Parallel compilation of CMS software

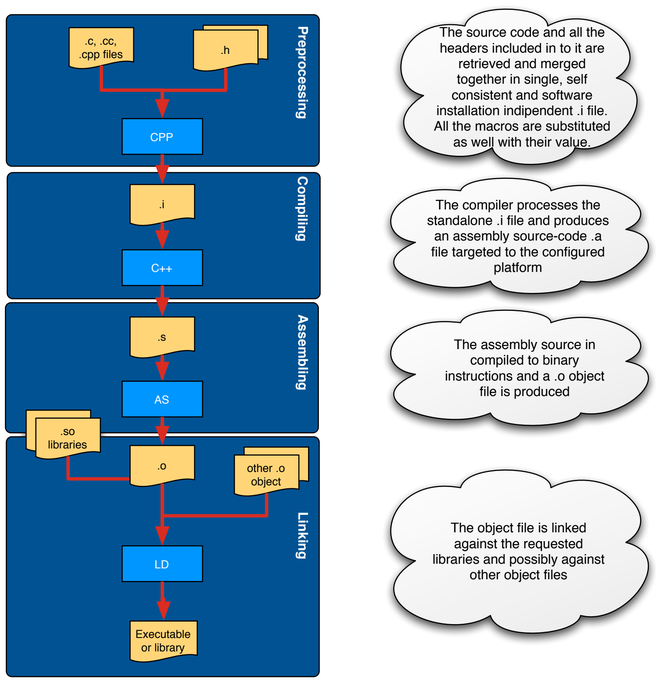
Giulio Eulisse*, Stefan Schmid**, Lassi A. Tuura*

*) Northeastern University, Boston (MA), U.S.A.
**) Eidgenössische Technische Hochschule (ETH), Zürich (ZH), Switzerland

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.

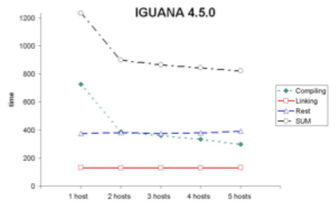
The compiling process

Albeit often seen as a single operation the compiling process -- or, better, the building process -- is actually made up of four different phases: preprocessing, the actual compilation, assembly and linking.



Performance test with scram V0.20

CMS uses a custom tool called SCRAM for building its software.
The current production version of SCRAM V0.20 adds several bottlenecks for parallelization besides the intrinsic ones present in the compilation task as such.
For example because of its recursive nature, Perl is started over and over again, adding overhead to the process.

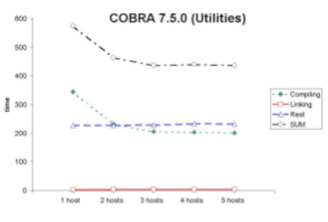


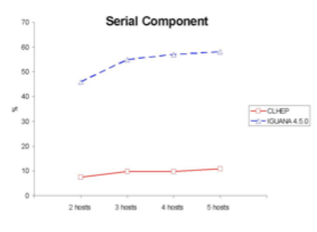
The picture above shows the building time for IGUANA 4.5.0 -- the interactive framework used by CMS -- also separating the different contributions to the building time.
As it can be seen, only the pure compilation can be sped up, the remaining components 'Linking' and 'Rest' --- consisting of SCRAM, preprocessing, and networking --- contribute a significant part to the whole execution time and are not sped up.

In contrast to IGUANA, COBRA -- CMS application framework -- has no inter-module dependencies which allows to compile different modules in parallel.

However, even if SCRAM is wrapped, the speedup with more then 2 hosts is small.

Finally, we integrated also the compiler cache ccache which yields a very good performance, see Figure "Cobra 7.5.0 (ccache)"
(cache hits only).





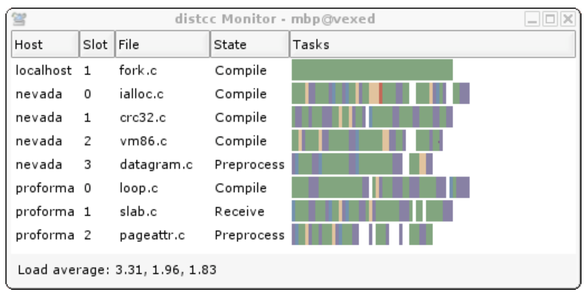
Parallelization of the compilation process (as performed by distcc)

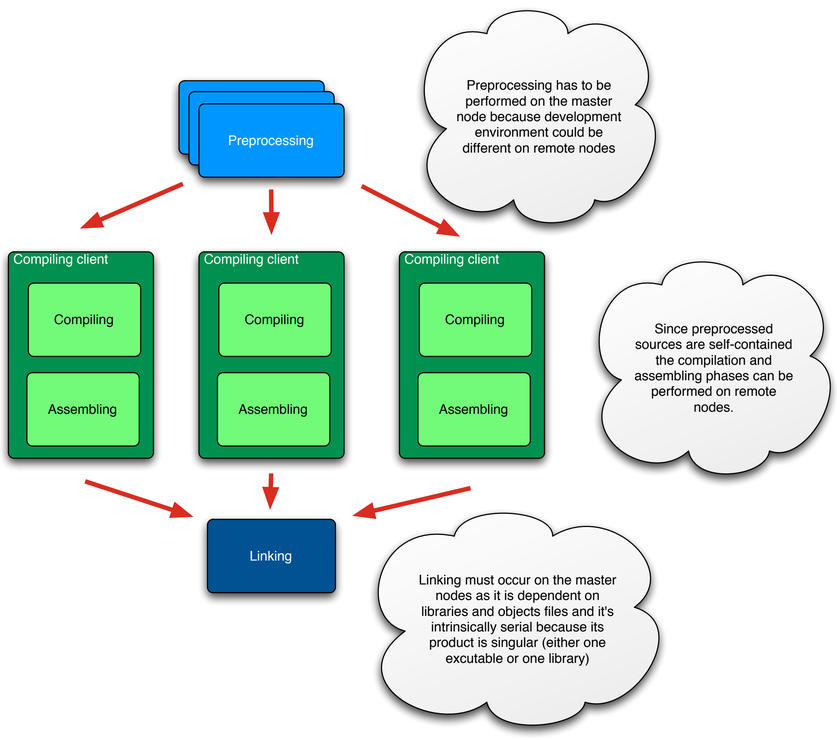
The problem of improving software building speed is probably the second most common problem in software development (second only to having the source actually compiling). For this reason there are already a number of projects that try to address it by sharing the workload to multiple machine in various ways.

After some investigation we have chosen the opensource tool **distcc** for mainly two reasons:

1) it works and it is easy to install.

2) it is opensource and big corporations (like Apple) are actually contributing to it.





Performance test with SCRAM V1

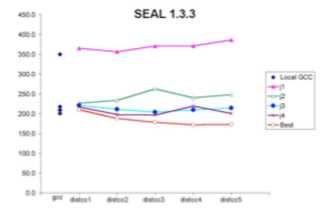
To overcome the bottlenecks of the old version of SCRAM a total rewrite effort of it has been started in late 2003 and it is now entering into "prime time".

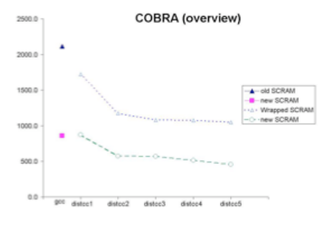
The new version of SCRAM -- dubbed V1 -- abandons the recursive nature of the old one and has only one makefile. This reduces the serial component due to SCRAM time-stamping work and uses different algorithms for dependencies. The overhead of a preliminary version of SCRAM v.1 turned out to be less than one second, which makes it very appealing when compared to the old V0.20 version.

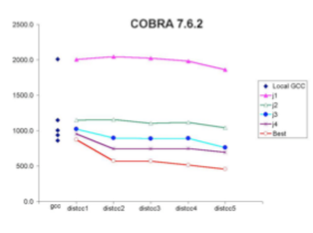
In our test we built the projects SEAL 1.3.3, POOL 1.5.0 and COBRA 7.6.2.
The result can be found in the following table.

project	1 vs 5 hosts	1 vs 10 CPU's
SEAL	1.17	2.03
POOL	1.87	4.11
COBRA	1.92	4.4

As it can be seen, the speedups are very different.
This is due to the dictionary generation that occurs in SEAL building process which cannot be parallelized using distcc.
COBRA, which does not have such a problem shows a good speedup.







Theory of Parallel Algorithms

The performance gain obtained by parallelizing a job on a cluster of machine is measured in terms of a quantity called speed-up which is defined as the ratio between the time required to perform the task on a single machine and the one obtained by doing it on a cluster:

$$speedup = \frac{t_{single}}{t_{cluster}}$$

Naively thinking, one could assume that by parallelizing a job on n different machines one could get a n-fold improvement in performances.
This is not actually true because every task has an intrinsic serial component, *f*, that cannot be parallelized.
This results in having the actual maximum speed up being:

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

so that even considering an infinite number of CPUs the maximum speedup is limited by the serial component *f*.

$$speedup_{\infty} = \frac{1}{f}$$

Automatic compiling peers discovery

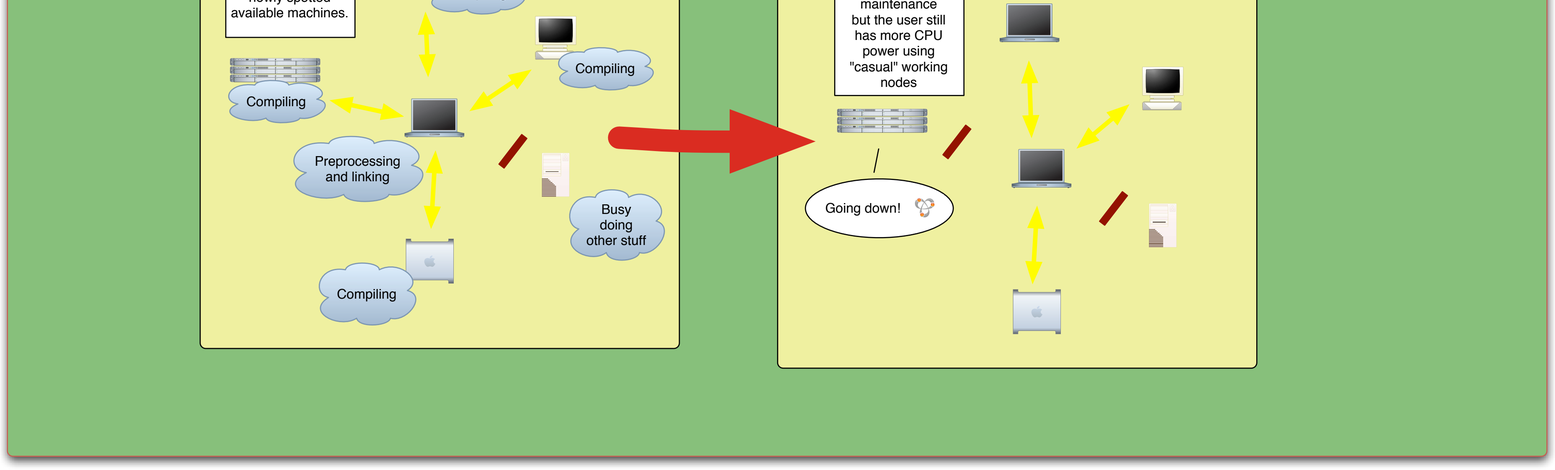
As is, distcc works very well in static environments where the peers that share the compilation workload are well defined.
This is not always the case, because it's not always possible to afford a separate building farm. Moreover one would like to take advantage of occasional spare resources that might appear under certain circumstances and he would like that this happened in automatic way.

For this reason we developed a simple server that, once installed on a client machine, would notify the network neighbourhood about the availability of the client once a certain policy-specified condition is satisfied.

This is similar to work already done by Apple to its private version of distcc but since it is not available under Linux we decided to develop our own prototype.

Like Apple we based our server on ZeroConf service discovery mechanism.
ZeroConf (A.K.A. Rendezvous) is an industry standard protocol, mainly developed by Apple Computers Inc., for advertising and discovering of services over IP based networks.

It works by using the well defined DNS-SD protocol and can be used in both server-less environments (via multicast DNS usage) as well as in managed ones.



Every machine in our prototyped system runs two processes and it can actually serve as compiling client as well as it could share it compilation workload with others.
The "server daemon" observes the status of its underlying host. Whenever the host is idling, the daemon advertises the machine as available on the network neighbourhood via multicast.
The "client daemon" notices the availability of new machines and adds them to the lists of compiling hosts,
making sure that they actually have a compatible compiler.

Other enhancements and future directions

As it as been explained, when dealing with parallelization, it is fundamental to reduce the intrinsic serial component of a job.
In the case of compilation a number of different additional tricks could be used, for example the usage of precompiled headers to reduce the preprocessing phase, using a compiler server to eliminate compiler startup time and the usage of compilers caches such as ccache to actually avoid recompiling of sources that have not changed across two different versions of the software.

Conclusions:

From the analysis made above, we can draw the following conclusions:

1. In most cases, one additional host reduces the execution time remarkably. The utility of a third or forth machine is less obvious.

2. It is crucial to reduce serial components like preprocessing, SCRAM, networking, and so on to get a good speed-up.

3. SCRAM V0.20 and the generation of SEAL's dictionary are significant bottlenecks for parallelization.

4. The speed-up depends on many parameters, which are not only related to the computer infrastructure (number, latency, bandwidth, ...) but also to the project's properties (e.g. number and size of its files). It is not possible to give a formula for an arbitrary project to calculate lower bounds of performance gains.

5. Pragmatic rules of thumb have to be applied, as it is difficult to predict which -j-option is best or whether to include local host or not.

6. Integration of ccache is easy and very useful.

7. It is important to avoid simultaneous writes to the same AFS volume. For example, temporary files should not be written to the home directory in a parallel algorithm. With the present technologies, the distribution of compilation jobs to idling hosts provides only moderate speedups, i.e. a factor of two can hardly be achieved even with dozens of desktops.

References

Hennessy, J.: Computer Architecture. A Quantitative Approach. (2002); 3rd Edition, San Francisco: Morgan Kaufmann Publishers

Tanenbaum, A.S.: Moderne Betriebssysteme (1995); 2nd Edition, Munich, Vienna: Hanser, p. 697ff.

Mark Lutz: Programming Python (2001); 2nd Edition, Sebastopol: O'Reilly.

Powers, S., Peerk, J., O'Reilly, T., Loukiedes, M.: Unix Power Tools (2002); 3rd Edition, O'Reilly.

http://iguana.web.cern.ch/iguana/

http://cobra.web.cern.ch/cobra/

http://cmsdoc.cern.ch/orca/

http://cmsdoc.cern.ch/oscar/

http://seal.web.cern.ch/seal/

http://pool.cern.ch

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

http://distcc.samba.org

http://www.zeroconf.org

http://dotlocal.org

http://ccache.samba.org

http://per.bothner.com/papers/GccSummit03-slides/

http://www.mosix.org