Advanced Topics in Numerical Analysis: High Performance Computing

Shared Memory Parallelism

Marsha Berger
Courant Institute, NYU

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

Feb 25, 2019

Outline

Organization issues

Parallel machines and programming models

Shared memory parallelism–OpenMP

Git—repository systems

Topics today

- Parallel programming models
- Shared memory parallelism—OpenMP
- Tool of the week: Git

Outline

Organization issues

Parallel machines and programming models

Shared memory parallelism–OpenMP

Git—repository systems

Parallel architectures (Flynn's taxonomy)

Characterization of architectures according to Flynn:

- **SISD:** Single instruction, single data. This is the conventional sequential model.
- **SIMD:** Single instruction, multiple data. Multiple processing units with identical instructions, each one working on different data. Useful when a lot of completely identical tasks are needed.
- **MIMD:** Multiple instructions, multiple data. Multiple processing units with separate (but often similar) instructions and data/memory access (shared or distributed). We will mainly use this approach.
- **MISD:** Multiple instructions, single data. Not practical.

SPMD: Single program, multiple data (most common style)

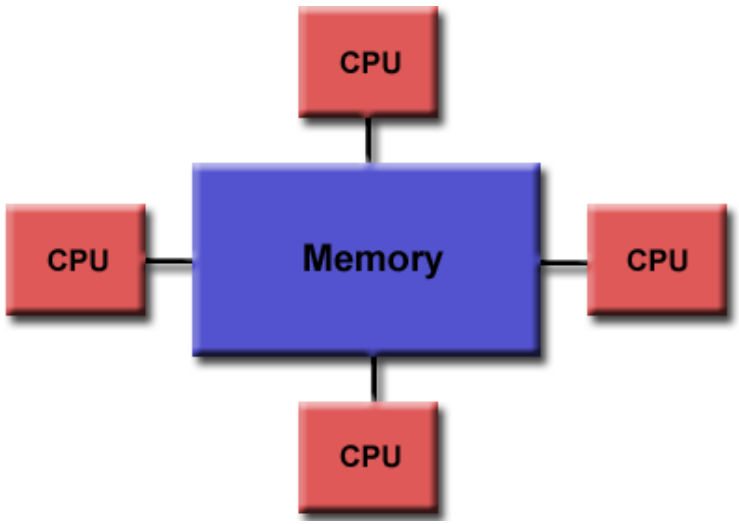
Programming model must reflect architecture

Example: Inner product between two (very long) vectors: $a^T b$:

- Where are a, b stored? Single memory or distributed?
- What work should be done by which processor?
- How do they coordinate their result?

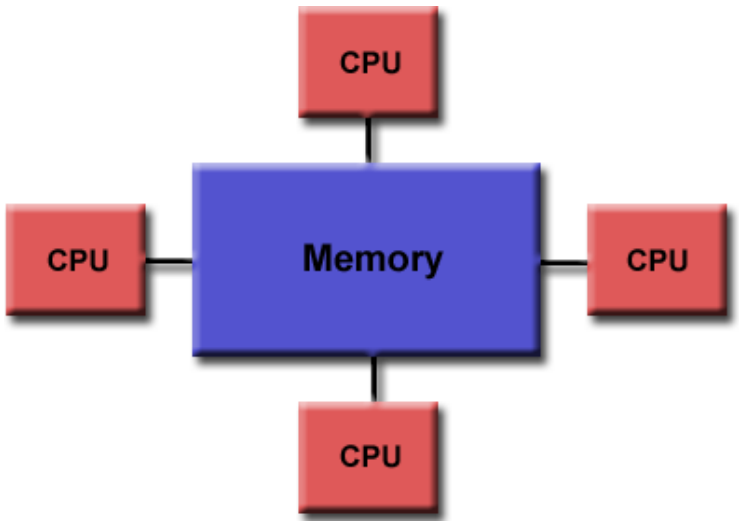
Shared memory programming model

- 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



Shared memory programming model

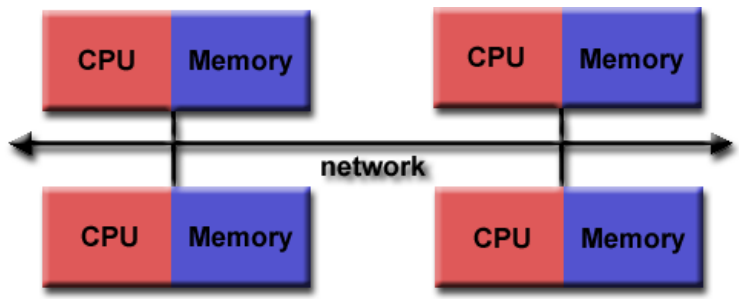
- Program is a collection of control threads, that are created dynamically
- Each thread has private and shared variables
- Threads can exchange data by reading/writing shared variables
- Danger: more than 1 processor core reads/writes to a memory location: race condition



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`

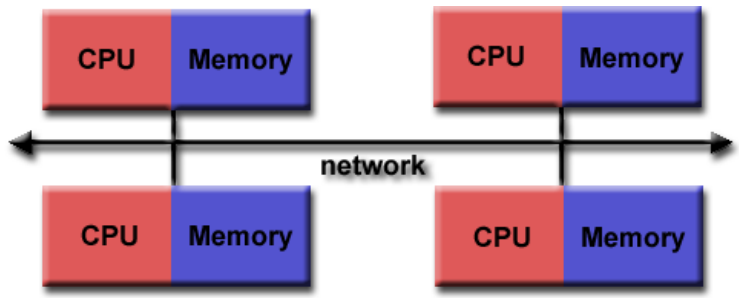
Distributed memory programming model

- 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



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

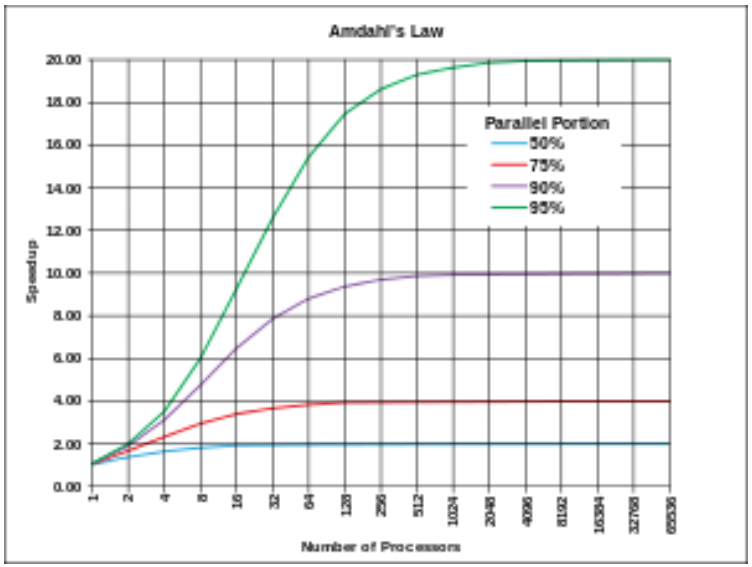
Amdahl's law:

- 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

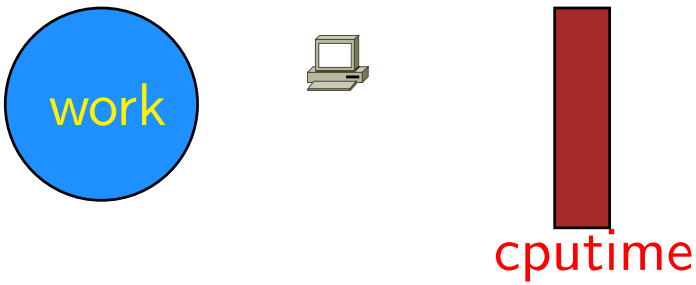
In parallel computations, the work should be distributed *evenly* across workers/processors.

- 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

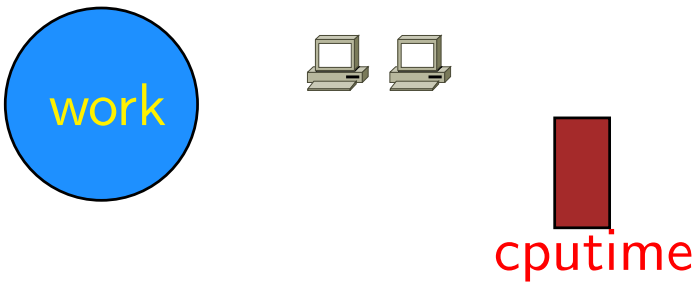
Strong scalability



Parallel scalability

Strong and weak scaling/speedup

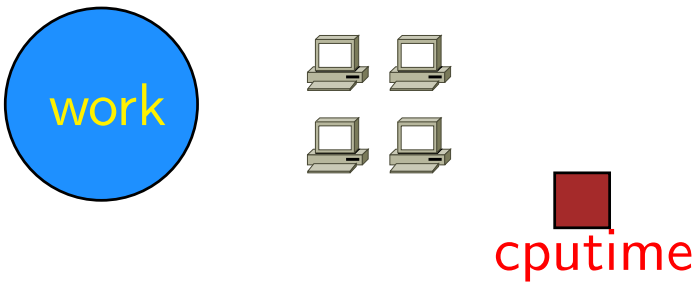
Strong scalability



Parallel scalability

Strong and weak scaling/speedup

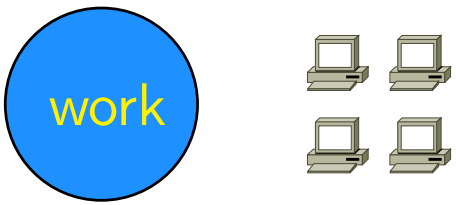
Strong scalability



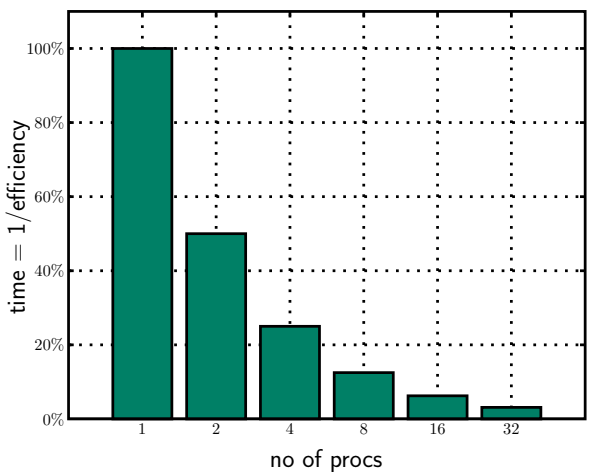
Parallel scalability

Strong and weak scaling/speedup

Strong scalability



cputime

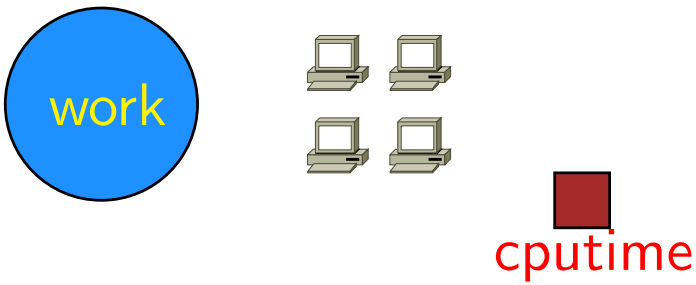


Fixed problem size
increase # of processor

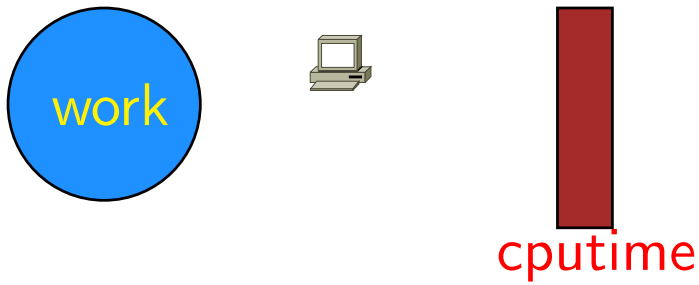
Parallel scalability

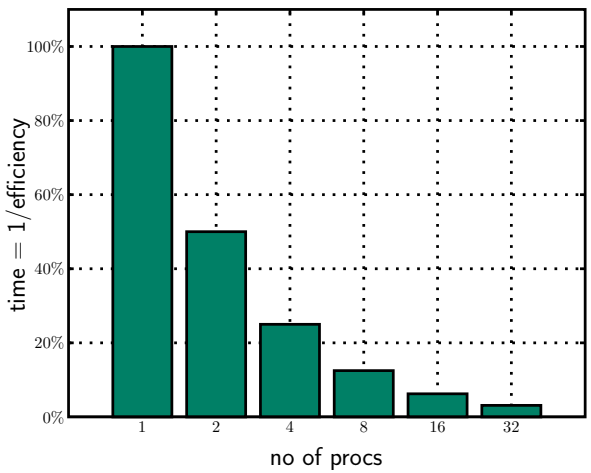
Strong and weak scaling/speedup

Strong scalability



Weak scalability



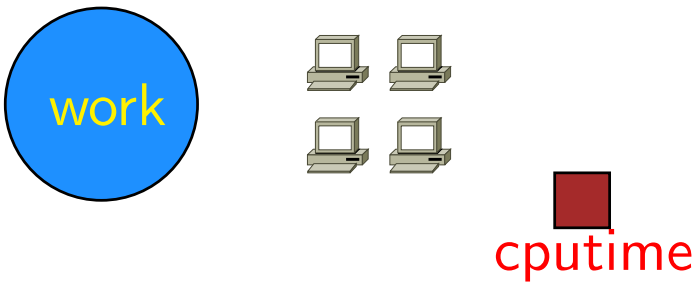


Fixed problem size
increase # of processor

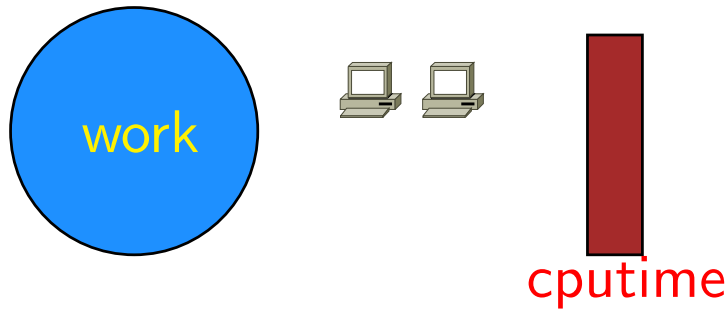
Parallel scalability

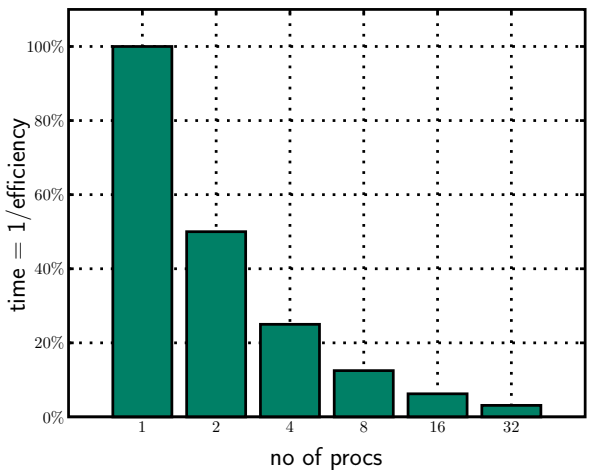
Strong and weak scaling/speedup

Strong scalability



Weak scalability



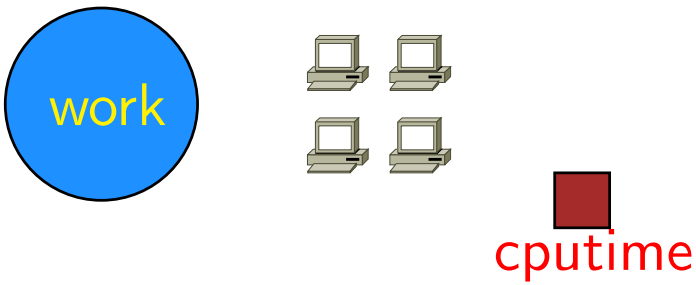


Fixed problem size
increase # of processor

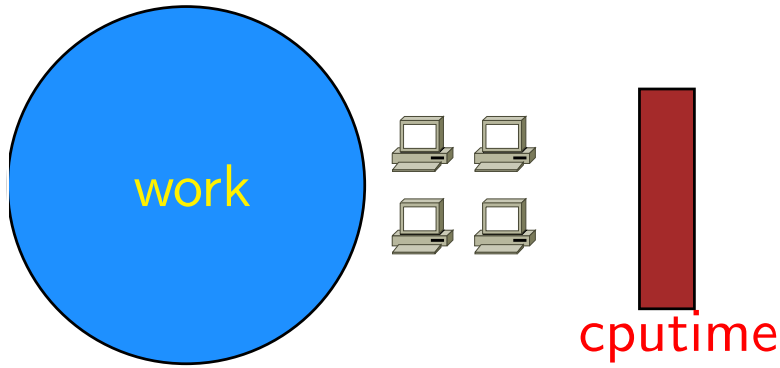
Parallel scalability

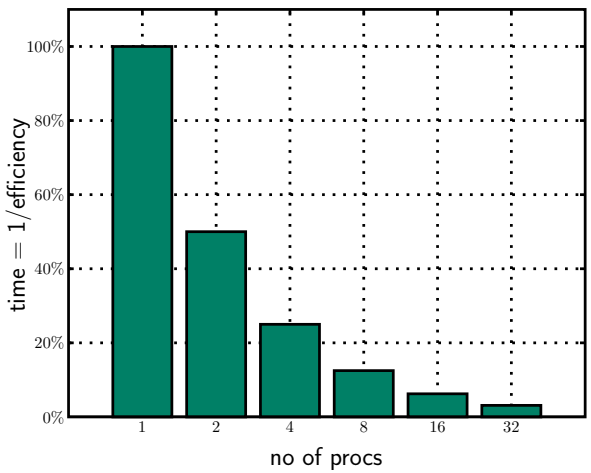
Strong and weak scaling/speedup

Strong scalability



Weak scalability



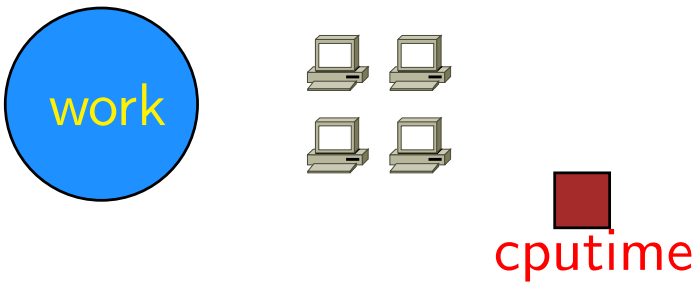


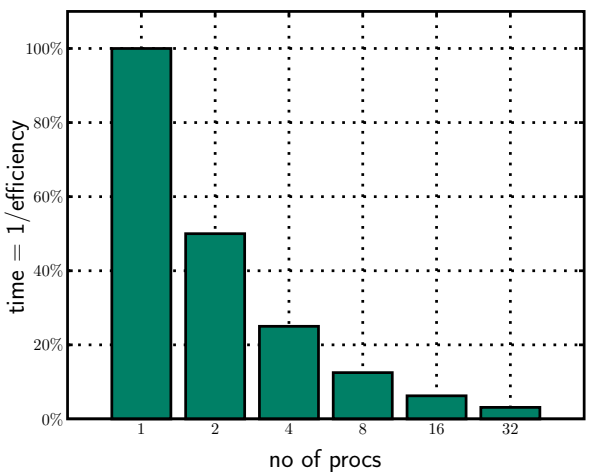
Fixed problem size
increase # of processor

Parallel scalability

Strong and weak scaling/speedup

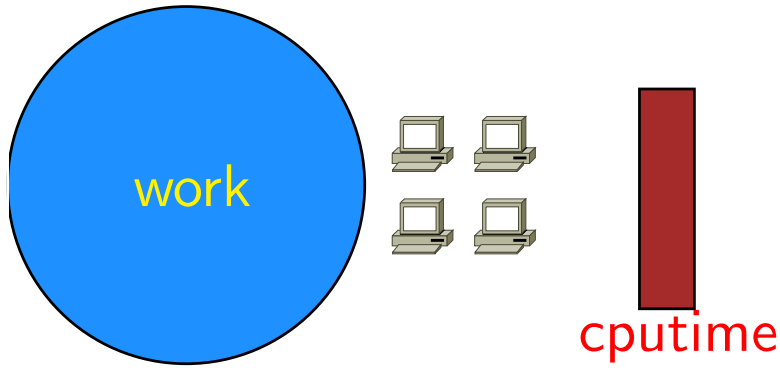
Strong scalability

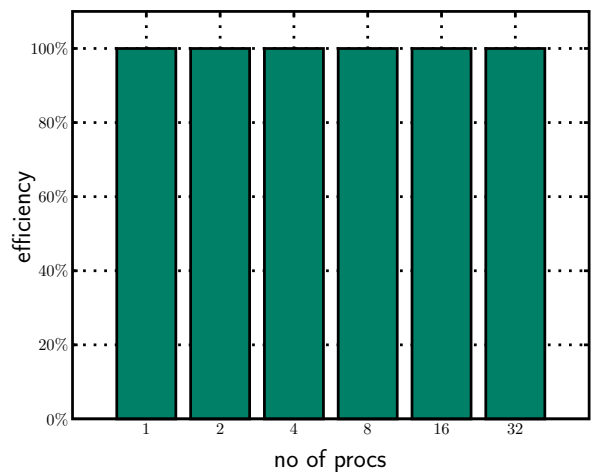




Fixed problem size
increase # of processor

Weak scalability





Increase problem size s.t.
fixed prob. size per processor

Outline

Organization issues

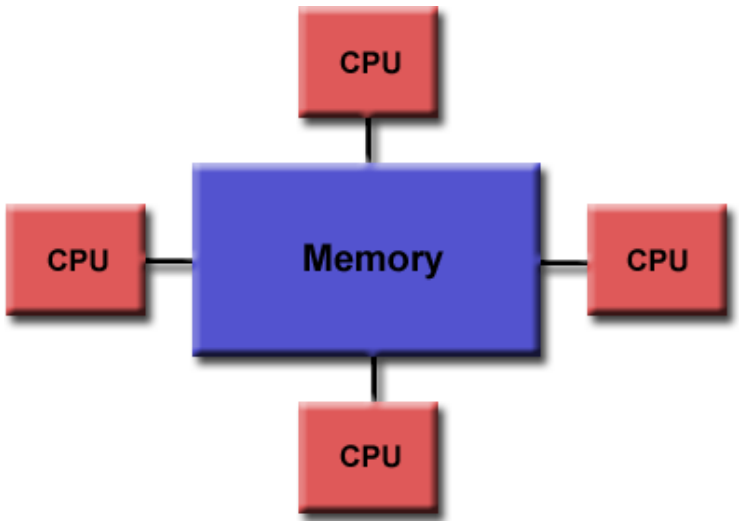
Parallel machines and programming models

Shared memory parallelism–OpenMP

Git—repository systems

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.

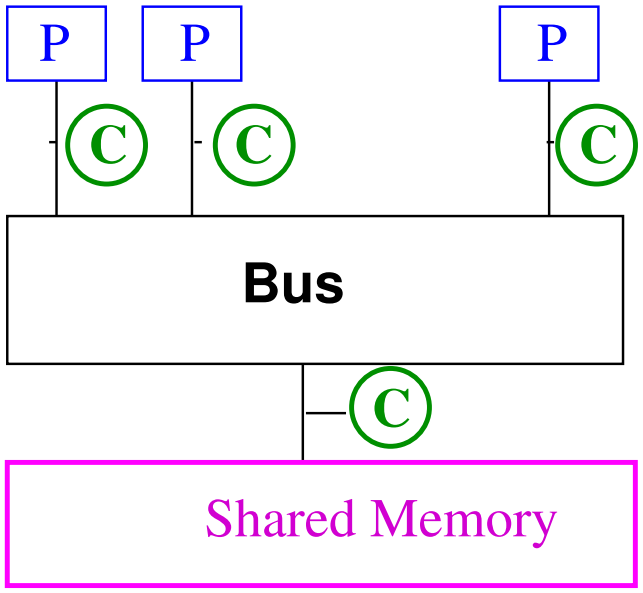
Shared memory programming

- POSIX Threads (`Pthreads`) library; more intrusive than OpenMP.
- PGAS languages: partitioned global address space: logically partitioned but can be programmed like a global memory address space (communication is taken care of in the background)
- OpenMP: open multi-processing is a light-weight application interface (API), that supports shared memory parallelism.

Shared Memory Machine Model

Symmetric Multiprocessors (SMP): processors all connected to a large shared memory. Examples are processors connected by crossbar, or multicore chips.

Key characteristic is *uniform memory access (UMA)*

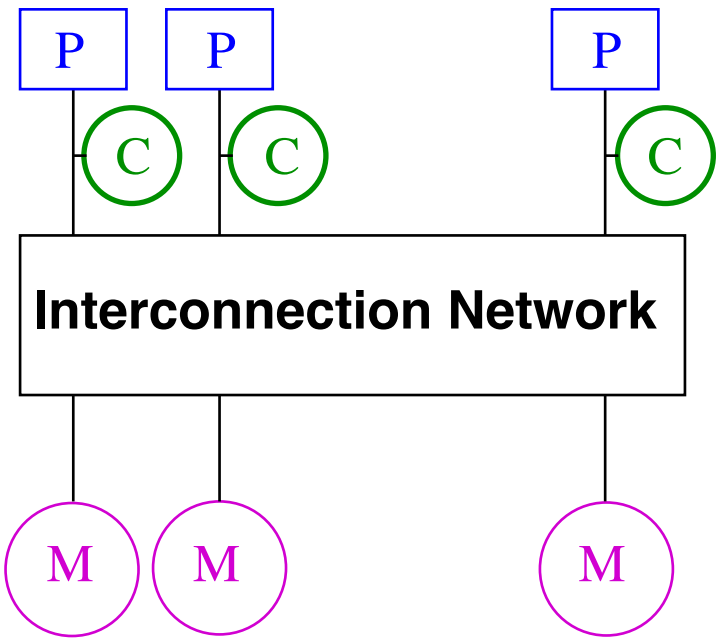


Caches are a problem - need to be kept *coherent* = when one CPU changes a value in memory, then all other CPUs will get the same value when they access it. All caches will show a coherent value.

Distributed Shared Memory

Memory is logically shared but physically distributed

- Any processor can access any address in memory
- Cache lines (or pages) passed around machine. Difficulty is *cache coherency protocols*.
- CC-NUMA architecture (if network is cache-coherent)



(SGI Altix at NASA Ames - had 10,240 cpus of Itanium 2 nodes connected by Infiniband, was ranked 84 in June 2010 list, ranked 3 in 2008)

Shared Memory Languages

- pthreads - POSIX (Portable Operating System Interface for Unix) threads; heavyweight, more clumsy
- PGAS languages - Partitioned Global Address Space UPC, Titanium, Co-Array Fortran; not yet popular enough, or efficient enough
- OpenMP - newer standard for shared memory parallel programming, lighter weight threads, not a programming language but an API for C and Fortran

OpenMP Overview

OpenMP is an API for multithreaded, shared memory parallelism.

- A set of compiler directives inserted in the source program
  - pragmas in C/C++ (pragma = compiler directive external to prog. lang. for giving additional info., usually non-portable, treated like comments if not understood)
  - (specially written) comments in fortran
- Library functions
- Environment variables

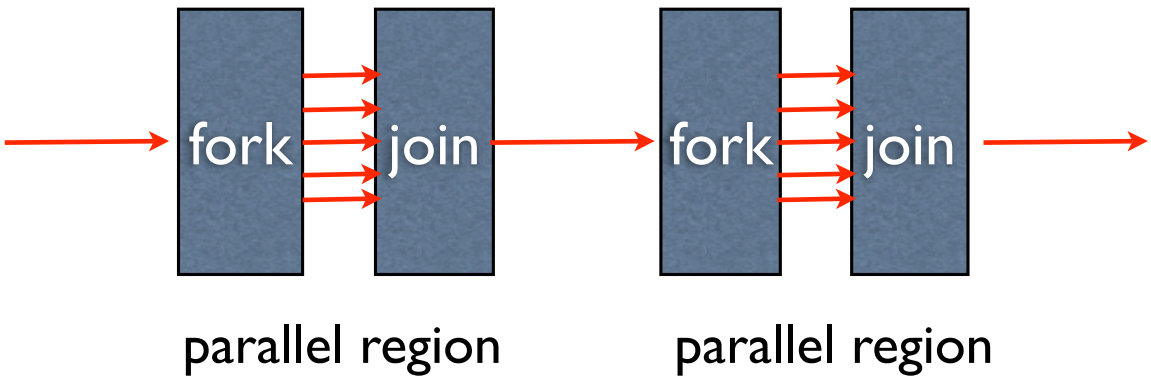
Goal is standardization, ease of use, portability. Allows incremental approach. Significant parallelism possible with just 3 or 4 directives. Works on SMPs and DSMs.

Allows fine and coarse-grained parallelism; loop level as well as explicit work assignment to threads as in SPMD.

Basic Idea

Explicit programmer control of parallelization using fork-join model of parallel execution

- all OpenMP programs begin as single process, the master thread, which executes until a parallel region construct encountered
- FORK: master thread creates team of parallel threads
- JOIN: When threads complete statements in parallel region construct they synchronize and terminate, leaving only the master thread. (similar to fork-join of Pthreads)



Basic Idea

- User inserts directives telling compiler how to execute statements
  - which parts are parallel
  - how to assign code in parallel regions to threads
  - what data is private (local) to threads
  - `#pragma omp` in C and `!$omp` in Fortran
- Compiler generates explicit threaded code
- Rule of thumb: One thread per core (2 or 4 with hyperthreading)
- Dependencies in parallel parts require synchronization between threads

Simple Example

```
#include <omp.h>
#include <stdio.h>

int main() {

#pragma omp parallel
printf("Hello world from thread %d\n",omp_get_thread_num());

return 0;
}
```

Compile line:

```
icc -qopenmp  helloWorld.c
gcc -fopenmp helloWorld.c
```

Simple Example

Sample Output:

```
MacBook-Pro% a.out
Hello world from thread 1
Hello world from thread 0
Hello world from thread 2
Hello world from thread 3

MacBook-Pro% a.out
Hello world from thread 0
Hello world from thread 3
Hello world from thread 2
Hello world from thread 1
```

how set # threads?

Setting the Number of Threads

Environment Variables:

`setenv OMP_NUM_THREADS 2` (cshell)

`export OMP_NUM_THREADS=2` (bash shell)

Library call:

`omp_set_num_threads(2)`

```
#include <omp.h>
#include <stdio.h>

int main() {

  omp_set_num_threads(2);

#pragma omp parallel
  printf("Hello world from thread %d\n",omp_get_thread_num());

return 0;
}
```

Parallel Construct

```
#include <omp.h>

int main(){
  int var1, var2, var3;

  ...serial Code

  #pragma omp parallel private(var1, var2) shared (var3)
 {
    ...parallel section
 }

 ...resume serial code

}
```

Parallel Directives

- When a thread reaches a PARALLEL directive, it becomes the master and has thread number 0.
- All threads execute the same code in the parallel region (Possibly redundant, or use work-sharing constructs to distribute the work)
- There is an implied barrier* at the end of a parallel section. Only the master thread continues past this point.
- If a thread terminates within a parallel region, all threads will terminates, and the result is undefined.
- Cannot branch into or out of a parallel region.

barrier - all threads wait for each other; no thread proceeds until all threads have reached that point

Parallel Directives

- If program compiled serially, openMP pragmas and comments ignored, stub library for omp library routines
- easy path to parallelization
- One source for both sequential and parallel helps maintenance.

Work-Sharing Constructs

- work-sharing construct divides work among member threads. Must be dynamically within a parallel region.
- No new threads launched. Construct must be encountered by all threads in the team.
- No implied barrier on entry to a work-sharing construct; Yes at end of construct.

3 types of work-sharing construct (4 in Fortran - array constructs):

- for loop: share iterates of `for` loop ("data parallelism") iterates must be independent
- sections: work broken into discrete section, each executed by a thread ("functional parallelism")
- single: section of code executed by one thread only

FOR directive schedule example

```
#include <stdio.h>
#include <omp.h>

#define N 20

int main(){

int sum = 0;
int a[N],i;

#pragma omp parallel

#pragma omp for
  for(i=0;i<N;i++){
    a[i] = i;
    printf(" iterate i=%3d by thread %3d\n",i,omp_get_thread_num());
  }

return 0;

}
```

FOR directive schedule example

```
energon1%  a.out
 iterate i=  0 by thread   0
 iterate i=  1 by thread   0
 iterate i=  2 by thread   0
 iterate i= 12 by thread   4
 iterate i= 13 by thread   4
 iterate i=  6 by thread   2
 iterate i=  7 by thread   2
 iterate i=  8 by thread   2
 iterate i=  9 by thread   3
 iterate i= 10 by thread   3
 iterate i= 11 by thread   3
 iterate i=  3 by thread   1
 iterate i=  4 by thread   1
 iterate i=  5 by thread   1
 iterate i= 18 by thread   7
 iterate i= 19 by thread   7
 iterate i= 16 by thread   6
 iterate i= 17 by thread   6
 iterate i= 14 by thread   5
 iterate i= 15 by thread   5
```

```
energon1%  a.out
 iterate i=  9 by thread   3
 iterate i= 10 by thread   3
 iterate i= 11 by thread   3
 iterate i=  6 by thread   2
 iterate i=  7 by thread   2
 iterate i=  8 by thread   2
 iterate i=  0 by thread   0
 iterate i=  1 by thread   0
 iterate i=  2 by thread   0
 iterate i= 12 by thread   4
 iterate i= 13 by thread   4
 iterate i= 14 by thread   4
 iterate i=  3 by thread   1
 iterate i=  4 by thread   1
 iterate i= 15 by thread   5
 iterate i= 16 by thread   5
 iterate i= 17 by thread   5
 iterate i= 18 by thread   6
 iterate i= 19 by thread   6
 iterate i=  5 by thread   1
```

for loop with 20 iterations and 8 threads:

icc: 4 threads get 3 iterations and 4 threads get 2
gcc: 6 threads get 3 iterations, 1 thread gets 2, 1 gets none

OMP Directives

All directives:

```
#pragma omp directive  [clause ...]
                        if (scalar_expression)
                        private (list)
                        shared (list)
                        default (shared | none)
                        firstprivate (list)
                        reduction (operator: list)
                        copyin (list)
                        num_threads (integer-expression)
```

Directives are:

- Case sensitive (not for Fortran)
- Only one directive-name per statement
- Directives apply to at most one succeeding statement, which must be a structured block.
- Continue on succeeding lines with backslash (`"\"`)

FOR directive

```
#pragma omp for [clause ...]
                    schedule (type [,chunk])
                    private (list)
                    firstprivate(list)
                    lastprivate(list)
                    shared (list)
                    reduction (operator: list)
                    nowait
```

SCHEDULE: describes how to divide the loop iterates

- **static** = divided into pieces of size *chunk*, and statically assigned to threads. Default is approx. equal sized chunks (at most 1 per thread)
- **dynamic** = divided into pieces of size *chunk* and dynamically scheduled as requested. Default chunk size 1.
- **guided** = size of chunk decreases over time. (Init. size proportional to the number of unassigned iterations divided by number of threads, decreasing to *chunk size*)
- **runtime**=schedule decision deferred to runtime, set by environment variable OMP_SCHEDULE.

Example: FOR directive

```
#include <omp.h>
#define  CHUNKSIZE 100
#define  N         1000

int main(){
   int i, chunk;
   float a[N], b[N], c[N];
   ...  /* initialize a, b  */
   chunk = CHUNKSIZE;

   #pragma omp parallel shared(a,b,c,chunk) private(i)
   {
      #pragma omp for schedule(dynamic,chunk) nowait
      for (i=0; i<N; i++)
          c[i] = a[i] + b[i];
    } /* end parallel section */
} /* end main */
```

FOR example

```
#pragma omp parallel shared(n,a,b,x,y), private(i)
{ // start parallel region

   #pragma omp for nowait
   for (i=0;i<n;i++)
     b[i] = += a[i];

   #pragma omp for nowait
   for (i=0;i<n;i++)
     x[i] = 1./y[i];

} // end parallel region  (implied barrier)
```

Spawning tasks is expensive: reuse if possible.
nowait clause: minimize synchronization.

SINGLE directive

```
#pragma omp single [clause ...]
                   private (list)
                   firstprivate(list)
                   nowait
 structured block
```

- SINGLE directive says only one thread in the team executes the enclosed code
- useful for code that isn't thread-safe (e.g. I/O)
- rest of threads wait at the end of enclosed code block (unless NOWAIT clause specified)
- no branching in or out of SINGLE block

Clauses

These clauses not strictly necessary but may be convenient (and may have performance penalties too).

- lastprivate private data is undefined after parallel construct. this gives it the value of last iteration (as if sequential) or sections construct (in lexical order).
- firstprivate pre-initialize private vars with value of variable with same name before parallel construct.
- default (`none | shared`). In fortran can also have private. Then only need to list exceptions. (none is better habit).
- nowait suppress implicit barrier at end of work sharing construct. Cannot ignore at end of parallel region. (But no guarantee that if have 2 `for` loops where second depends on data from first that same threads execute same iterates)

More Clauses

- if (logical expr) true = execute parallel region with team of threads; false = run serially (loop too small, too much overhead)
- reduction for assoc. and commutative operators compiler helps out; reduction variable is shared by default (no need to specify).

```
#pragma omp parallel for default(none) \
                         shared(n,a) \
                         reduction(+:sum)
  for (i=0;i<n;i++)
    sum += a[i]
  /* end of parallel reduction  */
```

  Also other arithmetic and logical ops., `min`,`max` instrinsics in Fortan only.

- copyprivate only with single direction. one thread reads and initializes private vars. which are copied to other threads before they leave barrier.
- threadprivate variables persist between different parallel sections (unlike private vars). (applies to global vars. must have dynamic false)

Synchronization

Synchronization: needed to protect access to shared data

- Implicit barrier synchronization at end of parallel region (no explicit support for synch. subset of threads). Can invoke explicitly with `#pragma omp barrier`. All threads must see same sequence of work-sharing and barrier regions .
- critical sections: only one thread at a time in critical region with the same name. `#pragma omp critical [(name)]`
- atomic operation: protects updates to individual memory loc. Only simple expressions allowed. `#pragma omp atomic`
- also flush, locks, master operations - perhaps discussed later.

At all these (implicit or explicit) synchronization points OpenMP ensures that threads have consistent values of shared data.

Critical Example

```
#pragma omp parallel sections
{
  #pragma parallel section
  {
      task = produce_task();
      #pragma omp critical (task_queue)
      {
         insert_into_queue(task);
      }
  }
  #pragma parallel section
  {
     #pragma omp critical (task_queue)
     {
       task = delete_from_queue(task);
     }
     consume_task(task);
  }
}
```

Atomic Examples

```
#pragma omp parallel shared(n,ic) private(i)
   for (i=0;i<n;i++){
     #pragma omp atomic
          ic = ic +1;
   }
```

`ic` incremented atomically

```
#pragma omp parallel shared(n,ic) private(i)
   for (i=0;i<n;i++){
     #pragma omp atomic
        ic = ic + bigfunc();
   }
```

`bigfunc` not atomic, only `ic` update

Atomic Example

```
int sum = 0;
#pragma omp parallel for shared(n,a,sum)
{
  for (i=0; i<n; i++){
  #pragma omp atomic
     sum = sum + a[i];
  }
}
```

Better to use a *reduction* clause:

```
int sum = 0;
#pragma omp parallel for shared(n,a)  \
                     reduction(+:sum)
{
  for (i=0; i<n; i++){
     sum += a[i];
  }
}
```

Reductions

Many different associative operations can be used in reductions:

Operator	Initial value
+	0
*	1
-	0

C/C++ only	
Operator	Initial value
&	~0
|	0
^	0
&&	1
||	0

Fortran Only	
Operator	Initial value
.AND.	.true.
.OR.	.false.
.NEQV.	.false.
.IEOR.	0
.IOR.	0
.IAND.	All bits on
.EQV.	.true.
MIN*	Largest pos. number
MAX*	Most neg. number

Outline

Organization issues

Parallel machines and programming models

Shared memory parallelism–OpenMP

Git—repository systems

Why Use Version Control?

Slides adapted from Andreas Skielboe

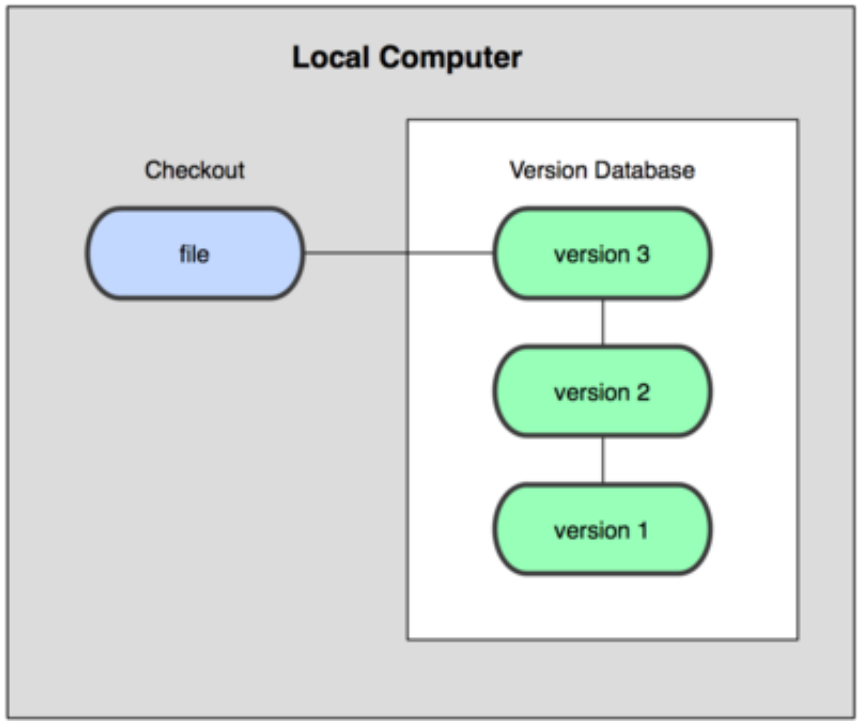
A Version Control System (VCS) is an integrated fool-proof framework for

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

... with minimal overhead.

Local Version Control Systems

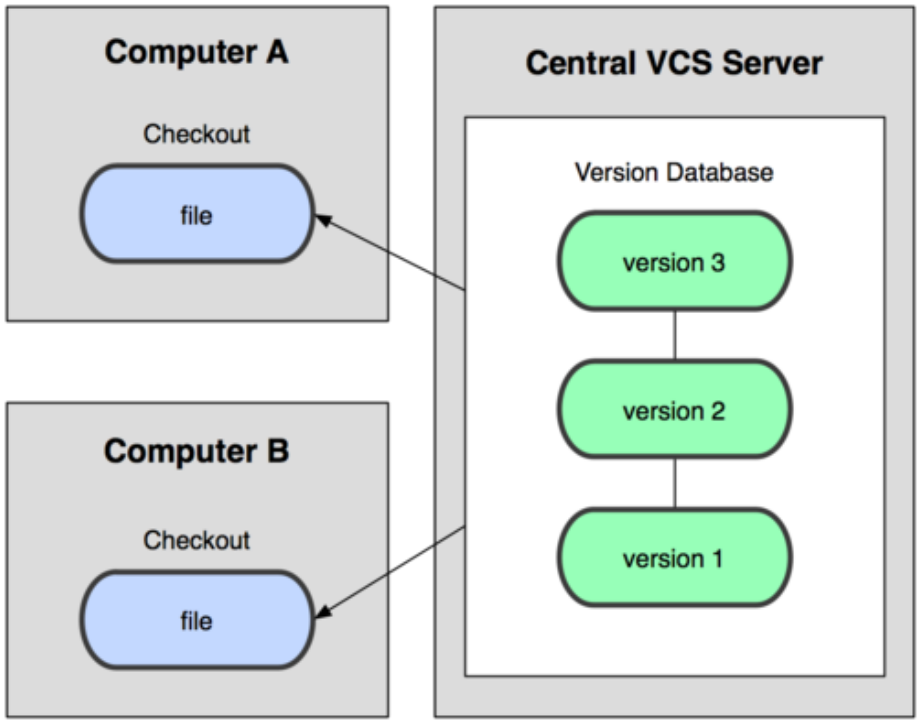
Conventional version control systems provides some of these features by making a local database with all changes made to files.



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

Centralized Version Control Systems

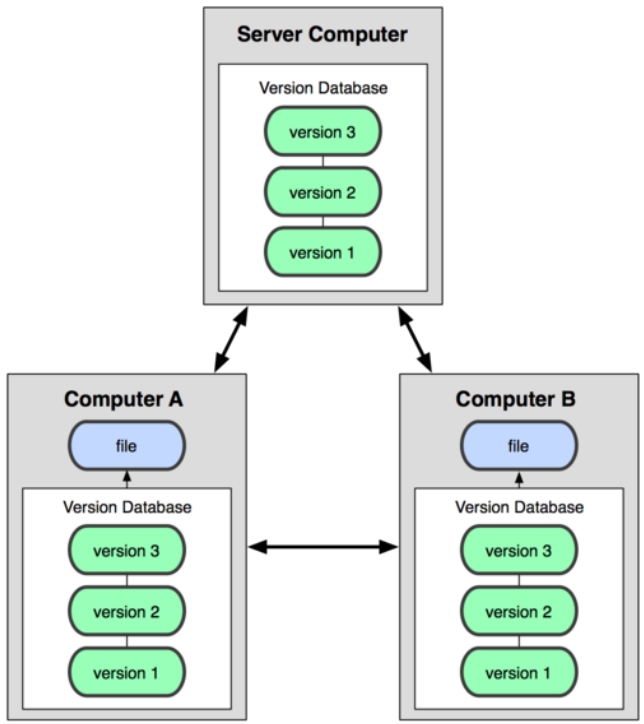
To enable synchronization and collaborative features the database is stored on a central VCS server, where everyone works in the same database.



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.

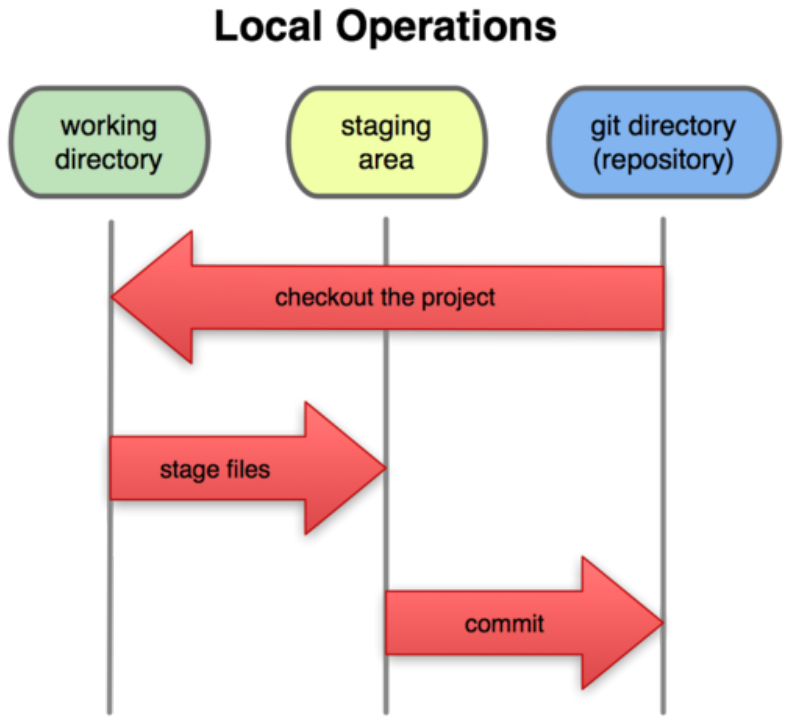
Git Basics - The Git Workflow

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*.

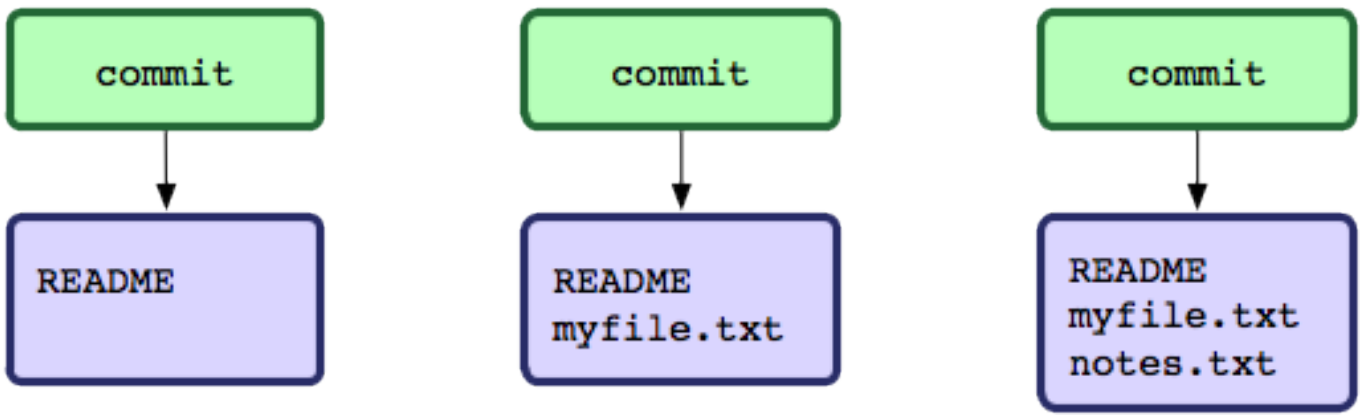
Git Basics - The Three States

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

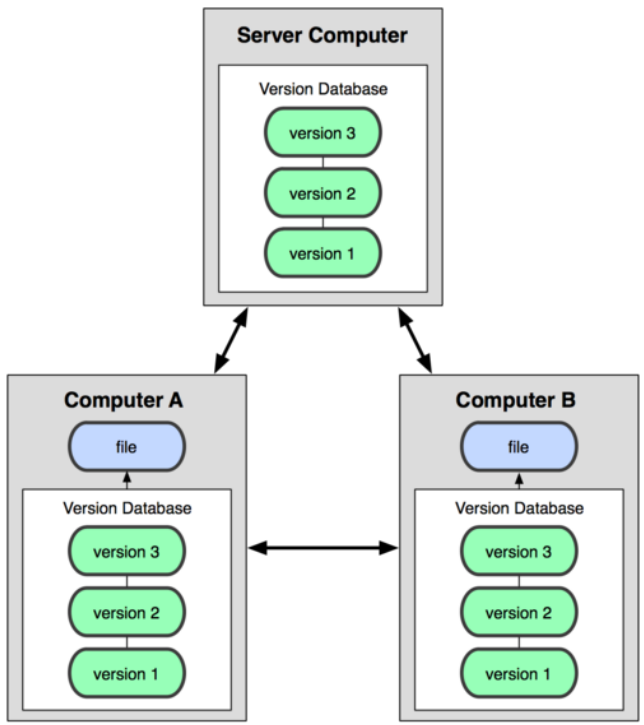
```
$ git config --list
```

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