Improving Parallelism in git-grep

Matheus Tavares Bernardino
https://matheustavares.gitlab.io
Supervisor: Prof. Dr. Alfredo Goldman

Institute of Mathematics and Statistics of the University of São Paulo

Introduction

A version control system (VCS), as defined in the Pro Git book[1], is *"a system that records changes to a file or set of files over time so that you can recall specific versions later"*. Among the version control tools available today, Git, an open source VCS, has become the most popular[2] for source code management.

Git is composed of many commands, such as add, commit, and grep. The last one is used to find lines in a project's files matching a given pattern. It can search both in files of the latest or earlier versions (reading from the internal *object store)*.

Since Git is used to manage many projects of different sizes, the community focuses heavily on scalability and performance. With that in mind, git-grep was made parallel in 2010. However, for object store searches, multithreading turned out to be slower than the sequential code, so threads were disabled for this case.

The main goal of this project was to **improve git-grep parallelism, especially in the object store case**, allowing to re-enable threads with a good speedup.

Theoretical Background

Git's object store works like an in-disk hashmap, where objects are referenced by the SHA1 hash of their contents. Each object is stored in a compressed format (using zlib). An object with content similar to another can also be stored as a delta.

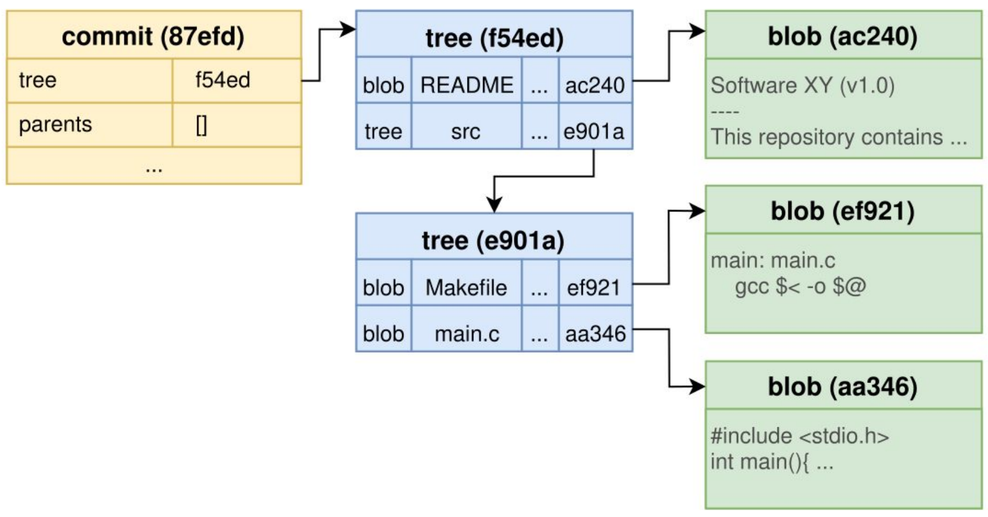

Figure 1: Diagram of some of Git's objects.

Development

A. Preparatory period

Getting to know the codebase, the community and the contribution workflow.

B. Finding the bottleneck(s)

We studied and profiled the code using *gprof* and *perf*. Zlib decompression seemed to be the perfect candidate to run in parallel: really time-consuming, already thread-safe and with multiple independent tasks.

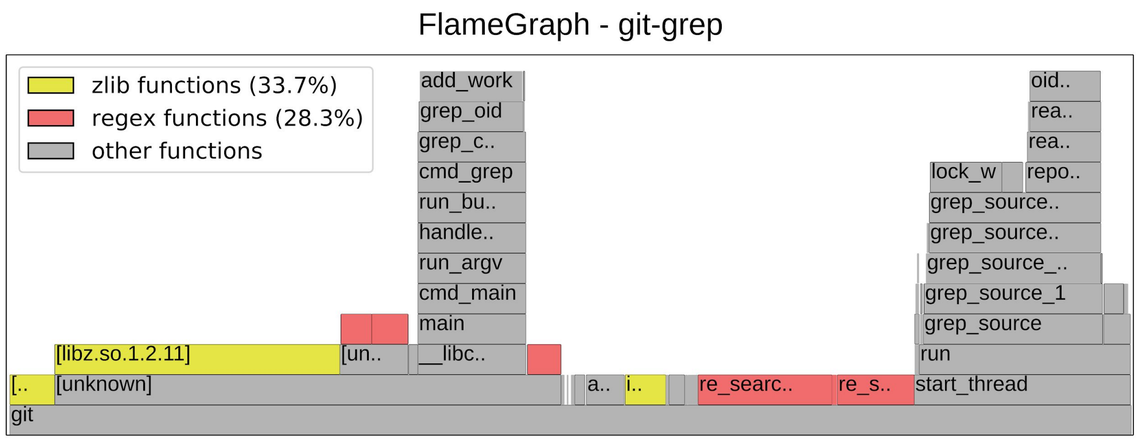

Figure 2: Profile of git-grep in the object store (threads re-enabled) using perf w/ FlameGraph.

C. Allowing zlib decompression to run in parallel

- We protected the object reading machinery from the inside with a recursive mutex (because it has recursive calls) and released the mutex before running decompression, so that it could be performed in parallel.
- Just by opening this hole in the previously one-pieced critical section we risked breaking invariants and introducing race conditions (as shown in figure 3). In fact, this happened with the delta base cache, accidentally allowing duplicate entries. We fixed it with a check before insertion.

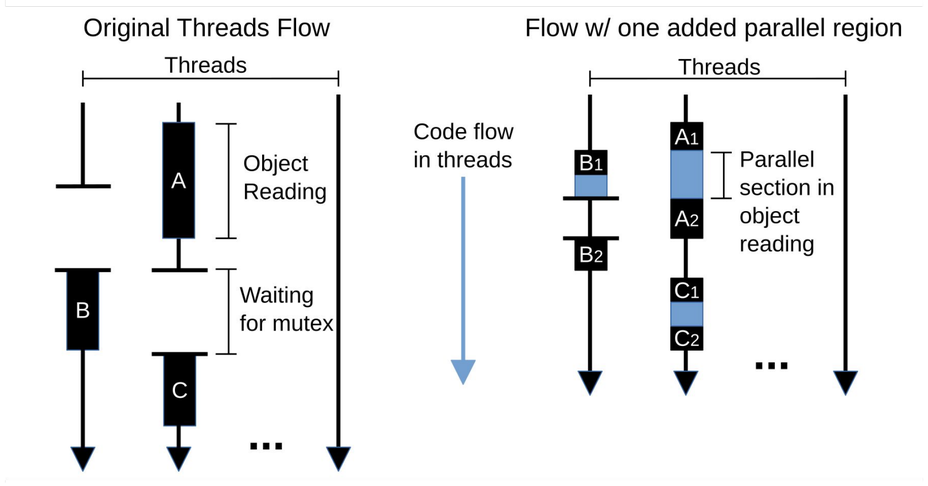

Figure 3: Diagrams of the threads' work in git-grep. The right one represents an introduction of one parallel section in object reading. Note that, if B_1 changes a global resource A_1 set for A_2, the latter might, now, failure if no proper care was taken.

- Replaced git-grep's object reading lock by the internally implemented version. git-grep used its lock to protect object reading **and** other operations that would otherwise conflict with it. So we had to expose the recursive mutex to be used externally, as well.
- Forced some lazy initializers to perform eagerly during git-grep startup to avoid race conditions when threaded.

Results

Tests were performed in a desktop running Debian 10.0 with the following architecture: Intel(R) Xeon(R) CPU E3-1230 V2 (4 cores w/ hyper-threading), 32GB of RAM and a 7200 rpm, SATA 3.1 HDD. We used the Chromium repository as data, for being relatively large in both content and history size. We used the extended regex *'(static|extern) (int|double) *'* for being a realistic use case and producing a quite time-demanding search.

Bellow are the mean elapsed times in intervals with 95% of confidence. In the "Original" code, we re-enabled threads in the object-store case, for comparison.

Conclusions

- Reached **speedups of more than 3.15x** over the original sequential code.
- Provided a safe and optimized way to perform object reading in threaded environments. (also useful for other commands in the future)
- Located and fixed 5 spots in git-grep's code that had possible race conditions.
- What is left: there is still a possible data race in *mmaped* sections of Git's packfiles (used as input to decompression). An idea using rwlocks was recently suggested in the mailing list as a solution, and we plan to work on that soon.

References

[1] Scott Chacon and Ben Straub. Pro Git. Apress, Nov. 2014
[2] Google. Google Trends: Comparing Git, Subversion, Mercurial, Perforce and CVS. 2019. url: http://bit.ly/gtrends-git (visited on 10/28/2019)

Acknowledgment

Part of this project was developed during **Google Summer of Code 2019.**
Many thanks to the mentors (Christian and Olga) and the entire community.