Advanced Topics in Numerical Analysis: High Performance Computing

More Shared Mmory Parallelism, Git, Make

Georg Stadler, Dhairya Malhotra
Courant Institute, NYU

Spring 2019, Monday, 5:10–7:00PM, WWH #1302

March 8, 2019

Outline

Organization

Scheduling:

- Thanks for coming to our makeup class!
- Next class on Monday, March 11 (we'll start with GPUs)
- Homework assignment #2 due on Monday, there will be a (short) new assignment handed out next week.

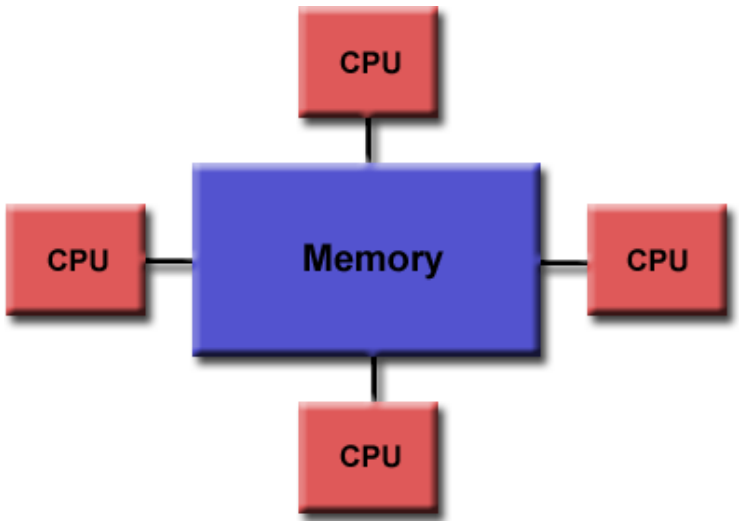
Topics today:

- Shared memory parallelism (cont'd)
- Tools of the week: Git (remove), Make

Outline

Shared memory programming model

- Program is a collection of control threads, that are created dynamically
- Each thread has private and shared variables
- Threads can exchange data by reading/writing shared variables
- Danger: more than 1 processor core reads/writes to a memory location: race condition



Programming model must manage different threads and avoid race conditions.
OpenMP: Open Multi-Processing is the application interface (API) that supports shared memory parallelism: `www.openmp.org`

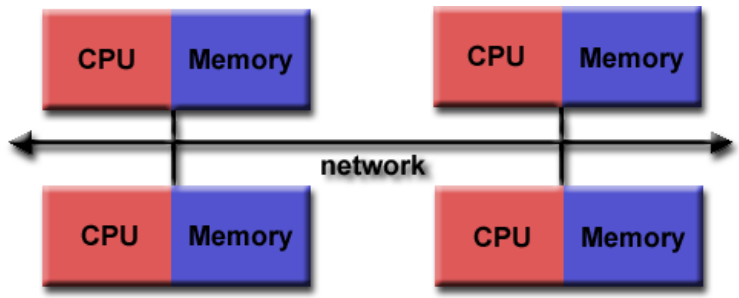
Shared memory programming model

Advantages and disadvantages:

+ Relative easy to parallelize loops etc. in serial programs
- Limited amount of parallelism possible in practice
- Memory bus becomes bottleneck if too many processors access the same memory (usually used with ≤ 50 cores)
- Cache coherency: Need to make sure that values stored in cache of each processor coincide (handled by hardware)
- Size of shared memory is limited and can get very expensive

Distributed memory programming model

- Program is run by a collection of named processes; fixed at start-up
- Local address space; no shared data
- logically shared data is distributed (e.g., every processor only has direct access to a chunk of rows of a matrix)
- Explicit communication through send/receive pairs



Programming model must accommodate communication.

MPI: Massage Passing Interface (different implementations: LAM, Open-MPI, Mpich, Mvapich), http://www.mpi-forum.org/

Hybrid distributed/shared programming model

- Pure MPI approach splits the memory of a multicore processor into independent memory pieces, and uses MPI to exchange information between them.
- Hybrid approach uses MPI across processors, and OpenMP for processor cores that have access to the same memory.
- A similar hybrid approach is also used for hybrid architectures, i.e., computers that contain CPUs and accelerators (GPGPUs, MICs).

Other parallel programming approaches

- Grid computing: loosely coupled problems, most famous example was `SETI@Home`.
- MapReduce: Introduced by Google; targets large data sets with parallel, distributed algorithms on a cluster.
- WebCL
- Pthreads
- CUDA
- Cilk
- ...

Amdahl's law

Is there enough parallelism in my problem?

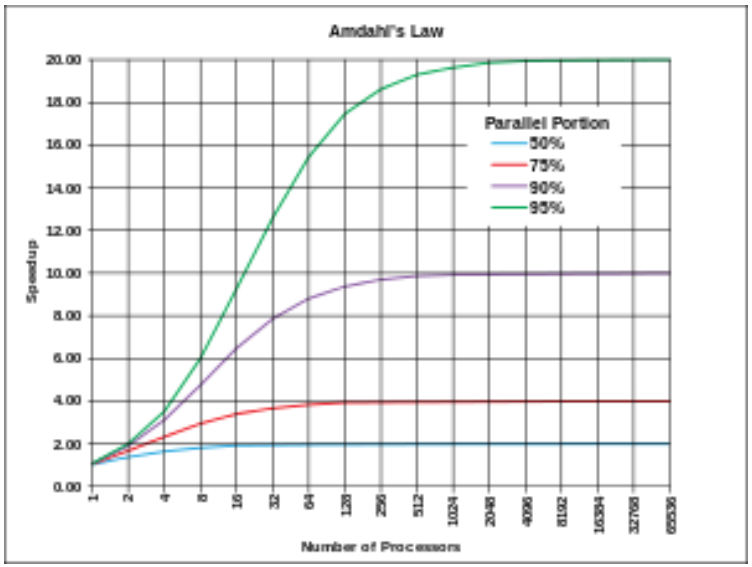
Suppose only part of the application is parallel
Amdahl's law:

- Let s be the fraction of work done sequentially, and $(1 - s)$ the part that is done in parallel
- p. . . number of parallel processor (cores).

Speedup:

$$\frac{\text{time}(1 \text{ proc})}{\text{time}(p \text{ proc})} \leq \frac{1}{s + (1 - s)/p} \leq \frac{1}{s}$$

Thus: Performance is limited by sequential part.



Source: Wikipedia

Load (im)balance in parallel computations

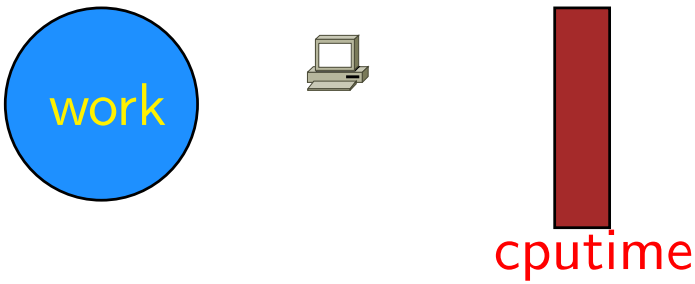
In parallel computations, the work should be distributed *evenly* across workers/processors.

- Load imbalance: Idle time due to insufficient parallelism or unequal sized tasks
- Initial/static load balancing: distribution of work at beginning of computation
- Dynamic load balancing: work load needs to be re-balanced during computation. Imbalance can occur, e.g., due to
  - adapting (mesh refinement)
  - in completely unstructured problems

Parallel scalability

Strong and weak scaling/speedup

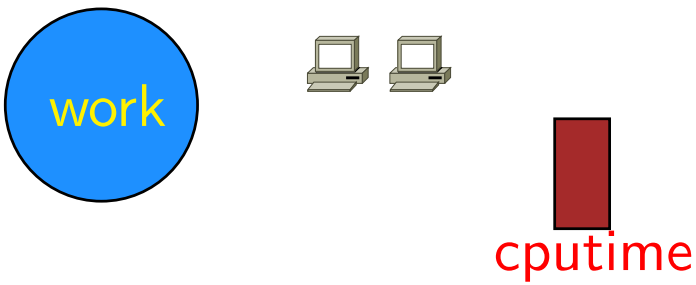
Strong scalability



Parallel scalability

Strong and weak scaling/speedup

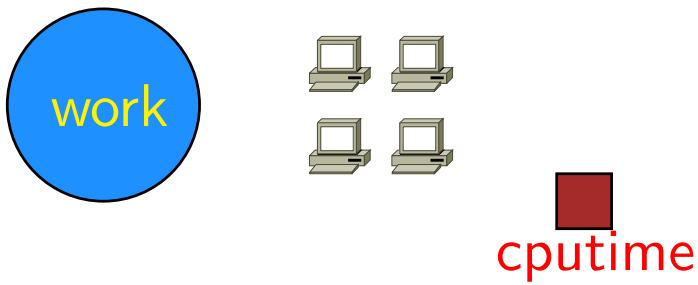
Strong scalability



Parallel scalability

Strong and weak scaling/speedup

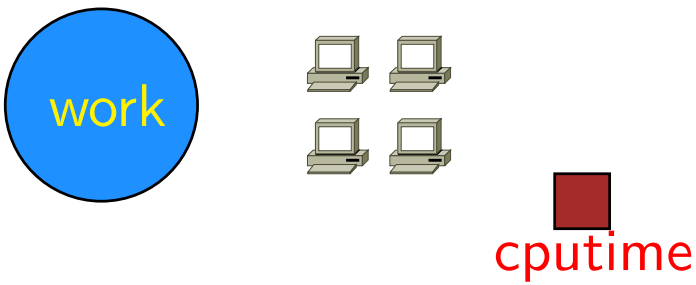
Strong scalability

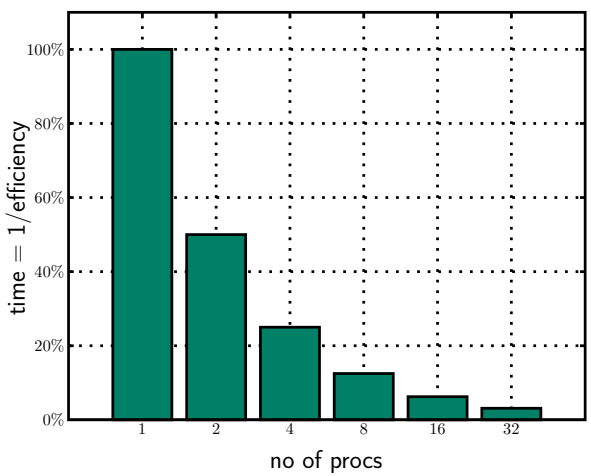


Parallel scalability

Strong and weak scaling/speedup

Strong scalability

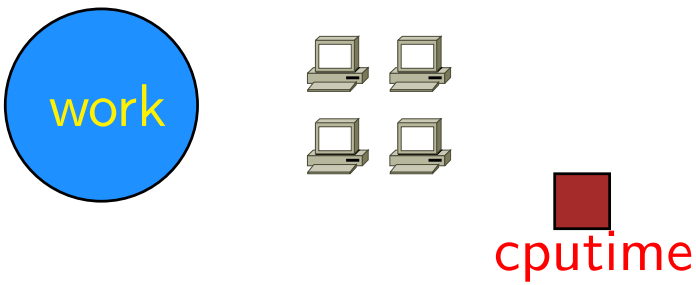


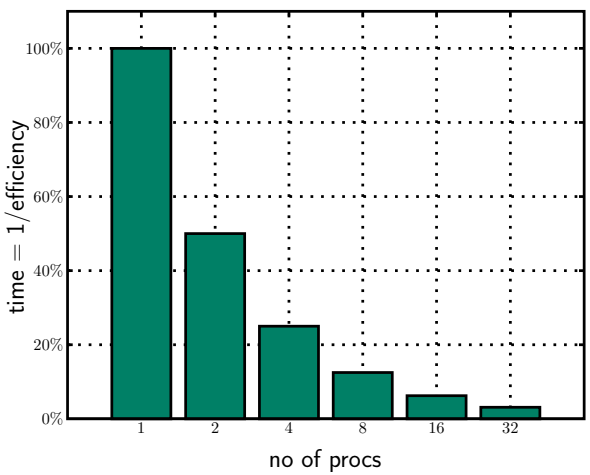


Parallel scalability

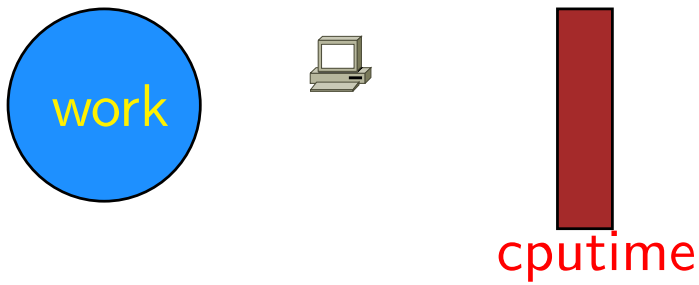
Strong and weak scaling/speedup

Strong scalability





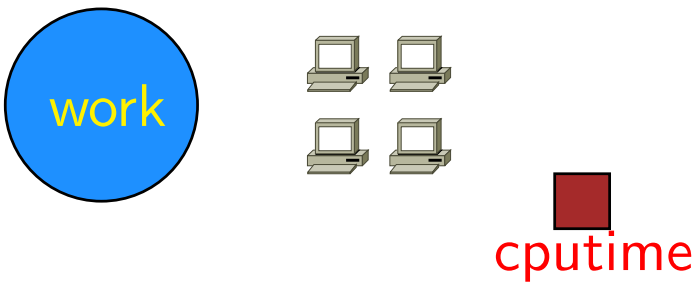
Weak scalability



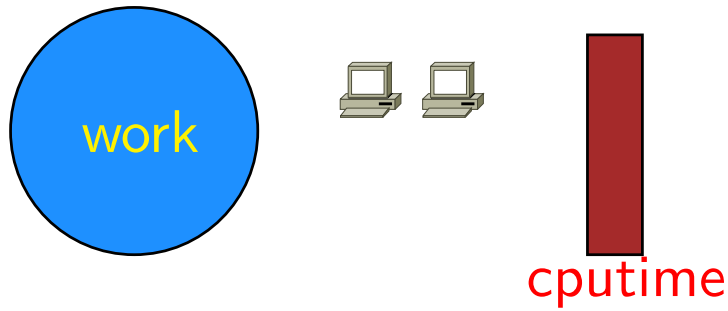
Parallel scalability

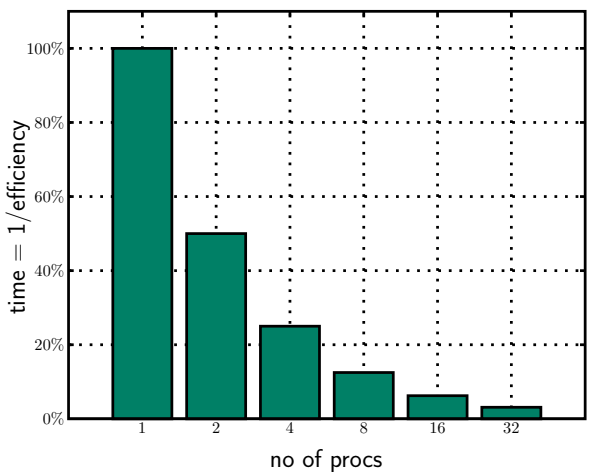
Strong and weak scaling/speedup

Strong scalability



Weak scalability

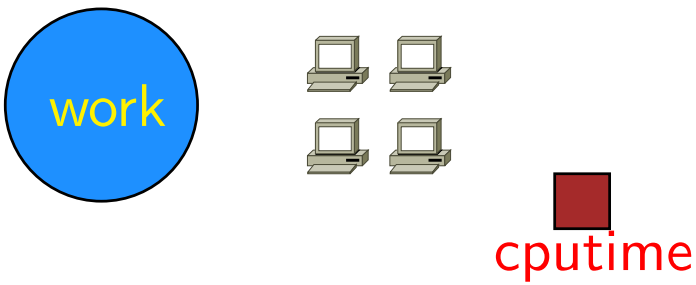




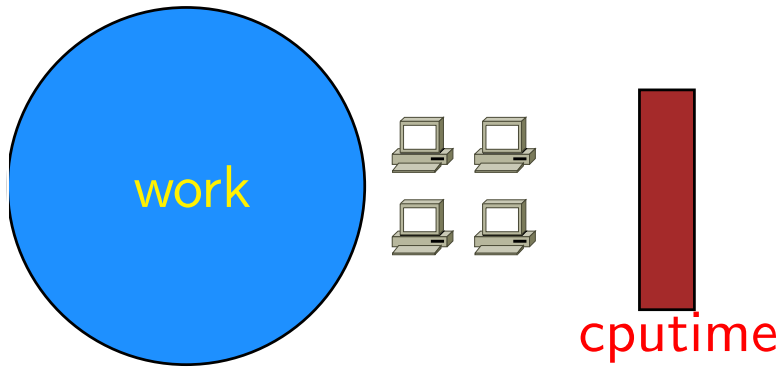
Parallel scalability

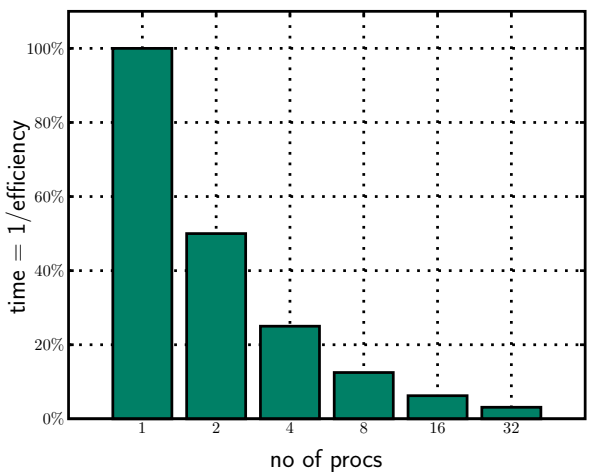
Strong and weak scaling/speedup

Strong scalability



Weak scalability

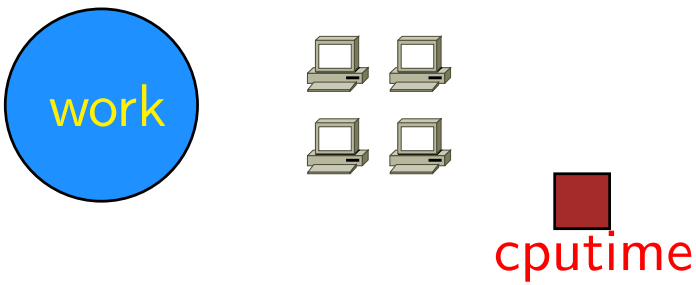


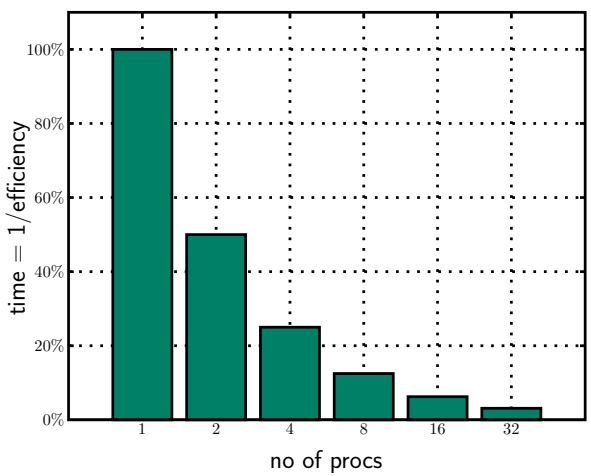


Parallel scalability

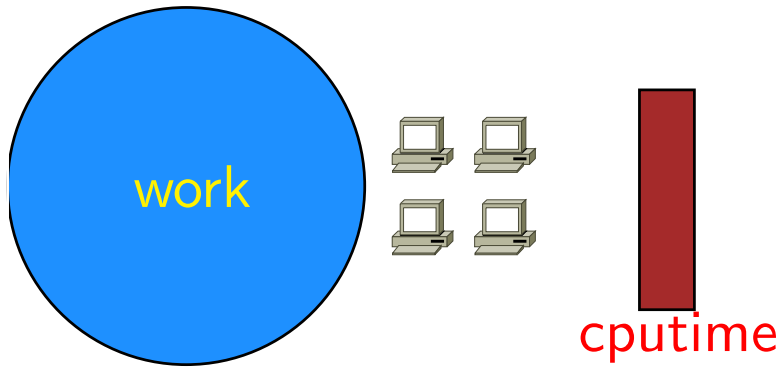
Strong and weak scaling/speedup

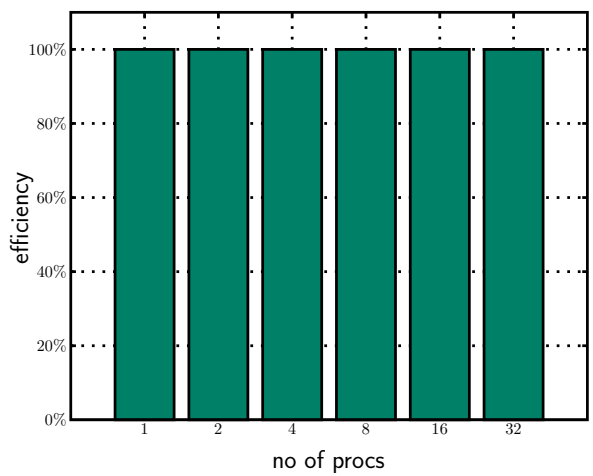
Strong scalability





Weak scalability

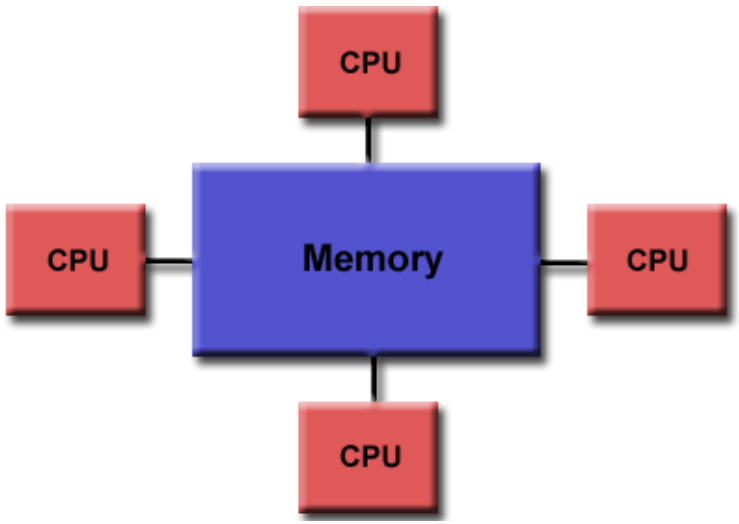




Outline

Shared memory programming model

- Program is a collection of control threads, that are created dynamically
- Each thread has private and shared variables
- Threads can exchange data by reading/writing shared variables
- Danger: more than 1 processor core reads/writes to a memory location: race condition



Only one process is running, which can fork into shared memory threads.

Threads versus process

- A process is an independent execution unit, which contains their own state information (pointers to instruction and stack). One process can contain several threads.
- Threads within a process share the same address space, and communicate directly using shared variables. Seperate stack but shared heap memory.
- Stack memory: Used for temporarily storing data; fast; last-in-first-out principle. Examples `int a=2;` `double b=2.11;` etc; no deallocation necessary; small size; static.
- Heap memory: Not managed automatically, manually allocate/de-allocate/re-allocate; slower; larger;
- Using several threads can also be useful on a single processor ("multithreading"), depending on the memory latency.

Shared memory programming with OpenMP

- Split program into serial and into parallel regions
- Race condition: more than one thread reads/writes to the same shared memory location
- Easy to parallelize loops without data dependencies by adding `#pragma` commands
- `pragmas` are compiler directive externals to the programming language; they are ignored by the compiler if it doesn't understand them

Outline

Why Use Version Control?

Slides adapted from Andreas Skielboe

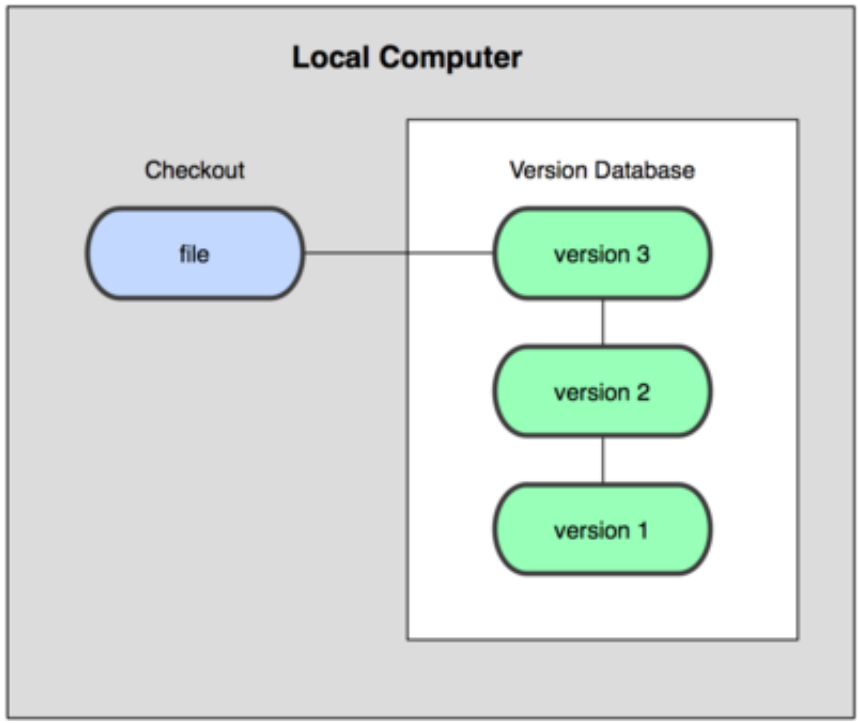
A Version Control System (VCS) is an integrated fool-proof framework for

- Backup and Restore
- Short and long-term undo
- Tracking changes
- Synchronization
- Collaborating
- Sandboxing

... with minimal overhead.

Local Version Control Systems

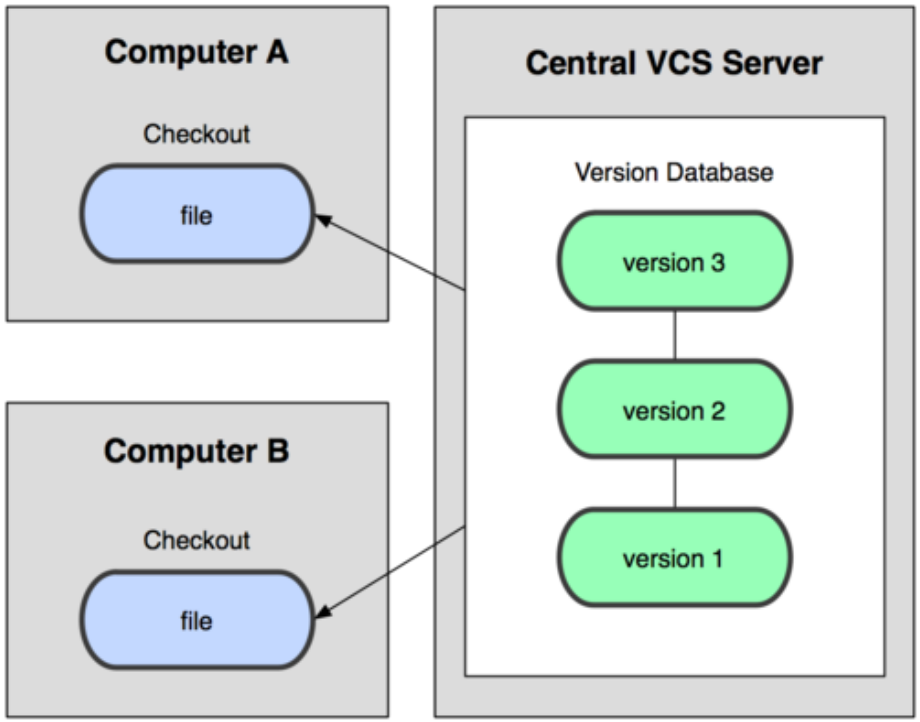
Conventional version control systems provides some of these features by making a local database with all changes made to files.



Any file can be recreated by getting changes from the database and patch them up.

Centralized Version Control Systems

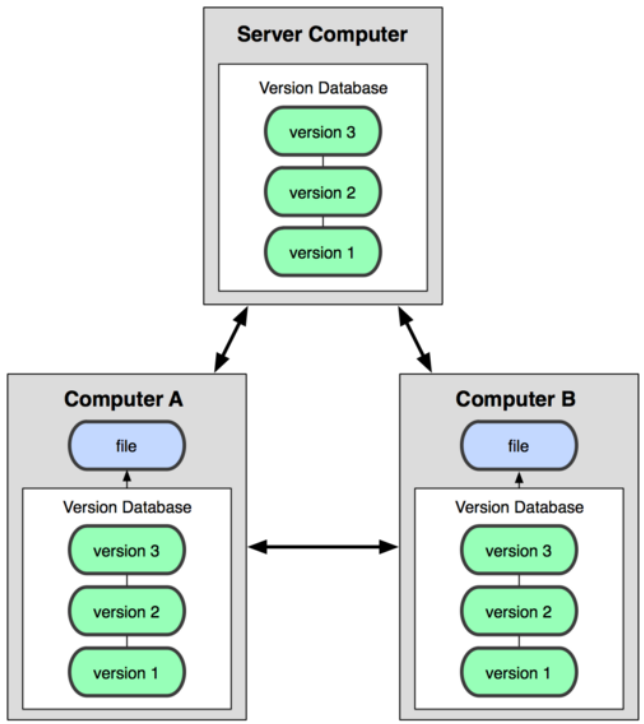
To enable synchronization and collaborative features the database is stored on a central VCS server, where everyone works in the same database.



Introduces problems: single point of failure, inability to work offline.

Distributed Version Control Systems

To overcome problems related to centralization, distributed VCSs (DVCSs) were invented. Keeping a complete copy of database in every working directory.



Actually the most **simple** and most **powerful** implementation of any VCS.

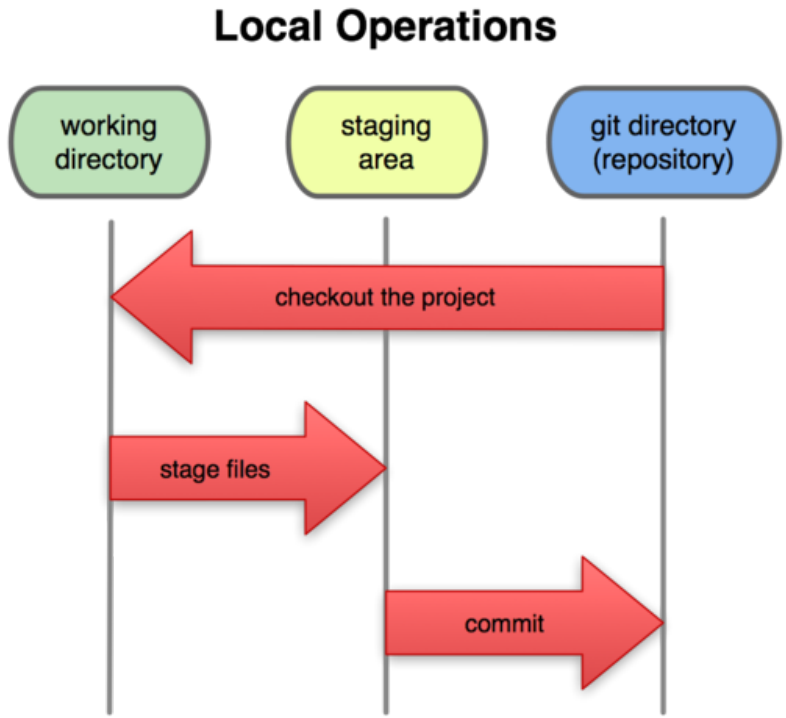
Git Basics - The Git Workflow

The simplest use of Git:

- **Modify** files in your *working directory*.
- **Stage** the files, adding snapshots of them to your *staging area*.
- **Commit**, takes files in the staging area and stores that snapshot permanently to your *Git directory*.

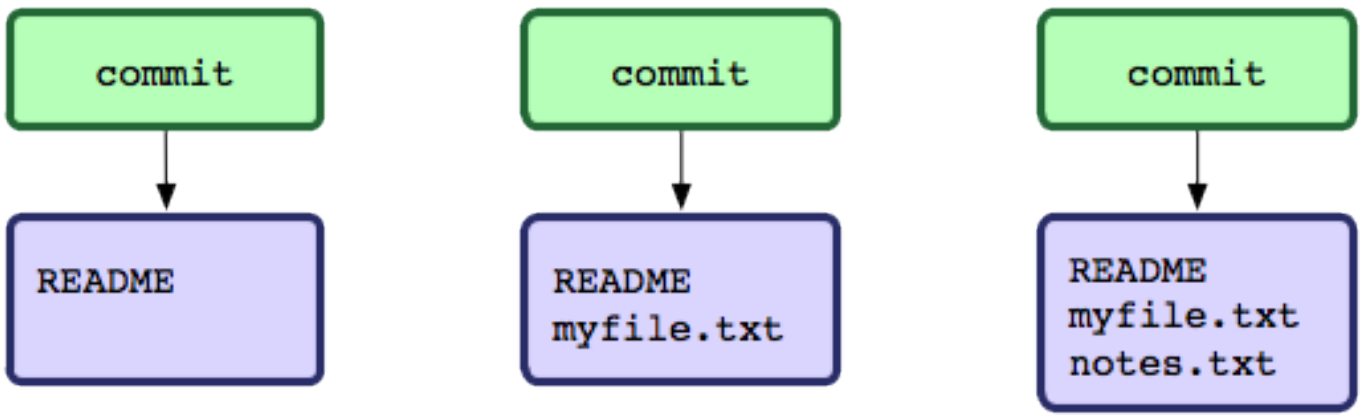
Git Basics - The Three States

The three basic states of files in your Git repository:



Git Basics - Commits

Each commit in the git directory holds a snapshot of the files that were staged and thus went into that commit, along with author information.



Each and every commit can always be looked at and retrieved.

Git Basics - Working with remotes

In Git **all remotes are equal**.

A *remote* in Git is nothing more than a link to another git directory.

Git Basics - Working with remotes

The easiest commands to get started working with a remote are

- *clone*: Cloning a remote will make a complete local copy.
- *pull*: Getting changes from a remote.
- *push*: Sending changes to a remote.

Fear not! We are starting to get into more advanced topics. So lets look at some examples.

Git Basics - Advantages

Basic advantages of using Git:

- Nearly every operation is local.
- Committed snapshots are always kept.
- Strong support for non-linear development.

Hands-on - First-Time Git Setup

Before using Git for the first time:

Pick your identity

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Check your settings

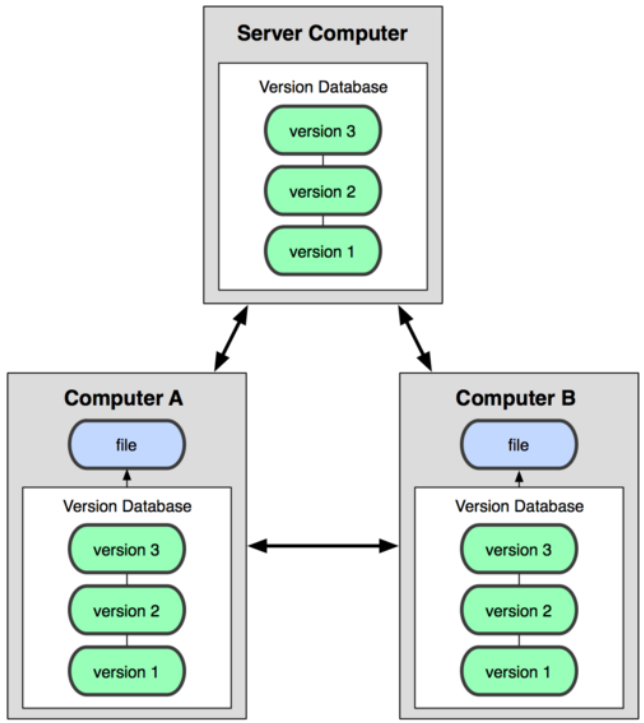
```
$ git config --list
```

Get help

```
$ git help <verb>
```

Hands-on - Getting started with a bare remote server

Using a Git server (ie. no working directory / *bare* repository) is the analogue to a regular centralized VCS in Git.



Hands-on - Getting started with remote server

When the remote server is set up with an initialized Git directory you can simply *clone* the repository:

Cloning a remote repository

```
$ git clone <repository>
```

You will then get a complete local copy of that repository, which you can edit.

Hands-on - Getting started with remote server

With your local working copy you can make any changes to the files in your working directory as you like. When satisfied with your changes you add any modified or new files to the staging area using *add*:

Adding files to the staging area

```
$ git add <filepattern>
```

Hands-on - Getting started with remote server

Finally to commit the files in the staging area you run *commit* supplying a *commit message*.

Committing staging area to the repository

```
$ git commit -m <msg>
```

Note that so far **everything is happening locally** in your working directory.

Hands-on - Getting started with remote server

To **share your commits** with the remote you invoke the *push* command:

Pushing local commits to the remote

```
$ git push
```

To recieve changes that other people have pushed to the remote server you can use the *pull* command:

Pulling remote commits to the local working directory

```
$ git pull
```

And **thats it**.

Hands-on - Summary

Summary of a minimal Git workflow:

- `clone` remote repository
- `add` you changes to the staging area
- `commit` those changes to the git directory
- `push` your changes to the remote repository

- `pull` remote changes to your local working directory

More advanced topics

Git is a powerful and flexible DVCS. Some very useful, but a bit more advanced features include:

- Branching
- Merging
- Tagging
- Rebasing

References

Some good Git sources for information:

- Git Community Book - http://book.git-scm.com/
- Pro Git - http://progit.org/
- Git Reference - http://gitref.org/
- GitHub - http://github.com/
- Git from the bottom up - http://ftp.newartisans.com/pub/git.from.bottom.up.pdf
- Understanding Git Conceptually - http://www.eecs.harvard.edu/~cduan/technical/git/
- Git Immersion - http://gitimmersion.com/

Applications

GUIs for Git:

- GitX (MacOS) - http://gitx.frim.nl/
- Giggle (Linux) - http://live.gnome.org/giggle

What should (not) be added to a repository?

Git tracks diff-files to keep its memory requirements small. Main rule: mostly add *source files that compile*.

- .c, .cpp, .f files

What should (not) be added to a repository?

Git tracks diff-files to keep its memory requirements small. Main rule: mostly add *source files that compile*.

- .c, .cpp, .f files YES!
- .tex files

What should (not) be added to a repository?

Git tracks diff-files to keep its memory requirements small. Main rule: mostly add *source files that compile*.

- .c, .cpp, .f files YES!
- .tex files YES!
- .aux, .out, .dvi. . . files

What should (not) be added to a repository?

Git tracks diff-files to keep its memory requirements small. Main rule: mostly add *source files that compile*.

- .c, .cpp, .f files YES!
- .tex files YES!
- .aux, .out, .dvi. . . files NO!
- compiled files, object files

What should (not) be added to a repository?

Git tracks diff-files to keep its memory requirements small. Main rule: mostly add *source files that compile*.

- .c, .cpp, .f files YES!
- .tex files YES!
- .aux, .out, .dvi. . . files NO!
- compiled files, object files NO! (large, no diffs possible, conflicts)
- .pdf files

What should (not) be added to a repository?

Git tracks diff-files to keep its memory requirements small. Main rule: mostly add *source files that compile*.

- .c, .cpp, .f files YES!
- .tex files YES!
- .aux, .out, .dvi. . . files NO!
- compiled files, object files NO! (large, no diffs possible, conflicts)
- .pdf files YES/NO!
- large data files

What should (not) be added to a repository?

Git tracks diff-files to keep its memory requirements small. Main rule: mostly add *source files that compile*.

- .c, .cpp, .f files YES!
- .tex files YES!
- .aux, .out, .dvi. . . files NO!
- compiled files, object files NO! (large, no diffs possible, conflicts)
- .pdf files YES/NO!
- large data files NO. . . sometimes maybe
- photos, movies etc.

What should (not) be added to a repository?

Git tracks diff-files to keep its memory requirements small. Main rule: mostly add *source files that compile*.

- .c, .cpp, .f files YES!
- .tex files YES!
- .aux, .out, .dvi. . . files NO!
- compiled files, object files NO! (large, no diffs possible, conflicts)
- .pdf files YES/NO!
- large data files NO. . . sometimes maybe
- photos, movies etc. NO! (unless unavoidable)

My rule of thumb: Files in the repository are permanent, only the best should make it in there (it's not your trash can!) They should compile (code/Latex), be (more or less) cleaned up, unless it's avoidable only source/text files.

Some of my git wisedom

Should I have a few large repositories or many small ones?

- I recommend many small ones (like I use for this class).
- Easier to manage, commit messages easier to monitor.
- Small memory footprint and faster!
- It's easy to link two repositories (e.g., code libraries) using git submodules (look it up)!

Some of my git wisedom

Should I have a few large repositories or many small ones?

- I recommend many small ones (like I use for this class).
- Easier to manage, commit messages easier to monitor.
- Small memory footprint and faster!
- It's easy to link two repositories (e.g., code libraries) using git submodules (look it up)!

How often should you commit?

- As often as you like (in case of doubt, more often)
- Makes it easier to monitor changes, track down bugs
- If you collaborate, better to avoid conflicts
- For me: feels like a (small) achievement, supports clean/systematic working style (always look at diff before committing)

Some of my git wisedom

Should I have a few large repositories or many small ones?

- I recommend many small ones (like I use for this class).
- Easier to manage, commit messages easier to monitor.
- Small memory footprint and faster!
- It's easy to link two repositories (e.g., code libraries) using git submodules (look it up)!

How often should you commit?

- As often as you like (in case of doubt, more often)
- Makes it easier to monitor changes, track down bugs
- If you collaborate, better to avoid conflicts
- For me: feels like a (small) achievement, supports clean/systematic working style (always look at diff before committing)

. . . any others??