Improving Parallelism on Git

Capstone Project Proposal - MAC0499 at IME-USP, 2019

Student Matheus Tavares Bernardino
Advisor Alfredo Goldman

About Git

Git is one of the most popular distributed version control systems, supporting the development of many FLOSS and proprietary software today. It was created in 2005 with the purpose of versioning no less than the Linux Kernel. Soon after, it achieved even greater extent thought platforms such as github and gitlab.

The project

As direct as possible, the goal with this capstone project is to **make more of Git's codebase thread-safe** and **improve parallelism in some commands**. More specifically, the non-thread-safe section I plan to be working on is the pack access code. Among other possibilities making this section thread-safe will provide means to improve git-grep's parallelism and to introduce threading to git-blame, which are my final goals.

The motivation behind this are the complaints from developers experiencing slow Git commands when working with large repositories[1], such as chromium and Android. And since nowadays, most personal computers have multi-core CPUs, it is a natural step trying to improve parallel support so that we can better use the available resources.

[1] Some of them can be seen here:
https://groups.google.com/a/chromium.org/forum/#!topic/chromium-dev/oYe69KzyG_U
https://bugs.chromium.org/p/git/issues/detail?id=18
https://bugs.chromium.org/p/git/issues/detail?id=16
https://code.fb.com/core-data/scaling-mercurial-at-facebook/
https://public-inbox.org/git/CA+TurHgyUK5sfCKrK+3xY8AeOg0t66vEvFxX=JiA9wXww7eZXQ@mail.gmail.com/
https://public-inbox.org/git/20140213014229.GE4582@vauxhall.crustytoothpaste.net/
https://public-inbox.org/git/CACBZZX6A+35wGBYAYj7E9d4XwLby21TLbTh-zRX+fkSt_e2zeg@mail.gmail.com/

With this in mind, pack access code is a good target for improvement, since it's used by many Git commands (e.g., checkout, grep, blame, diff, log, etc.). This section of the codebase is still sequential and has many global states, which should be made local, removed or protected before we can work to improve parallelism. And after we have that, git-grep and git-blame are two natural choices to take benefit from it:

- git-grep is already parallel and it shouldn't be hard to refactor its critical sections to use more fine-grained locks, taking advantage of the thread-safe pack access.
- git-blame is sequential and its performance is a big issue for developers at large repositories. Besides that, it seems to have great independence between its main tasks, which make them very good to be run in parallel.

The Pack Access Code

To better describe what the pack access code is, we must talk about Git's object storage (in a simplified way): Besides what are called *loose object files*, Git has a very optimized mechanism to compactly store objects (blobs, trees, commits, etc.) in packfiles[2]. These files are created by[3]:

1. listing objects;
2. sorting the list with some good heuristics;
3. traversing the list with a sliding window to find similar objects in the window, in order to do delta decomposing;
4. compress the objects with zlib and write them to the packfile.

What we are calling *pack access code* in this document, is the set of functions responsible for retrieving the objects stored at the packfiles. This process consists, roughly speaking, in three parts:

1. Locate and read the blob from packfile, using the index file;
2. If the blob is a delta, locate and read the base object to apply the delta on top of it;
3. Once the full content is read, decompress it (using zlib inflate).

Note: *There is a delta-base cache for the second step so that if another delta depends on the same base object, it is already in memory. This cache is global.*

Note 2: *When reading from the packfile, Git uses windows to map regions of the packfile into memory for better performance. (Do not confuse these with the windows used when* ***writing*** *to packfiles). These windows are global per packfile and all packfile attributes are hold by a global variable named "the_repository".*

[2] https://git-scm.com/book/en/v2/Git-Internals-Packfiles
[3] https://github.com/git/git/blob/master/Documentation/technical/pack-heuristics.txt

If the previously shown steps were thread-safe, the ability to perform the delta reconstruction (together with the delta-base cache lookup) and zlib inflation in parallel could bring a good speedup. At git-blame, for example, 24%[4] of the time is spent in the call stack originated at *read_object_file_extended.* Not only this but once we have this big section of the codebase thread-safe, we can work to parallelize even more work at higher levels of the call stack.

Plan

To make pack access thread-safe, I will probably be working mainly with *packfile.c*, *sha1-file.c*, *object-store.h, object.c* and *pack.h* (however, I may also need to tackle other files). I will be focusing on the following three pack access call chains (which are not thread-safe yet), found in git-grep and/or git-blame:

read_object_file → *repo_read_object_file* → *read_object_file_extended* → *read_object* → *oid_object_info_extended* → *find_pack_entry* → *fill_pack_entry* → *find_pack_entry_one* → *bsearch_pack* and *nth_packed_object_offset*
oid_object_info → *oid_object_info_extended* → *<same as previous>*
read_object_with_reference → *read_object_file* → *<same as previous>*

To accomplish thread-safety, some of the point I will have to work on are:

- Protect *packfile.c* global variables such as *pack_open_windows* and *pack_open_fds*, which are read and updated in sections that must be thread-safety.
- Just like the previous item, protect *sha1-file.c* global states such as the *cached_objects* array used by read_object_file(). This cache is intended to hold a small number of objects in memory pretending that they are available at the object storage but not really writing them to disk.
- Protect operations on the delta-base cache. Here we should study whether to add mutexes to the cache itself or to the underlying hashmap. There are advantages and disadvantages in both cases, so it should still be discussed with the community. A third option would be to try making the cache thread-local. But this could perform badly since some objects may be in one thread's cache but not the other's. And memory usage would increase because of duplicate information.

[4] With gprof and gprof2dot I generated the following image: https://i.imgur.com/XmyJMuE.png; which shows some of the most time consuming functions when git-blame is invoked. The call stack originate at read_object_file_extended() is responsible for pack access and consumes 24% of the total execution time.

- Make sure tests cover functions I'll be working on and refactor/add tests as needed.

  After that is done, the focus will be on git-grep and git-blame, going thought the following:

- Refactor the critical sections at git-grep to use more fine-grained mutexes, taking advantage of the now thread-safe pack access. This will hopefully increase git-grep performance, especially in large repositories.
- Check other mutex protected functions git-grep uses, not related with pack access, to see if we can implement a more fined-grained parallelism there. This functions are: *fill_textconv, is_submodule_active, repo_submodule_init, repo_read_gitmodules* and *add_to_alternates_memory.*
- Once pack access is thread-safe, ensure xdiff code used by git-blame has thread-safety. I expect this to be easier.
- Add threads to git-blame. This step should be well discussed with the community when we get to the point of implementing it, to figure out what is the best way to do it. We could work a producer-consumer mechanism at blame.c's assign_blame() function, for a very good work sharing assignment (90% of git-blame's time is spent here[5]). Or try threading at lower functions on the call stack that still uses a lot of execution time such as the libxdiff ones.

Schedule

Here is an approximated schedule:

Period	**Main Task**	**Side Tasks**
February to April	Community bounding and study pack access code	• Get to know the community and send first contributions [done] • Study the project and discuss ideas with the community [done] • Trace pack access call chain used by git commands like blame and checkout. [partially done] • Gather information of global states.
May	Work on sha1-file.c global states and begin monograph	• Protect object cache at sha1-file.c. • Work on other sha1-file.c

[5] https://i.imgur.com/XmyJMuE.png

		global states and non-thread-safe functions. ● Start writting the monograph with all I've already done and studied
June	Work on packfile windows and other packfile.c global states	● Protect access to the windows used to map packfiles' regions into memory. ● Protect packfile.c global states (*pack_open_windows*, *pack_open_fds* and etc.) and work on its non-protected functions.
July	Work on delta-base cache	● Conclude work on packfile windows. ● Protect delta-base cache operations.
August	Work on git-grep	● Use more fine-grained mutexes ● Check other git-grep locks that do not call pack access functions and try to make them more fine-grained too.
September	Prepare the ground for a threaded git-blame	● Check xdiff code for non-thread-safe operations and make them safe ● Profile git-blame and study how to introduce threading to it. Also discuss the ideas with the community. ● Start working on parallel git-blame
October	Work on git-blame	● Finish work on parallel git-blame
November	Work on any leftovers and monograph	This time will be reserved to finish any leftovers from the previous periods and conclude my monograph

Progress so far

Community Bounding

I started to learn more about the Git community in February, among other things, reading about Git internals at the Git Pro book and the development process at the project documentation. I also joined the community on Git's mailing list and IRC channel and began to follow up some of the conversations happening on these media. Since then, I've already studied a few sections of the codebase and sent the following patches:

Patch	**Status**
clone: test for our behavior on odd objects/* content	v5 sent for review
clone: better handle symlinked files at .git/objects/	v5 sent for review
dir-iterator: add flags parameter to dir_iterator_begin	v5 sent for review
clone: copy hidden paths at local clone	v5 sent for review
clone: extract function from copy_or_link_directory	v5 sent for review
clone: use dir-iterator to avoid explicit dir traversal	v5 sent for review
clone: Replace strcmp by fspathcmp	v5 sent for review

And three more patches for git.github.io:

Patch	**Status**
rn-50: Add git-send-email links to light readings	Merged
SoC-2019-Microprojects: Remove git-credential-cache	Merged
SoC-2019-Microprojects: Remove all trailing spaces	Merged

The parallelism project

For some weeks now I have been studying and discussing with the community the ideas presented here to improve Git's parallelism. The community helped me a lot to mature my initial ideas and to think even further.

I also plan to to participate in GSoC (Google Summer of Code), which is a program aimed to bring more students to open source development. I've already submitted the initial phase of this project as my proposal for the program. So if I get approved, I will be working on making pack access thread-safe for GSoC.

Here are some threads on Git's mailing list where I started discussing my project:

- https://public-inbox.org/git/CAHd-oW7onvn4ugEjXzAX_OSVEfCboH3-FnGR00dU8iaoc+b8=Q@mail.gmail.com/
- https://public-inbox.org/git/20190402005245.4983-1-matheus.bernardino@usp.br/
- https://public-inbox.org/git/CAHd-oW7KMrDJ-cyzk63oqW9-QVpag6fKnDp+Mo5bWxg1KfzY3g@mail.gmail.com/

And also a conversation I had at the cromium mailing list to better understand their issues with Git in terms of performance: https://groups.google.com/a/chromium.org/forum/#!topic/chromium-dev/oYe69KzyG_U