PARALLEL COMPILATION OF CMS SOFTWARE

Shaun Ashby, Giulio Eulisse, Lassi A. Tuura (CERN, Geneva, Switzerland)
Stefan Schmid (ETH Zurich, Switzerland)

Abstract

LHC experiments have large amounts of software to build. CMS has studied ways to shorten project build times using parallel and distributed builds as well as improved ways to decide what to rebuild. We have experimented with making idle desktop and server machines easily available as a virtual build cluster using *distcc* and *zeroconf*. We have also tested variations of *ccache* and more traditional make dependency analysis. We report on our test results, with analysis of the factors that most improve or limit build performance.

INTRODUCTION

The development of big software projects requires frequent recompilation for rapid prototyping. We have developed a system to perform distributed compilation using spare cycles of available machines (running GNU/Linux).

This document is organized as follows: In this introductory part we first describe the "classic" compilation process and point out which parts will be parallelized by our system. We then give some background about the theory of parallelization in general which is crucial to understand the potential bottlenecks for the speed-up. Finally we present the build environment at CMS and also the infrastructure on which our tests have been performed. The next section then introduces our compilation system and its components. After the presentation of the performance tests, the document concludes with a short summary of our findings and an outlook to the future.

Phases of Compilation

Figure 1 outlines the phases of compilation. The C-preprocessor expands the macros in the C or C++ source file and does header files include processing. The generated intermediate file is the input file of the C or C++ compiler.[1] In the next step, the assembler generates the binary object file, which is finally linked with other object files and the libraries. The result is the executable program or shared library.

The compilation and the assembly phase depend only on the preprocessed input and are therefore self-contained. These two steps are parallelized in our system.

[1]Modern compilers usually integrate the preprocessor, so there is no intermediate file in practice.

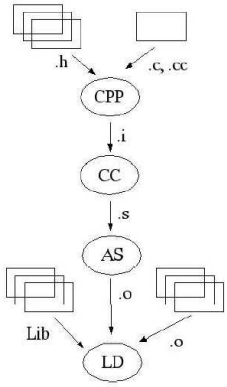

Figure 1: Phases of compilation.

Parallel Algorithms

The *speed-up* of a parallel algorithm is defined as the execution time of an algorithm on a single machine, divided by the execution time on a cluster of machines, i.e. $speedup = \frac{t_{single}}{t_{cluster}}$.

There is no algorithm which can be parallelized ad infinitum. Every algorithm has parts that have to be executed serially, for example we can not distribute a single hard disk read. Given an algorithm with a fraction f of the serial execution time that can not be parallelized, according to *Amdahl's Law* [2], having P CPUs, the speedup is at most

$$speedup_{max} = \frac{1}{f + \frac{1-f}{P}}.$$

So even for infinitely many CPUs the speedup can not exceed $\frac{1}{f}$.

In our case of distributed compilation, we find several factors that limit parallelization:

- We distribute the work on a *per file* basis, and each file is compiled on a machine sequentially.
- In addition to compilation, we have two other tasks: pre-processing and linking.[2] Whereas we can compile in parallel, with a granularity of entire files, these two tasks are not only inherently serial, but are in our case also done on the same machine, that is, on the local host. However, several independent libraries or executables can be linked simultaneously, but this will happen on a single computer.
- Very practical issues reduce the gain of additional

[2]And SCRAM, see later.

hosts even further. For example the networking overhead increases with the number of participating hosts.

In order to maximize the speed-up, it is crucial to understand where these bottlenecks are and how the number of serial steps can be minimized.

Environment at CMS

There are two different build systems at CMS (see [1]): SCRAM V0.20, the "old SCRAM," and SCRAM V1, the "new SCRAM." While SCRAM V0.20 has several bottlenecks for parallelization — for example, it recursively descends into each subdirectory — SCRAM V1 makes more standard use of make and is very fast.

For our tests we used a cluster of five Intel P4 Xeon 2.8 GHz (two CPUs each) hosts running GNU/Linux. The bandwidth is more than 10 MBytes/s and a ping takes around 0.1 ms. The local compiler of the machines is GCC 3.2.

The project's source files are held on local disk, but the header files except for the system headers and the libraries are located on the network file system AFS.

SYSTEM DESCRIPTION

In this section, we present the different components of our system: The distributed compiler, the service discovery protocol which allows to find the idling host that may participate in the build, the compiler cache, and the daemons which make use of the service discovery mechanism and decide whether the underlying machine is busy or not. Finally, the issue of integration into the build system SCRAM is addressed.

Distributed Compiler

Our system uses the distributed compiler *distcc* [3]. Given a list of server hostnames, either in a file or in an environment variable, it distributes compilation to these hosts and collects the results. The remote machines use a locally installed compiler, so we have to ensure that incompatible versions are not mixed.

Distcc preprocesses the source files locally and then ships the output to the server hosts. This ensures correctness, but it is also more expensive compared to remote hosts preprocessing the sources themselves. On a low-bandwidth network, shipping preprocessed sources may become a bottleneck.

If distcc encounters a problem while connecting to the remote host, it removes the server from the list and masks the error by compiling locally. It is therefore not a problem if the host list contains crashed hosts.

Note that distcc does not check whether it is really worth to compile remotely.

Service Discovery

To discover the hosts currently not busy and therefore able to help in compilation, we use a service discovery protocol, called *zero-conf* [4]. In our setup, all hosts in a cluster run a daemon which automatically updates the host's registration in a DNS server depending on how busy the underlying machine is. When a client wants to compile a project, it simply looks up the registered servers and passes their names to distcc. We use the multicast DNS server *PyRendezvous* [4], but it is possible to use a central DNS server as well.

Compiler Cache

We studied also the effects of using the compiler cache *ccache* [5] in our system. Ccache acts as a caching preprocessor to C/C++ compilers, using the -E compiler switch to get the output of the preprocessor and a hash to detect if a compilation can be satisfied from cache. The present version of ccache has the disadvantage that it considers the whole path of the files only, which means that we can not benefit from the cache when we compile in a different directory.

Daemons

There are two processes running on every host in the system. A "server daemon" applies different strategies to observe its underlying host. Whenever the host is idling, the daemon registers the machine and the machine's local compiler version using the service discovery protocol. There are several possible strategies to decide whether a host is idle: if the screen-saver is running, if less than a certain number of users are logged at the machine, if the CPU-usage falls beyond a threshold, etc.

The "client daemon" periodically checks for available servers; as the servers have to use the same gcc-compiler as the client, it looks for the machines offering the same version as it has itself. The daemon writes the names of the idling hosts into a file. When distcc is called, it uses this file instead of performing a service discovery. As distcc is able to handle crashed hosts, it is not necessary for the host list to be completely up-to-date.

Integration into SCRAM

The integration of the distributed compiler into SCRAM V1 is simple: one can just override the corresponding variables and execute `scram build MAKEFLAGS=-jX CC=... CXX=...` Here, the -j-option sets the level of parallelization: the maximum number of concurrently spawned processes. The CC and CXX variables stand for the C-compiler and C++-compiler respectively and can be pointed to distcc.

For SCRAM V0.20 a different approach is necessary. We instructed SCRAM to use a script called pgcc as the compiler. While the old SCRAM usually waits for the compiler to return before compiling the next file, we trick it

to parallelize compilation as follows: pgcc spawns another process which calls distcc, while pgcc itself returns immediately. This fools the old SCRAM to behave as if the first compilation job was already done and starts the next compilation. Of course, for the linking process we have to wait until all source files are compiled using lock files.

Hence, the situation for old SCRAM is as follows: We compile the source files of a module in parallel, resynchronize before linking, and then start to compile the next module. If the project has no inter-module dependencies (for example COBRA [6], but not IGUANA [7]), it is not necessary to compile one module after another and we can spawn a new process also for linking, letting pgcc return immediately.

We observed that the old SCRAM and make spend a lot of time doing dependency checking, that is, to find out which files can be reused from the previous execution. While this is useful in a situation where we use a local compiler, it is a bottleneck for distributed compilation, because this work can not be parallelized. For this reason, pgcc forces SCRAM not to do these checks by providing a trivial dependency output for each source file consisting of the object file, the source file and a dummy target.

PERFORMANCE TESTS

In the tests presented in this section, all hosts have been idling and we did not run the service discovery protocol in order to concentrate on the pure speed-ups of distributed compilation.

SCRAM V0.20

We first tested IGUANA 4.5.0 [7], see Figure 2. The speed-up is quite small, and we see that the serial components “linking” and “rest” (consisting of time spent for SCRAM, preprocessing and networking) make up almost half of the execution time. As has been mentioned before, the old SCRAM changes into the corresponding directory of every module and then calls `scram b` recursively, which is expensive and is a crucial reason for the poor parallelization.

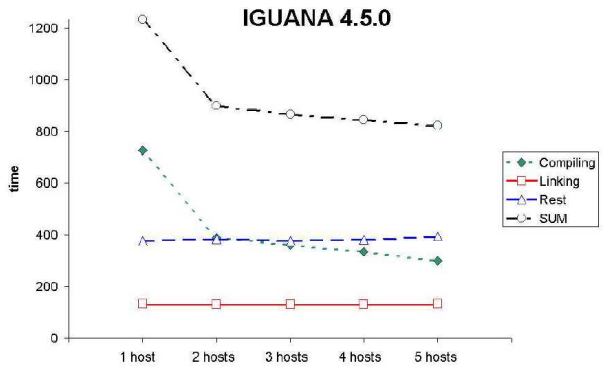

Figure 2: IGUANA 4.5.0.

For COBRA 7.5.0 [6] which has no inter-module dependencies, the speed-up is moderate (cf. Figure 3), even in the case of a wrapped SCRAM which gets rid of a part of the serial component. Finally, we tested ccache (full cache),

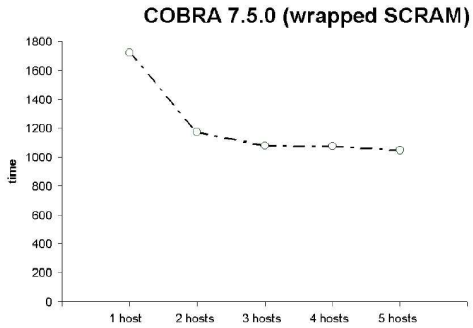

Figure 3: COBRA 7.5.0.

which yields good results, see Figure 4.

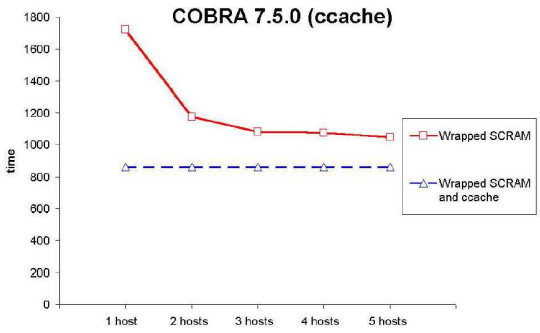

Figure 4: Ccache for COBRA 7.5.0.

SCRAM V1

New SCRAM has just one makefile and is not recursive. It reduces the time-stamping work and uses different algorithms for dependencies. The overhead of SCRAM turned out to be less than one second. We tested a prototype version of new SCRAM for the projects SEAL 1.3.3 [8], POOL 1.5.0 [9] and COBRA 7.6.2. The results are shown in Table 1. Unlike the former test, the sources here are also located on AFS. Obviously, the speed-up for SEAL is still very small. The reason for this is that the generation of SEAL’s dictionary is a huge serial component. Moreover, SEAL has many small source files which reduces the speed-up even further. The results for POOL and COBRA are quite good.

Table 1: Speed-Up for SCRAM V1.

project	**1 vs. 10 CPUs**
SEAL	2.03
POOL	4.11
COBRA	4.4

Figure 5 gives the detailed test for COBRA. It shows that more than five hosts may still be useful. Moreover, from our tests, we can find a reasonable rule of thumb for the -j option: $\frac{3}{2} \cdot$ # CPUs.

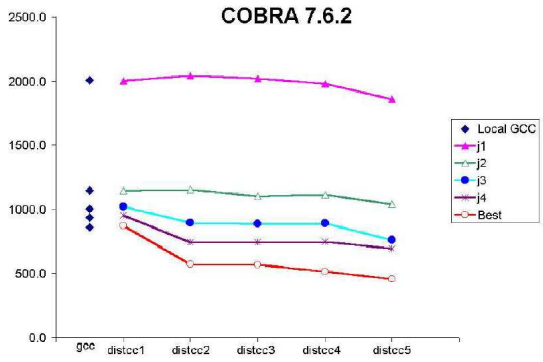

Figure 5: COBRA 7.6.2.

CONCLUSIONS

From our studies we can draw several conclusions:

- In most cases, one additional host reduces the execution time remarkably. The utility of a third or forth machine is less obvious.
- It is crucial to reduce serial components like preprocessing, SCRAM, networking, and so on to get a good speed-up.
- SCRAM V0.20 and the generation of SEAL's dictionary are significant bottlenecks for parallelization.
- The speed-up depends on many parameters, which are not only related to the computer infrastructure (number of hosts, latency, bandwidth...) but also to the project's properties (e.g. number and size of files). It is not possible to give a formula for an arbitrary project to calculate lower bounds of performance gains.
- Pragmatic rules of thumb have to be applied to predict certain system parameters. For the -j-option, we recommend $\frac{3}{2} \cdot$ # CPUs.
- Integration of ccache is easy and useful.

With the present technologies, the distribution of compilation jobs to idling hosts provides only moderate speed-ups, i.e. a factor of two can hardly be achieved even with dozens of desktops. However, as will be discussed in the next section, this may change in the near future.

In our system, the sources have usually been local, but all the headers (except the system headers) and libraries were on AFS. Although AFS caches files, many operations take more time on AFS than on local disk. Therefore, a completely local compilation may result in better performance than in our environment.

FUTURE WORK

Besides using local header files and libraries, we can identify several measures to reduce serial execution times further. The following sub-sections present two interesting technologies which may be useful to speed-up the compilation process further.

Moreover, a completely different approach for distributed compilation is possible, e.g. by using MOSIX [10] which distributes the compiling processes automatically on a cluster — without distcc. However, in contrast to distcc this solution requires changes at kernel level.

Pre-compiled Header Files (PCH)

It is possible to pre-compile header files [11], so they have not to be processed over and over again if they appear in many source files. To use PCH, it is necessary to change the makefiles.

Compiler Server

Apple is working on a compiler server [12] which also allows reuse of compiled headers. The idea is that the compiler will run as a kind of a server process which waits for compilation requests and always checks if the file has already been compiled. A compiler server does not just remember the text, but stores the semantic data trees resulting from the header files in memory. This is a very sophisticated approach compared to PCH. However, this new version of gcc is still under construction.

REFERENCES

[1] http://cmsdoc.cern.ch/Releases/SCRAM/current/doc/html/SCRAM.html.

[2] J. Hennessy, *Computer Architecture. A Quantitative Approach* (2002), Morgan Kaufmann Publishers, USA.

[3] http://distcc.samba.org.

[4] http://dotlocal.org.

[5] http://ccache.samba.org.

[6] http://cobra.web.cern.ch/cobra/.

[7] http://iguana.web.cern.ch/iguana/.

[8] http://seal.web.cern.ch/seal/.

[9] http://pool.cern.ch.

[10] http://www.mosix.org.

[11] http://gcc.gnu.org/onlinedocs/gcc/Precompiled-Headers.html.

[12] http://per.bothner.com/papers/GccSummit03-slides/.