Fast distributed compilation and testing of large C++ projects

Rosen Matev[1,*]

[1]CERN

Abstract. High energy physics experiments traditionally have large software codebases primarily written in C++ and the LHCb physics software stack is no exception. Compiling from scratch can easily take 5 hours or more for the full stack even on an 8-core VM. In a development workflow, incremental builds often do not significantly speed up compilation because even just a change of the modification time of a widely used header leads to many compiler and linker invokations. Using powerful shared servers is not practical as users have no control and maintenance is an issue. Even though support for building partial checkouts on top of published project versions exists, by far the most practical development workflow involves full project checkouts because of off-the-shelf tool support (git, intellisense, etc.)

This paper details a deployment of distcc, a distributed compilation server, on opportunistic resources such as development machines. The best performance operation mode is achieved when preprocessing remotely and profiting from the shared CernVM File System. A 10 (30) fold speedup of elapsed (real) time is achieved when compiling Gaudi, the base of the LHCb stack, when comparing local compilation on a 4 core VM to remote compilation on 80 cores, where the bottleneck becomes non-distributed work such as linking. Compilation results are cached locally using ccache, allowing for even faster rebuilding. A recent distributed memcached-based shared cache is tested as well as a more modern distributed system by Mozilla, sccache, backed by S3 storage. These allow for global sharing of compilation work, which can speed up both central CI builds and local development builds. Finally, we explore remote caching and execution services based on Bazel, and how they apply to Gaudi-based software for distributing not only compilation but also linking and even testing.

1 Introduction

The LHCb physics software stack builds on top of Gaudi [1] and is comprised of a number of interdependent projects, some of which provide libraries while others define applications (see Figure 1). The largest part of the codebase is written in C++, which also consumes the most resources while building the software. The LHCb software projects inherit the Gaudi build system, which is based on CMake.

The typical complexity of building the software can be illustrated as follows. If we consider Gaudi alone, at present its build defines about 600 targets out of which 450 are object

*e-mail: Rosen.Matev@cern.ch

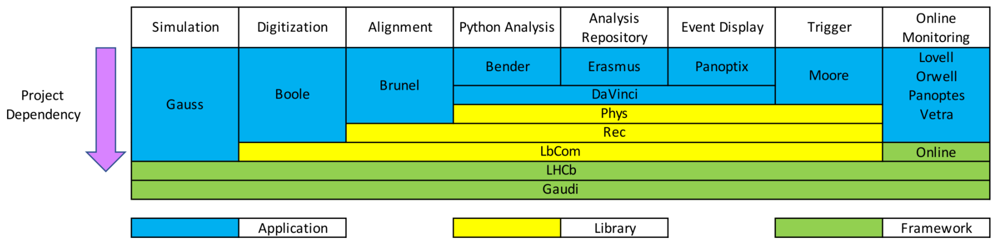

Figure 1. LHCb physics software stack.

files produced by the C++ compiler. A more complete example is the LHCb trigger application, called Moore, which together with all dependent projects (up to and including Gaudi) defines about 4800 targets, out of which 3500 are object files. It is worth noting that there are very few dependencies that limit the concurrency of building those object files, which means that the bulk of the build can be parallelised if many cores are available.

Developing on such a codebase can prove difficult due to the amount of resources required for building. Typically, development happens on hosts that are readily available but often not very powerful, such as LXPLUS, 4-core CERN OpenStack virtual machines (VMs), desktop PCs. After building once, incremental builds are supported, but often the increments are very large due to modifications in headers or when switching branches. To combat the limited resources, custom tools exist for working on partial checkouts on top of released project versions [2], however they are not well suited for other than trivial changes due to, for instance, the lack of good support for standard git workflows, or the need to checkout a large chunk of the stack.

Automated nightly builds suffer from a similar problem. The machines they run on are also quite limited (8 core VMs) and they always configure and build from scratch in order to ensure consistency. It is clear that faster iteration and feedback, both locally and in automated builds, can help significantly the software development speed.

Throughout the following a test case is used of building Gaudi alone on a 4-core CERN OpenStack VM. Only the build step is timed, which takes about 13 min using six simultaneous jobs.

2 Caching build products

When building locally or centrally, the exact same compilation work is easily repeated since often few or no sources change with respect to the last time the build was performed. An example is clean nightly builds when nothing was merged or temporarily switching branches locally.[1] Therefore, caching of the compilation can be advantageous and can be achieved with tool called ccache [3]. ccache is a tool that aims for speed, is easy to use and is widely used, including in the CI builds of Gaudi and in the LHCb nightly builds [4, 5].

The main task of a compilation cache is to determine whether an object file was already build by calculating a lookup key (hash) based on sources, compilation flags, etc. In the case of compiling C++, the main difficulty stems from the preprocessing where include directives are expanded. The simplest mode of operation of ccache is the so called "preprocessor mode", in which the C preprocessor is run and its output is used to compute the hash, as shown in

[1]Build systems such as Make or Ninja rely on the modification timestamp of files to figure out whether rules need to be rerun. As a result, when one temporarily switches branches with git, the timestamp of files will be modified, necessitating a rebuild even if the sources are strictly identical.

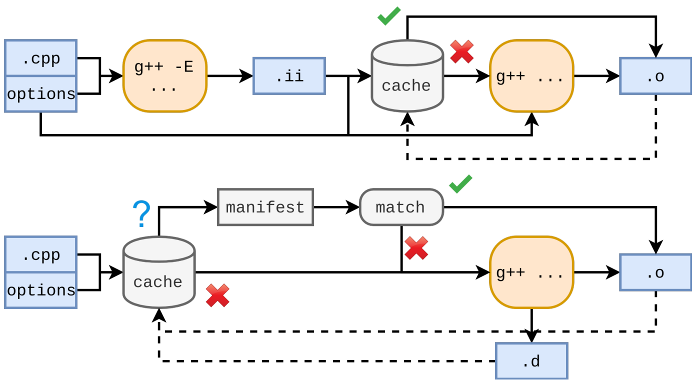

Figure 2. Schematic view of the (top) "preprocessor" and (bottom) "depend" mode of ccache. In the former the C preprocessor is invoked every time, while in the latter the dependencies produced by the compiler are used.

Figure 2. Using this mode leads to a fourfold improvement in building Gaudi with a hot cache at a price of about 30% overhead when the cache is cold. This unimpressive performance is easily explained by the fact that preprocessing, done for every target, is slow.

On the other hand, the default "direct" mode runs the preprocessor only on a cache miss, in which case its output is parsed to find the include files that were read. The content of the latter is hashed and included in a "manifest" in the cache so that future lookups can compare their content. Using this mode provides another fourfold improvement when the cache is hot. Finally, a similar performance is measured with the newly introduced "depend" mode, which avoids the preprocessor altogether by passing the `-MD` flag to the compiler to gather the header dependencies (see Figure 2). The advantage of the "depend" mode will become apparent in the following section.

3 Distributed compilation

Often within an organisation there are CPU resources that are not directly accessible to every developer, such as desktop PCs or restricted, shared machines. distcc [6] is "a fast, free distributed C/C++ compiler", almost as old as Gaudi, which allows offloading compilation to some voluntary or dedicated resources. In this work, distcc 3.2rc1 was installed on two servers running CentOS 7, both hosting a dual socket Intel® Xeon® E5-2630 v4 (2.20 GHz) and totalling to 80 logical cores. The distcc daemon listens on a TCP port, open to the internal network, which provides the lowest overhead per compilation. The daemon is compiled with GSSAPI authentication support and each host is registered to the CERN central database (KDC). This allows for clients to authenticate using a kerberos token and for the server to apply a whitelist of allowed users.

To profit from the many available cores, the Ninja build system is instructed to run 100 jobs in parallel. Ninja "pools" are used to limit concurrency for non-compilation targets, which cannot be distributed, such as linking. The standard mode of operation of the distcc client is to run the preprocessor locally and only send its output to the server for compilation, as seen in Figure 3. However, the local preprocessing represents a bottleneck that leads to underutilisation of the remote servers, achieving only about a fourfold improvement in build time. distcc can of course be combined with ccache, but care has to be taken to limit local

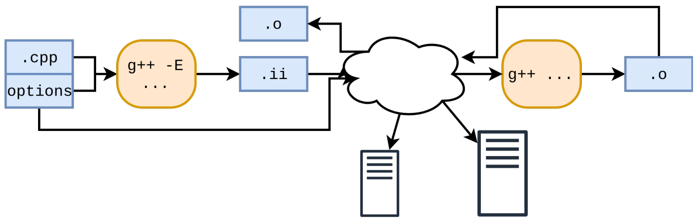

Figure 3. Schematic view of the standard mode of operation of distcc. The C preprocessor runs locally and only the compilation of its output is distributed.

preprocessing in order not to overload the host. For the ccache modes where this is relevant the local concurrency was limited using GNU parallel [7].

In order to overcome the bottleneck of local preprocessing, the distcc "pump" mode can be used. In that mode, an "include server" runs locally and statically analyses the source to be compiled to find the transitive header dependencies. The lifetime of the include server spans the entire build in order to gain efficiency by memoising sub-trees while recursing the headers. The headers are sent to the compilation server along with the source file, such that preprocessing can happen there. An exception is made for system headers, which are expected to be the same on the server. This rather stringent requirement is ensured in our case by LCG releases being distributed by the globally shared CernVM File System (CVMFS) and by installing the common operating system dependencies using the `HEP_OSlibs` meta package. The fact that the LCG releases are available on the compilation server improves both the local analysis of headers and also lowers the amount of data that needs to be transferred.[2] An optimised build without debug symbols generates only moderate upstream (downstream) traffic peaking at 1.5 (4) MiB/s and totalling 40 (75) MiB.

The pump mode couples well with the depend mode of ccache, entirely avoiding local preprocessing. In this configuration, a build with a cold cache takes 90 s, with very little resources used locally and full utilisation of the 80 remote logical cores. As shown in Figure 4, out of the 90 s, only about 60 s are needed to compile the object files, while the rest is taken by linking and other local rules.

4 Distributed cache

Ideally, the compilation cache will be shared between hosts, those of developers or those used for centralised builds. However, there are a number of difficulties in sharing a cache. For instance, developers can use different directories and some information included in the lookup key (hash) will correctly contain absolute paths, e.g. in the case of including debug symbols. ccache covers such cases but at present relies on a local file system for its cache. A way around that is to use a shared file system such as NFS, but the access to one and its performance are not obvious issues to solve.

sccache [9] solves this problem by natively supporting remote storage such as S3, Redis or others. Similarly to ccache, it is used as a compiled wrapper but has a client-server architecture for efficient communication with the remote storage. It even comes with distributed compilation support. However, unlike ccache, the lookup key (hash) determination of sccache is quite simple (e.g. no support of options like `base_dir`), which results in fewer

[2]A small patch to the distcc include server was necessary to support the exclusion of arbitrary paths such as `/cvmfs`.

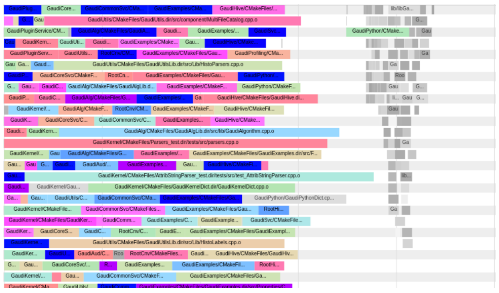

Figure 4. Profile of a distributed build of Gaudi extracted from the Ninja log and visualised with Chromium's trace event profiling tool [8]. The horizontal axis denotes elapsed time and spans between 0 and 100 s. Each row represents one of the 100 Ninja jobs (only the first quarter is shown for brevity). Coloured boxes show remote compilation while grey denotes the execution of local build rules.

cache hits, effectively offloading the cache but not sharing it between developers. In addition, sccache always runs the preprocessor, and as a result performs similarly to the case of using the preprocessor mode of ccache.

Finally, a ccache fork with a memcached remote storage backend is investigated. The fork does not include recent ccache developments such as the depend mode. Nevertheless, it demonstrates excellent performance with a hot cache being only about 10% slower compared to a local cache. A generic storage extension mechanism via plugins is planned for ccache, thus it can be expected that memcached or other remote storage backends will be available in a future ccache release.

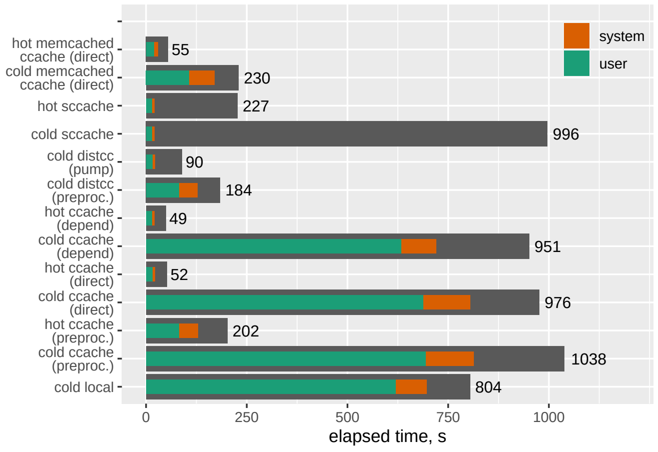

Figure 5. Elapsed time in seconds of all performed benchmarks. The user and system CPU time normalised to the number of cores (4) are shown with green and red, respectively. Cold (hot) signifies a build with 0% (100%) cache hits.

5 Conclusion

The aforementioned toolchain improvements, especially the distributed compilation, is well received by developers, who effectively gain access to powerful machines without the need move development there. The server logs show that about 15% of compilations do not finish successfully, where most are due to user interruption, some are due to authentication failures and only a small part is because of timeouts due to server overloading. The results of all benchmarks performed as part of this work are shown in Figure 5.

Both ccache and distcc are nearly twenty years old but are still maintained and very useful. Compared to a local build using four cores, a nearly tenfold reduction of Gaudi build time is observed when distributing compilation to 80 cores. Similar gains are seen across the LHCb software stack. Such developments have the potential to speed up automated builds, typically constrained on limited-resource VMs, where timely feedback is also essential.

More modern software is starting to gain traction, such as sccache that was explored here. Another very promising example is the Bazel build system [10], which has the goal of fully specifying the target inputs and outputs. Among other benefits, this allows for the remote execution of rules and caching of targets, which in turn provide safer (hermetic) and faster build and test execution, and a consistent environment for developers. The ecosystem is relatively mature, with a number of self-hosted remote execution services existing to date. However, using Bazel for the Gaudi-based LHCb software would mean a complete rewrite of the existing CMake build. One particular complication for C++ projects is that one cannot specify dynamic dependencies (i.e. those unknown at declaration time, but known at build time, such as header dependencies of source files), which makes a potential migration particularly complicated.

References

[1] G. Barrand et al., Comput. Phys. Commun. 140, 45 (2001)

[2] M. Clemencic, B. Couturier, J. Closier and M. Cattaneo, Journal of Physics: Conference Series 898, 072024 (2017)

[3] https://ccache.dev/

[4] K. Kruzelecki, S. Roiser and H. Degaudenzi, Journal of Physics: Conference Series 219, 042042 (2010)

[5] M. Clemencic and B. Couturier, Journal of Physics: Conference Series 513, 052007 (2014)

[6] https://github.com/distcc/distcc

[7] O. Tange, ;login: The USENIX Magazine, February, 42-47 (2011)

[8] https://www.chromium.org/developers/how-tos/trace-event-profiling-tool

[9] https://github.com/mozilla/sccache

[10] https://bazel.build/