Are modules fast?

Rene Rivera – grafikrobot@gmail.com – P1441R1, 2019-03-06

Table of Contents

1. Abstract
2. Changes
   2.1. R1
   2.2. R0 (Initial)
3. Introduction
4. Parallel Build
   4.1. Method
      4.1.1. Modular Source
      4.1.2. Non-modular Source
   4.2. Limitations
   4.3. Results
   4.4. Execution
5. Conclusion
6. Acknowledgements
7. References

Document number	ISO/IEC/JTC1/SC22/WG21/P1441R1
Date	2019-03-06
Reply-to	Rene Rivera, grafikrobot@gmail.com
Audience	WG21

1. Abstract

Measurements of the performance of building C++ modules and relevant comparisons.

2. Changes

2.1. R1

Fix bug in test generation and DAG calculation that resulted in non-uniform growth in the modular test times. Fix grammatical errors. Ran tests on a high CPU machine from the GCC Farm Project. Additional data of parallel building from higher thread and faster hardware, thanks to Robert Maynard for running the tests and for the GCC Compile Farm for access to their machines. Replaced simple level based scheduler with a full parallel DAG executor to avoid CPU underutilization. Support for Clang 7.0.1 module compilation for alternate data.

2.2. R0 (Initial)

Initial performance measurements for synthetic tests of gcc merged modules implementation (rev 268043, 2019-01-17 10:58:47 -0600 (Thu, 17 Jan 2019)) .

3. Introduction

One of the stated goals of the modules proposals was performance over existing non-modular source specifically in terms of scalable builds. This paper aims to answer the important question of whether that goal was achieved by current modules implementations.

Building software does not happen in the solitude of the C++ compiler. It is a careful orchestration of a collection of tools from the preprocessor, compiler, linker, assembler, etc controlled by the build system. These have all been optimized to deal with the current separate compilation model and generally perform gargantuan feats. But as such it means that performance measurement needs to take into account, to some degree, all those in a controlled method to generate meaningful measurements.

To that end the measurements presented here are "synthetic". They are structured such that they simulate and isolate certain components, like the build system, to facilitate the extraction of relevance in the data. Such methods are described below for individual measurements.

4. Parallel Build

Although there are various aspects comprising performance scalability of key interest is in seeing how the potential algorithmic savings of modules "caching" compiler work at the cost of longer DAG build chains compares against the current, almost unlimited, parallelized building of plain source compiles. This test aims to measure the overall compile times of modular source across varied dependency chain depth against equivalent non-modular source.

4.1. Method

Overall we perform the basic task of compiling 150 C++ source files into corresponding object files only. We perform the compile at different dependency DAG chain depth counts. Each source depends, either as an `import` or `#include`, on some number of sources (headers or modules) from the previous DAG levels. Which means that at DAG level 150 each source depends on at least the one source from the previous DAG level. The overall wall clock time to compile all 150 files is taken as the result of the measurements. An important observation is that at the highest DAG depth the arrangement creates a completely linear dependency chain.

4.1.1. Modular Source

For modular source this test simulates the behavior of a build system to compile individual modular source files in parallel executions of the compiler to the limit of the available CPU threads. It does so in appropriate, but simulated, dependency DAG order; compiling each DAG level as a group. The process goes as such:

1) Generate a synthetic, and simplified, DAG to describe the build. Where all the sources in the level can be compiled in parallel.

2) Generate all source files similar to this:

```C++
export module m148;

import m0;
import m15;
import m84;

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

Where the imports are randomly chosen from the set of all already compiled modules (in previous DAG levels).

3) For each DAG level compile all the source files therein with a GCC invocation similar to:

```BASH
g++ -fmodules-ts -c -O0 m148.cpp
```

4.1.2. Non-modular Source

For no-modular source this test simulates the behavior of a build system to compile individually all source files in parallel executions of the compiler to the limit of the available CPU threads. As the non-modular sources only depend on already existing header source files all source files can be attempted to be compiled at once. The process goes as such:

1) Generate a synthetic, and simplified, DAG to describe the build with parity to the modular build. But which is not used in the build itself.

2) Generate all source header files similar to this:

```C++
#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
```

Where the includes are randomly chosen from the set of all headers files in previous DAG levels. Although not needed to limit to previous DAG levels this is done to keep parity with the modular source in terms of statistical complexity of the included source size.

3) Generate all source files similar to this:

```C++
#include "h148.hpp"
```

4) Compile all source files as one group with a GCC invocation similar to:

```BASH
g++ -c -O0 h148.cpp
```

4.2. Limitations

This test has some real-life limitations borne out of the software and hardware used for testing. Some of these limitations were discovered through experimentation, for example with internal compiler errors.

Being an experimental language feature the support for compiling modules is fragile and placed severe limits on what the modular sources could contain:

- Most current C++ constructs fail to compile reliably on GCC, for example templates, and cause ICEs. This is why the generated sources only contain `int` variable definitions.
- Along that same line the current GCC modular compiler also seems to have stability limits on how many imports can be done. This is why only a maximum of three (3) `import` statements are included in the source. As the compiler errors with four (4) or more `import` statements.
- Not to be outdone, compiling the header test also has one limitation. The GCC and Clang compilers have a hard limit of number of include header depth of 150. Which is why the tests only cover up to 150 files and consequently a matching DAG depth.
- The Clang compiler does not implement the merged modules proposal. Which means we avoid features like the global fragment module when generating test source.
- The Clang compiler uses two stages for modules. One to generate the BMI and another to generate the object code. This potentially increases the amount of parallelism but also increases the number of processes that need to run.

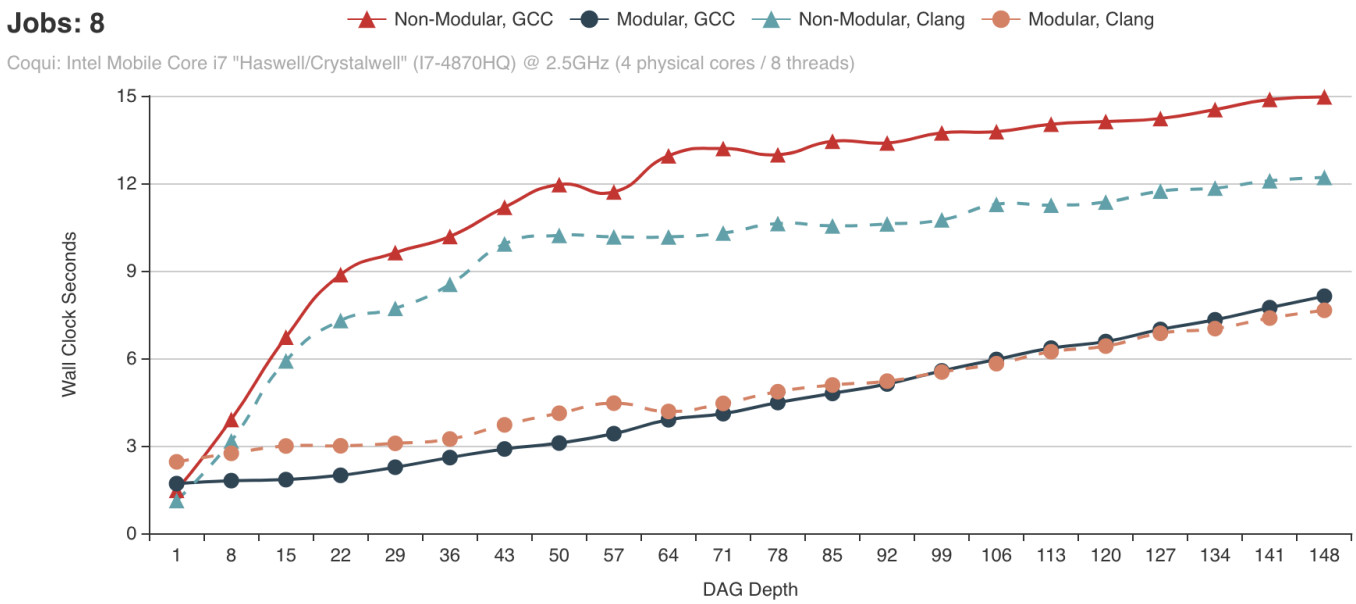
This is a simulation with "perfect" build knowledge as all dependency and source information is known before building starts. Hence it is only a very rough approximation of reality where build system have to deal with dependency discovery while building and in the face of generated source files.

The executor that schedules build commands is greedy and executes the first available task as soon as there’s a slot for it.

There is some amount of overhead on the Python side for running the threads in the executor. But this is stable through the lifetime of each test rune, and hence can be ignored as we only care about relative performance.

4.3. Results

The charts below use a triangle icon for non-modular builds, and a circle for modular builds.

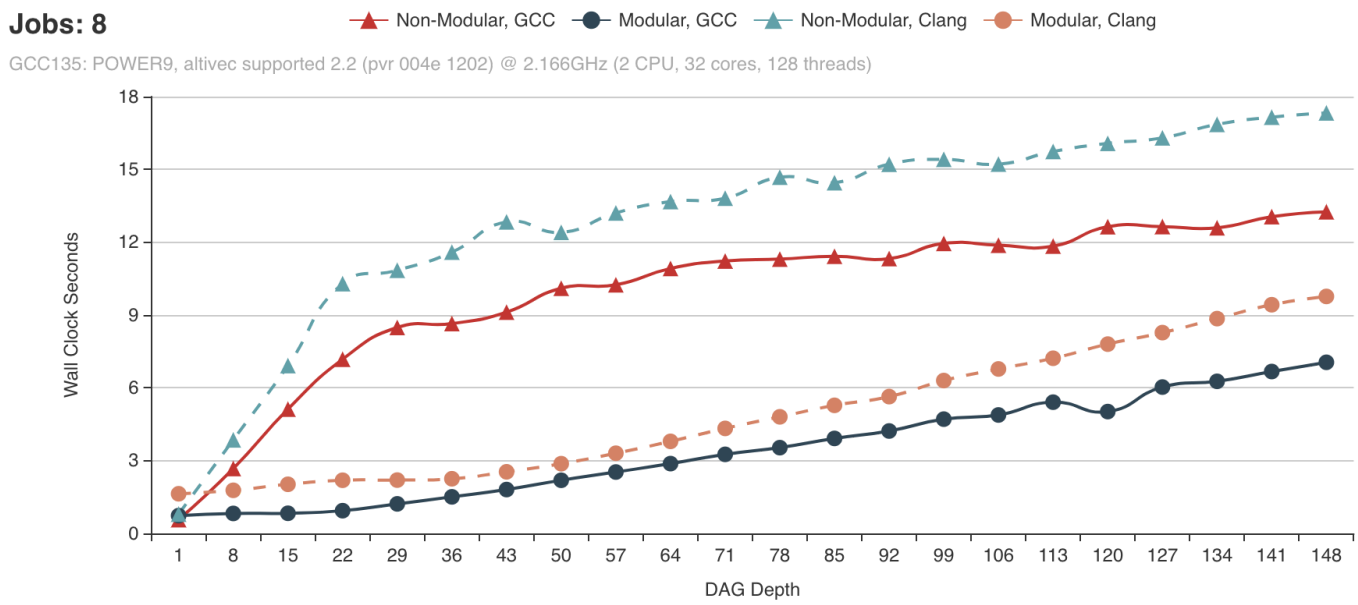


First looking at the GCC performance, the solid lines, we can see that modular compiles have a consistent advantage over all DAG depths. The modular compiles rise slowly, and steadily, as the DAG depths increase. Whereas the non-modular compiles initially rise quickly and then a bit slower. But they still rise over time as more source code is consumed from the deepening header inclusions. This certainly looks really good for a modular future.

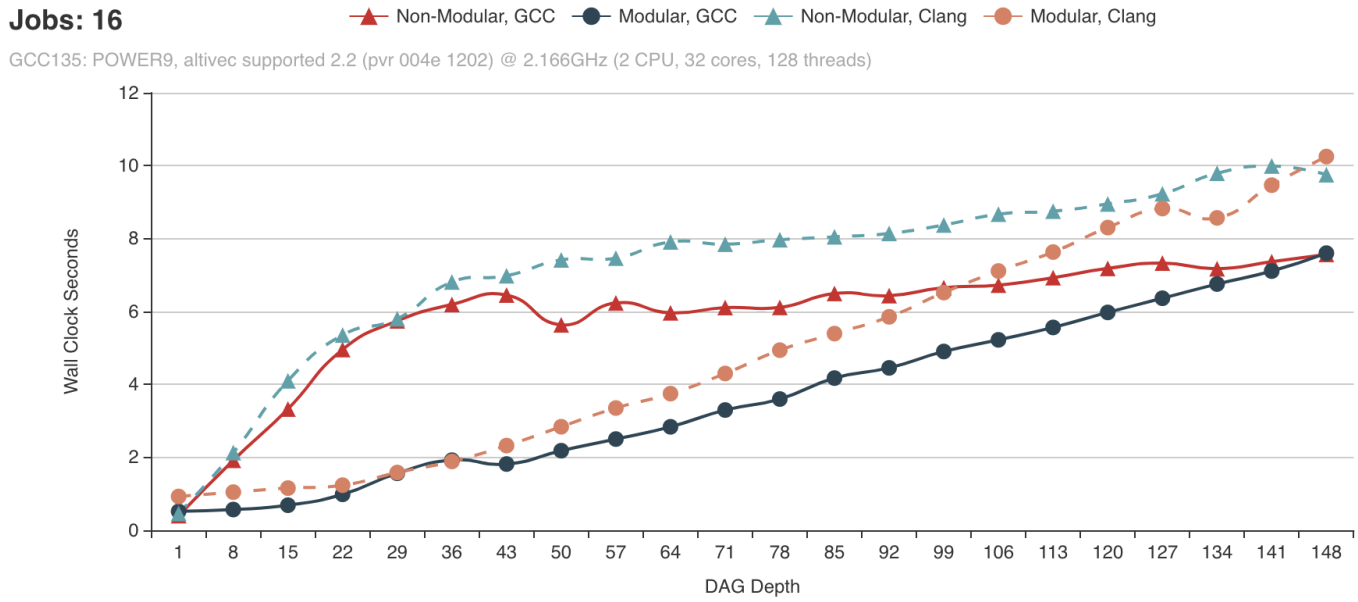
Now, lets look at Clang, the dashed lines, like GCC it follows a similar almost linear increase. And similarly the non-modular Clang compiles perform like the GCC non-modular builds, rising quickly and then slowly. An observation worth noting is that Clang performs slightly better than GCC over the increasing DAG depths.

Both compilers perform very well at the not high DAG depths and scale well as DAG depth increases. But this is at what is considered low parallelism levels. Even phones are approaching this level of performance.

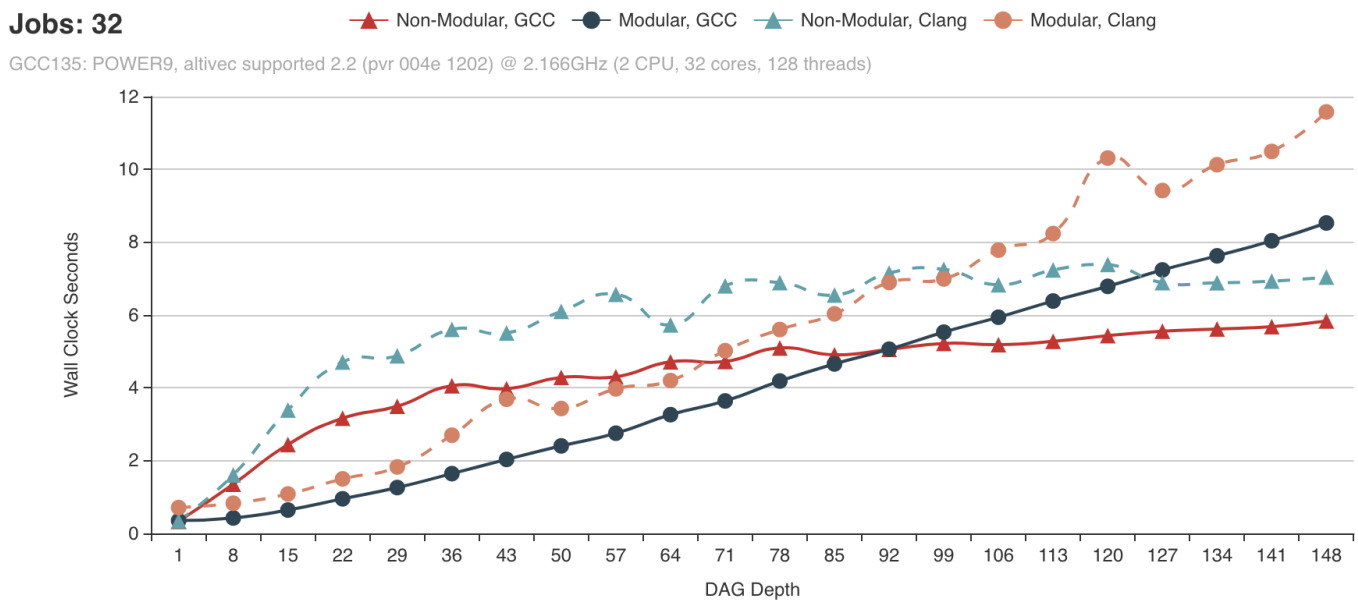
Lets now look at higher levels of parallelism with a machine that has only slightly lower CPU core speed but is capable of executing up 128 threads with its 32 physical cores. The first sample set is a base line comparison at 8 parallel jobs. As we can see it performs comparably with the above results. The one big difference is that now GCC performs better than the Clang builds.

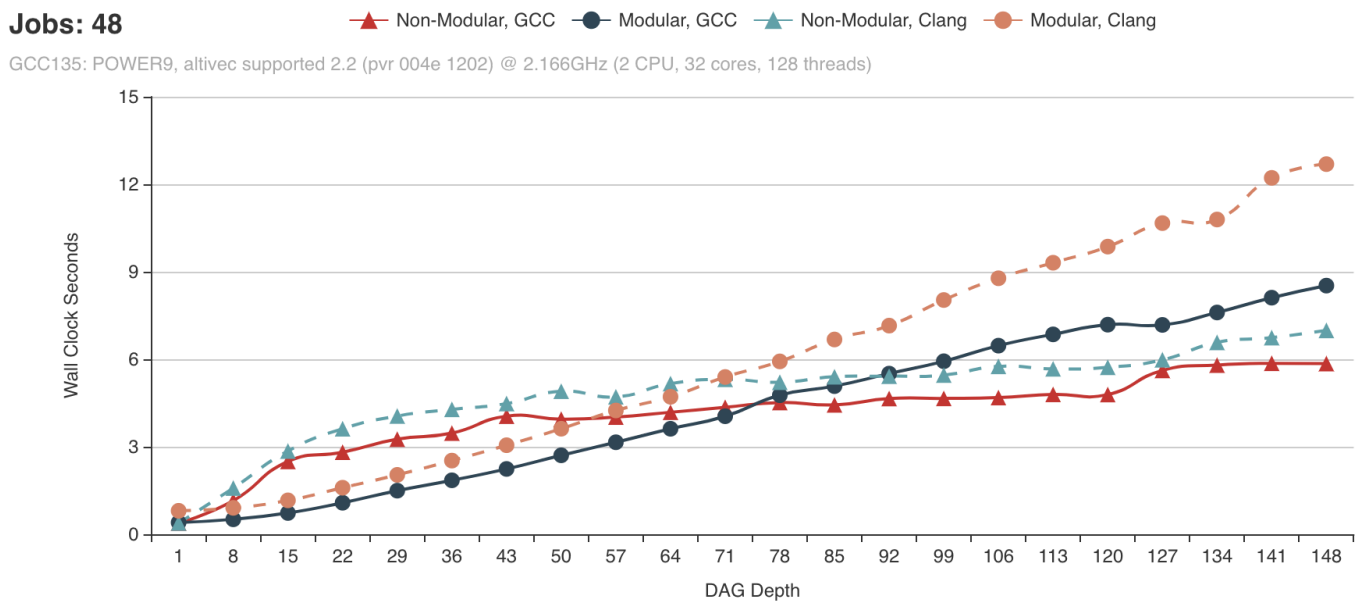


We'll now jump to 16 parallel jobs and onward step incrementally 16 jobs at a time until we reach the machine thread maximum of 128.



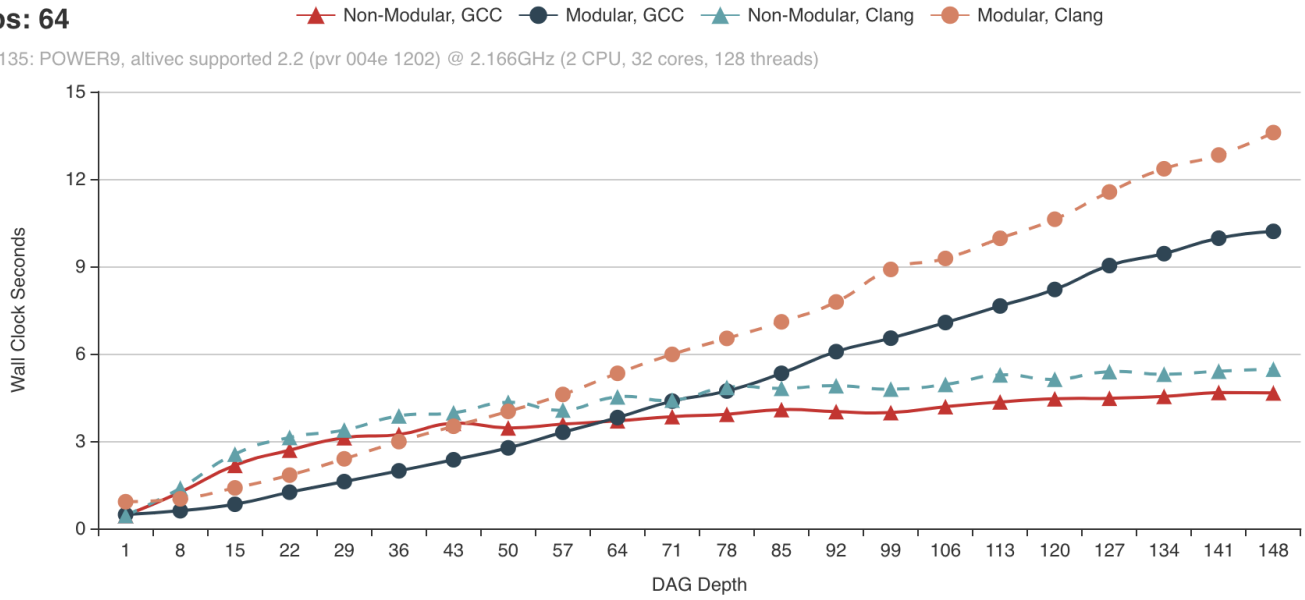
Immediately we can see that the non-modular builds roughly half in clock time matching the doubling of the available parallelism. And are now very close to the performance of GCC modular builds. What is striking is that the modular builds don't substantially change from the 8 parallel job case. And while before modular builds where faster at all depths now the non-modular builds have caught up at the highest depth of this test.



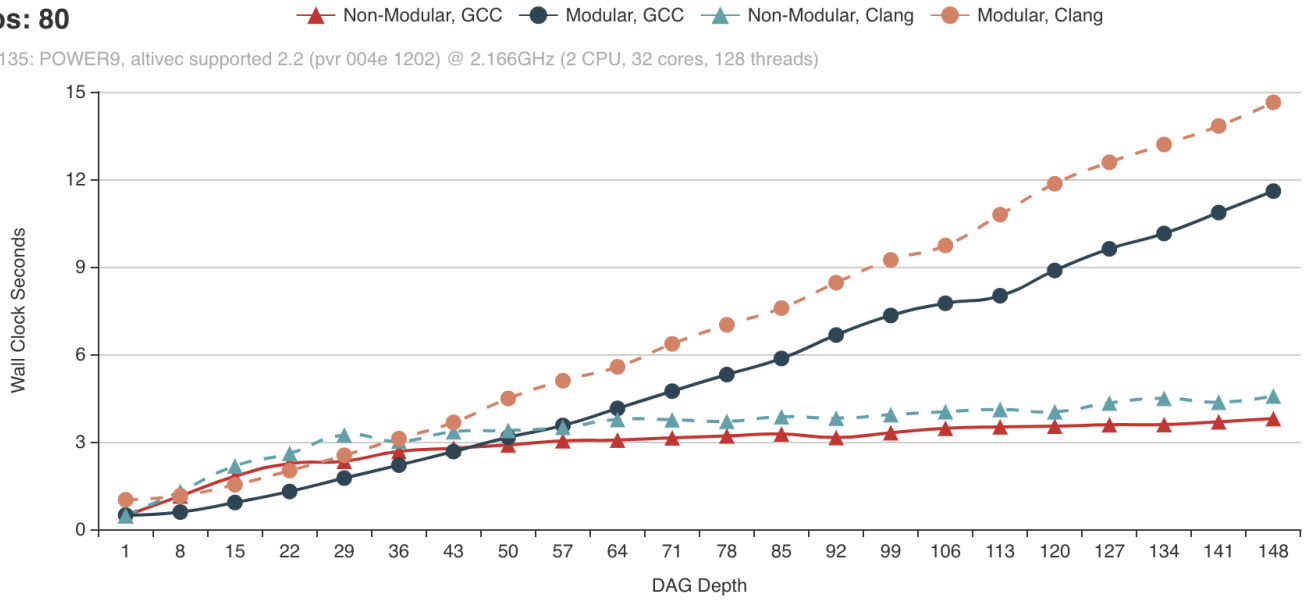


As we can see the increasing number of parallel jobs keeps improving the performance of the non-modular builds while the modular builds are consistent.

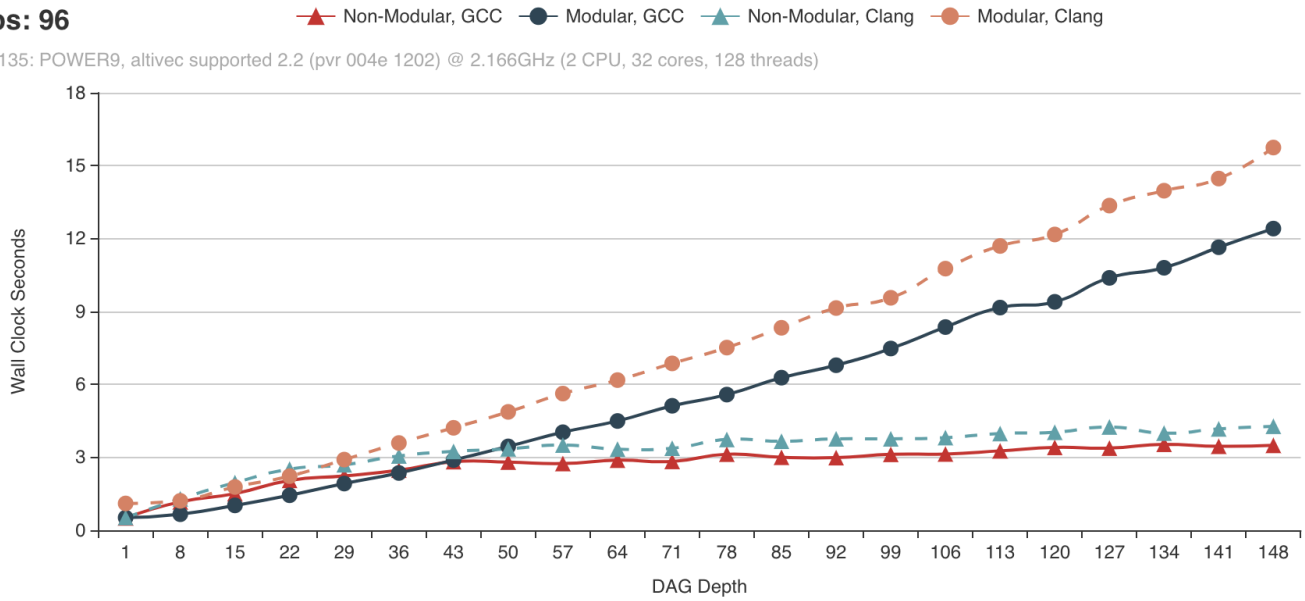
Jobs: 64

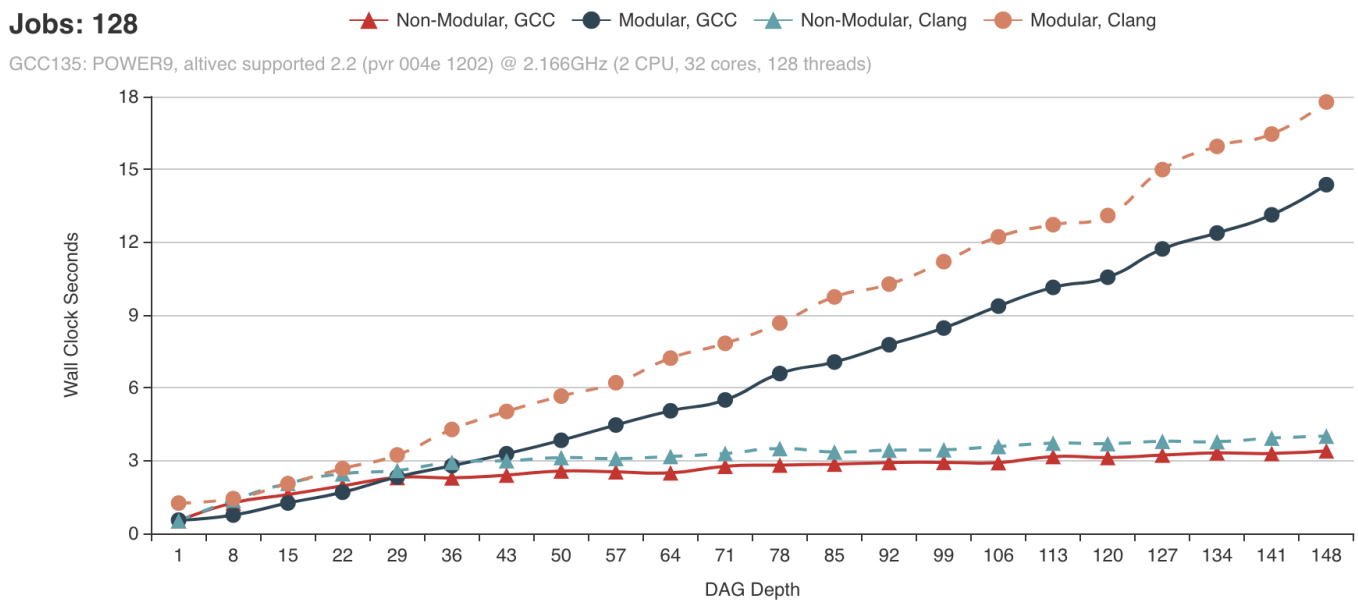


Jobs: 80



Jobs: 96

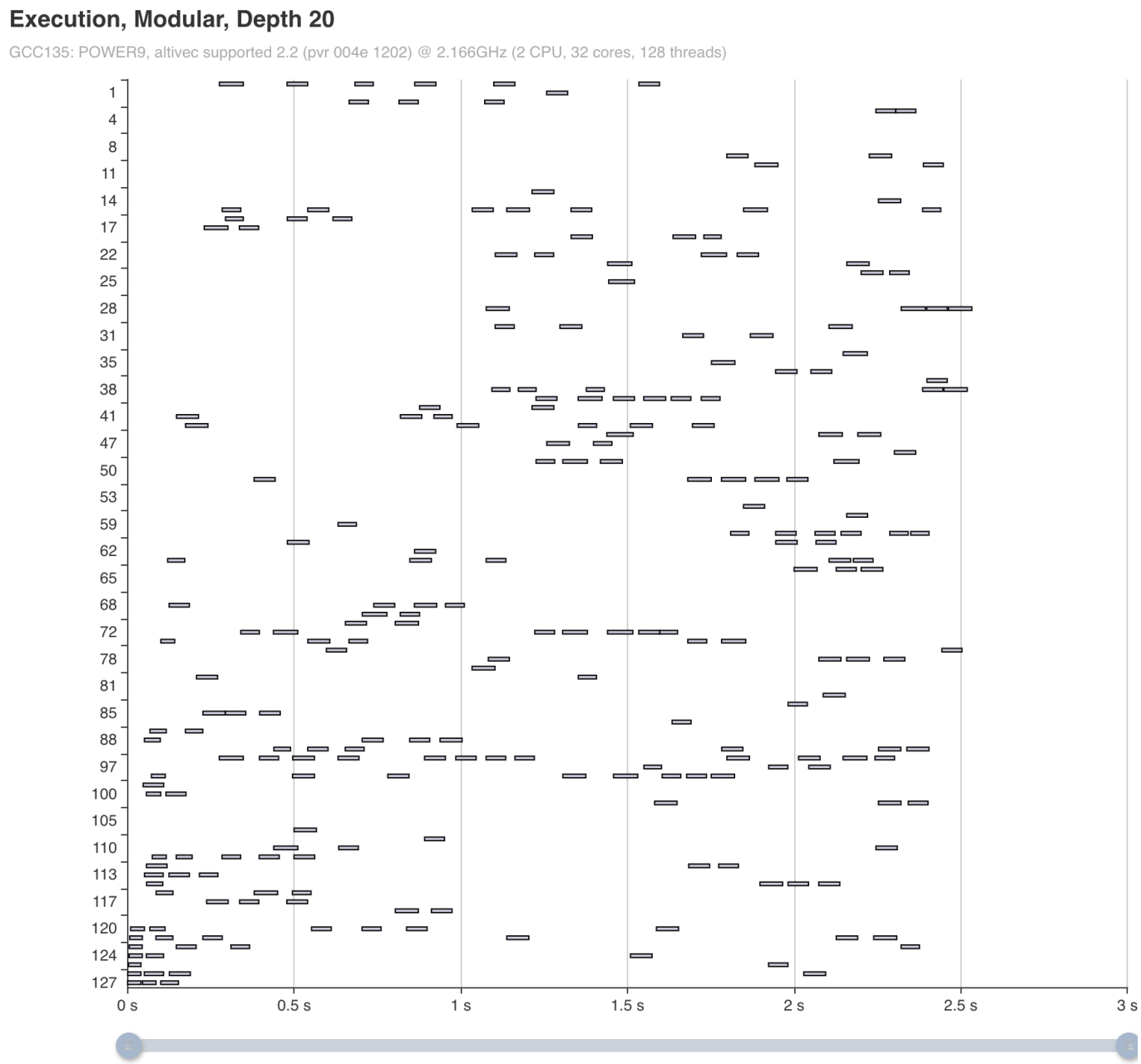




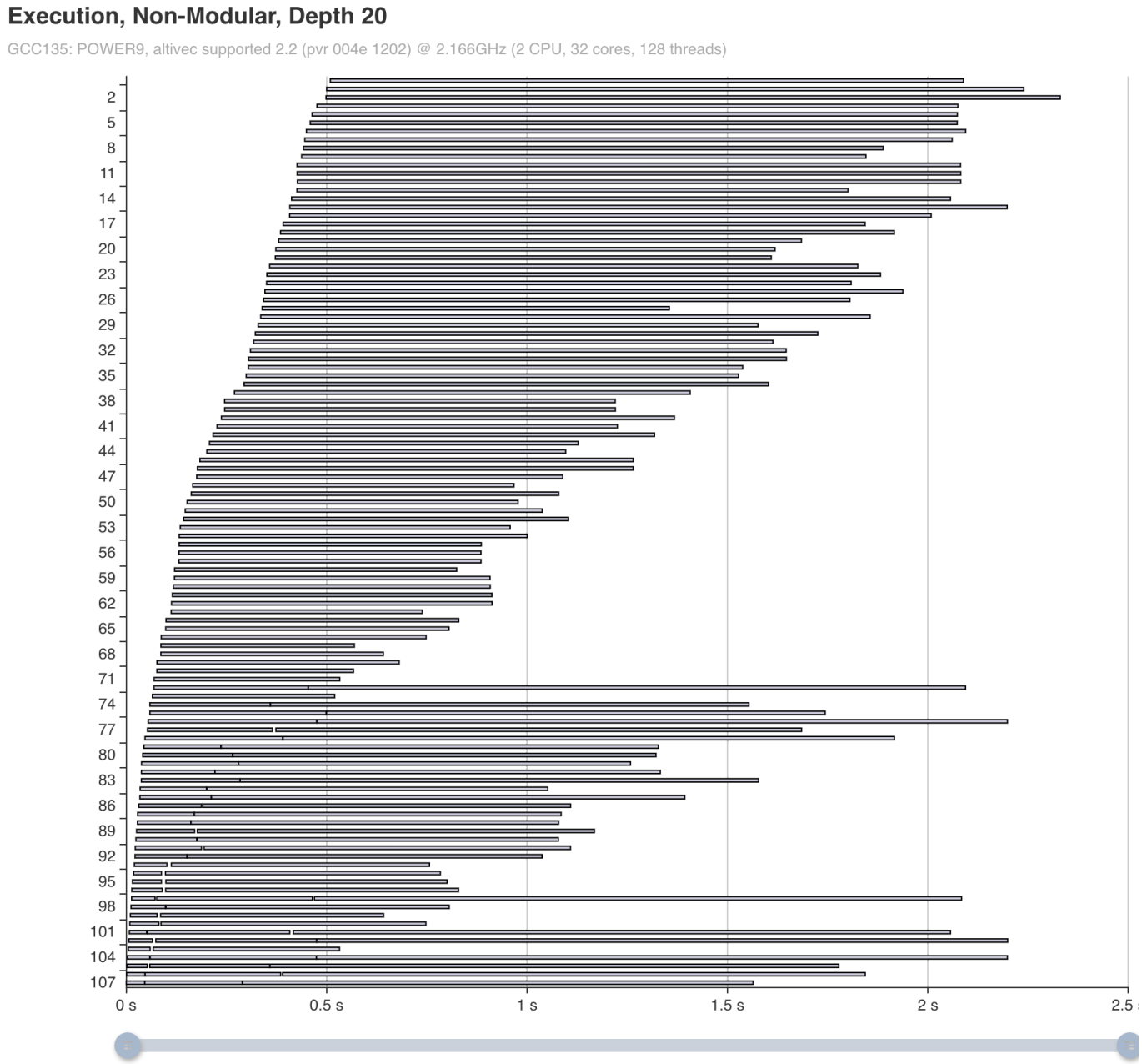
We've now reached the point where non-modular builds are faster at most DAG depths. And while the modular builds have stayed the same in the face of increasing parallelism, the non-modular builds have gained ground at each step.

4.4. Execution

Looking at wall clock performance isn't complete without also looking at the execution distribution also. Here we look at the timing chart of compilation tasks performed during one build cycle, i.e. when tasks ran and on which job thread. In this case we chose a DAG depth of 20 as that is likely a reasonably common structure for larger projects. But we do use the maximum local parallelism of 128 jobs to highlight the possible differences between modular and non-modular builds.



We can see above one key observation, there is a lot of unused CPU time during the overall build execution. This would be due to tasks needing to wait for the needed dependent tasks to complete. Below is the equivalent non-modular build.



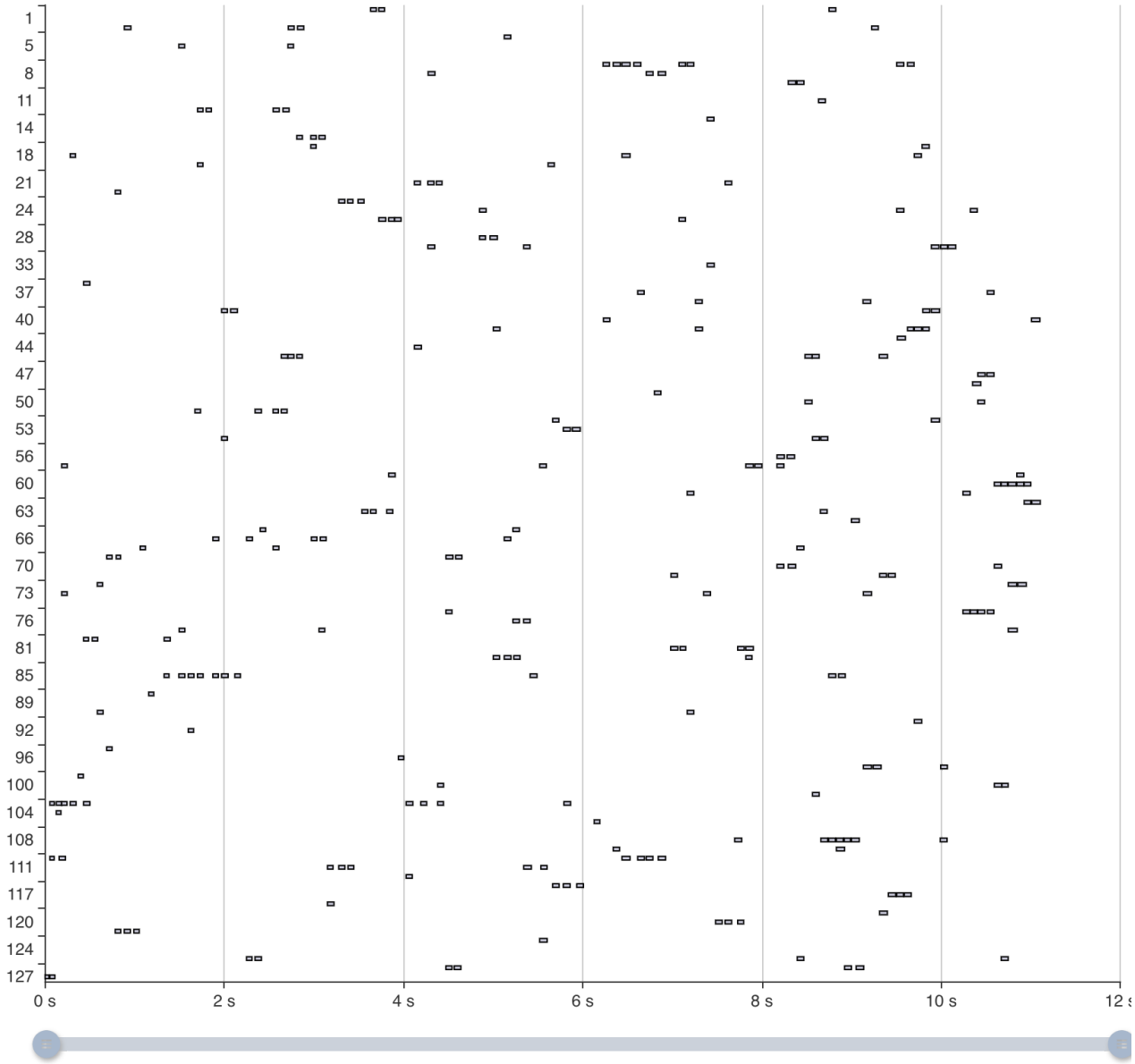
As expected there is more parallel CPU being used in this case as nothing needs to wait for dependencies. But we also notice that the individual tasks tasks take considerably longer than the modular compile tasks.

> The initial delays in each job thread is due to the limited build simulation. There's a single mutex locked data queue that all the job threads pull from. Which creates a race for all of them to get in line for jobs.

To extrapolate lets also look at the same build except at a DAG depth of 100.

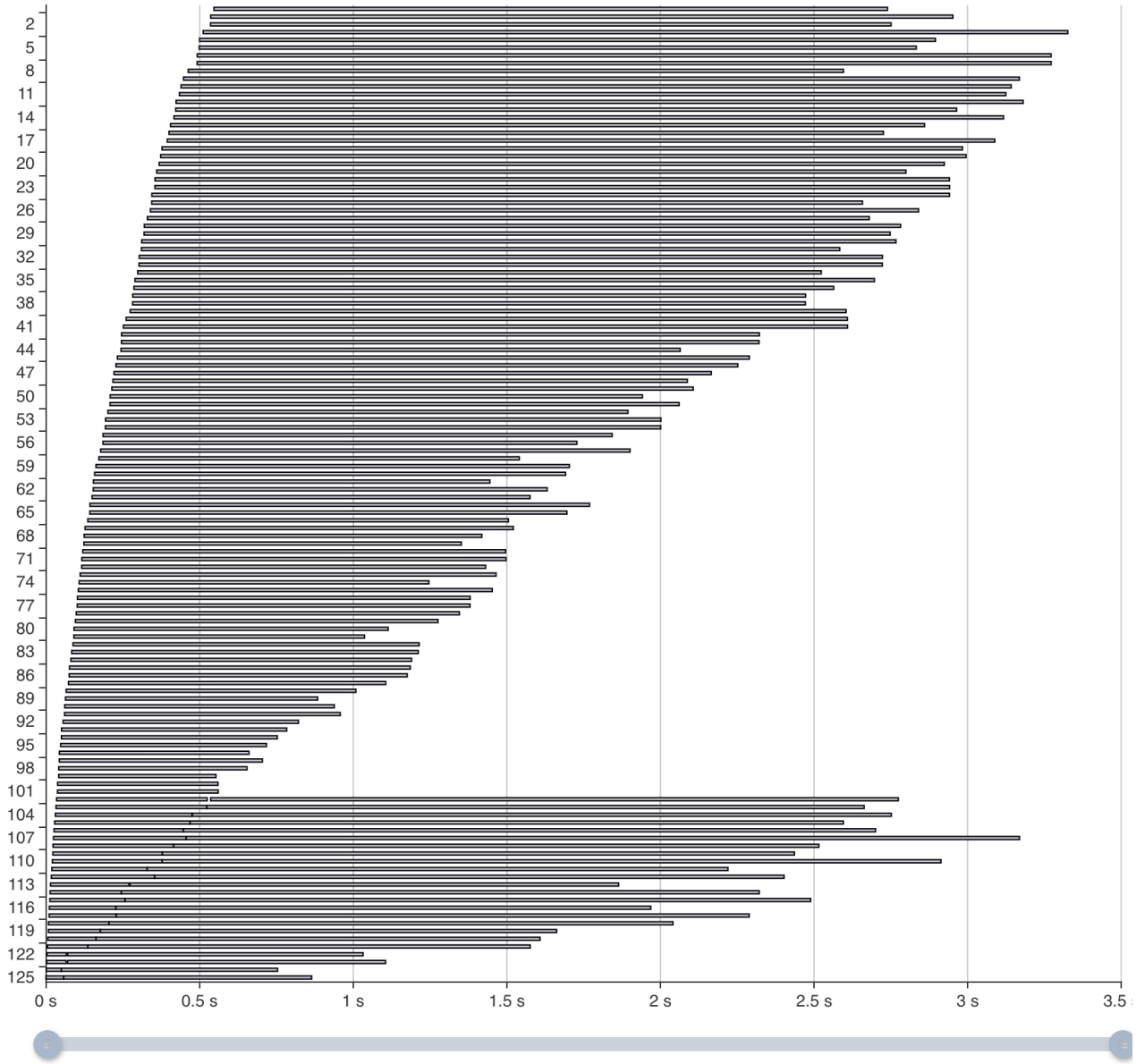
Execution, Modular, Depth 100

GCC135: POWER9, altivec supported 2.2 (pvr 004e 1202) @ 2.166GHz (2 CPU, 32 cores, 128 threads)



Execution, Non-Modular, Depth 100

GCC135: POWER9, altivec supported 2.2 (pvr 004e 1202) @ 2.166GHz (2 CPU, 32 cores, 128 threads)



We can now see how the modular builds delay the execution of tasks as the dependency chain grows longer.

5. Conclusion

With the limitations of the capabilities of current compilers one can only conclude that modular builds are very advantageous at lower parallelism levels environments. But that it's unclear if they are an advantage in highly parallel build environments. In other words, that modules currently do not scale in the same ways as traditional compilation.

6. Acknowledgements

Thanks to Nathan Sidwell for the work to implement the latest merged modules proposal in GCC. And the citizens of the Internet and more specifically the contributors to StackOverflow for the hints that made it possible to decipher how to get the GCC svn checkout to build in OSX.

And many thanks to Mathias Stearn for the logic on how to compile the Clang modules. And Ninja generation for comparison in a different build system context.

7. References

GCC cxx-modules implementation, Nathan Sidwell https://gcc.gnu.org/wiki/cxx-modules (https://gcc.gnu.org/wiki/cxx-modules)

C++ Tooling Stats, Rene Rivera https://github.com/bfgroup/cpp_tooling_stats (https://github.com/bfgroup/cpp_tooling_stats)