherent — there was no consistent decrease of speed when the text grows. More investigation would be necessary to understand how cache affects running times of algorithms in SMART.

6 Effects of Shared Memory

The SMART environment [7] uses shared memory for storing the text. The obvious reason for that is to make it easier to execute multiple tests in one run. We noticed inconsistent differences in timing results of certain algorithms in HS and SMART (see Table 3). When we investigated those findings carefully, we found out that the use of shared memory was the reason.

While running the data cache experiments, SMART exhibited unexplained behavior. Invalidating cache lines with `clflush` resulted in significantly faster running times of algorithms. A similar effect was observable whenever the shared memory was touched in any way in `alg` before string matching. The reason turns out to be minor page faults that occur on every first page access which is explored next.

To inspect this, let us count the minor page faults that occur during a string matching. On our test computer, PAPI includes a native event `perf::MINOR-FAULT` to count minor page faults. Figure 2 plots the counts over multiple text sizes for both HS and SMART. The difference between the two is very apparent. HS incurs no page faults during string matching whereas the number of SMART grows linearly at roughly the rate of 250 page faults per 1 MiB of text. With a 4 KiB page size, 250 page faults equate 4 KiB * 250 = 1000 KiB ≈ 1 MiB of memory, which matches the text size. Basically, every page backing the shared memory faults once. These minor page faults are irrespective of any other parameter such as the algorithm used, pattern size, the test computer, etc.

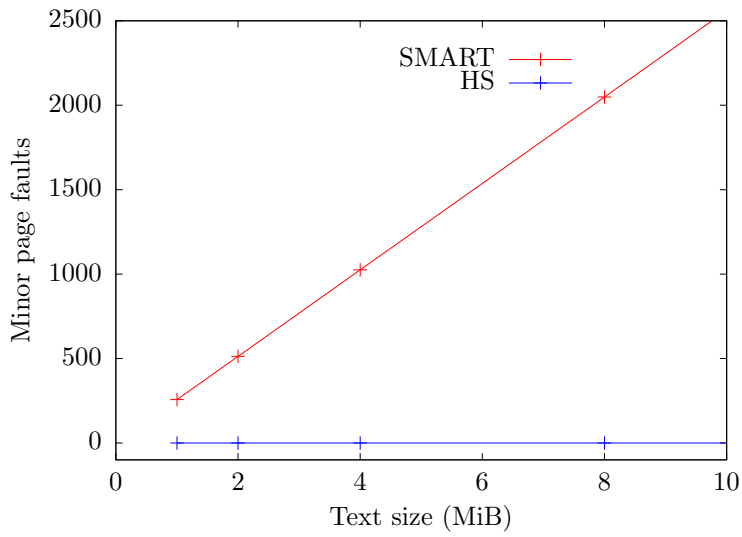

Figure 2. Minor page faults during each string matching.

The reason for the minor page faults is not entirely apparent, but it most likely traces back to the memory management techniques of Linux. Rusling [17] briefly describes System V shared memory in the book *The Linux Kernel*. According to him, attaching a process to shared memory merely modifies the virtual address space of the process, but does not back it up with physical memory. Thus, the first memory access of a process to any page of shared memory generates a page fault which supports the above observation. The actual reason is not further explored and other resources on the topic seem to be scarce. However, Linux generally employs lazy allocation of main memory with techniques such as demand paging, copy-on-write, and memory overcommitment. Perhaps the page faults to shared memory line up with this philosophy.

Whatever the cause, the underlying implications are problematic for running times. First, the continuous interrupts disrupt normal execution of algorithms. The interrupt handler requires context switching, modifying page tables, etc. Second, SMART uses wall-clock time to measure running times. The timings comprise both user space and kernel space execution, which includes time spent on resolving the page faults. These add up given the little time spent on one page.

Figure 3 illustrates these effects. It shows the difference in running times when pages are prefaulted compared to faulting during string matching. The figure shows the differences for multiple pattern lengths for all the implemented algorithms in SMART. Prefaulting can be achieved by explicitly accessing each page or by locking the memory (e.g. with `mlockall`[7]). The former is effectively what invalidating the cache lines did. The latter locks the virtual address space of a process into main memory to ensure it never faults. We used memory locking to measure the prefaulted running times, which additionally required reconfiguring system-wide maximum locked memory size. The change to `alg` is a simple addition given in Figure 4.

Figure 3 reveals insights on the effects of page faults. For $m > 4$, the page faults result in a fairly steady ∼1.2 ms average increase in running times. The increase has little variation across all the algorithms, but there still exists a dozen outliers consistently over the different pattern sizes. For $m \leq 4$, the differences seem more erratic

[7] Defined in POSIX Realtime Extensions 1003.1b-1993 and 1003.1i-1995

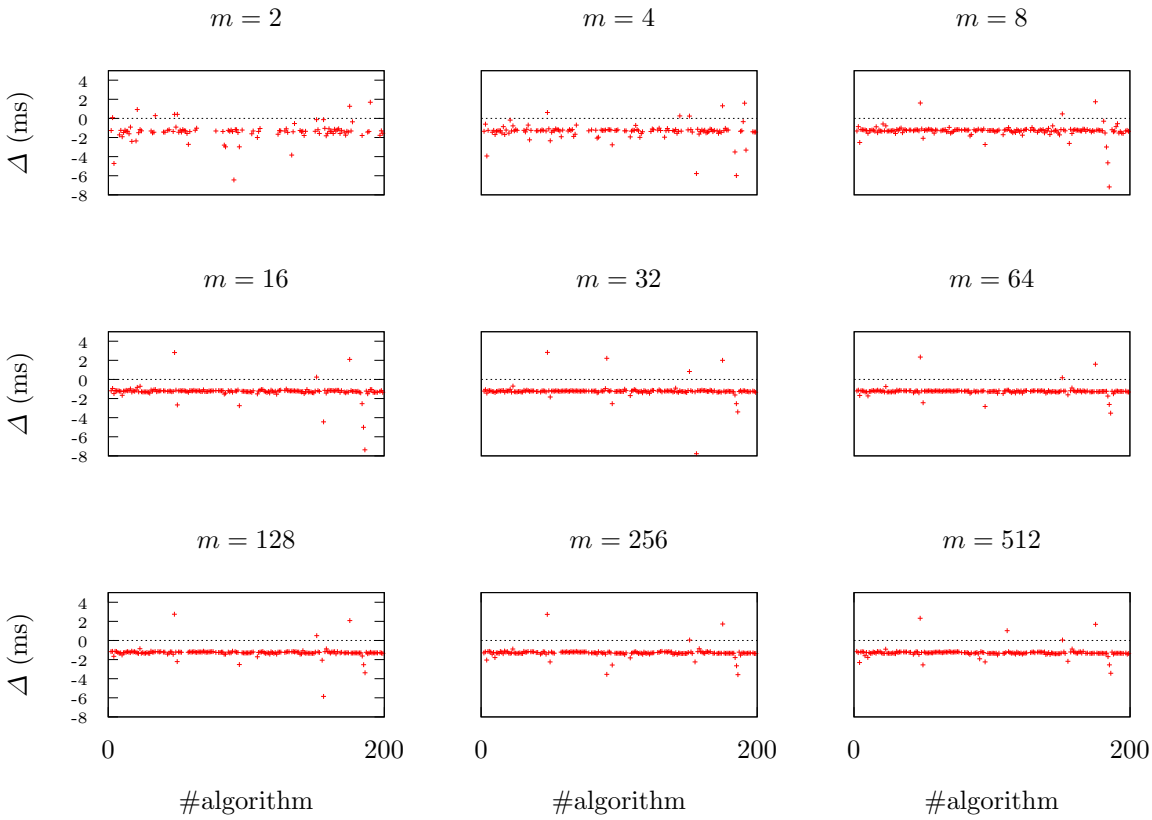

Figure 3. Change in running times (new running time − old running time) for each algorithm in SMART when prefaulting pages on the Skylake computer given 4 MiB KJV.

```
#include <sys/mman.h>
...
if (mlockall(MCL_CURRENT) == -1) {
  perror("mlockall");
  return 1;
}
int count = search(p,m,t,n);
...
```

Figure 4. An addition to `alg` to lock memory.

and, admittedly, a few algorithms deviate from the average increase that otherwise do not. However, less algorithms work for these pattern sizes which contributes to the impression of more deviation.

Figure 3 also gives a view to the accuracy of the time measurement method of SMART. The cause of the farthest outliers needs further study.

Overall, the changes in running times skew algorithm comparisons. Increasing the running time of (almost) every algorithm dilutes relative differences. For example, a closer inspection on the fastest algorithms according to the original running times is presented in Table 2. The largest increase at $m = 512$ is almost 90%. Comparing the original time 1.47 ms to another algorithm with an original time 1.57 ms yields a difference of only ∼7%. Comparing the prefaulted time 0.16 ms to the prefaulted time 0.26 ms of the other algorithm yields a difference of ∼63%, which is almost an order of magnitude different. This effect is more pronounced on large patterns as they tend

to be faster. Moreover, the outliers are evaluated unfairly as they might overtake or fall behind other algorithms after prefaulting.

Table 2. Original and prefaulted running times (ms) of the fastest algorithms according to the original running times.

m	2	4	8	16	32	64	128	256	512
Original	1.73	1.86	2.04	1.84	1.57	1.56	1.57	1.52	1.47
Prefaulted	0.54	0.66	0.83	0.62	0.37	0.36	0.25	0.19	0.16

Lastly, the diluted relative differences can be further demonstrated by comparing running times in HS, SMART, and SMART with prefaulting. Table 3 shows such running times for bf, hor, and sbndmq2 introduced in Section 5. The HS and the prefaulted SMART running times are quite close to one another, while the running times of actual SMART are larger. For instance, sbndmq2 is 15% and 16% of the running time of bf in HS and prefaulted SMART, respectively, while it is 27% in SMART.

Table 3. Average per pattern running times (ms) for 8 MiB KJV and $m = 16$.

	bf	hor	sbndmq2
HS	16.31	5.09	2.44
SMART	18.75	7.75	5.04
prefaulted SMART	16.28	5.31	2.61

It is unfortunate that the use of shared memory disturbs running times, though the original aim was obviously to achieve more dependable results. Our correction eliminates the disturbance. However, the correction is rude and it is not yet suitable for production use.

7 Other Issues

The space character is typically the most frequent character in a text of natural language. Therefore, the result of an experimental comparison may depend on whether the patterns contain spaces or not [11,4]. Especially, the space as the last character of a pattern slows down many algorithms of the Boyer–Moore type if the pattern contains another space.

One problem in comparing algorithms is the tuning/tune-up level. Should one compare original versions or versions at the same tune-up level? Skip loop, sentinel, guard, and multicharacter read are all tune-ups which may greatly affect the running time. Even the implementations in the SMART repository are not fully comparable in this respect. For example, in the past it was a well-known fact that the memcmp function is slower than an ordinary match loop. So the SMART implementation[8] of Horspool's algorithm uses a match loop instead of memcmp applied in the original algorithm [10]. However, memcmp is now faster than a match loop on many new processors.

The results of a comparison may depend on the technology used. Thus, results of old comparisons may not hold any more. We demonstrate this by an example. We ran an experiment of two algorithms sbndm4 [4] with 16-bit reads and ufast-rev-md2 [11]

[8] Release 13.04

on two processors of different age. These processors (Intel Pentium and Intel Core i7-4578U) were introduced in 1993 and 2014, respectively. The text was KJV and m was 10. Sbndm4 was considerably faster on i7 — its running time was only 23% of the running time of ufast-rev-md2, but the situation was the opposite on Pentium: the running time of ufast-rev-md2 was 32% of the running time of sbndm4. Potential sources for the great difference are changes in relative memory speed, cache size, and penalty for misaligned memory accesses.

Likewise, two compilers may produce dissimilar results — see, for example, the running time of NSN in [12]. Sometimes, an old algorithm using much memory can become relatively faster in a newer computer — the algorithm by Kim and Shawe-Taylor [13] is such an example. This reflects another downside of technology development: you may find an old "inefficient" and rejected algorithm idea of yours has recently become viable, and is then published by someone else.

Which then could be a more universal measure than execution time to compare algorithms? Some researchers count character comparisons. When the first string matching algorithms were introduced, the number of comparisons was an important measure to reflect the work load of an algorithm. Because many of newer algorithms, like bndm [15], do not use comparisons, researchers started to use the number of read text characters. When the technology advances, even the number of read text characters is no longer a good estimate for speed, as the brute force algorithm with a q-gram guard [16] shows.

One problem of the area of string matching is that the developers are enthusiastic about too small improvements. Differences less than 5% are not significant in practice. Small changes in the code, like reordering of variables and arrays, or switching the computer may contribute a similar difference. We think that 20% is a fair threshold for a significant improvement.

8 Conclusions

Mostly we reviewed good testing practices but there are issues which may lead incorrect conclusions in algorithm comparisons. Experimental algorithm rankings are never absolute because evolving technology affects them. The ranking order of algorithms may even change when the comparison is repeated on another processor of the same age or generation. Therefore, conclusions based on a difference of less than 5% in running times are not acceptable. When selecting data for experiments, the length of text should be at least 1.5 times the cache size in order to avoid cache interference with running times. The most remarkable finding of this paper is how the use of shared memory may disturb running times.

References

1. R. A. Baeza-Yates and G. H. Gonnet: *A new approach to text searching.* Commun. ACM, 35(10) 1992, pp. 74–82.
2. J. Boháč: *Reliable TSC-based timekeeping for platforms with unstable TSC*, Master's thesis, Charles University, Prague, Czech Republic, 2008.
3. S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci: *A portable programming interface for performance evaluation on modern processors.* International Journal of High Performance Computing Applications, 14(3) 2000, pp. 189–204.
4. B. Durian, J. Holub, H. Peltola, and J. Tarhio: *Improving practical exact string matching.* Inf. Process. Lett., 110(4) 2010, pp. 148–152.

5. S. Faro: *SMART*. `https://github.com/smart-tool/smart`, 2016, Commit cd7464526d41396e11912c6a681eddb965e17f58. Accessed 12.6.2020.
6. S. Faro and T. Lecroq: *The exact online string matching problem: A review of the most recent results*. ACM Computing Surveys, 45(2) 2013.
7. S. Faro, T. Lecroq, S. Borzì, S. D. Mauro, and A. Maggio: *The string matching algorithms research tool*, in Proceedings of the Prague Stringology Conference 2016, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2016, pp. 99–111.
8. P. J. Fleming and J. J. Wallace: *How not to lie with statistics: The correct way to summarize benchmark results*. Commun. ACM, 29(3) 1986, pp. 218–221.
9. T. Hirvola: *Bit-parallel approximate matching of circular strings with k mismatches*. `https://github.com/hirvola/bsa`, 2017, Commit 1f5264c481ea4152c68c47cdfe2c76657448ba7c. Accessed 20.11.2020.
10. R. N. Horspool: *Practical fast searching in strings*. Software: Practice and Experience, 10(6) 1980, pp. 501–506.
11. A. Hume and D. Sunday: *Fast string searching*. Software: Practice and Experience, 21(11) 1991, pp. 1221–1248.
12. M. A. Khan: *A transformation for optimizing string-matching algorithms for long patterns*. The Computer Journal, 59(12) 2016, pp. 1749–1759.
13. J. Y. Kim and J. Shawe-Taylor: *Fast string matching using an n-gram algorithm*. Software: Practice and Experience, 24(1) 1994, pp. 79–88.
14. D. Levinthal: *Performance analysis guide for Intel® Core i7 Processor and Intel® Xeon 5500 processors*. Intel, 2009.
15. G. Navarro and M. Raffinot: *A bit-parallel approach to suffix automata: Fast extended string matching*, in Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching, M. Farach-Colton, ed., Berlin, Heidelberg, 1998, Springer-Verlag, pp. 14–33.
16. W. Pakalén, J. Tarhio, and B. W. Watson: *Searching with extended guard and pivot loop*, in Proceedings of the Prague Stringology Conference 2021, Czech Technical University in Prague, Czech Republic, 2021.
17. D. A. Rusling: *The Linux Kernel*, New Riders Pub, 2000.