CS601: Software Development for Scientific Computing

Programming Assignment 1 - Version Control System (Git), performance analysis with PAPI, and Matrix Multiplication
Due: 15/9/2022

The objective of this assignment is to gain a hands-on experience with:

1. A version control system such as Git and a build-tool such as Make
2. Implementing one of the motifs, matrix multplication, and analyzig the performance using PAPI.

1 Version Control Systems and `Git`

Have you ever spent hours writing code (or a term paper) only to realize, when your computer crashes, that you haven't saved your progress? Have you ever spent an hour making a change to a program only to realize that all your changes were wrong and you had to go back to the way the code was? Version control helps solve these problems.

Every non-trivial program is written in stages. To write a good program, you need to divide it into stages and finish one stage at a time. After you finish one stage, commit a version so that you have a record. Version control provides many advantages. One of them is a simple way to back up your code. If, for any reason, you want to go back to a previous version, it is very easy. Version control does many more things than backing up your code.

Please understand that you must commit changes often if you want to use version control. If you do not commit, version control cannot help you.

This class uses `git` for version control. Git is a distributed version control system. That means there are two repositories: local and remote. When you commit changes, only the local repository is changed. This makes commits fast and independent of network connections. If your computer is damaged, you still lose the local repository.

To make changes to the remote repository, you need to push the changes done on the local repository. If your computer is damaged, you can retrieve the code from the remote repository.

Please read the guide at github about how to use version control.

Please remember that you must commit and push often to take advantage of version control.

Version control is required in this class. So, the instructors will not accept any excuse like "I accidentally deleted my code and please give me an extension." Neither will "my computer crashed" be accepted as an excuse.

You must commit and push often to demonstrate your progress of the assignments. *If there is any doubt about academic dishonesty, your commit history would be an important piece of evidence proving your innocence.*

Please read the book about how to use version control: https://git-scm.com/book/en/v2

1.1 Prerequisites

Create a Github account (if you do not already have one). This is the account you should use to create and submit all of your assignments this semester.

Fill the Google form shared with you earlier and inform the TA and Instructors with your GitHub username.

1.2 `Git` - setup and submission instructions

The `Git` setup and submission instructions remain the same for all assignments. Please replace the assignment number in the examples with the number of the assignment you are submitting.

1. Create a Git repository for the assignment.
   (a) Log in to your Github account.
   (b) Visit the Github Teams Discussion page to find the link for the assignment cs601PA1. Click the link. This will create a repository on Github for the assignment (you will follow a similar procedure for all future assignments). Make sure that the repository is called 'IITDhCSE/CS601PA1-<your GitHub username here>'.
   (c) Clone the repository to develop your assignment. Cloning a repository creates a local copy. Change your directory to whichever directory you want to create your local copy in, and type:
   ```
   > git clone git@github.com:IITDhCSE/CS601PA1-<your GitHub username here>.git CS601PA1
   ```
   This will create a subdirectory called `CS601PA1`, where you will work on your code.
   In this command: `git clone` copies a repository.
   `git@github.com:IITDhCSE/CS601PA1-<your GitHub username here>.git` tells `git` where the server (remote copy) of your code is.
   `CS601PA1` tells `git` to place the code in a local directory named `CS601PA1`
   If you change to directory `CS601PA1` and list the contents, you should see the files you will need for this assignment:
   ```
   > cd CS601PA1
   ```
   ```
   > ls
   ```
   And you should see all of the files required to get started with this assignment.

   Sometimes, you may see an error accessing the repository and the clone command may fail to recognize the existance of a repository. One of the reasons could be setting up access credentials:

   Setting up SSH key with GitHub Set up a public SSH key in your GitHub account (if you haven't already). To do this, first generate a new ssh key:
   ```
   > ssh-keygen
   ```
   Hit enter three times (to accept the default location, then to set and confirm an empty passphrase). This will create two files: /.ssh/id_rsa (your private key) and /.ssh/id_rsa.pub (your public key) Then print out your public key:
   ```
   > cat  /.ssh/id_rsa.pub
   ```
   And copy it to the clipboard. Then follow steps at: https://help.github.com/en/articles/adding-a-new-ssh-key-to-your-github-account

2. As you develop your code, you can commit a local version of your changes (just to make sure that you can back up if you break something) by typing:
   ```
   > git add <file name that you want to commit>
   ```
   ```
   > git commit -m "< describe your changes>"
   ```
   `git add <filename>` tells `git` to "stage" a file for committing. Staging files is useful if you want to make changes to several files at once and treat them as one logical change to your code. You need to call `git` add every time you want to commit a file that you have changed.

   `git commit` tells `git` to save a new version of your code including all the changes you staged with `git add`. Note that until you execute `git commit`, none of your changes will have a version associated with them. You can save/commit the changes many times. It is a good habit committing often. It is very reasonable if you commit every ten minutes (or more often).

Do not type `git add *` because you will likely add unnecessary files to the repository. When your repository has many unnecessary files, committing becomes slower. If the unnecessary files are large (such as executables or core files), committing can take several minutes and your assignments may be considered late.

3. The changes you saved by executing `git commit` in the previous step are local to your development environment i.e. they are saved in a local repository. To copy your changes back to Github (to make sure they are saved if your computer crashes, or if you want to continue developing your code from another machine, or you want to make your code visible to a collaborator), type

   ```
   > git push
   ```

   If you do not push, the teaching staff cannot see your solutions.

4. you will use `git`'s "tagging" functionality to submit assignments. Rather than using any submission system, you will use `git` to *tag* which version of the code you want to grade. To tag the latest version of the code, type:

   ```
   > git tag -a <tagname> -m "<describe the tag>"
   ```

   This will attach a tag with name <tagname> to your latest commit. Once you have a version of your program that you want to submit, when you run the following commands:

   ```
   > git tag -a cs601pa1submission -m "Turnin PA1"
   > git push --tags
   ```

   it would associate your source code files with a release tag name `cs601pa1submission` and push it to the remote server. The grading system will check out the source code version with appropriate tag name and grade that. If you want to update your submission (and tell the grading system to ignore any previous submissions), you would type:

   ```
   > git tag -a -f cs601pa1submission -m "Turnin PA1"
   > git push -f --tags
   ```

   to indicate:

   `> git tag -a -f cs601pa1submission -m "Turnin PA1"` overwrites the tag named "cs601pa1submission" on the local repository.

   `git push -f -tags`, pushes the updates and overwrites the tag on the remote repository (on Github).

   These commands will overwrite any other tag named `cs601pa1submission` with one for the current commit. Please be careful about the following rules:

   (a) For each assignment, you should tag only one version with the name `cs601pa1submission`. It is your responsibility to tag the correct one. You CANNOT request regrading if the grading program retrieves the version that you do not want to submit.

   (b) After tagging a version `cs601pa1submission`, any modifications you make to your program WILL NOT BE GRADED (unless you update the tag, as described above).

   (c) The grading program starts retrieving soon after the submission deadline of each assignment. If your repository has no version tagged `cs601pa1submission`, it is considered that you are late.

   (d) The grading program checks every student's repository 120 hours after the submission deadline. If a version tagged `submission` is found, the grading program retrieves and grades that version.

   (e) The grading program uses only the version tagged `cs601pa1submission`. It does NOT choose the higher score before and after the submission deadline. If a later version has the `cs601pa1submission` tag, this later version will be graded with the late discount. Thus, you should tag a late version with `cs601pa1submission` only if you are confident that the new score, with the late discount, is higher.

(f) The time of submission is the time when you push the code to the repository, not the time when the grading program retrieves your code. If you push the code after the deadline, it is late. Even though you push before the grading program starts retrieving your program, it is still considered late.

(g) You should push at least fifteen minutes before the deadline. Give yourself some time to accommodate unexpected situations (such as slow networks).

(h) You are encouraged to tag partially working programs for submission early. In case anything occurs (for example, your computer is broken), you may receive some points. Please remember to tag improved version as you make progress.

Do not submit any binaries. Your git repo should only contain source files; no products of compilation (.o, .exe, a.out etc.).

Do not send your code for grading. The only acceptable way for grading is to tag your repository.

Under absolutely no circumstance will the teaching staff (instructors and teaching assistants) debug your programs without your presence. Such email is ALWAYS ignored. If you need help, go to office hours, or post on the discussion forum.

2 Problem Statements

1. Organize your local repository that you downloaded to associate different source code files with different release tags as described in the table shown next:

Tag Name	source code file(s)	comments
`VERSION1`	`vectorprod.cpp`	this should be the first version of your software. When a user downloads the code based on this tag, the *only* source code file visible in your downloaded repository must be called `vectorprod.cpp` and this file should exactly match the contents of `vectorprod_v1.cpp` given to you in the downloaded repository.
`VERSION2`	`vectorprod.cpp`	When a user downloads the code based on this tag, the *only* source code file visible in your downloaded repository must be `vectorprod.cpp` and and this file should exactly match the contents of `vectorprod_v2.cpp` given to you in the downloaded repository.
`VERSION3`	`vectorprod.cpp`, `scprod.h`, `scprod.cpp`	When a user downloads the code based on this tag, the source code files visible in your downloaded repository must be: `vectorprod.cpp` that exactly matches the contents of `vectorprod_v3.cpp`, `scprod.h` and `scprod.cpp` (`vectorprod_v3.cpp, scprod.cpp, and scprod.h` given to you in the downloaded repository.). Also, you should organize your repository such that: the header files reside in a directory called `inc`, the `.cpp` files must reside in a directory called `src`, the executable resides in a directory called `bin`, and all temporary files reside in the `build` directory.
`VERSION4`	`vectorprod.cpp`, `scprod.h`, `scprod.cpp`	When a user downloads the code based on this tag, the source code files visible in your downloaded repository must be: `vectorprod.cpp` that exactly matches the contents of `vectorprod_v4.cpp`, `scprod.h` that exactly matches the contents of `scprod_v4.h`, and `scprod_v4.cpp` that exactly matches the contents of `scprod_v4.cpp` (`vectorprod_v4.cpp, scprod_v4.cpp, and scprod_v4.h` are given to you in the downloaded repository.). Now, note that within your `vectorprod.cpp` and `scprod.cpp`, there are lines that still refer to `scprod_v4.h`. You should change these lines to refer to the 'correct' header file. Also, you should organize your repository such that: the header files reside in a directory called `inc`, the `.cpp` files must reside in a directory called `src`, the executable resides in a directory called `bin`, and all temporary files reside in the `build` directory.

You should also provide a `makefile` that is used to build each of the versions described above. Your source code must include only one `makefile` and this file must reside at the top-level directory i.e. not inside any of the directories `src`, `inc`, etc. You should use the compiler flags `-g, -Wall, -Werror`[1] while building the target. The name of the target must be `dotprod`. Your makefile must contain a single rule (default) and the exact command that we will use to build target is: `make` (without any arguments).

2. Implement a C++ program that involves computing the product of two matrices as follows: `C=C+A*B`, where `C`, `A`, and `B` are square matrices of size `N`. The product `A*B` is computed and updated to matrix `C`, which is zero initialized. A simple matrix-multiplication consists of 3 nested loops (referred to as `ijk` loop) shown in the below pseudocode:

```
for i=0 to N-1
  for j=0 to N-1
    for k=0 to N-1
      $c_{ij} = c_{ij} + a_{ik} * b_{kj}$
```

a_{xy}, b_{xy}, and c_{xy} are elements of matrices `A`, `B`, and `C` respectively with x and y are suitable indices in the range $[0, N-1]$. All the source code must reside in a separate directory called `matmul`. You should

[1] You will also need other compiler flags. See slide 13, week2.pdf

implement multiple versions of this program as follows:

target	comments
`matmul1`	implementation of the simple `ijk` loop as shown above.
`matmul2`	implementation of the `kij` loop, which uses the outer-product formulation we discussed in class.
`matmul3`	implementation of the blocked-matmul, where instead of the column-blocking we discussed in class, you should implement row-blocking.

While implementing, you should:

- provide one `makefile` with 5 rules, where there is a rule that builds each of the versions descibed above. The default rule (i.e. when the command `make` without any arguments is executed) that is fired builds all the versions. There is also a rule to remove all the temporary files built.
- carefully choose matrix layout (row-major) for each of the matrices involved.
- include the code to validate the result.
- use `double` type with elements of matrices.
- organize source code based on the guidelines mentioned in the previous problem and also discussed in class.

choose values of N as follows: 1024, 2048, and 4096. For each N chosen, choose block size values (`matmul3` only) as follows: 16, 32, 64, 128. Report overall execution times and discuss the results.

3. Install Performance API (PAPI) and measure performance counters i) total instructions ii) L1 cache misses iii) L2 cache misses iv) L3 cache misses using PAPI for each `matmul` version. You should measure the counters by placing the PAPI apis around the loop code only. Report the performance counters for each `matmul` version executed with each value of N chosen previously.

2.1 What you need to submit

- (for problem statement 1:) Modified initial repository, with all the tags described above. The latest source code files (those that correspond to the tag name `VERSION4` described above).
- (for problem statement 2:) A directory called `matmul` that contains all the related source code.
- (for problem statement 2 and 3:) A brief report that shows the execution times (described for problem 2 above) and performance counters. Your report must contain a table with at least 18 rows: 6 rows for `matmul1`/`matmul2-N` pair. In addition, there are 12 rows for `matmul3-N-block-size` combination. Each of those rows should report the 4 counter values in addition to the execution time. You should also discuss, based on the table presented, why you see the execution times that you see.

You must tag your source code (changes that you made to the repository after the `VERSION4` release) and submit as described earlier. The tag name to be used is: `cs601pa1submission`. All tag names are case-sensitive..