1 *Assumptions & layout of this document*

The basic assumptions on which the commands given in this document are predicated are as follows:

- You are using a Linux-based system and so can open a terminal emulator.
- There is sufficient free space to clone the git repository to your workstation.
- The *git* and *gitk* packages are installed.

This document takes the form of an extended tutorial. Firstly, the structure of the remote git repository on Bitbucket is discussed. Then this remote repository is cloned to the user's workstation to create a local repository. A new file is added to the local git repository. The file is then modified and this change is committed in git. The same file is deleted from the repository.

The tutorial then explains how to push an updated local branch to the remote repository and how to get the master branches in the local and remote repositories up-to-date. Finally the issue of merging branches is discussed, in particular how to merge the master branch (tracking upstream LAMMPS) into our version of LAMMPS.

Commands which are entered in the terminal are written `in a monospaced font like this`.

2 *The remote repository on Bitbucket*

2.1 Gaining access

If you are not already a member of the GranLAMMPS team, you need to ask an administrator to become a member. Currently the administrators are Catherine O'Sullivan (primary) and Kevin Hanley. Once the administrator makes you a member of the GranLAMMPS team, you will receive an e-mail from Bitbucket containing instructions. If you don't already have a Bitbucket account, you will be asked to create one which requires picking a username. Your chosen username will appear in the URL used to access the remote repository. There are three access levels to the gitLAMMPS repository: administrator (full permissions), developer (read and write permissions) and viewer (read permissions only).

2.2 Structure of the remote repository

The gitLAMMPS repository currently has five branches, only two of which are of interest to us: master and common.

master This branch does not contain any of our additions or modifications. It is simply the current version of LAMMPS distributed by the Sandia developers.

common This is the version of LAMMPS containing all of our changes that we use for running simulations.

Each branch can be treated a bit like a sub-repository within the main repository.

3 *Creating a local repository*

In this section, we are going to create a copy of the remote repository on our workstation, i.e., create a *local* version of the gitLAMMPS repository. This is essential to commit changes or otherwise modify the repository. For this tutorial, it is assumed that the local repository will be in the user's Documents directory and will be named 'gitLAMMPS'. Obviously the user needs to obtain access to the remote repository by following the instructions in Subsection 2.1 to proceed with this tutorial.

3.1 Cloning the remote repository

Cloning the gitLAMMPS repository to the user's Documents directory is quite straightforward (the second command should all be written on a single line):

```
[user ~]$ cd ~/Documents/
[user Documents]$ git clone
            https://USERNAME@bitbucket.org/granlammps/gitlammps.git gitLAMMPS/
```

You will be asked to input the password you set in Subsection 2.1. A message will appear saying that an empty git repository is being initialised. As cloning the remote repository requires the whole repository to be downloaded, it may take a few minutes to finish. Once complete, a local repository will be available at "~/Documents/gitLAMMPS". Look at the current status of the local repository:

```
[user Documents]$ cd gitLAMMPS/
[user gitLAMMPS]$ git branch
* master
```

It *seems* that only one branch is present: the master branch tracking upstream LAMMPS. The "*" preceding the branch name indicates the branch which is currently active.

3.2 Setting up branch aliases

Although it doesn't seem like it, all of the branches are present in the local repository created in Subsection 3.1; all branches except master are merely hidden. We can see all branches by issuing a `git branch` command with its optional "-a" switch:

```
[user gitLAMMPS]$ git branch -a
* master
  remotes/origin/GMmain
  remotes/origin/HEAD -> origin/master
  remotes/origin/Pluviation
```

```
  remotes/origin/common
  remotes/origin/master
  remotes/origin/tunnel2D
```

To add the hidden common branch and switch to it, the `git checkout` command is used:

```
[user gitLAMMPS]$ git checkout -b common remotes/origin/common
Branch common set up to track remote branch common from origin.
Switched to a new branch 'common'
[user gitLAMMPS]$ git branch
* common
  master
```

A simpler invocation of the `git checkout` command is used to switch between branches. For example, switch back to the master branch by doing the following:

```
[user gitLAMMPS]$ git checkout master
Switched to branch 'master'
```

If we type `git branch` again, the asterisk will now be beside master. Assuming we are on the master branch, let us look at what is happening in more detail:

```
[user gitLAMMPS]$ cd src/GRANULAR/
[user GRANULAR]$ ls
```

All files in the GRANULAR directory are listed. At the time of writing (May 2017), there are 14 files in the upstream GRANULAR directory of LAMMPS. Now switch back to the common directory and list the files again:

```
[user GRANULAR]$ git checkout common
Switched to branch 'common'
[user GRANULAR]$ ls
```

The number of files increases to 36: fix_fluiddrag.{cpp/h}, compute_energy_gran.{cpp/h}, etc. have been added by the Imperial group. This shows that those files which have been added by the Imperial group appear in the common branch but *not* the master branch.

3.3 Checking the default remote repository

Our local repository was created by cloning the remote repository. git automatically set the remote repository for our local repository to the parent that was cloned. We can verify this:

```
[user gitLAMMPS]$ git remote -v
origin https://USERNAME@bitbucket.org/granlammps/gitlammps.git (fetch)
origin https://USERNAME@bitbucket.org/granlammps/gitlammps.git (push)
```

This means that if we push/pull (i.e., upload/download) changes without specifying a remote repository explicitly, git will use the parent as the remote.

4 *Viewing and committing changes*

We will work almost exclusively with our local git repository. The purpose of git is to allow all changes to be tracked in an ordered way, so we need to be able to see the change history easily: this is the first topic discussed in Section 4. Furthermore, there are several changes we might like to make to the common branch (we *never* commit changes to the master branch as explained in Subsection 2.2), including adding a new file, modifying an existing file or deleting a file. We will perform these three operations in that order in Subsections 4.2–4.4.

4.1 Viewing previous changes to the repository

The inbuilt command to view the change history of a branch is `git log`:

```
[user gitLAMMPS]$ git checkout common
[user gitLAMMPS]$ git log
commit 57cce400b92d09fea532d075b2419a1b6885ab82
Merge: b265c88 039af55
Author: Kevin Hanley <kh@cvcosulliv01.cv.ic.ac.uk>
Date:   Wed Mar 12 15:19:36 2014 +0000

    Merge branch 'master' into common

commit b265c88bbc03ad252a218c8b3726006422a4773d
Author: Kevin Hanley <kh@cvcosulliv01.cv.ic.ac.uk>
Date:   Wed Mar 12 15:05:52 2014 +0000

    Added energy tracing code to FIX WALL GRAN for ball-wall contacts.
...                    ...                    ...
```

Press <SPACE> or <RETURN> to scroll downwards or <Q> to exit. Using `git log` can be cumbersome. A better option is to use the `gitk` command provided by the package of the same name. This opens a nice GUI which shows clearly all changes that have been made to the branch (Figure 1 on page 6) and allows the user to filter changes by various criteria which can be useful.

4.2 Adding a new file to the common repository

We can add a new empty file, "test.txt", to the GRANULAR directory of the common branch of our local repository as follows:

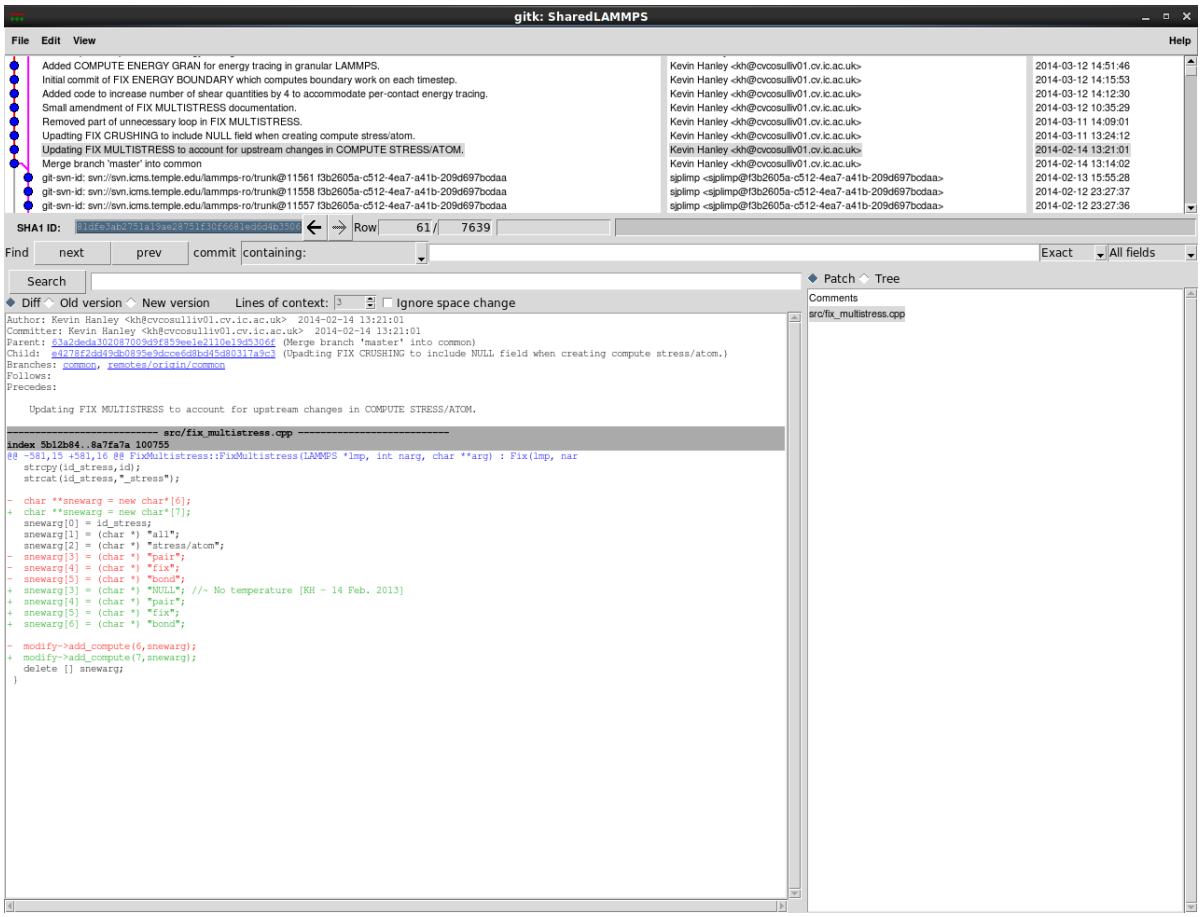

Figure 1: Screenshot of *gitk* showing the common branch of the local repository

```
[user gitLAMMPS]$ cd src/GRANULAR/
[user GRANULAR]$ touch test.txt
```

If you query the status of the branch using `git status`, git will tell you that untracked files are present but it won't do anything about it:

```
[user gitLAMMPS]$ git status
# On branch common
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
# test.txt
nothing added to commit but untracked files present (use "git add" to track)
```

You need to add the file yourself using the `git add` command. Afterwards…

```
[user gitLAMMPS]$ git add test.txt
[user gitLAMMPS]$ git status
# On branch common
```

```
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# new file:   test.txt
```

The final step is committing the changes by running `git commit -a`, which ensures the file will henceforth be tracked by git. In the process of committing the file, you are required to add a commit message: a brief description of the changes being committed to the repository. Try to make your commit messages vaguely descriptive. The commit messages are added using the vi editor (see basics of vi below).

```
[user GRANULAR]$ git commit -a
[common a97c50b] Adding a new file with this commit message.
 0 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 src/GRANULAR/test.txt
```

Basics of adding a commit message with vi

1. *Change to the insert mode of vi by pressing <I>.*
2. *Type your message as normal.*
3. *Press the <ESC> key to exit text insertion mode.*
4. *Type ":wq" to write out and quit vi.*

In summary, adding a new file to git is a two-step process (after moving the file to the appropriate directory in gitLAMMPS):

```
[user gitLAMMPS]$ git add test.txt
[user gitLAMMPS]$ git commit -a
```

4.3 Modifying an existing file

Modifying an existing file is easier than adding a new file. Add some text to test.txt, save and close the file. We commit the change as before, writing a suitable commit message:

```
[user GRANULAR]$ git commit -a
[common 0ab5959] Added a line to this test file.
 1 files changed, 1 insertions(+), 0 deletions(-)
```

4.4 Deleting a file from the common repository

This is basically the reverse of adding a file; simply replace `git add` with `git rm`:

```
[user GRANULAR]$ git rm test.txt
rm 'src/GRANULAR/test.txt'
[user GRANULAR]$ git commit -a
[common c7c3787] Deleted test file.
 1 files changed, 0 insertions(+), 1 deletions(-)
 delete mode 100644 src/GRANULAR/test.txt
```

The log for the complete add/modify/delete sequence of operations looks as follows:

```
[user GRANULAR]$ git log
commit c7c3787bbbf2fb71670e0f509ce004c2ad9cc970
Author: Kevin Hanley <kh@cvcosulliv01.cv.ic.ac.uk>
Date:   Thu Mar 13 15:08:25 2014 +0000

    Deleted test file.

commit 0ab59598a3b19f8f4ca3f0ae9566dba744783189
Author: Kevin Hanley <kh@cvcosulliv01.cv.ic.ac.uk>
Date:   Thu Mar 13 15:06:00 2014 +0000

    Added a line to this test file.

commit a97c50b8abd35b266308a314f9ff434f0b3511a4
Author: Kevin Hanley <kh@cvcosulliv01.cv.ic.ac.uk>
Date:   Thu Mar 13 14:55:02 2014 +0000

    Adding a new file with this commit message.
...                    ...                    ...
```

5 *Interaction of local and remote repositories*

The local repository has been created on a workstation in Section 3 and we have seen how to make changes to the common branch in Section 4. We now need to learn how to transfer our changes to the remote repository on Bitbucket (Subsection 5.1) and how to update the master branch, both in the local (Subsection 5.2) and remote (Subsection 5.3) repositories.

5.1 Pushing changes from local common branch to remote common branch

The easiest way to transfer our changes from local to remote is to use the `git push` command:

```
[user ~]$ cd ~/Documents/gitLAMMPS/
[user gitLAMMPS]$ git push origin common
```

...and enter your password when requested. The arguments of git push give the destination (origin) and the branch to be pushed. If there are no changes to commit, you will see the message "`Everything up-to-date`"; otherwise you will see a summary of the changes made to the remote common branch.

5.2 Updating the local master branch

In order to update the master branch by dragging in the developments made by the upstream developers at Sandia, we create a new alias to a remote destination: the LAMMPS repository on GitHub (described at https://github.com/lammps/lammps). Use the `git remote` command previously seen in Subsection 3.3 to add a new remote called 'upstream'.

```
[user ~]$ cd ~/Documents/gitLAMMPS/
[user gitLAMMPS]$ git remote add upstream https://github.com/lammps/lammps.git
[user gitLAMMPS]$ git remote -v
origin https://USERNAME@bitbucket.org/granlammps/gitlammps.git (fetch)
origin https://USERNAME@bitbucket.org/granlammps/gitlammps.git (push)
upstream https://github.com/lammps/lammps.git (fetch)
upstream https://github.com/lammps/lammps.git (push)
```

Now the `git pull` command can be used to update the master branch of the local repository.

WARNING: IT IS ESSENTIAL THAT YOU SWITCH TO THE MASTER BRANCH *BEFORE* RUNNING `GIT PULL`. THE `GIT PULL` COMMAND IS POTENTIALLY DANGEROUS AS IT ALWAYS MERGES INTO THE BRANCH WHICH IS CURRENTLY CHECKED OUT, WHATEVER THAT HAPPENS TO BE.

```
[user ~]$ cd ~/Documents/gitLAMMPS/
[user gitLAMMPS]$ git checkout master
[user gitLAMMPS]$ git pull upstream master
```

This may take a few minutes to run so be patient. The output in the terminal window will list the files that have been modified since the branch was last updated, indicate the degree of change to each of these files, tell you the numbers of files changed/insertions/deletions, and list any files which have been added/deleted:

```
From https://github.com/lammps/lammps
 * branch            master     -> FETCH_HEAD
Updating 039af55..be64063
Fast-forward
 doc/Manual.html                 |    4 +-
 doc/Manual.txt                  |    4 +-
 src/neighbor.cpp                |  140 ++++++++++++++++++++++++++++++---------------
...                    ...                         ...
 src/version.h                   |    2 +-
 22 files changed, 229 insertions(+), 118 deletions(-)
 create mode 100644 src/rcb.cpp
 create mode 100644 src/rcb.h
```

5.3 Updating the remote master branch

Updating the remote master branch on Bitbucket is a two-step process:

1. Update the local master branch as described in Subsection 5.2.
2. Use `git push` to transfer changes made in the local master branch to the remote master branch, the commands for which are given below:

```
[user ~]$ cd ~/Documents/gitLAMMPS/
[user gitLAMMPS]$ git push origin master
Counting objects: 61, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (24/24), done.
Writing objects: 100% (35/35), 719.86 KiB, done.
Total 35 (delta 27), reused 29 (delta 21)
To https://USERNAME@bitbucket.org/granlammps/gitlammps
   f89510a..ebc7b99  master -> master
```

6 *Merging master branch into common branch*

Periodically merging the master branch into the common branch is the mechanism by which our version of LAMMPS, including all of our in-house additions and modifications, is kept up-to-date with the upstream LAMMPS distribution. This approach has worked well in the past and is infinitely preferable to trying to update the code manually.

There is, however, a potential problem. A change made to the upstream code may affect some of the added/modified files in our version of LAMMPS. To give a simple example, the syntax of

compute_stress_atom.cpp was changed by the main developers in February 2014 to add a temperature argument. Two files in our version of LAMMPS, fix_multistress.cpp and fix_crushing.cpp, implicitly set up a stress compute by calling ComputeStressAtom; this meant that both of these files needed updating. Another example is that changes made to dump_custom.{cpp/h} usually necessitate similar changes in dump_vtk.{cpp/h}. The only way to know the knock-on effects of upstream changes is to have an intimate knowledge of our version of LAMMPS. Updating the code by doing a merge without this understanding will almost certainly lead to problems.

Merging operations can be done only with the local repository. Even if this were not the case, it is the most sensible approach as if anything goes wrong (though it shouldn't if you follow this procedure), you can easily delete and recreate the local repository by cloning the remote repository (Section 3).

6.1 Before merging

In preparation for merging the master branch into the common branch, this sequence is recommended:

1. Get the local master branch up-to-date (Subsection 5.2).
2. Switch to the common branch of the local repository. Use `gitk` (Subsection 4.1) to identify the date and time of the last merged upstream commit and make a note of this. For example, the last upstream commit shown on Figure 1 on page 6 is "*2014-02-13 15:55:28*".
3. Switch to the master branch. Look carefully at each of the upstream changes that have been made since this date. Make note of any upstream changes that will require some code modification on your part. If the number of files requiring modification is large (say, more than five) and/or there are a huge number of upstream commits (likely if you haven't merged in the last month), it may be worthwhile doing several smaller merges rather then one enormous one.

After assiduously following these steps, you will have a list of files that will need modification post-merge. You will also know whether it would be better to do one merge or to divide the merge into several smaller ones.

6.2 Doing a merge

The command to use, unsurprisingly, is `git merge`.

WARNING: MAKE CERTAIN THAT YOU CHECK OUT THE COMMON BRANCH BEFORE MERGING.

As the warning states, it is of paramount importance that you are in the common branch as `git merge` merges another branch into the current one that is checked out, whatever that may be. To do a full merge of all changes:

```
[user ~]$ cd ~/Documents/gitLAMMPS/
[user gitLAMMPS]$ git checkout common
[user gitLAMMPS]$ git merge master
```

One of two things will occur. Either the merge will work perfectly or else a message about a *merge conflict* will be returned. If the merge works perfectly, git automatically commits the file changes so you don't need to do this yourself. Dealing with merge conflicts is the subject of Subsection 6.3. You need to make the necessary modifications to the files identified in Subsection 6.1 regardless of whether a merge conflict occurs or not.

It is also possible to do a partial merge; this means that you merge only up to a specified commit on the master branch. On the master branch, take note of the relevant *commit ID*. A commit ID is a long alphanumeric string, unique to each commit, which looks like this: `c7c3787bbbf2fb71670e0f509ce004c2ad9cc970`. The commit ID can be found easily using `git log` or `gitk` (as "SHA1 ID" on the left-hand side). A partial merge can be done by appending this commit ID to the usual `git merge` command instead of specifying a branch:

```
[user ~]$ cd ~/Documents/gitLAMMPS/
[user gitLAMMPS]$ git checkout common
[user gitLAMMPS]$ git merge c7c3787bbbf2fb71670e0f509ce004c2ad9cc970
```

6.3 Resolving merge conflicts

Suppose that a merge conflict occurs (caused by "pair.cpp"):

```
[user gitLAMMPS]$ git merge master
Auto-merging src/fix.h
Auto-merging src/pair.cpp
CONFLICT (content): Merge conflict in src/pair.cpp
Auto-merging src/pair.h
Automatic merge failed; fix conflicts and then commit the result.
```

This message indicates that some problem has arisen when auto-merging src/pair.cpp. We can also summarise all of the files affected by the merge and their status using the `git status` command with the optional "-s" (for "short") switch:

```
[user gitLAMMPS]$ git status -s
M  doc/Manual.html
...                    ...                    ...
UU src/pair.cpp
...                    ...                    ...
M  src/version.h
```

Any files preceded by "M" have been merged without conflict; files preceded by "UU" contain a conflict. Why has a conflict occurred in pair.cpp? This is what the relevant part of the file used to look like in the common branch (and what pair.cpp still looks like in the common branch of the remote repository, if everything is up-to-date):

```
vatom = NULL;

datamask = ALL_MASK;
datamask_ext = ALL_MASK;

/*~ Added to initialise the status of the rolling resistance
  model as disabled [KH - 23 October 2013]*/
rolling = 0;

//~ Initialise at 1 [KH - 25 October 2013]
model_type = 1;

/*~ Initialise the status of per-contact energy tracing as
  inactive [KH - 6 March 2014]*/
trace_energy = 0;
```

The upstream version of pair.cpp in the master branch (which, remember, we are trying to merge with the code shown immediately above) looks like this:

```
vatom = NULL;

// CUDA and KOKKOS per-fix data masks

datamask = ALL_MASK;
datamask_ext = ALL_MASK;

execution_space = Host;
datamask_read = ALL_MASK;
datamask_modify = ALL_MASK;
```

Clearly, four new lines (one comment and three lines of code) have been added by the upstream developers. The conflict has occurred because although git is smart enough to insert the comment at the correct place, it is confused by the disparity at the end of the code section. After the merge conflict, pair.cpp in the local common branch looks like this:

```
  vatom = NULL;

  // CUDA and KOKKOS per-fix data masks

  datamask = ALL_MASK;
  datamask_ext = ALL_MASK;

<<<<<<< HEAD
  /*~ Added to initialise the status of the rolling resistance
    model as disabled [KH - 23 October 2013]*/
  rolling = 0;

  //~ Initialise at 1 [KH - 25 October 2013]
  model_type = 1;

  /*~ Initialise the status of per-contact energy tracing as
    inactive [KH - 6 March 2014]*/
  trace_energy = 0;
=======
  execution_space = Host;
  datamask_read = ALL_MASK;
  datamask_modify = ALL_MASK;
>>>>>>> master
```

Wherever there is a merge conflict, git inserts *markers* around the relevant code in the following way:

```
<<<<<<< HEAD
  Code from file in common version.
=======
  Code from upstream file in master version.
>>>>>>> master
```

The code won't compile while these markers are present. This must be resolved manually by opening the file in a suitable editor (such as Emacs; if you use a Windows OS, refrain from using Notepad/ Wordpad), correcting the code including deleting the markers, then saving and closing the file. After the author did this, pair.cpp looked like the following:

```
  vatom = NULL;

  // CUDA and KOKKOS per-fix data masks

  datamask = ALL_MASK;
  datamask_ext = ALL_MASK;

  execution_space = Host;
  datamask_read = ALL_MASK;
  datamask_modify = ALL_MASK;

  /*~ Added to initialise the status of the rolling resistance
    model as disabled [KH - 23 October 2013]*/
  rolling = 0;

  //~ Initialise at 1 [KH - 25 October 2013]
  model_type = 1;

  /*~ Initialise the status of per-contact energy tracing as
    inactive [KH - 6 March 2014]*/
  trace_energy = 0;
```

After resolving the merge conflict, we need to *add* the file to git again (which is slightly counter-intuitive) to change the status of the file from "UU" to "M". Finally we need to commit the merge, a step which is necessary only when a merge conflict has occurred:

```
[user gitLAMMPS]$ git add src/pair.cpp
[user gitLAMMPS]$ git status -s
...                     ...                     ...
M  src/pair.cpp
...                     ...                     ...
[user gitLAMMPS]$ git commit -a
```

6.4 After merging

1. After merging, the priority is to make the necessary modifications to the files identified in Subsection 6.1 and commit these changes.

2. Then try compiling a serial build of LAMMPS. If problems arise during compilation, these must be fixed before continuing any further. The common branch of LAMMPS in the remote repository must *always* compile; what you do in the local repository on your own workstation is your own business.

3. Once the code compiles and all seems to be well, push the updated common branch to the remote repository as described in Subsection 5.1.