C++ Modules: the good, the bad, the ugly

Uladzislau Chekhareu

- Technical Lead Developer at Solarwinds
- Love cats, C++ and riding a bike

Introduction

- Modules - one of the most anticipated things in C++ 20
- But it’s not discussed much comparing to some others!
- I’ll try to explain, what is this beast and how to deal with it

The state of modules

- Paper p1103r2.pdf approved for merge into International Standard
- To some extent implemented by major compilers
- Examples from slides were tested using gcc

What do we have now

Organizing code through headers

- Old good #include directives
- Preprocessed, not part of the language
- Simple text replacement

What do we have now

Bad:

- Include guards
- Compilation times
- Weak physical encapsulation
- No isolation
- Sometimes include order matters
- ODR violations

What do we have now

Good:

- Easy
- Parallel compilation
- Good support by build systems
- Established practices to prevent smelling code

Modules goals

- Modular interfaces
- Physical encapsulation
- Isolation
- Compilation speed

New terms

module, import - new special identifiers

export - reused and repurposed keyword

Module unit - translation unit that contains module declaration

Module - collection of module units with the same name

Module units

Primary interface unit

```cpp
// hello.cxx
export module hello;

export int foo() {
   return 42;
}

// main.cxx
import hello;

int main() {
   foo();
}
```

```cpp
// valid module names
hello
hello.core
hello_util
hello101


// invalid module names
1111hello
.
.hello
```

Interface partition unit

```cpp
// hello_world.cxx
export module hello:world;

export int bar() {
   return 50;
}
```

```cpp
// hello.cxx
export module hello;
export import :world;

export int foo() {
   return 42;
}
```

Implementation unit

```
// hello_impl.cxx
module hello;

int foo() {
   return 42;
}
```

```
// hello.cxx
export module hello;
export import :world;

export int foo();
```

Implementation partition unit

```
// hello_helper.cxx
module hello:helper;

int doStuff(int bar) {
   return bar + 1;
}
```

```
// hello_world.cxx
export module
hello:world;

import :helper;

export int bar() {
   return doStuff(50);
}
```

```
// hello_impl.cxx
module hello;

import :helper;

int foo() {
   return doStuff(42);
}
```

Client code

```
// main.cxx
import hello;          // Ok
import hello:world;    // Error, can't explicitly import interface partition
import hello:helper;   // Error, can't import implementation partition

int main() {
   foo();              // Ok
   bar();              // Ok
   doStuff(1);         // Error
}
```

Exporting and importing

Export can only appear at namespace scope

```
export void foo() {}                          // Ok, global namespace

export template<typename T>                   // Ok
void foot(T input) { /* snip */ }

export enum class Errors { /* snip */ }  // Ok, also allowed for old-style enums

namespace Bar {
    export int foo = 42;                      // Ok, namespace scope
    export struct FooBar {                    // Ok, also allowed for classes
        int Baz = 2;
    }
}
```

Export can only appear at namespace scope

```
class Gadget {
   export Widget w;                          // illegal
};

void bar() { export int i = 5; }             // bad

void baz(export std::string input) {}        // please don't

template<export typename T>                  // stop
export class my_container{};

enum class Errors {
    export InvalidArgument = 1000            // error
};
```

Convenient exporting

```
export namespace Goods { // exports all enclosing entities
    int const Count = 5;
    template<typename T>
    void foot(T input) { /* snip */ }
}
```

```
namespace Goods {
    void bar {} // NOT exported!
}
```

```
export {
    auto Value = 10.f;
    class Gadget {};
    /* more exported entities */
}
```

Can’t export entities with internal linkage

```
namespace {
    export int foo = 42;            // bad - anonymous namespace
    export void bar() {}
    export class MyClass() {};
}

export static int baz = 30;         // bad - static
export static void foobar() {}
```

Can’t export macroses!

```
export #define FOO 42    // hell no!

#define BAR 42
export BAR;              // you shall not pass
```

Re-export of imported module

```
// gadget.cxx
export module gadget;

export struct Gadget
{
   int Detail;
};
```

```
// widget.cxx
export module widget;

// re-export
export import gadget;

export struct Widget {
   Gadget First;
};
```

```
// main.cxx
#include <iostream>
import widget;

using namespace std;

int main() {
   Gadget g{42};
   Widget w{g};
   cout << w.First.Detail << endl;
}
```

Modular code: all imports inside preamble

```
// foo.cxx
export module foo;
import bar;     // Ok
import baz;     // Ok

class baz {};  // Preamble ends, no more imports allowed

import foobar; // Error
```

Non-modular code: import everywhere

```
// main.cxx
#include <stdio.h>
import foo;             // Can use names from foo from this point

void foobar() {
    func_from_foo();  // Ok
    func_from_bar();  // Error
}

import bar;             // Can use names from bar from this point

void barfoo() {
    func_from_bar();  // Ok now
}
```

Can’t import partitions belonging to other modules

```
// a_p1.cxx
module a:p1;

/* snip */
```

```
// b_p1.cxx
module b:p1;

/* snip */
```

```
// b.cxx
export module b;
import :p1;        // Ok
import a:p1;       // Error
import b;          // Error

/* snip */
```

Cyclic import is not allowed

```
// a.cxx
export module a;
import b;

/* snip */
```

```
// b.cxx
export module b;
import c;

/* snip */
```

```
// c.cxx
export module c;
import a;          // not allowed

/* snip */
```

Partial backward compatibility

```
// foo.cxx
export module foo;

class import {};     // Ok, but please don't do it

import i1;           // Error, treated as keyword
::import i2;         // Ok, please don't
```

Modules in-depth

Name mangling

Before modules:

1. Internal linkage
2. External linkage
3. No linkage

Modules introduce:

4. Module linkage

Name mangling

```
// alinkage.cxx
export module alinkage;

export int foo = 42;     // external
static int baz = 50;     // internal
int bar = 10;            // module

export void doStuff() { // external
   int something = 5;   // no linkage
}

static void doInternalStuff() {} // internal
void doModuleStuff() {}          // module
```

Symbol table from alinkage.o:

Bind	Name
LOCAL	_ZW8alinkageEL3baz
LOCAL	_ZW8alinkageEL15doInternalStuffv
GLOBAL	foo
GLOBAL	bar
GLOBAL	_Z7doStuffv
GLOBAL	_ZW8alinkageE13doModuleStuffv

Name mangling

```
// alinkage_impl.cxx
// implementation unit of module alinkage
module alinkage;

void doBar() {
   bar = 110;          // Ok, module linkage
   baz = 150;          // Error, internal linkage
   doModuleStuff();    // Ok
   doInternalStuff(); // Error
}
```

Symbol table from alinkage.o:

Bind	Name
LOCAL	_ZW8alinkageEL3baz
LOCAL	_ZW8alinkageEL15doInternalStuffv
GLOBAL	foo
GLOBAL	bar
GLOBAL	_Z7doStuffv
GLOBAL	_ZW8alinkageE13doModuleStuffv

Name mangling

```
// blinkage.cxx
// separate module blinkage
export module blinkage;

export int foo = 50;     // Link time error
export int baz = 55;     // Ok
int bar = 15;            // Should be ok
void doModuleStuff() {} // Ok
```

Symbol table from alinkage.o:

Bind	Name
LOCAL	_ZW8alinkageEL3baz
LOCAL	_ZW8alinkageEL15doInternalStuffv
GLOBAL	foo
GLOBAL	bar
GLOBAL	_Z7doStuffv
GLOBAL	_ZW8alinkageE13doModuleStuffv

Name mangling implications

1. Modules are **not** name scoping mechanism!
2. Place exported names inside namespace
3. Use static and anonymous namespaces carefully

Compilation

Headers

- Assuming m headers and n sources
- Source code size is m + n
- Header is compiled for each inclusion
- So compilation takes m * n time

Modules

- Source code size is still m + n
- Module is compiled once
- Compilation takes m + n time!
- How does it work?

Compilation

BMI - binary module interface

- Produced by compiling module interface
- Read by importers
- Contains information required by importers to use module (for example, AST)
- Compiled once per build!

Compilation

- BMI is produced by compiling module interface...
- And read by importers
- Dependency!
- Module interface **must** be compiled before importers
- Building modular code is not fully parallelizable
- Headers, on the other hand...

Compilation

Paper concerning modular code compilation speed:

https://bfgroup.github.io/cpp_tooling_stats/modules/modules_perf_D1441R1.html

Author: Rene Rivera, committee member, participant of SG15

Twitter: https://twitter.com/grafikrobot

Compilation

Method used:

- Compile 150 source files
- Test for various (up to 150) dependency DAG chain depths
- On each level include (or import) 3 components from previous levels
- Simple code - only *int* variables definitions

Compilation

Modular source

```
export module m148;

import m0;
import m15;
import m84;

namespace m148_ns
{
export int n = 0;
export int i1 = 1;
// ...
export int i300 = 300;
}
```

Header

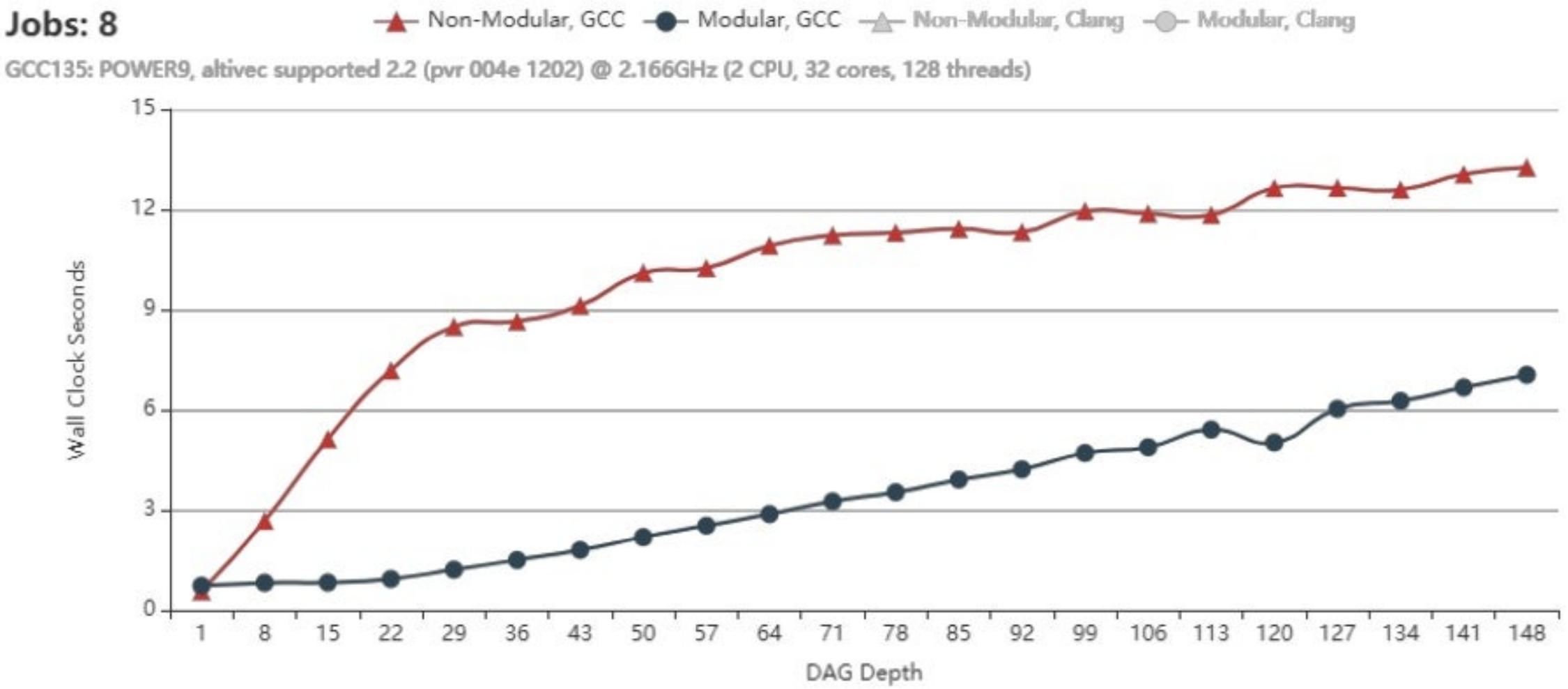
```
#ifndef H_GUARD_h148
#define H_GUARD_h148
#include "h77.hpp"
#include "h78.hpp"
#include "h92.hpp"

namespace h148_ns
{
int n = 0;
int i1 = 1;
// ...
int i300 = 300;
}
#endif
```

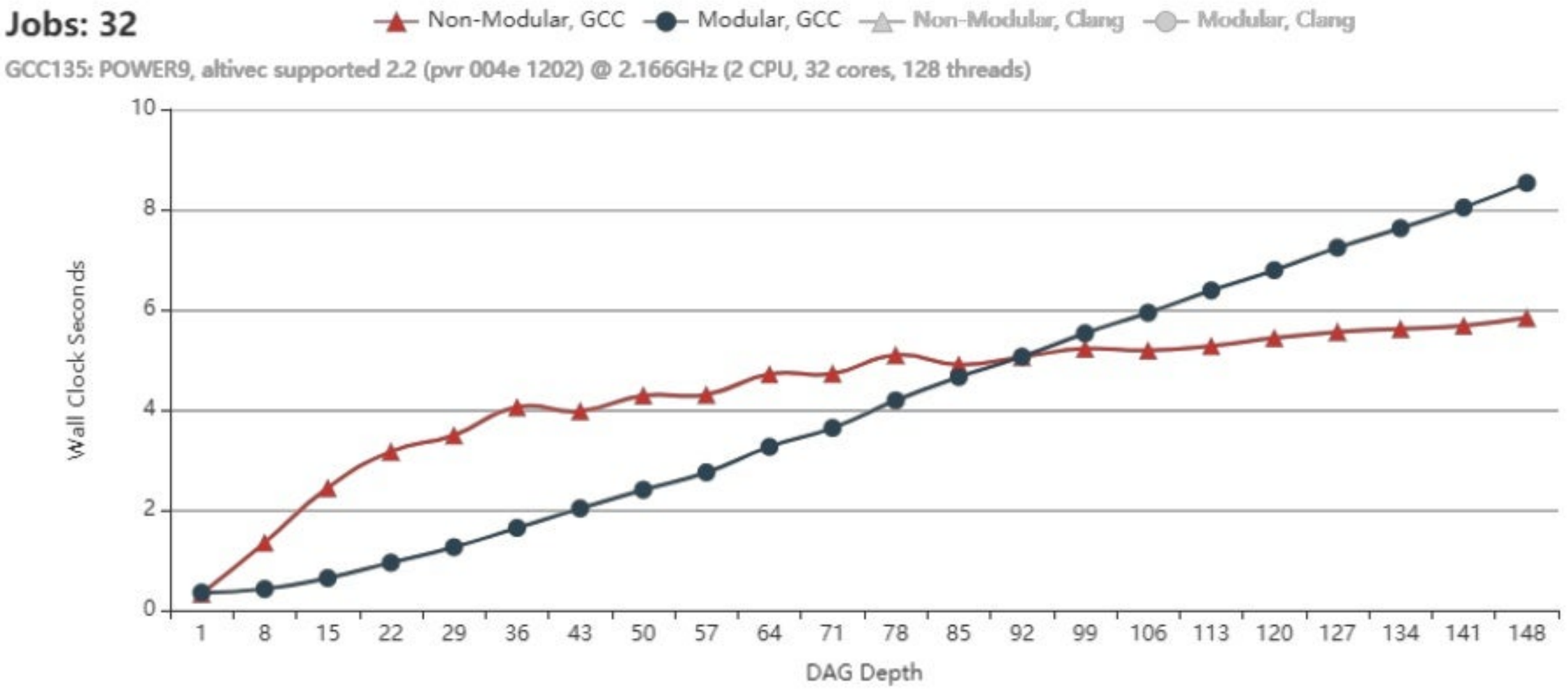
Source

```
#include "h148.hpp"
```

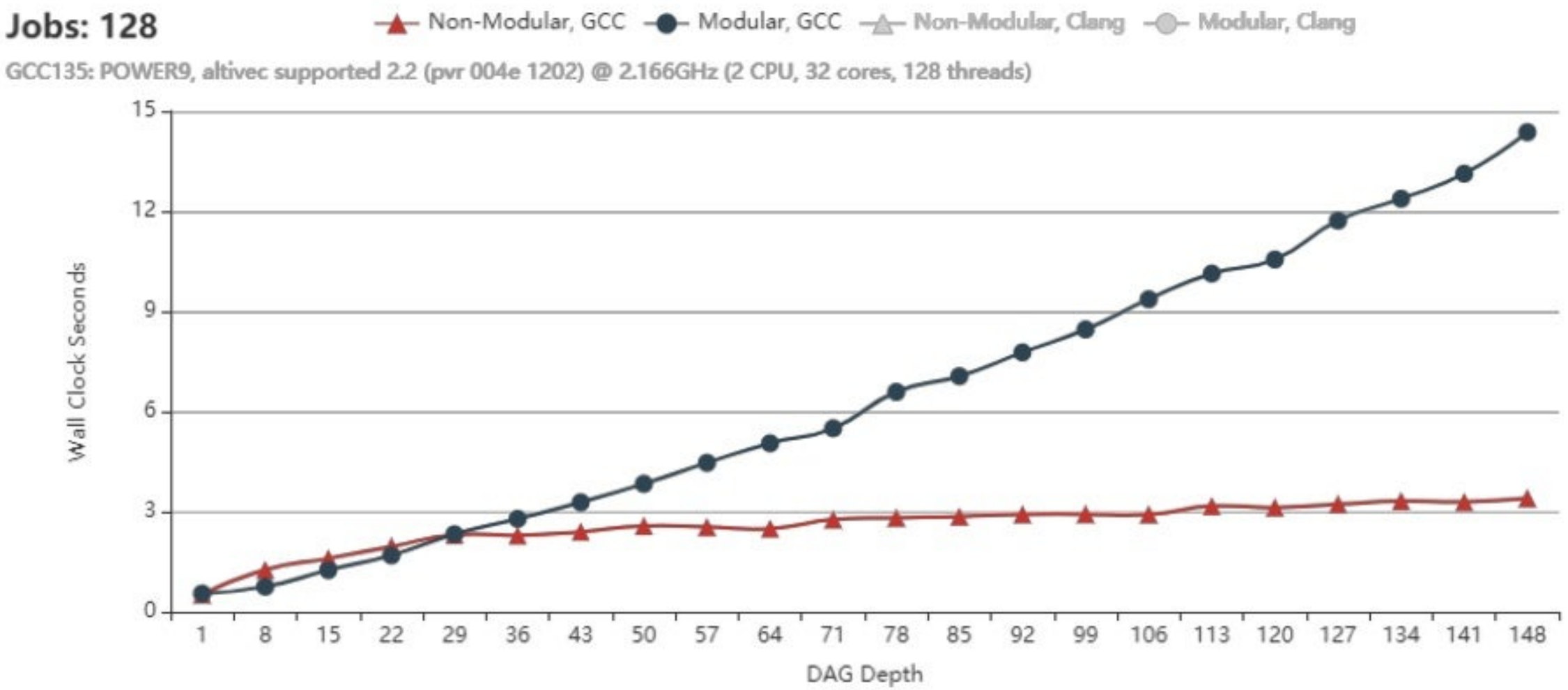
Compilation



Compilation

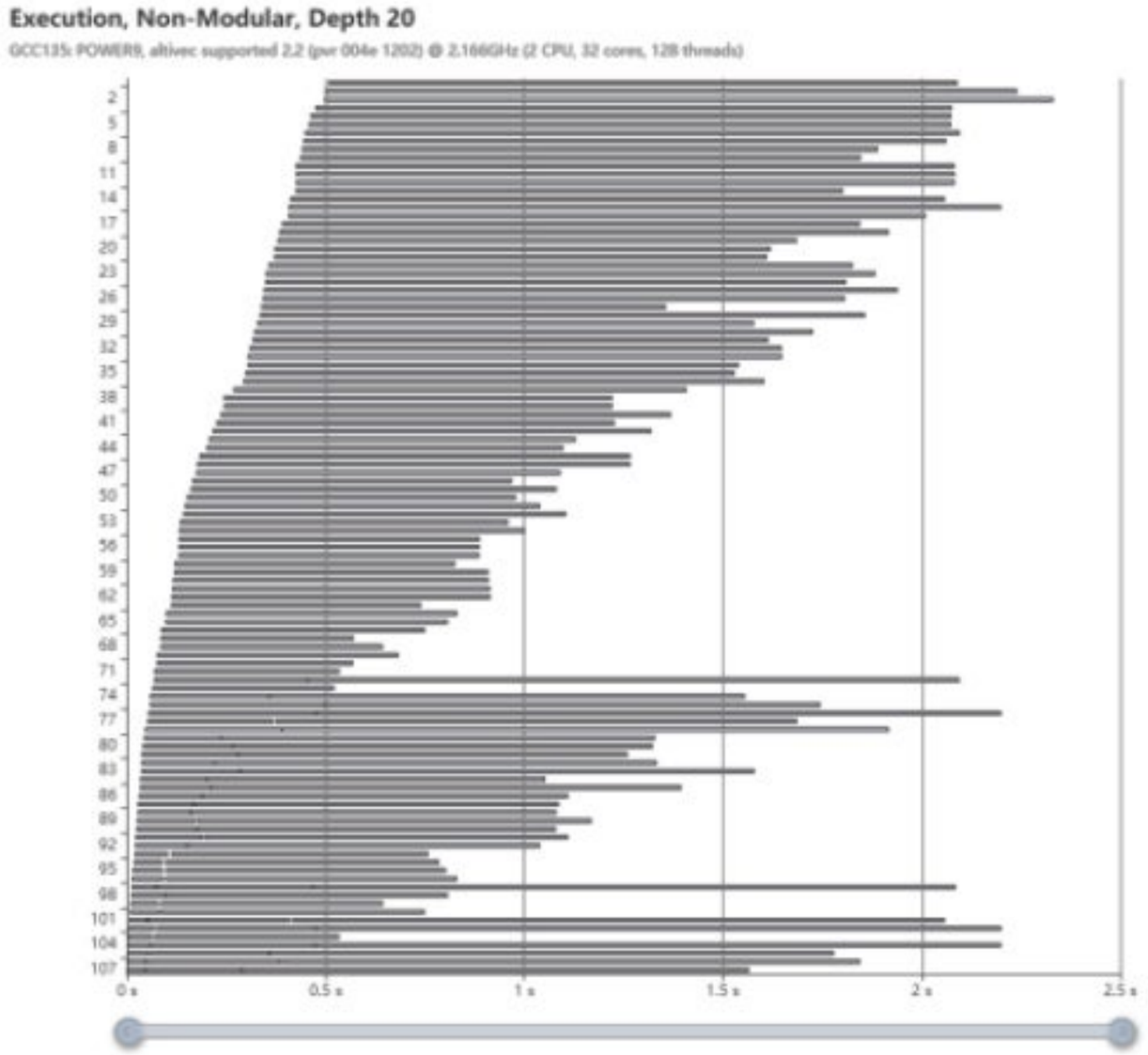


Compilation



Compilation

Compilation

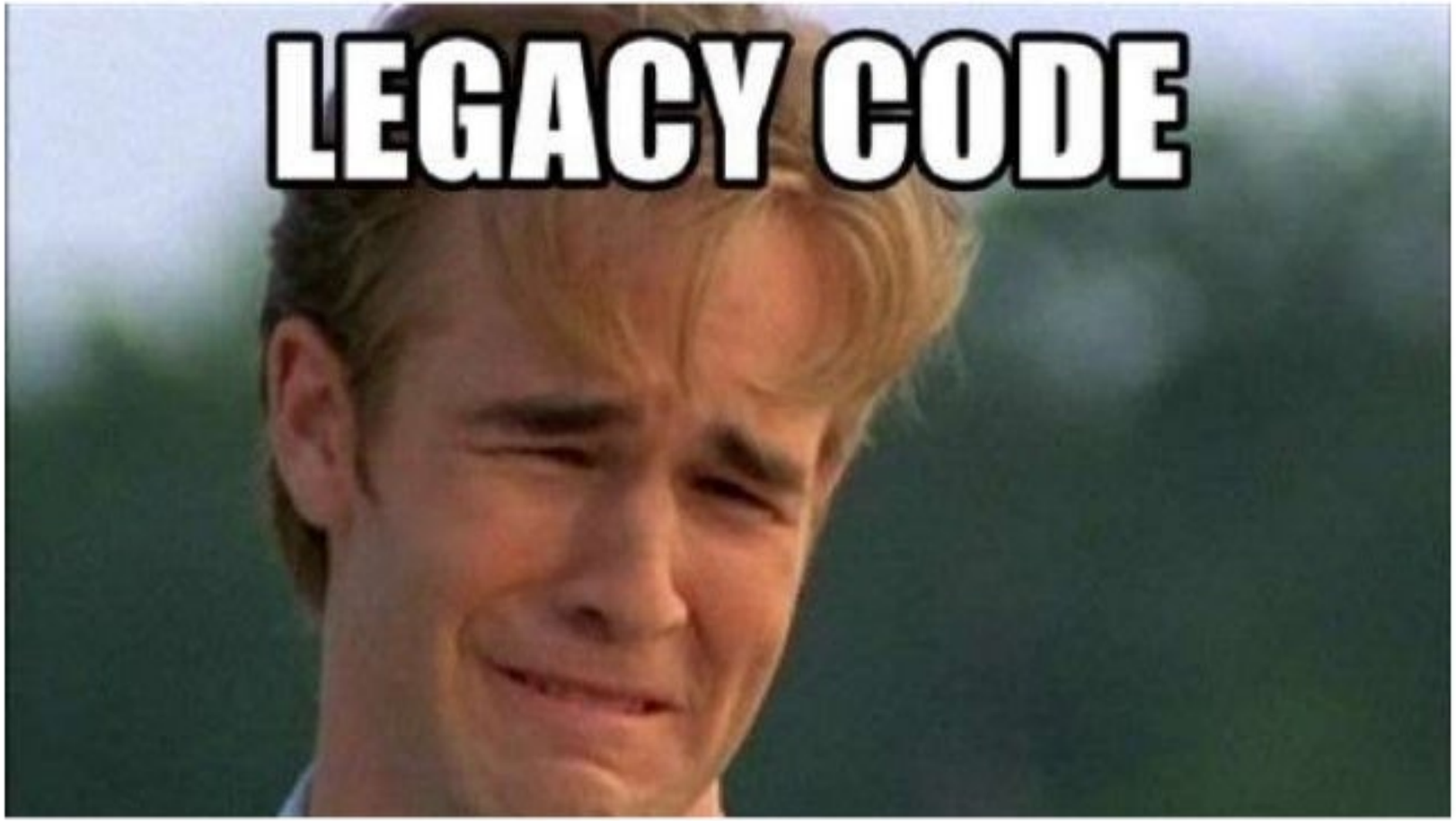


Compilation

Observations:

- Unlike headers, modules build can not fully utilize system resources in highly parallel environments
- At least we can watch youtube while building cool modular code :)
- Moore's law
- But test is not perfect
- And compilers and build systems module support will mature

Legacy



Legacy

- #include "legacy_header.h" - inside global module fragment
- import "legacy_header.h" - inside module preamble
- Leaks macroses with all related problems
- Huge theme for another talk

Summary

Modules goals ?

- Modular interfaces
- Physical encapsulation
- Isolation (except exported names)
- Compilation speed (hopefully)

What do we do now?

- Wait and hope?
- Try it out yourself!
- Consider good practices for writing modular code (don’t forget to share!)
- Prepare your legacy

How to try out

MSVC 2019

- /experimental:module
- /std:c++latest
- Modular code must be in ***.icc** files
- Only basic features are supported
- Somehow “supports” modular **stl** (*import std.core;*)

How to try out

GCC development branch ‘c++-modules’

- Wiki link: https://gcc.gnu.org/wiki/cxx-modules
- -fmodules-ts
- -fmodule-header
- Mostly supports merging proposal
- Sometimes ICEs (specifically on some header units)

Some guidelines

Module naming:

- Module name should be unique
- Avoid names like **util** or **core**
- Prefix with company and/or product name

Some guidelines

Exporting:

- All exports should be done within single top-level namespace
- Do not export multiple top-level namespaces
- Do not re-export everything, keep interfaces minimal

Conclusion

So are C++ Modules:

- Good?
- Bad?
- Ugly?

THANK YOU!

Q&A

Links

- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1103r2.pdf - merging module proposal
- https://gcc.gnu.org/wiki/cxx-modules - GCC module implementation state
- https://bfgroup.github.io/cpp_tooling_stats/modules/modules_perf_D1441R1.html - compilation speed comparison
- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1427r0.pdf - module toolability concerns