@hankadusikova

Regular Expressions

Hana Dusíková

@hankadusikova

Who am I?

- I live in Brno, Czechia.
- I'm a researcher at Avast.
- Organizer of Avast C++ Prague Meetup.
- (soon) Czech national body to WG21.
- Amateur photographer ~~and occasional hiker~~.

Let's talk about Regular Expressions...

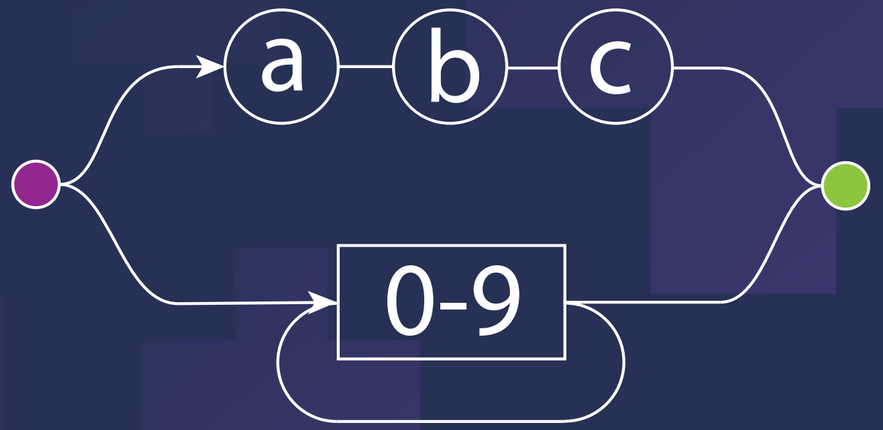
What can I do with a regular expression?*

- Match the subject against a pattern.
- Extract the content of the subject.
- Modify parts of the subject.

*Depending on algorithm used internally.

What does a regular expression look like*

abc|[0-9]+



*There are many slightly different dialects (Perl, ECMA, Posix).

How can I use regular expressions in C++?

```
std::regex re{"abc|[0-9]+"} // compile the RE only once;

bool match(std::string_view sv) {
    return std::regex_match(sv.begin(), sv.end(), re);
}
```

How can I use regular expressions in C?

```c
const unsigned char * p = "^[0-9]{4,16}?[aA]";

int err = 0;
size_t errOff = 0;

pcre2_code * re = pcre2_compile(p, PCRE2_ZERO_TERMINATED, 0, &err, &errOff, NULL);

int match(const char * str, size_t sz) {
    return pcre2_match(re, str, sz, 0, 0, NULL, NULL);
}

// TODO: don't forget to call pcre2_code_free at the end
```

What's going on in the constructor?

- Syntax parsing of the pattern.
- Building a data-structure (tree or table) representing the pattern.
- The calculations are very expensive (in comparison to matching alone).

How can I avoid calling the constructor?

You can write a finite state machine by yourself.

But that's hard to write properly.
You can easily introduce an error.

You can use an external utility to generate the parser.

Really? There is no other way?

There is...

Compile Time Regular Expressions

Hana Dusíková

@hankadusikova

What do I want?

- Work with REs as with any other code.
- Don't pay any runtime cost for parsing the pattern.
- Have REs as quick as possible with zero-overhead.

How do I think REs should be used in C++?

```
bool match(std::string_view sv) noexcept {
    return std::regex_match<"abc|[0-9]+">(sv);
}
```

This is possible with C++20's language support for Non Type Template Parameters.

The Compile Time Regular Expression library with C++20

```
bool match(std::string_view sv) noexcept {
    return ctre::match<"abc|[0-9]+">(sv);
}
```

Disabled for compilers without class NTTP support.

The Compile Time Regular Expression library with C++17*

```
bool match(std::string_view sv) noexcept {
    using namespace ctre::literals;
    return "abc|[0-9]+"_ctre.match(sv);
}
```

* with N3599 language extension (in GCC & Clang)

Example: Constexpr

```
static_assert("[0-9]+\\.[0-9]+"_ctre.match("123.456"sv));
```

Example: Constexpr

```
static_assert("[0-9]+\\.([0-9]+)"_ctre.match("123.456"sv).get<1>() == "456"sv);
```

Example: Extracting a name from a CSV-like input

```
struct name { std::string_view first, family; };

std::optional<name> extract(std::string_view sv) noexcept {
    using namespace ctre::literals;
    if (auto r = "([A-Za-z]++),([A-Za-z]++)"_ctre.match(sv)) {
        return name{ r.get<1>(), r.get<2>() };
    } else {
        return std::nullopt;
    }
}
```

Example: Extracting a date from a string

```
struct date { unsigned year, month, day; };

std::optional<date> extract(std::string_view sv) noexcept {
    using namespace ctre::literals;
    if (auto [re, y, m, d] = "([0-9]+)/([0-9]{1,2})/([0-9]{1,2})"_ctre.match(sv); re) {
        return date{ conv(y), conv(m), conv(d) };
    } else {
        return std::nullopt;
    }
}
```

But this must generate horrible assembly!

Not really :)

Optimized assembly is just 64 LoC long!

x86-64 gcc 7.3 | -std=c++17 -O3

11010 | .LX0: | .text | // | \s+ | Intel | Demangle | A | Libraries

```
1  extract(std::basic_string_view<char, std::char_traits<char> >):
2          add     rsi, rdx
3          mov     rax, rdi
4          cmp     rdx, rsi
5          je      .L2
6          movzx   edi, BYTE PTR [rdx]
7          lea     ecx, [rdi-97]
8          cmp     cl, 25
9          jbe     .L19
10 .L2:
11         mov     BYTE PTR [rax+32], 0
12         ret
13 .L19:
14         lea     rcx, [rdx+1]
15         cmp     rsi, rcx
16         je      .L2
17         movzx   r8d, BYTE PTR [rdx+1]
18         lea     edi, [r8-97]
19         cmp     dil, 25
20         ja      .L4
21 .L3:
22         add     rcx, 1
23         cmp     rsi, rcx
24         je      .L2
25         movzx   r8d, BYTE PTR [rcx]
26         lea     edi, [r8-97]
27         cmp     dil, 25
28         jbe     .L3
29 .L4:
30         cmp     r8b, 44
31         jne     .L2
32         lea     r9, [rcx+1]
33         cmp     rsi, r9
34         je      .L2
35         movzx   edi, BYTE PTR [rcx+1]
36         sub     edi, 97
37         cmp     dil, 25
38         ja      .L2
39         lea     rdi, [rcx+2]
40         cmp     rsi, rdi
41         je      .L6
42         movzx   r11d, BYTE PTR [rcx+2]
43         lea     r8d, [r11-97]
44         cmp     r8b, 25
45         ja      .L2
```

(In theory) The optimizer should be able to see the same intent with std::regex too...

Unfortunately it doesn't.

The equivalent code is ~18.7 kLoC.

x86-64 gcc 7.3 | -std=c++17 -O3 -c

11010 | .LX0: | .text | // | \s+ | Intel | Demangle | A | Libraries

```
std::ctype<char>::do_widen(char) const:
        mov     eax, esi
        ret
std::ctype<char>::do_narrow(char, char) const:
        mov     eax, esi
        ret
std::_Function_base::_Base_manager<std::__detail::_AnyMatcher<std::__cxx11::regex_traits<char>, false, false, false> >::_M_manager(std::_Any_data&, std::_Any_data const&, std::_Manager_oper
        test    edx, edx
        je      .L6
        cmp     edx, 1
        jne     .L5
        mov     QWORD PTR [rdi], rsi
.L5:
        xor     eax, eax
        ret
.L6:
        mov     QWORD PTR [rdi], OFFSET FLAT:typeinfo for std::__detail::_AnyMatcher<std::__cxx11::regex_traits<char>, false, false, false>
        xor     eax, eax
        ret
std::_Function_base::_Base_manager<std::__detail::_AnyMatcher<std::__cxx11::regex_traits<char>, false, false, true> >::_M_manager(std::_Any_data&, std::_Any_data const&, std::_Manager_opera
        cmp     edx, 1
        je      .L10
        jb      .L11
        cmp     edx, 2
        jne     .L9
        mov     rax, QWORD PTR [rsi]
        mov     QWORD PTR [rdi], rax
.L9:
        xor     eax, eax
        ret
.L11:
        mov     QWORD PTR [rdi], OFFSET FLAT:typeinfo for std::__detail::_AnyMatcher<std::__cxx11::regex_traits<char>, false, false, true>
        xor     eax, eax
        ret
.L10:
        mov     QWORD PTR [rdi], rsi
        xor     eax, eax
        ret
std::_Function_base::_Base_manager<std::__detail::_AnyMatcher<std::__cxx11::regex_traits<char>, false, true, false> >::_M_manager(std::_Any_data&, std::_Any_data const&, std::_Manager_opera
        cmp     edx, 1
        je      .L15
        jb      .L16
        cmp     edx, 2
        jne     .L14
        mov     rax, QWORD PTR [rsi]
```

What about errors?

```
return ")abc+"_ctre.match(sv);
```

```
In file included from example.cpp:1:
In file included from include/ctre.hpp:4:
ctre/literals.hpp:63:2: error: static_assert failed due to requirement 'correct()'
"Regular Expression contains syntax error."
        static_assert(correct(), "Regular Expression syntax error.");
        ^             ~~~~~~~
example.cpp:5:18: note: in instantiation of function template specialization
'ctre::literals::operator""_ctre<char, ')', 'a', 'b', 'c'>' requested here
    return ")abc"_ctre.match(sv);
```

Compile Time Regular Expression library

- is easy to use,
- uses existing and well known RE syntax,
- is perfect for checking inputs of your program,
- emits nice assembly,
- and we will talk later about performance...

The Parser

Disclaimer: all examples were simplified.

How does it work?

- It uses a generic LL(1) parser for converting a pattern into a type.
- The parser uses a provided PCRE compatible grammar.
- Output type encodes the pattern as an expression template.
- Evaluation of the expression template matches the subject.

How does the LL(1) parser work?

- Starts with a start symbol on the stack.
- On every step it pops one symbol from the stack and checks the current character at the input.
- Based on the pair of symbol and character it decides to:
  - push a string of symbols to the stack,
  - pop a character from the input,
  - or reject.
- Repeat until the stack and input are empty then accept.

What does a LL(1) grammar look like?

f(*symbol*,*char*) →

	(	)	*	+	?	\	a	d	|	other	ϵ
→ ***S***	(*alt0*) *mod seq alt*					\ *esc mod seq alt*	a *mod seq alt*	d *mod seq alt*		other *mod seq alt*	ϵ
alt0	(*alt0*) *mod seq alt*					\ *esc mod seq alt*	a *mod seq alt*	d *mod seq alt*		other *mod seq alt*	
alt		ϵ							| *seq0 alt*		ϵ
esc							a	d			
mod	ϵ	ϵ	*	+	?	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
seq0	(*alt0*) *mod seq*					\ *esc mod seq*	a *mod seq*	d *mod seq*		other *mod seq*	
seq	(*alt0*) *mod seq*	ϵ				\ *esc mod seq*	a *mod seq*	d *mod seq*	ϵ	other *mod seq*	ϵ
(	pop										
)		pop									
*			pop								
+				pop							
?					pop						
\						pop					
a							pop				
d								pop			
|									pop		
other										pop	
Z_0											accept

How can I write it in C++?

There are many ways to encode it,
but not many for the compile-time parsing.

How can I represent symbols in C++?

```
struct my_grammar {
    struct S {};
    struct alt0 {};
    struct alt {};
    struct esc {};
    struct mod {};
    struct seq0 {};
    struct seq {};

    using start_symbol = S;

    // ...
}
```

Just a bunch of empty types.

How can I represent an LL(1) table in C++?

f (*symbol*, *character*) → (...)

f (*symbol*, *symbol*) → pop_input

f (*symbol*, *character*) → reject

f (Z_0, ϵ) → accept

```
struct my_grammar {
 //...
 auto f (symbol, term<'character'>) -> list{...};
 template <auto S> auto f (term<S>, term<S>) -> pop_input;
 auto f (...) -> reject;
 auto f (empty_stack, epsilon) -> accept;
};
```

The decision of which step of the LL(1) table to use is based on the function overload resolution of the C++ language. Return type is the value. There is no need for a function body.

Why is the grammar inside a struct?

```
template <typename Grammar, ...> struct parser {
    //...
    auto next_move = Grammar::f(top(stack), get_character<Pos>());
    //...
}
```

It allows me to pass the grammar as an argument to the parser.

How do I implement the stack?

Just use (simplest) type list and a set of a few free functions:

```
template <typename... Ts> struct list { };


template <typename... Ts, typename... As>
 constexpr auto push(list<Ts...>, As...) -> list<As..., Ts...>;

template <typename T, typename... As>
 constexpr auto pop(list<T, Ts...>) -> list<Ts...>;

template <typename T, typename... Ts>
 constexpr auto top(list<T, Ts...>) -> T;

struct empty_stack { };

constexpr auto top(list<>) -> empty_stack;
```

Compile-time programming starts to be easy when you think functionally.

How do I implement the input string?

```
template <typename CharT, size_t N> struct fixed_string {
    std::array<CharT, N> data;

    // constexpr constructor from const char[N]

    constexpr auto operator[](size_t i) const noexcept { return data[i]; }
    constexpr size_t size() const noexcept { return N; }
};

template <typename CharT, size_t N>
 fixed_string(const CharT[N]) -> fixed_string<CharT, N>;

// more info about class NTTP in p0732 by Jeff Snyder and Louis Dionne
```

std::fixed_string will (probably) be part of C++20.

How do I use the input string?

```
template <..., fixed_string Str> struct parser {
    // ...
    template <size_t Pos> constexpr auto get_character() const noexcept() {
        if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
        else return epsilon{};
    }
    // ...
};
```

Formally, the symbol on input is either terminal or epsilon.

How does it fit together?

```
constexpr bool ok = parser<pcre_grammar, "^abc">::correct;
```

Think functionally, your tools are:
function overloading resolution and (tail) recursion...

```
template <typename Grammar, fixed_string Str> struct parser {
    static constexpr bool correct = parse(list<Grammar::start_symbol>());

    // prepare each step and move to next
    template <size_t Pos = 0, typename S> static constexpr bool parse(S stack) {
        // reminder: all these types are empty structs
        // and are all known during compile time

        using next_stack = decltype(  pop(stack)                          );
        using symbol     = decltype(  top(stack)                          );
        using current    = decltype(  get_character<Pos>()                );
        using next_move  = decltype(  Grammar::f(symbol{}, current{})     );
        return move<Pos>(next_move{}, next_stack{});
    }
    //...
```

```
// pushing something to stack (epsilon case included)
template <size_t Pos, typename... Push, typename Stack>
 static constexpr bool move(list<Push...>, Stack stack) {
    using next_stack = decltype(  push(stack, Push{}...)  )
    return parse<Pos>(next_stack{});
}

// move to next character
template <size_t Pos, typename Stack>
 static constexpr bool move(pop_input, Stack stack) {
    return parse<Pos+1>(stack);
}

//...
```

```
template <size_t Pos, typename Stack>
  static constexpr bool move(reject, Stack) {
    return false;
}

template <size_t Pos, typename Stack>
  static constexpr bool move(accept, Stack) {
    return true;
}

}; // end of Parser struct
```

But this is returning a boolean value! Is there something missing?

Yes!

We need to build
an expression template as the output of the parser.

Building
The Expression Template

How can I build a type from a string?

Use the same parsing algorithm with an added argument
and new type of special symbols (called semantic actions)
in a grammar to modify the argument.

If you want to do calculation
the best type of argument is a type list (again).

Where are the semantic actions placed?

	(	)	*	+	?	\	a	d	|	other	ϵ
→ ***S***	(*alt0*) *mod seq alt*					\ *esc mod seq alt*	a **char** *mod seq alt*	d **char** *mod seq alt*		other **char** *mod seq alt*	ϵ
alt0	(*alt0*) *mod seq alt*					\ *esc mod seq alt*	a **char** *mod seq alt*	d **char** *mod seq alt*		other **char** *mod seq alt*	
alt		ϵ							| *seq0* **alt** *alt*		ϵ
esc							a **alpha**	d **digit**			
mod	ϵ	ϵ	* **star**	+ **plus**	? **opt**	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
seq0	(*alt0*) *mod seq*					\ *esc mod seq*	a **char** *mod seq*	d **char** *mod seq*		other **char** *mod seq*	
seq	(*alt0*) *mod* **seq** *seq*	ϵ				\ *esc mod* **seq** *seq*	a **char** *mod* **seq** *seq*	d **char** *mod* **seq** *seq*	ϵ	other **char** *mod* **seq** *seq*	ϵ
(	pop										
)		pop									
*			pop								
+				pop							
?					pop						
\						pop					
a							pop				
d								pop			
|									pop		
other										pop	
Z_0											accept

What do the SemanticAction symbols look like?

```
struct my_grammar {
    // ...
    struct _char: action {};
    struct alpha: action {};
    struct digit: action {};
    struct seq: action {};
    struct star: action {};
    struct plus: action {};
    struct opt: action {};
    // ...
}
```

What are the changes in the parser?

```
template <typename Grammar, fixed_string Str> struct parser {

    template <size_t Pos = 0, typename S, typename T = list<>>
     static constexpr auto parse(S stack, T subject = {}) {
        using next_stack = decltype(  pop(stack)  );
        using symbol     = decltype(  top(stack)  );
        if constexpr (SemanticAction<symbol>) {
            using previous = decltype(  prev_character<Pos>()  );
            using next_subject = decltype( modify(symbol{}, previous{}, subject) );
            return parse<Pos>(next_stack(), next_subject{});
        } else {
            using current    = decltype(  get_character<Pos>()              );
            using next_move  = decltype(  Grammar::f(symbol{}, current{})  );
            return move<Pos>(next_move{}, next_stack{}, subject);
        }
```

```
// pushing something to stack (epsilon case included)
template <size_t Pos, typename... Push, typename Stack, typename T>
 static constexpr auto move(list<Push...>, Stack stack, T subject) {
    using next_stack = decltype(  push(stack, Push{}...)  )
    return parse<Pos>(next_stack{}, subject);
}

// move to next character
template <size_t Pos, typename Stack, typename T>
 static constexpr auto move(pop_input, Stack stack, T subject) {
    return parse<Pos+1>(stack, subject);
}

//...
```

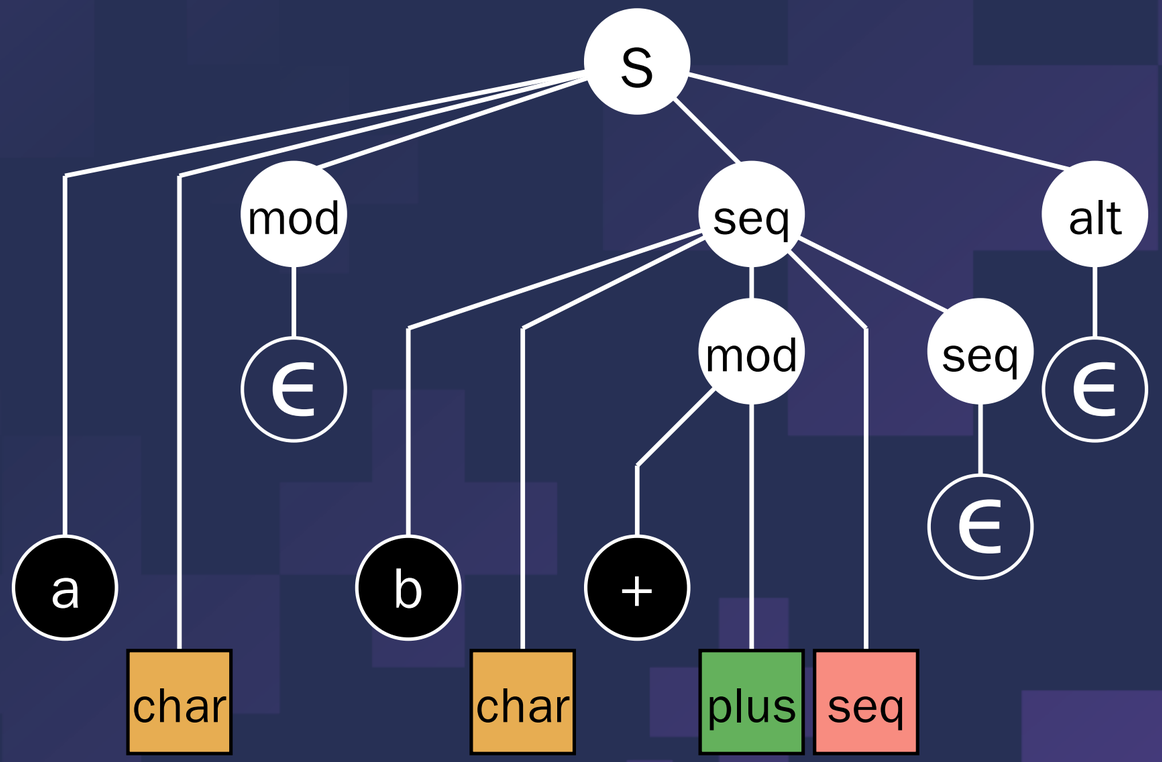
```
template <size_t Pos, typename Stack, typename T>
 static constexpr auto move(reject, Stack, T subject) {
    return pair{false, subject};
}

template <size_t Pos, typename Stack, typename T>
 static constexpr auto move(accept, Stack, T subject) {
    return pair{true, subject};
}

}; // end of Parser struct
```

What does building from a string look like?

ab+




```
seq<ch<'a'>,plus<ch<'b'>>>
```

What about the modify function?

Pushing character

```
template <auto C, typename... Ts>
 auto modify(my_grammar::_char, term<C>, list<Ts...>) -> list<ch<C>, Ts...>;
```

(Notice the term and ch types.)

Making something optional

```
template <auto C, typename Opt, typename... Ts>
 auto modify(my_grammar::opt, term<C>, list<Opt, Ts...>)
  -> list<opt<Opt>, Ts...>;
```

Making cycles

```
template <auto C, typename Plus, typename... Ts>
 auto modify(my_grammar::plus, term<C>, list<Plus, Ts...>)
  -> list<plus<Plus>, Ts...>;
```

```
template <auto C, typename Star, typename... Ts>
 auto modify(my_grammar::star, term<C>, list<Star, Ts...>)
  -> list<star<Star>, Ts...>;
```

Creating a sequence

```
template <auto C, typename A, typename B, typename... Ts>
 auto modify(my_grammar::seq, term<C>, list<B, A, Ts...>) -> list<seq<A, B>, Ts...>;
```

(Notice the switched order of A & B on the stack.)

Adding to a sequence

```
template <auto C, typename... As, typename B, typename... Ts>
 auto modify(my_grammar::seq, term<C>, list<B, seq<As...>, Ts...>)
  -> list<seq<As..., B>, Ts...>;
```

Making an alternate

```
template <auto C, typename A, typename B, typename... Ts>
 auto modify(my_grammar::alt, term<C>, list<B, A, Ts...>) -> list<alt<A, B>, Ts...>;
```

(Notice the switched order of A & B on the stack.)

Adding to an alternate

```
template <auto C, typename... As, typename B, typename... Ts>
 auto modify(my_grammar::alt, term<C>, list<B, alt<As...>, Ts...>)
  -> list<alt<As..., B>, Ts...>;
```

Adding a digit or alpha class

```
template <auto C, typename... Ts>
 auto modify(my_grammar::alpha, term<C>, list<Ts...>) -> list<alpha, Ts...>;
```

```
template <auto C, typename... Ts>
 auto modify(my_grammar::digit, term<C>, list<Ts...>) -> list<digit, Ts...>;
```

We have the expression template. What's next?

```
static_assert(std::is_same_v<
     seq<ch<'a'>,plus<ch<'b'>>>,
     parser<my_grammar, "ab+">::output_type
>);
```

Evaluating
The Expression Template

How are REs matched against a string?

```
template <fixed_string re> bool match(std::string_view sv) {
    static_assert(parser<my_grammar, re>::correct);
    using RE = parser<my_grammar, re>::output_type;

    return match(sv.begin(), sv.begin(), sv.end(), list<RE>{}).success;
}
```

```
match<"[a-z]\\d+">("a42"sv);
```

List? Again?!

Yes!

But first what's the output type of match(...)?

```
template <ForwardIterator It> struct pair {
    ForwardIterator it;
    bool success;
    // constexpr constructor etc...
};
```

How do we know it's the end?

```
// and at the end... we need to check for the end :)
template <ForwardIterator It>
 bool match(It begin, It it, It end, list<>{}) {
    // if we are at the end input we should accept
    return {it, it == end};
}
```

(Helper for finishing the regex matching.)

Matching a character

```
template <ForwardIterator It, auto C, typename... Ts>
 pair<It> match(It begin, It it, It end, list<ch<C>, Ts...>) {

    if (it == end) return {it, false};
    if (*it != C) return {it, false};

    return match(begin, std::next(it), end, list<Ts...>{});
}
```

Matching an alpha or a digit class

```
template <ForwardIterator It, typename... Ts>
 pair<It> match(It begin, It it, It end, list<digit, Ts...>) {

    if (it == end) return {it, false};
    if (!(*it >= '0' && *it <= '9')) return {it, false};

    return match(begin, std::next(it), end, list<Ts...>{});
}
```

```
template <ForwardIterator It, typename... Ts>
 pair<It> match(It begin, It it, It end, list<alpha, Ts...>) {

    if (it == end) return {it, false};
    if (!(*it >= 'a' && *it <= 'z')) return {it, false};
    return match(begin, std::next(it), end, list<Ts...>{});
}
```

Matching a sequence

```
template <ForwardIterator It, typename... Seq, typename... Ts>
 pair<It> match(It begin, It it, It end, list<seq<Seq...>, Ts...>) {

    return match(begin, it, end, list<Seq..., Ts...>{});
}
```

Matching an optional

```
template <ForwardIterator It, typename... Opt, typename... Ts>
 pair<It> match(It begin, It it, It end, list<opt<Opt...>, Ts...>) {

    if (auto out = match(begin, it, end, list<Opt..., Ts...>{})) {
        return out;
    } else {
        // try it without the content of opt<...>
        return match(begin, it, end, list<Ts...>{}));
    }
}
```

Matching an alternate

```
template <ForwardIterator It, typename Head, typename... Tail, typename... Ts>
 pair<It> match(It begin, It it, It end, list<alt<Head, Tail...>, Ts...>) {
    if (auto out = match(begin, it, end, list<Head, Ts...>{})) {
        return out;
    } else {
        // try the next one
        return match(begin, it, end, list<alt<Tail...>, Ts...>{}));
    }
}

template <ForwardIterator It, typename... Ts>
 pair<It> match(It begin, It it, It end, list<alt<>, Ts...>) {
    // no option from alternation was successful
    return {it, false};
}
```

Matching a plus cycle

```
template <ForwardIterator It, typename First, typename... Alt, typename... Ts>
 pair<It> match(It begin, It it, It end, list<plus<Plus...>, Ts...>) {
    for (;;) {
        if (auto inner = match(begin, it, end, list<Plus..., end_of_cycle>{})) {
            if (auto out = match(begin, it, end, list<Ts...>{})) {
                return out;
            } else {
                it = inner.it;
            }
        } else return {it, false};
    }
}
```

```
struct end_of_cycle {};

template <ForwardIterator It>
 pair<It> match(It, It it, It, list<end_of_cycle>) {
    return {it, true};
}
```

(The cycle is lazy.)

Matching a star cycle

```
template <ForwardIterator It, typename First, typename... Alt, typename... Ts>
 pair<It> match(It begin, It it, It end, list<star<Star...>, Ts...>) {
    for (;;) {
        if (auto out = match(begin, it, end, list<Ts...>{})) {
            return out;
        } else if (auto inner = match(begin, it, end, list<Star..., end_of_cycle>{})) {
            if (auto out = match(begin, it, end, list<Ts...>{})) {
                return out;
            } else {
                it = inner.it;
            }
        } else return {it, false};
    }
}
```

(The cycle is lazy.)

In the examples there are no captures or any advanced RE functionality.

Benchmarks

How quick or slow is this thing?

Measured code with CTRE

```
int main (int argc, char ** argv) {
    using namespace ctre::literals;
    constexpr auto re = "PATTERN"_ctre;
    std::ifstream stream{argv[1], std::ifstream::in};
    std::string line;
    while (std::getline(stream, line)) {
        if (re.search(line)) {
            std::cout << line << '\n';
        }
    }
}
```

Measured code with std::regex

```
int main (int argc, char ** argv) {
    std::regex re("PATTERN");
    std::ifstream stream{argv[1], std::ifstream::in};
    std::string line;
    while (std::getline(stream, line)) {
        if (std::regex_search(line, re)) {
            std::cout << line << '\n';
        }
    }
}
```

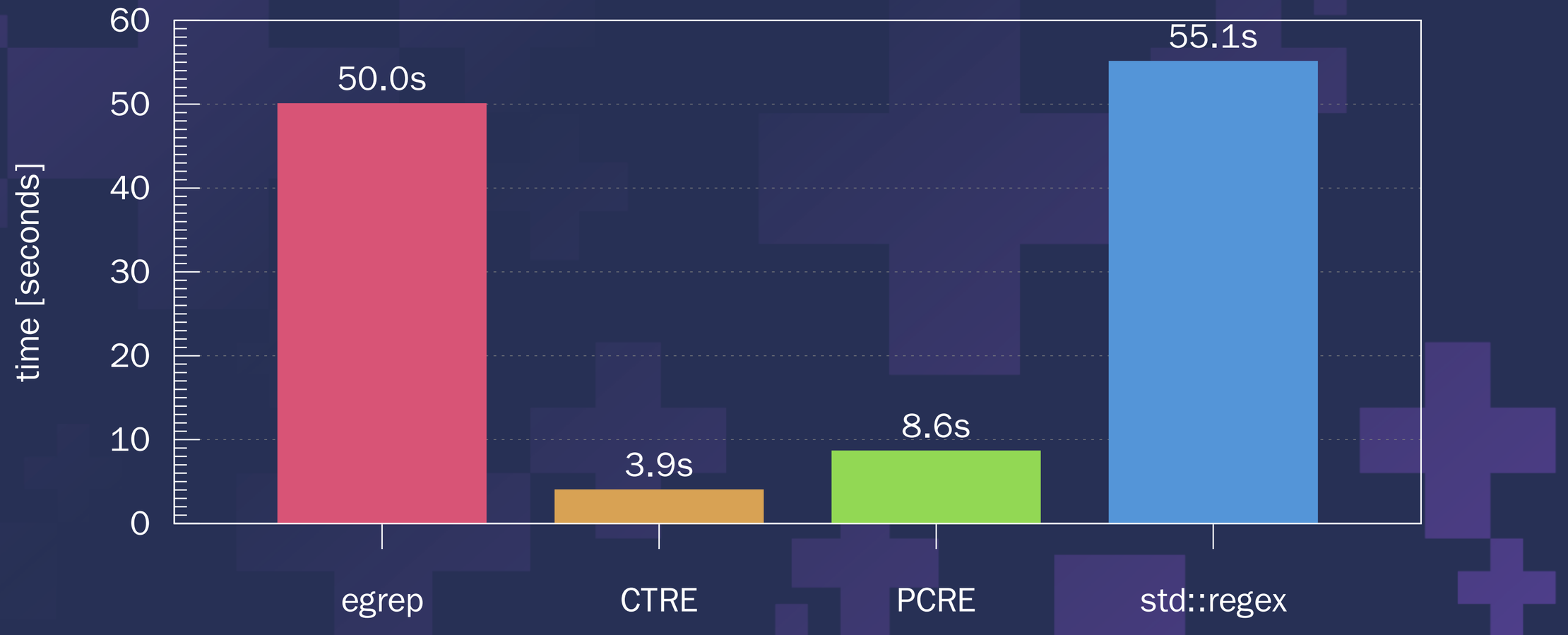
Measured code with PCRE2

```
int main (int argc, char ** argv) {
    auto * pattern = "PATTERN";
    pcre2_code * re = pcre2_compile(pattern, PCRE2_ZERO_TERMINATED, 0,
      &errornumber, &erroroffset, nullptr);

    std::ifstream stream{argv[1], std::ifstream::in};
    std::string line;
    while (std::getline(stream, line)) {
        if (pcre2_match(re, line.c_str(), line.length(), 0, 0, NULL, NULL) >= 0) {
            std::cout << line << '\n';
        }
    }
}
```

Runtime Matching (GCC): `ABCD|DEFGH|EFGHI|A{4,}`

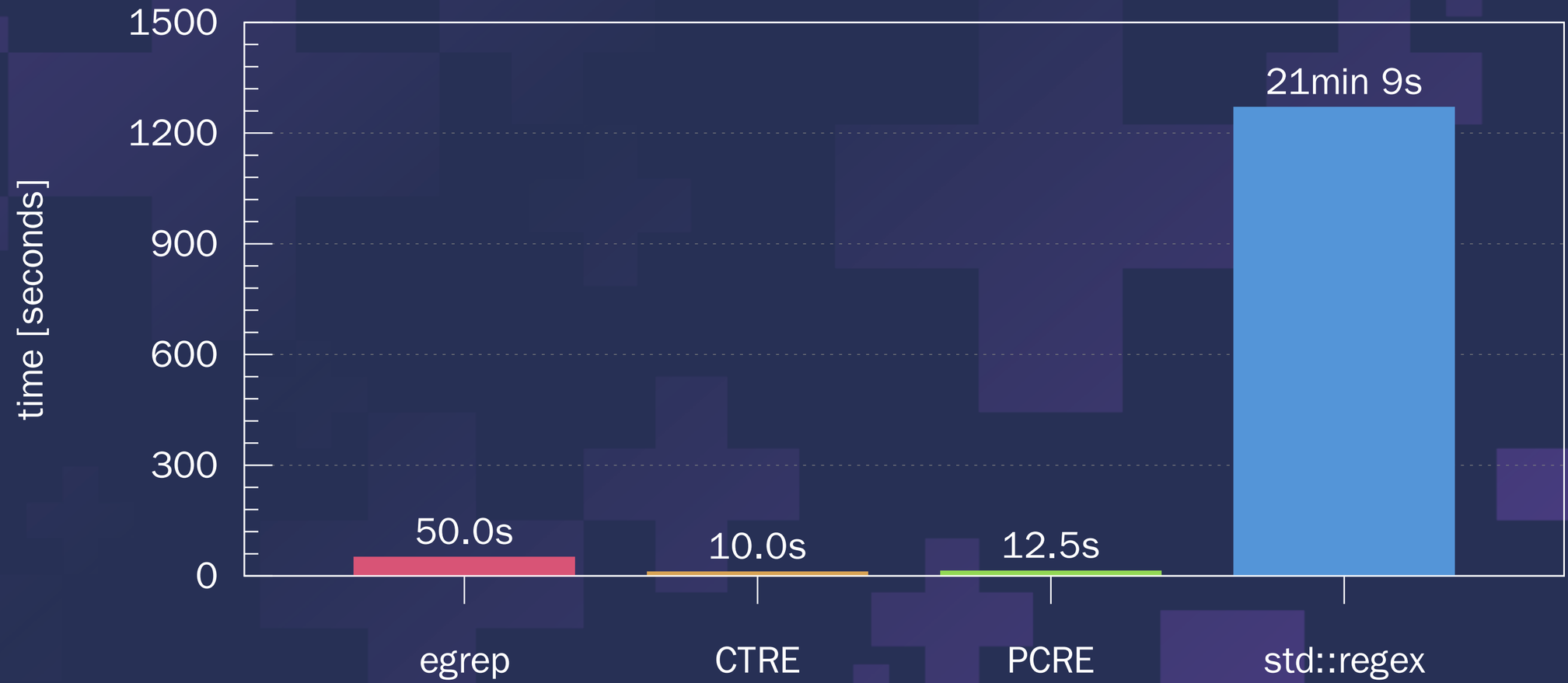
Matching pattern against Big CSV file (1.3 GiB, 6.5 MLoC).



(less is better)

Runtime Matching (Clang): `ABCD|DEFGH|EFGHI|A{4,}`

Matching pattern against Big CSV file (1.3 GiB, 6.5 MLoC).

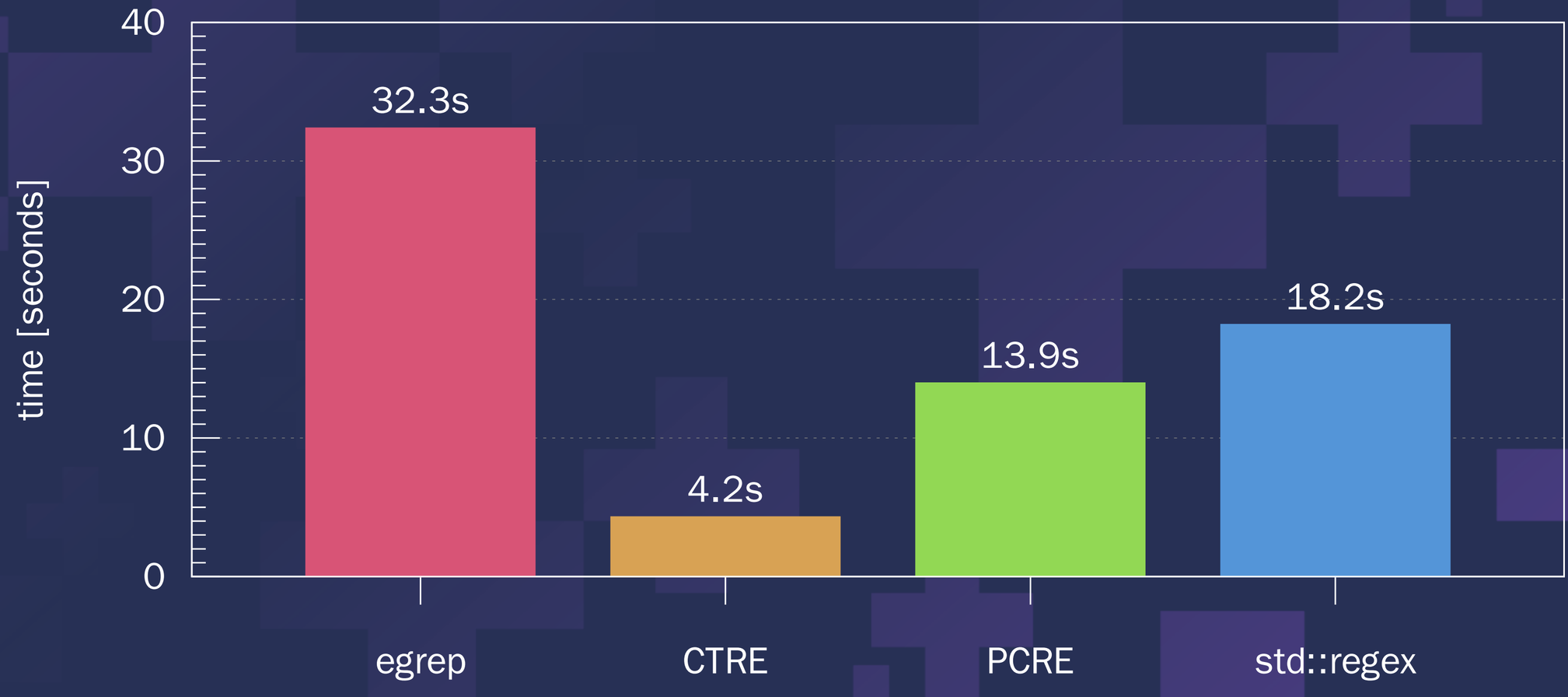


(less is better)

Looks like std::regex in libc++ is slow...

Runtime Matching (GCC): `[0-9a-fA-F]{8,16}`

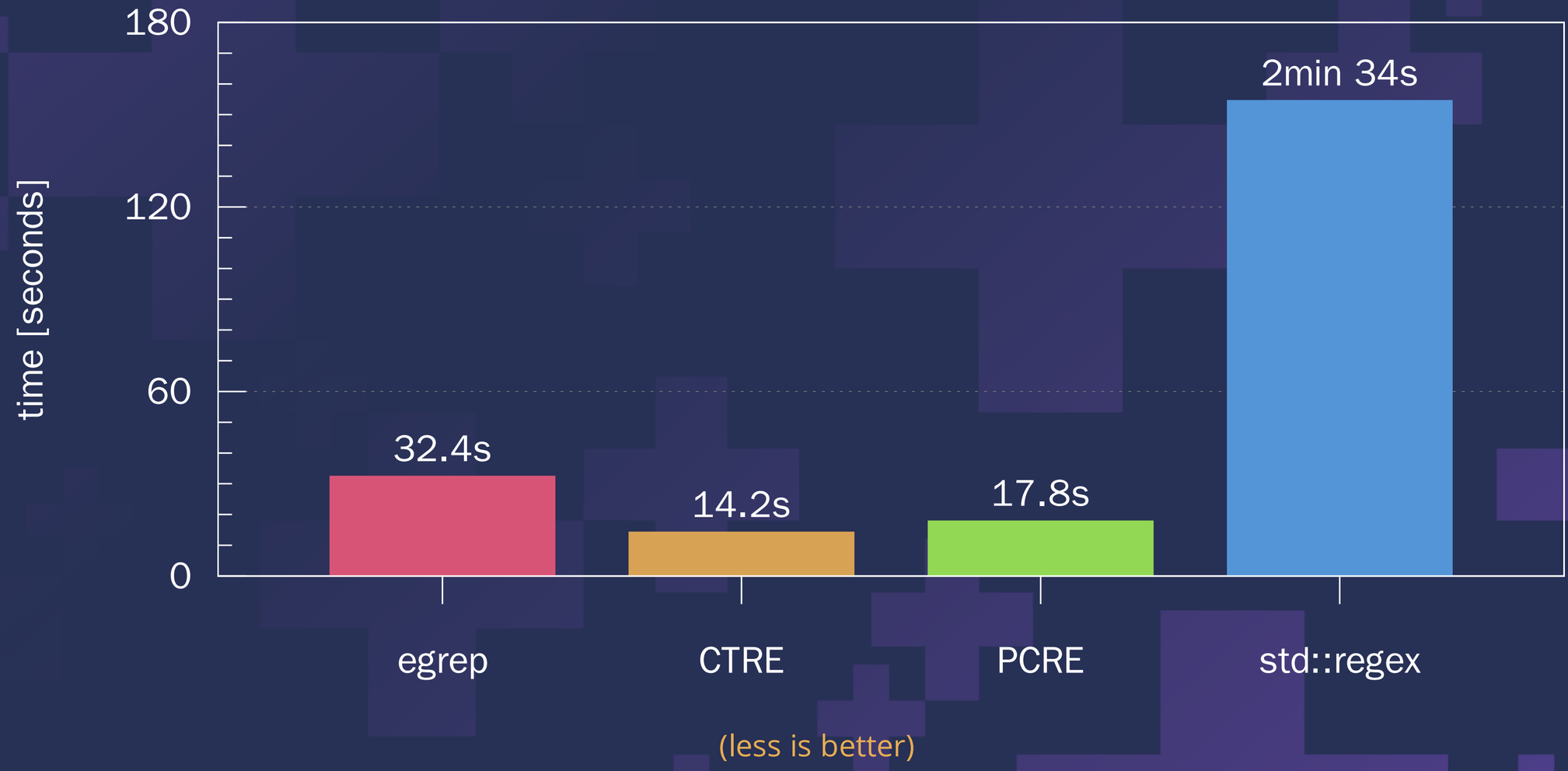
Matching pattern against Big CSV file (1.3 GiB, 6.5 MLoC).



(less is better)

Runtime Matching (Clang): `[0-9a-fA-F]{8,16}`

Matching pattern against Big CSV file (1.3 GiB, 6.5 MLoC).

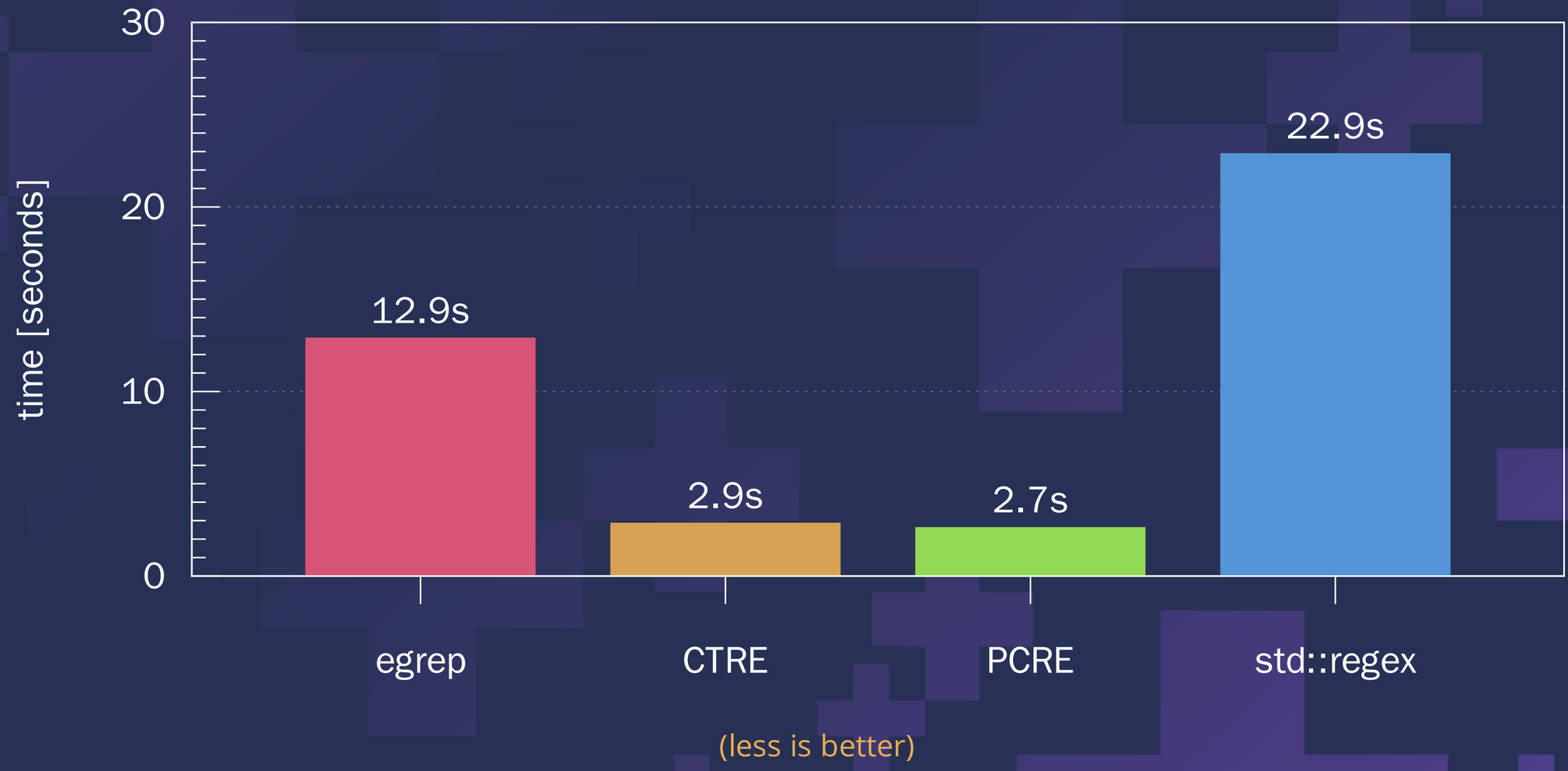


(less is better)

So the std::regex in libc++ is consistently bad.

Runtime Matching (GCC): `([0-9]{4,16})?[aA]`

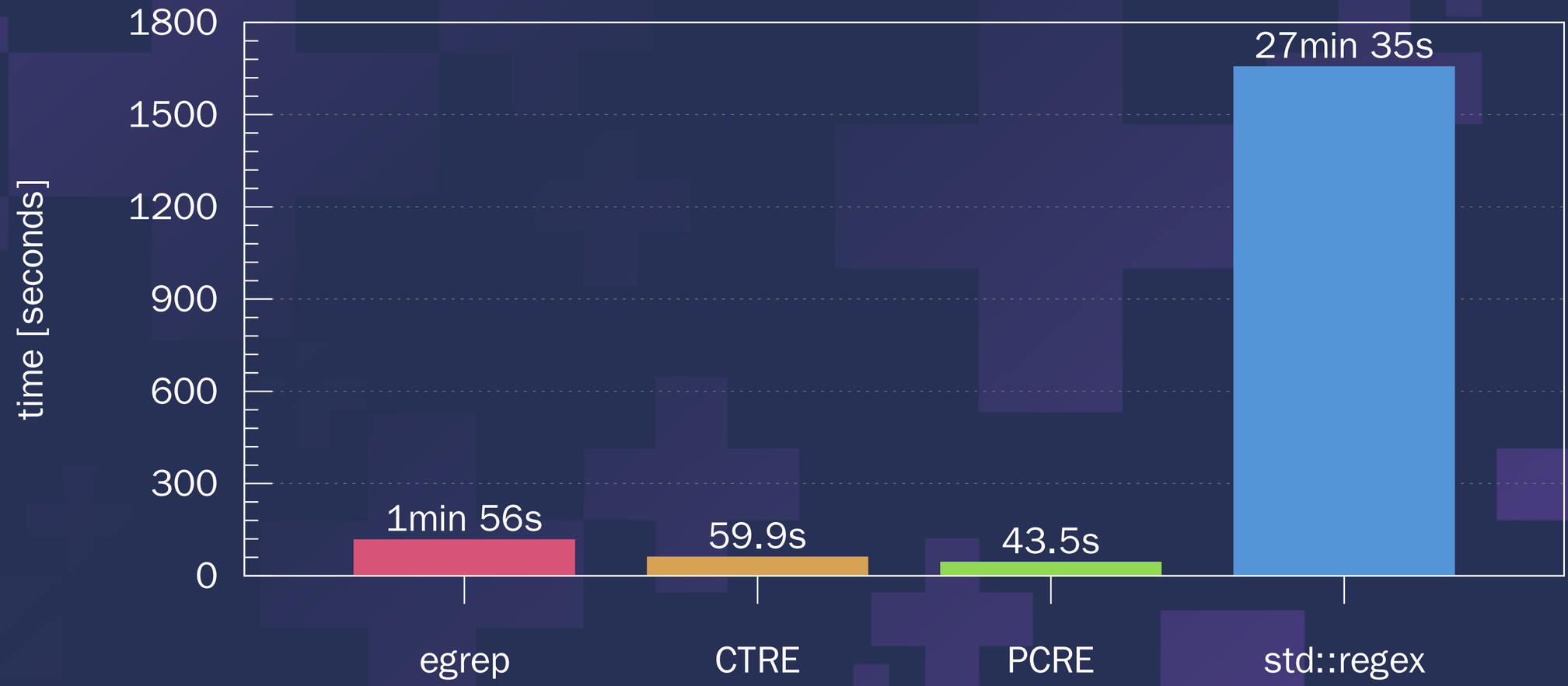
Matching pattern against Big CSV file (1.3 GiB, 6.5 MLoC).



(less is better)

Runtime Matching (Clang): `([0-9]{4,16})?[aA]`

Matching pattern against Big CSV file (1.3 GiB, 6.5 MLoC).



(less is better)

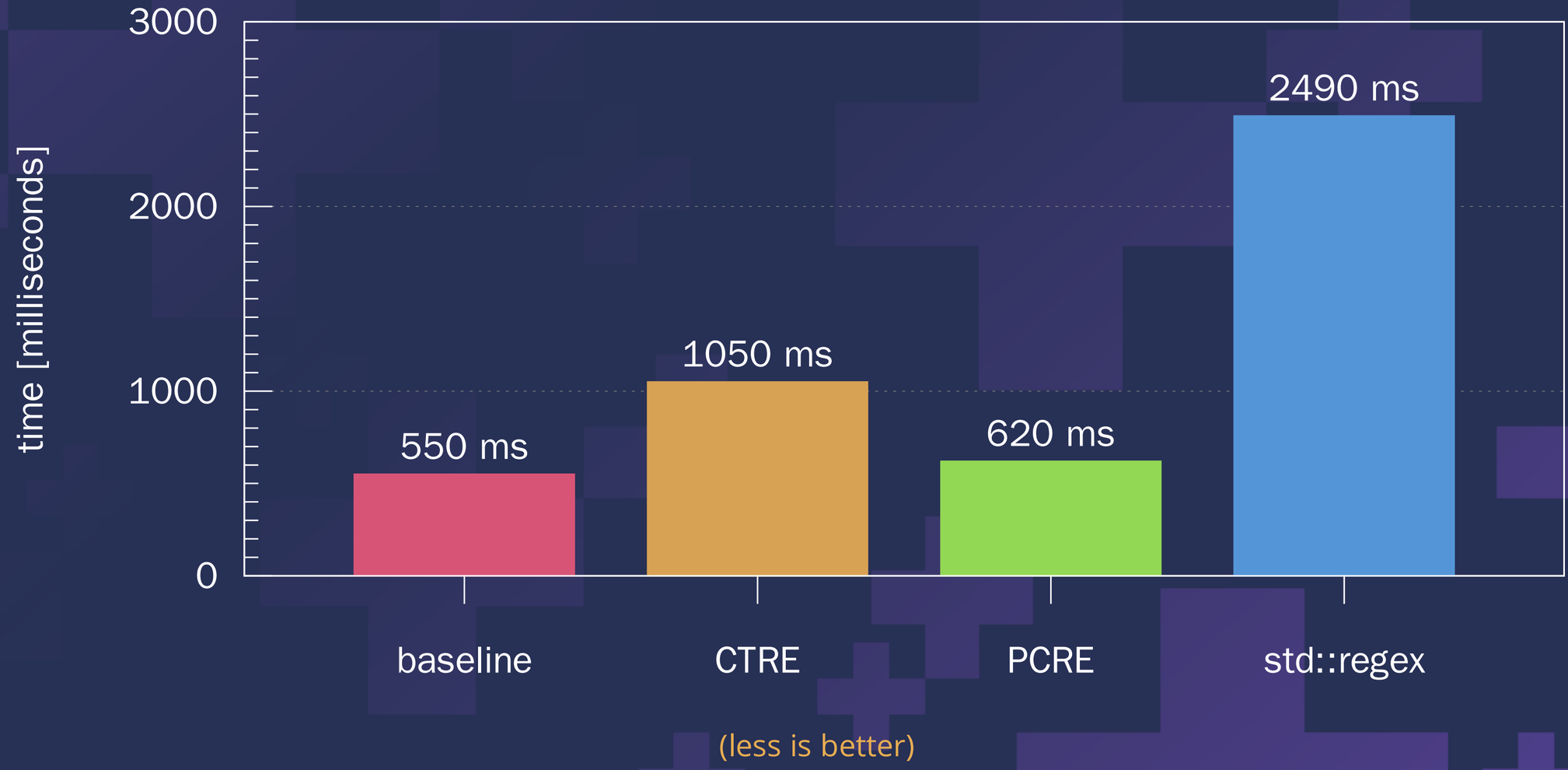
As I was benchmarking
I found something interesting...

Compile Time Benchmark

- All tests include the same set of headers.
- I measured compilation time only, without linking.

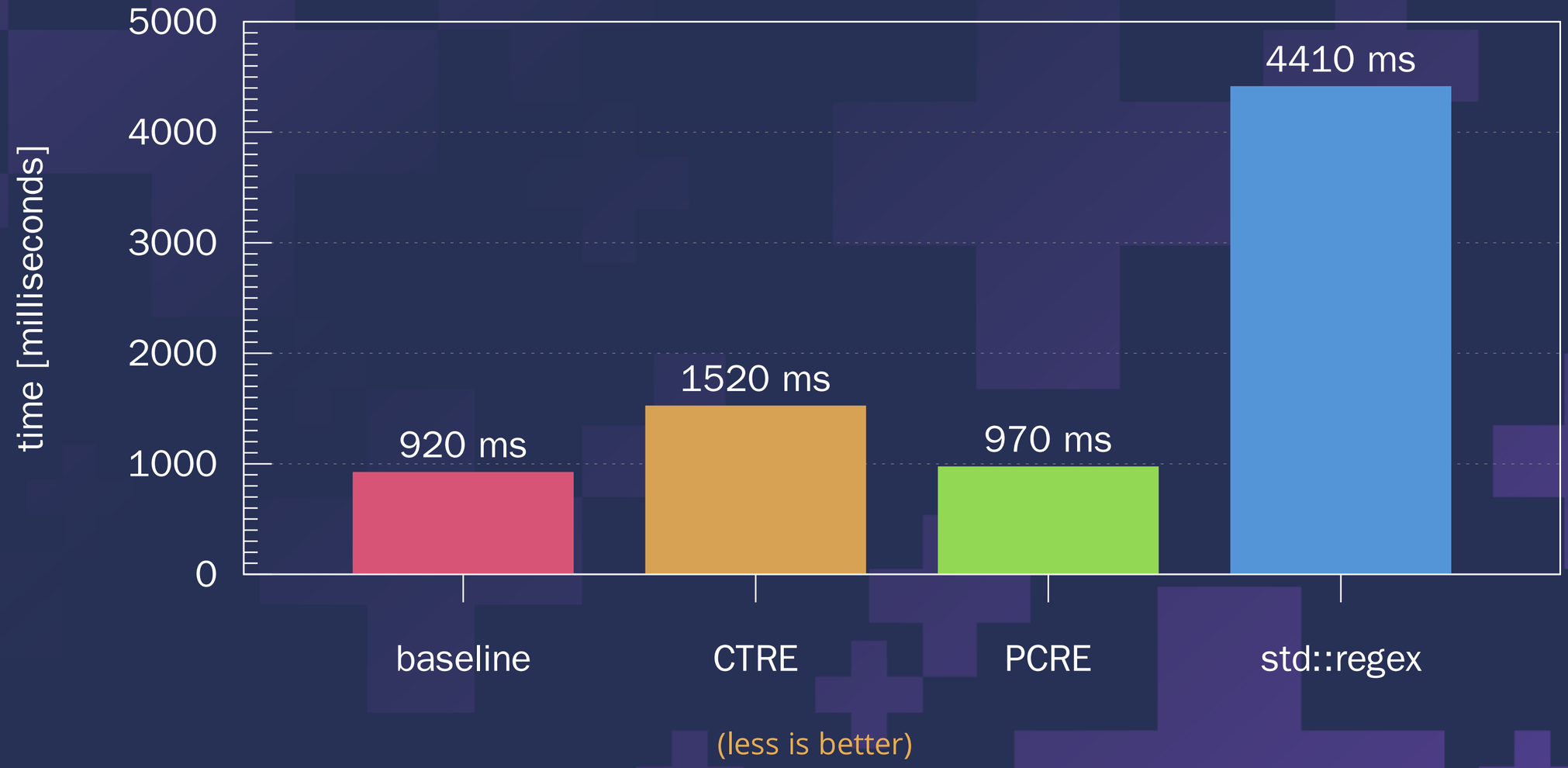
Optimized Compile Time of The Benchmark code.

`clang -c -O3`



Optimized Compile Time of The Benchmark code.

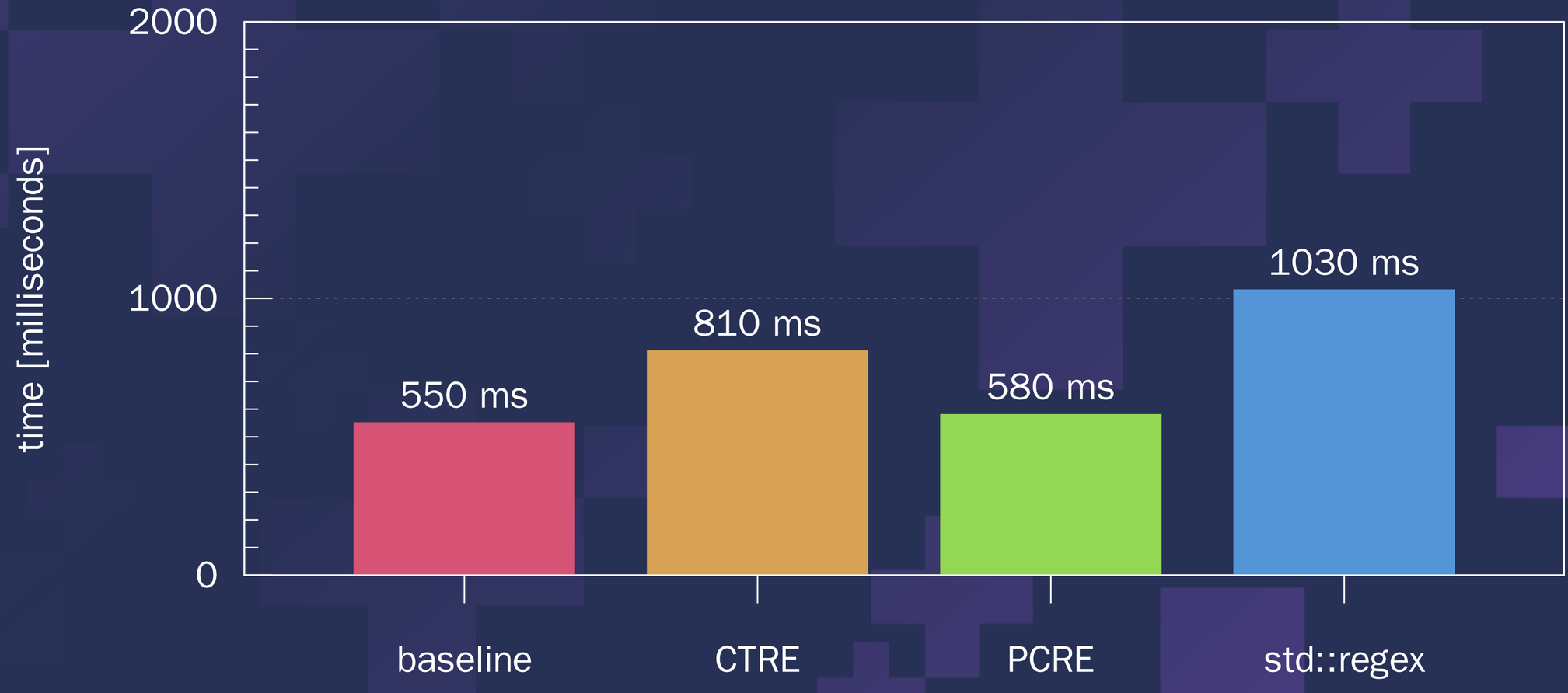
`gcc -c -O3`



(less is better)

Compile Time of The Benchmark code.

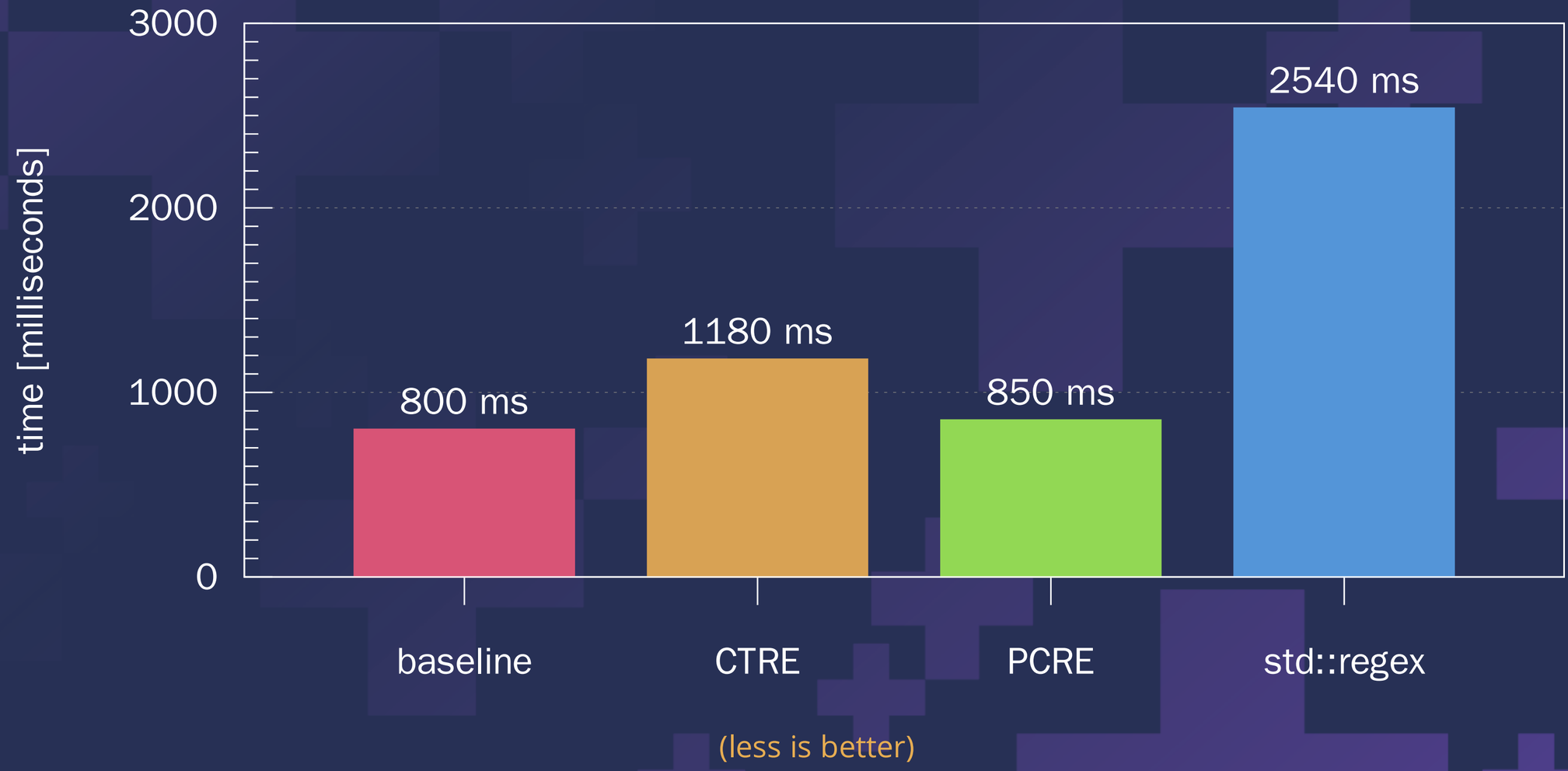
`clang -c -O0`



(less is better)

Compile Time of The Benchmark code.

`gcc -c -O0`



(less is better)

What's the behaviour of the library?

- Multiple different REs cost ~200ms each.
- Repeating the same REs cost virtually nothing (due to caching).
- Cost of one RE parsing is linear with its complexity.

What did we learn?

- Recursion and function overloading resolution are powerful tools.
- Pure functional programming keeps your code simple.
- Trust your compiler (but check its output).

Thank You!

You can find the library at: cpp.fail/ctre

Q & A

You can find the library at: cpp.fail/ctre