How to write a fast plain text searching tool using modern C++


Why?

Goals

- Build reusable and fast text processing libraries.
- Create an usable text searching command-line utility which is as fast as grep, ripgrep, and/or ag.

Background

Generic programming

Generic programming is a style of computer programming in which algorithms are written in terms of types to-be-specified-later that are then instantiated when needed for specific types provided as parameters. This approach, pioneered by ML in 1973,[1][2] permits writing common functions or types that differ only in the set of types on which they operate when used, thus reducing duplication.

Policy based design

```
Policy-based design, also known as policy-based class design or
policy-based programming, is a computer programming paradigm based
on an idiom for C++ known as policies. It has been described as a
compile-time variant of the strategy pattern, and has connections
with C++ template metaprogramming. It was first popularized by
Andrei Alexandrescu with his 2001 book Modern C++ Design and his
column Generic<Programming> in the C/C++ Users Journal.
```

What is SIMD?

Single instruction, multiple data (SIMD) is a class of parallel computers in Flynn's taxonomy. It describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously. Such machines exploit data level parallelism, but not concurrency: there are simultaneous (parallel) computations, but only a single process (instruction) at a given moment. SIMD is particularly applicable to common tasks such as adjusting the contrast in a digital image or adjusting the volume of digital audio. Most modern CPU designs include SIMD instructions to improve the performance of multimedia use.

Benchmark tools

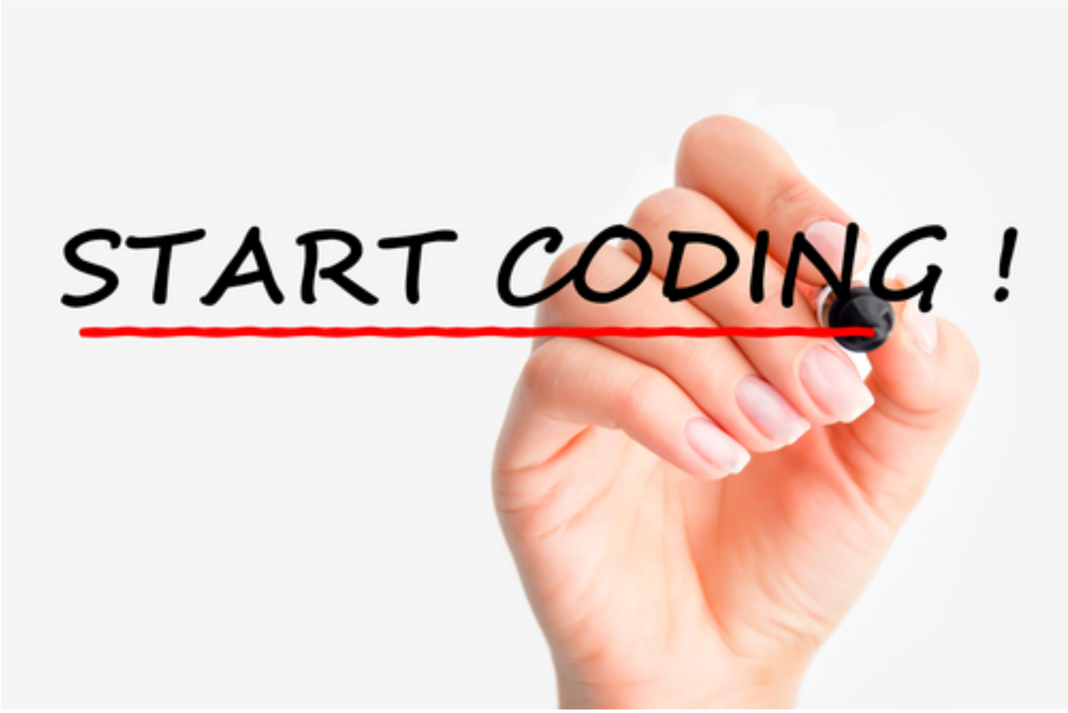
- Google benchmark
- Celero
- perf
- strace

Test environments

- Linux:
    - CPU: Xeon(R) E5-2699, Core i7 i920, Core i7 6th
    - Memory: 773519 MBytes
    - Storage: SSD and network storage
    - Kernel: 3.8.13 and 4.17
- Mac OS:
    - CPU: Intel(R) Core(TM) i7-4770HQ CPU @ 2.20GHz
    - Memory: 16 GB
    - Storage: SSD
- Compiler
    - gcc-5.5 and gcc-7.3
    - clang-900.0.39.2

Test data

- Test data and test patterns are discussed here
- Log data


START CODING !

The anatomy of a text-searching tool

- Gather files to search.
- Read text data from files
- Search for a pattern from the text data.
- Print out the search results

What is the most efficient way to read data from a text file?

A typical C++ solution

```
    size_t iostream_linestats(const std::string &afile) {
        std::ifstream t(afile);
        size_t lines = 0;
        std::for_each(std::istreambuf_iterator<char>(t), std::istreambuf_iterator<
                      [&lines](auto const item) {
                          if (item == EOL) ++lines;
                      });
        return lines;
}
```

A memory mapped solution using Boost

```
size_t read_memmap(const char *afile) {
    boost::iostreams::mapped_file mmap(afile, boost::iostreams::mapped_file::r
    auto begin = mmap.const_data();
    auto end = begin + mmap.size();
    size_t counter = 0;
    for (auto iter = begin; iter != end; ++iter) {
        counter += *iter == EOL;
    }
    return counter;
}
```

A memory mapped solution using low-level APIs

```
// Open data file for reading
int fd = open(datafile, O_RDONLY);
if (fd == -1) { handle_error("Cannot open "); }

// Obtain file size
struct stat info;
if (fstat(fd, &info) == -1) { handle_error("Cannot get information of "); }
size_t length = info.st_size;

// Create mapped memory
const int flags = MAP_PRIVATE;
char *begin = static_cast<char *>(mmap(nullptr, length, PROT_READ, flags, fd,
```

Read data in blocks using low-level APIs

```
    size_t block_count = (buf.st_size / BUFFER_SIZE) + (buf.st_size % BUFFER_SIZE
    for (size_t blk = 0; blk < block_count; ++blk) {
        long nbytes = ::read(fd, read_buffer, BUFFER_SIZE);
        if (nbytes < 0) {
            const std::string msg =
                std::string("Cannot read from file \"") + std::string(datafile) +
            throw(std::runtime_error(msg));
        };

        // Apply a given policy to read_buffer.
        Policy::process(read_buffer, nbytes);
    }
```

Benchmark results

```
hungptit@hungptit ~/w/i/benchmark> ./file_read
Celero
Timer resolution: 0.001000 us
-----------------------------------------------------------------------------------
Group           |   Experiment    |   Prob. Space   |     Samples     |   Iteratio
-----------------------------------------------------------------------------------
read            | boost_memmap    |            Null |              40 |
read            | mmap_reader_mem |            Null |              40 |
read            | memmap          |            Null |              40 |
read            | read_chunk      |            Null |              40 |
read            | ioutils_std     |            Null |              40 |
read            | ioutils_memchr  |            Null |              40 |
Complete.
```

Benchmark results (cont)

```
hungptit@hungptit ~/w/i/benchmark> strace -c ./file_read chunk 3200.txt
Number of lines: 302278
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
86.58    0.001419           5       250           read
 4.21    0.000069           4        17           mmap
 3.29    0.000054           4        12           mprotect
 2.01    0.000033           4         7           openat
 1.10    0.000018           2         7           close
 1.10    0.000018           2         8           fstat
 0.49    0.000008           8         1           write
 0.37    0.000006           6         1           munmap
 0.31    0.000005           1         3           brk
 0.12    0.000002           1         2           rt_sigaction
 0.12    0.000002           2         1           arch_prctl
 0.12    0.000002           2         1           set_tid_address
 0.06    0.000001           1         1           rt_sigprocmask
 0.06    0.000001           1         1           set_robust_list
 0.06    0.000001           1         1           prlimit64
 0.00    0.000000           0         1         1 access
 0.00    0.000000           0         1           execve
------ ----------- ----------- --------- --------- ----------------
100.00    0.001639                   315         1 total
```

Benchmark results (cont)

```
hungptit@hungptit ~/w/i/benchmark> strace -c ./file_read mmap 3200.txt
Number of lines: 302278
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
36.33    0.000311         155         2           munmap
26.52    0.000227          12        18           mmap
11.33    0.000097           8        12           mprotect
 7.59    0.000065           9         7           openat
 3.27    0.000028           3         8           fstat
 2.92    0.000025           5         5           read
 2.80    0.000024           3         7           close
 2.22    0.000019          19         1         1 access
 1.17    0.000010          10         1           execve
 1.05    0.000009           9         1           write
 1.05    0.000009           3         3           brk
 0.93    0.000008           8         1           madvise
 0.70    0.000006           3         2           rt_sigaction
 0.47    0.000004           4         1           arch_prctl
 0.47    0.000004           4         1           set_tid_address
 0.47    0.000004           4         1           prlimit64
 0.35    0.000003           3         1           rt_sigprocmask
 0.35    0.000003           3         1           set_robust_list
------ ----------- ----------- --------- --------- ----------------
100.00    0.000856                    73         1 total
```

Why do we use memchr?

```
#
# Overhead  Command    Shared Object     Symbol
# ........  .........  ................  ......................................
#
61.98%  file_read  libc-2.26.so     [.] __memchr_sse2
6.55%  file_read  [kernel.vmlinux]  [k] filemap_map_pages
6.06%  file_read  file_read         [.] main
3.76%  file_read  ld-2.26.so        [.] strcmp
3.75%  file_read  ld-2.26.so        [.] _dl_check_map_versions
3.58%  file_read  ld-2.26.so        [.] _dl_lookup_symbol_x
3.49%  file_read  libc-2.26.so      [.] _dl_addr
3.41%  file_read  [kernel.vmlinux]  [k] unlock_page
3.07%  file_read  [kernel.vmlinux]  [k] lock_page_memcg
3.05%  file_read  [kernel.vmlinux]  [k] page_remove_rmap
1.15%  file_read  [kernel.vmlinux]  [k] vmacache_find
0.14%  perf       [kernel.vmlinux]  [k] apic_timer_interrupt
0.01%  perf       [kernel.vmlinux]  [k] end_repeat_nmi
0.00%  perf       [kernel.vmlinux]  [k] __intel_pmu_enable_all.constprop.21
```

Summary

- Low level memory mapped files and reading in chunks are fastest solutions.
- memchr does significantly speedup our line counting algorithms.
- Both memory mapped files and reading in chunks algorithms can be used for our purposes.

How to search a pattern from the text data fast?

Simple exact text matching algorithm

```
void process(const char *begin, const size_t len) {
  const char *start = begin;
  const char *end = begin + len;
  const char *ptr = begin;
  while ((ptr = static_cast<const char *>(memchr(ptr, EOL, end - ptr)))) {
    linebuf.append(start, ptr - start + 1);
    process_linebuf();
    linebuf.clear();

    // Update parameters
    start = ++ptr;
    ++lines;

    // Stop if we reach the end of the buffer.
    if (start == end)
      break;
  }

  // Update the line buffer with leftover data.
  if (start != end) {
    linebuf.append(start, end - start);
    process_linebuf();
  }
  pos += len;
}
```

Simple exact text matching alg (cont)

```
void process_line(const char *begin, const size_t len) {
  if (matcher.is_matched(begin, len)) {
    fmt::print("{0}:{1}", lines, std::string(begin, len));
  }
}

void process_linebuf() { process_line(linebuf.data(), linebuf.size()); }

struct ExactMatch {
  explicit ExactMatch(const std::string &patt) : pattern(patt) {}
  bool is_matched(const std::string &line) {
    if (line.size() < pattern.size()) {
      return false;
    }
    return line.find(pattern) != std::string::npos;
  }
  const std::string pattern;
};
```

fgrep vs grep

```
Performance counter stats for './fgrep "LEVEL":"error" /mnt/weblogs/scribe/workqu

    12247.710568 task-clock                #    0.999 CPUs utilized          (
              28 context-switches          #    0.002 K/sec                  (
               7 cpu-migrations            #    0.001 K/sec                  (
             745 page-faults               #    0.061 K/sec                  (
  26,970,906,925 cycles                    #    2.202 GHz                    (
 <not supported> stalled-cycles-frontend
 <not supported> stalled-cycles-backend
  66,706,695,883 instructions              #    2.47  insns per cycle        (
  18,238,272,894 branches                  # 1489.117 M/sec                  (
     223,213,329 branch-misses             #    1.22% of all branches        (

    12.265244742 seconds time elapsed                                        (
```

fgrep vs grep (cont)

```
hdang@dev115 ~/w/f/commands> /usr/sbin/perf stat -r 3 grep '"LEVEL":"error"' /mnt/

 Performance counter stats for 'grep "LEVEL":"error" /mnt/weblogs/scribe/workqueue

       3923.676736 task-clock                #    0.998 CPUs utilized            (
                19 context-switches          #    0.005 K/sec                    (
                 9 cpu-migrations            #    0.002 K/sec                    (
               804 page-faults               #    0.205 K/sec
     8,640,378,363 cycles                    #    2.202 GHz                      (
   <not supported> stalled-cycles-frontend
   <not supported> stalled-cycles-backend
     4,413,214,598 instructions              #    0.51  insns per cycle          (
       866,115,263 branches                  #  220.741 M/sec                    (
        44,463,657 branch-misses             #    5.13% of all branches          (

       3.929624555 seconds time elapsed                                          (
```


IM POSSIBLE

Technology Beats Algorithms (in Exact String Matching)

Jorma Tarhio, Jan Holub, Emanuele Giaquinta

(Submitted on 5 Dec 2016)

More than 120 algorithms have been developed for exact string matching within the last 40 years. We show by experiments that the \naive{} algorithm exploiting SIMD instructions of modern CPUs (with symbols compared in a special order) is the fastest one for patterns of length up to about 50 symbols and extremely good for longer patterns and small alphabets. The algorithm compares 16 or 32 characters in parallel by applying SSE2 or AVX2 instructions, respectively. Moreover, it uses loop peeling to further speed up the searching phase. We tried several orders for comparisons of pattern symbols and the increasing order of their probabilities in the text was the best.

Subjects: **Data Structures and Algorithms (cs.DS)**
Cite as: **arXiv:1612.01506 [cs.DS]**
(or **arXiv:1612.01506v1 [cs.DS]** for this version)

Submission history

From: Jan Holub [view email]
[v1] Mon, 5 Dec 2016 20:34:39 GMT (64kb)

SSE2 version of std::string::find

```
size_t sse2_strstr_anysize(const char *s, size_t n, const char *needle,
                           size_t k) {
  const __m128i first = _mm_set1_epi8(needle[0]);
  const __m128i last = _mm_set1_epi8(needle[k - 1]);
  for (size_t i = 0; i < n; i += 16) {
    const __m128i block_first =
        _mm_loadu_si128(reinterpret_cast<const __m128i *>(s + i));
    const __m128i block_last =
        _mm_loadu_si128(reinterpret_cast<const __m128i *>(s + i + k - 1));
    const __m128i eq_first = _mm_cmpeq_epi8(first, block_first);
    const __m128i eq_last = _mm_cmpeq_epi8(last, block_last);
    uint16_t mask = _mm_movemask_epi8(_mm_and_si128(eq_first, eq_last));
    while (mask != 0) {
      const auto bitpos = bits::get_first_bit_set(mask);
      if (memcmp(s + i + bitpos + 1, needle + 1, k - 2) == 0) {
        return i + bitpos;
      }
      mask = bits::clear_leftmost_set(mask);
    }
  }
  return std::string::npos;
}
```

AVX2 version of std::string::find

```
size_t FORCE_INLINE avx2_strstr_anysize(const char *s, size_t n,
                                        const char *needle, size_t k) {
  const __m256i first = _mm256_set1_epi8(needle[0]);
  const __m256i last = _mm256_set1_epi8(needle[k - 1]);
  for (size_t i = 0; i < n; i += 32) {
    const __m256i block_first =
        _mm256_loadu_si256(reinterpret_cast<const __m256i *>(s + i));
    const __m256i block_last =
        _mm256_loadu_si256(reinterpret_cast<const __m256i *>(s + i + k - 1));
    const __m256i eq_first = _mm256_cmpeq_epi8(first, block_first);
    const __m256i eq_last = _mm256_cmpeq_epi8(last, block_last);
    uint32_t mask = _mm256_movemask_epi8(_mm256_and_si256(eq_first, eq_last));
    while (mask != 0) {
      const auto bitpos = bits::get_first_bit_set(mask);
      if (memcmp(s + i + bitpos + 1, needle + 1, k - 2) == 0) {
        return i + bitpos;
      }
      mask = bits::clear_leftmost_set(mask);
    }
  }
  return std::string::npos;
}
```

Micro-benchmark results

```
2018-06-12 17:52:34
Running ./string
Run on (88 X 2199.78 MHz CPU s)
CPU Caches:
  L1 Data 32K (x44)
  L1 Instruction 32K (x44)
  L2 Unified 256K (x44)
  L3 Unified 56320K (x2)
--------------------------------------------------------
Benchmark                  Time           CPU Iterations
--------------------------------------------------------
std_string_find          162 ns        161 ns    4373243
sse2_string_find          21 ns         21 ns   33215481
avx2_string_find          14 ns         14 ns   49368340
```

SSE2-fgrep benchmark results

```
Performance counter stats for 'grep "LEVEL":"error" /mnt/weblogs/scribe/workqueue

      3995.421584 task-clock                #    0.998 CPUs utilized            (
               18 context-switches          #    0.004 K/sec                    (
               11 cpu-migrations            #    0.003 K/sec                    (
              804 page-faults               #    0.201 K/sec
    8,798,363,579 cycles                    #    2.202 GHz                      (
  <not supported> stalled-cycles-frontend
  <not supported> stalled-cycles-backend
    4,399,286,215 instructions              #    0.50  insns per cycle          (
      863,854,226 branches                  #  216.211 M/sec                    (
       44,460,390 branch-misses             #    5.15% of all branches          (

      4.001814923 seconds time elapsed                                          (
```

AVX2-fgrep benchmark results

```
hdang@dev115 ~/w/f/commands> /usr/sbin/perf stat -r 3 ./fgrep '"LEVEL":"error"' /m

 Performance counter stats for './fgrep "LEVEL":"error" /mnt/weblogs/scribe/workqu

       3102.228162 task-clock                #    0.998 CPUs utilized            (
                23 context-switches          #    0.008 K/sec                    (
                 2 cpu-migrations            #    0.001 K/sec                    (
               744 page-faults               #    0.240 K/sec                    (
     6,831,415,116 cycles                    #    2.202 GHz                      (
   <not supported> stalled-cycles-frontend
   <not supported> stalled-cycles-backend
     9,108,286,074 instructions              #    1.33  insns per cycle          (
     1,675,216,078 branches                  #  540.004 M/sec                    (
        45,603,179 branch-misses             #    2.72% of all branches          (

       3.109485988 seconds time elapsed                                          (
```

Summary

- std::string::find is not optimized.
- We do need std::string_view feature to support in-place parsing.
- SSE2 and AVX2 version of strstr does significantly speedup the exact matching algorithms.

std::regex

```
struct RegexMatcher {
  RegexMatcher(const std::string &patt) : search_pattern(patt) {}
  bool is_matched(const char *begin, const size_t len) {
    return std::regex_search(begin, begin + len, search_pattern);
  }
  std::regex search_pattern;
};
```

fgrep vs grep vs ripgrep vs ag

```
hungptit@hungptit ~/w/f/benchmark> ./grep_bench -g pattern1
Celero
Timer resolution: 0.001000 us
----------------------------------------------------------------------------------
Group           |   Experiment    |   Prob. Space   |     Samples     |   Iteratio
----------------------------------------------------------------------------------
pattern1        | gnu_grep        |            Null |              10 |
pattern1        | ag              |            Null |              10 |
pattern1        | ripgrep         |            Null |              10 |
pattern1        | fgrep           |            Null |              10 |
Complete.
```

Regular expression

- Regular Expression Matching Can Be Simple And Fast
- Comparison of regex engines.

A simple matcher policy using hyperscan

```
const bool is_matched(const char *data, const size_t len) {
  if (data == nullptr)
    return true;
  auto errcode =
      hs_scan(database, data, len, 0, scratch, event_handler, nullptr);
  if (errcode == HS_SUCCESS) {
    return false;
  } else if (errcode == HS_SCAN_TERMINATED) {
    return true;
  } else {
    throw std::runtime_error("Unable to scan the input buffer");
  }
}
```

fgrep vs grep vs ripgrep vs ag

```
hungptit@hungptit ~/w/f/benchmark> ./all_tests
Celero
Timer resolution: 0.001000 us
-----------------------------------------------------------------------------------
Group           |   Experiment    |   Prob. Space   |     Samples     |   Iteratio
-----------------------------------------------------------------------------------
mark_twain      | grep_brew       |            Null |               5 |
mark_twain      | ag              |            Null |               5 |
mark_twain      | ripgrep         |            Null |               5 |
mark_twain      | fgrep           |            Null |               5 |
Complete.
```

fgrep vs grep vs ripgrep vs ag

How to print out search results fast?

- iostream
- fprintf
- [fmt]

Demo

Useful tips

- std::string::find is not efficient.
- std::regex is very slow.
- iostream is very slow for reading files.
- boost::iostream is reasonable fast, however, it is still 50% slower then the low-level implementation.
- We do need to minimize memory copy when writing high performance code.
- The public interface does affect our code performance.

Conclusions

- fgrep's raw performance is comparable to that of ripgrep and GNU grep. From our benchmark the_silver_searcher is slower than grep and ripgrep.
- Generic programming paradigm is a big win. It helps to create reusable, flexible, and high performance algorithms.
- Creating efficient solutions using modern C++ is not a trivial task. We have demonstrated that a clean C++ solution using iostream and std::regex can be 100x slower than GNU grep or ripgrep commands.

Todo list

- Improve the usability of fgrep.
- Improve the performance of fgrep.
- Add more tests.

Acknowledgment

- SSE2/AVX2 code is the modified version of sse4-strstr
- My fast file reading algorithm ideas come from
    - Limere's blog post
    - GNU wc command
    - grep
    - ripgrep

Used libraries and tools

- Catch2
- boost
- STL
- fmt
- hyperscan
- cereal
- benchmark
- Celero
- utils
- ioutils
- CMake
- gcc
- clang
- perf
- strace
- ripgrep
- GNU grep

Q/A