Advanced Topics in Numerical Analysis: High Performance Computing

MATH-GA 2012.001 & CSCI-GA 2945.001

Georg Stadler
Courant Institute, NYU
stadler@cims.nyu.edu

Spring 2017, Thursday, 5:10–7:00PM, WWH #512

Feb. 24, 2017

Outline

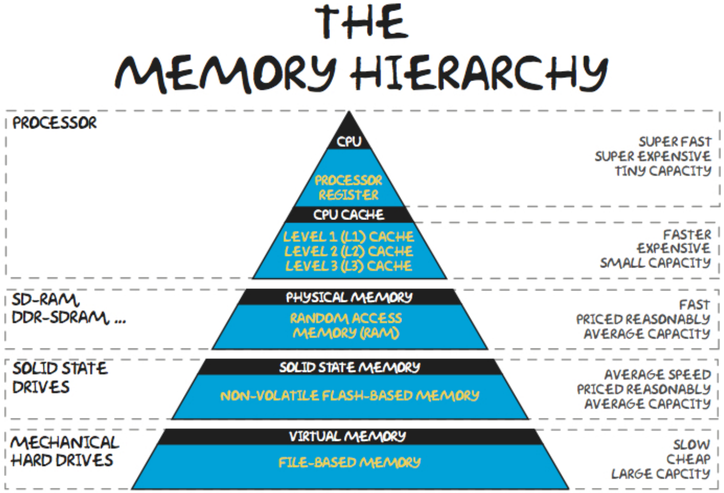
Organization issues

- Please register for an XSEDE account such that Bill can add you to the Stampede allocation (see his post on Piazza). You're welcome to do that even if you only audit the course
- Please fill out the poll for makeup class either on March 6 or 7
- Next week, my colleague Marsha Berger will fill in for me
- And yes, there will be new homework at some point. . .

Outline

Memory hierarchies

On my Mac Book Pro: 32KB L1 Cache, 256KB L2 Cache, 3MB Cache, 8GB RAM



CPU: $\mathcal{O}(1\text{ns})$, L2/L3: $\mathcal{O}(10\text{ns})$, RAM: $\mathcal{O}(100\text{ns})$, disc: $\mathcal{O}(10\text{ms})$

Memory hierarchies

Important terms:

- latency: time it takes to load/write data from/at a specific location in RAM to/from the CPU registers (in seconds)

Memory hierarchies

Important terms:

- latency: time it takes to load/write data from/at a specific location in RAM to/from the CPU registers (in seconds)
- bandwidth: rate at which data can be read/written (for large data); in (bytes/second);

Memory hierarchies

Important terms:

- latency: time it takes to load/write data from/at a specific location in RAM to/from the CPU registers (in seconds)
- bandwidth: rate at which data can be read/written (for large data); in (bytes/second);
- cache-hit: required data is available in cache $\Rightarrow$ fast access

Memory hierarchies

Important terms:

- latency: time it takes to load/write data from/at a specific location in RAM to/from the CPU registers (in seconds)
- bandwidth: rate at which data can be read/written (for large data); in (bytes/second);
- cache-hit: required data is available in cache $\Rightarrow$ fast access
- cache-miss: required data is not in cache and must be loaded from main memory (RAM) $\Rightarrow$ slow access

Memory hierarchies

Important terms:

- latency: time it takes to load/write data from/at a specific location in RAM to/from the CPU registers (in seconds)
- bandwidth: rate at which data can be read/written (for large data); in (bytes/second);
- cache-hit: required data is available in cache $\Rightarrow$ fast access
- cache-miss: required data is not in cache and must be loaded from main memory (RAM) $\Rightarrow$ slow access

Computer architecture is complicated. We need a basic performance model.

- Processor needs to be “fed” with data to work on.
- Memory access is slow; memory hierarchies help.

Memory hierarchy
Simple model

1. Only consider two levels in hierarchy, fast (cache) and slow (RAM) memory
2. All data is initially in slow memory
3. Simplifications:
   - Ignore that memory access and arithmetic operations can happen at the same time
   - assume time for access to fast memory is 0
4. Computational intensity: flops per slow memory access

$$q = \frac{f}{m}, \text{ where } f \ldots \#\text{flops}, m \ldots \#\text{slow memop.}$$

Computational intensity should be as large as possible.

Memory hierarchy

Example: Matrix-matrix multiply Comparison between naive and blocked optimized matrix-matrix multiplication for different matrix sizes: Different algorithms can increase the computational intensity significantly.
BLAS: Optimized Basic Linear Algebra Subprograms

Memory hierarchy

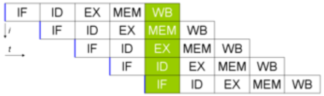
Example: Matrix-matrix multiply Comparison between naive and blocked optimized matrix-matrix multiplication for different matrix sizes: Different algorithms can increase the computational intensity significantly.
BLAS: Optimized Basic Linear Algebra Subprograms

- Temporal and spatial locality is key for fast performance.
- Since arithmetic is cheap compared to memory access, one can consider making extra flops if it reduces the memory access.
- In distributed-memory parallel computations, the memory hierarchy is extended to data stored on other processors, which is only available through communication over the network.

Outline

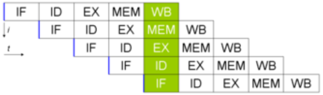
Levels of parallelism

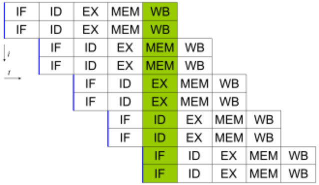
- Parallelism at the bit level (64-bit operations)



Levels of parallelism

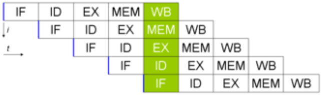
- Parallelism at the bit level (64-bit operations)
- Parallelism by pipelining (overlapping of execution of multiple instructions); “assembly line” parallelism, Instruction-Level-Parallelism (ILP); several operators per cycle

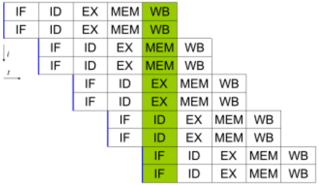




Levels of parallelism

- Parallelism at the bit level (64-bit operations)
- Parallelism by pipelining (overlapping of execution of multiple instructions); "assembly line" parallelism, Instruction-Level-Parallelism (ILP); several operators per cycle
- multiple functional units parallelism: ALUs (algorithmic logical units), FPUs (floating point units), load/store memory units,...

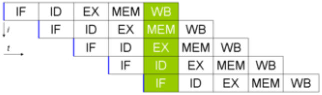


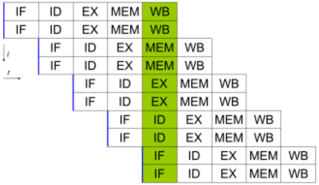


all of the above assume single sequential control flow

Levels of parallelism

- Parallelism at the bit level (64-bit operations)
- Parallelism by pipelining (overlapping of execution of multiple instructions); "assembly line" parallelism, Instruction-Level-Parallelism (ILP); several operators per cycle
- multiple functional units parallelism: ALUs (algorithmic logical units), FPUs (floating point units), load/store memory units,...





all of the above assume single sequential control flow

- process/thread level parallelism: independent processor cores, multicore processors; parallel control flow

Amdahl's law

Is there enough parallelism in my problem?

Suppose only part of the application is parallel

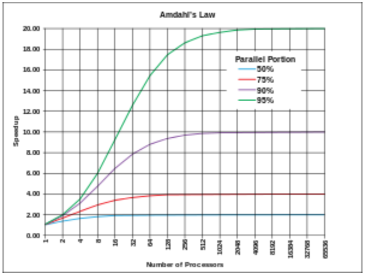
Amdahl's law:

- Let s be the fraction of work done sequentially, and $(1-s)$ the part that is done in parallel
- $p \ldots$ 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

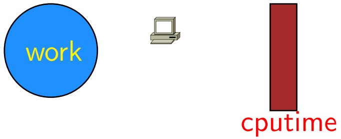
In parallel computations, the work should be distributed *evenly* across workers/processors.

- 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

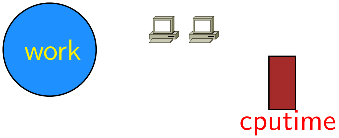
Strong scalability



Parallel scalability

Strong and weak scaling/speedup

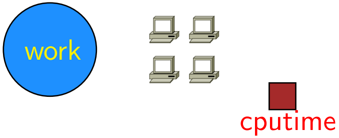
Strong scalability



Parallel scalability

Strong and weak scaling/speedup

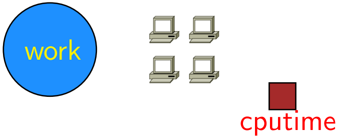
Strong scalability

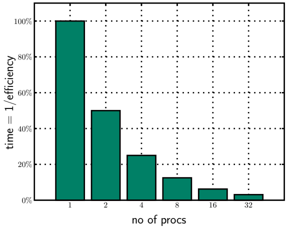


Parallel scalability

Strong and weak scaling/speedup

Strong scalability

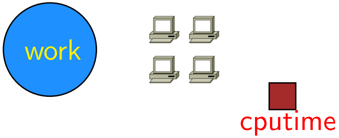


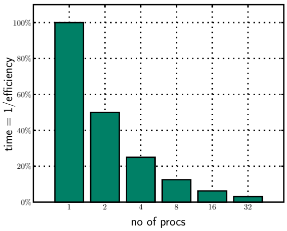


Parallel scalability

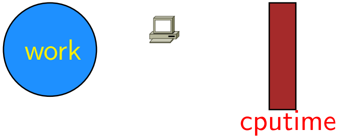
Strong and weak scaling/speedup

Strong scalability





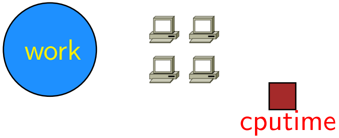
Weak scalability

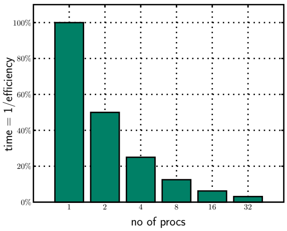


Parallel scalability

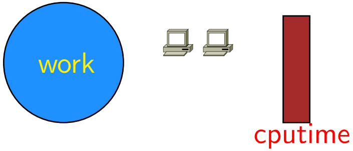
Strong and weak scaling/speedup

Strong scalability





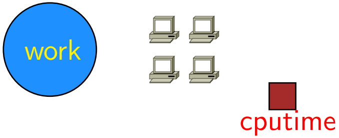
Weak scalability

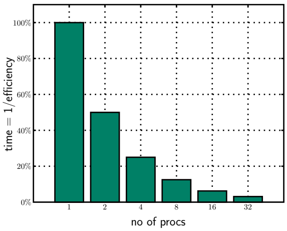


Parallel scalability

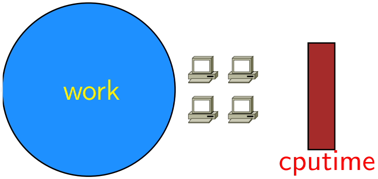
Strong and weak scaling/speedup

Strong scalability





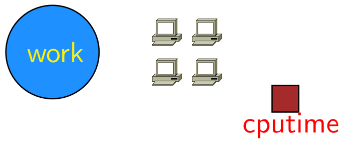
Weak scalability

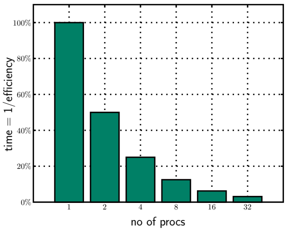


Parallel scalability

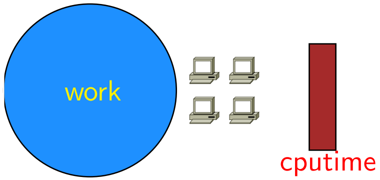
Strong and weak scaling/speedup

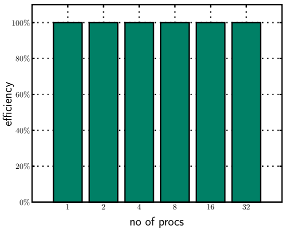
Strong scalability





Weak scalability

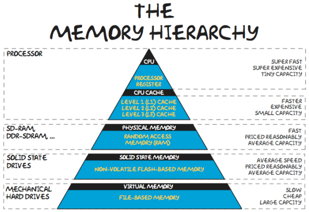




Locality and parallelism

Locality of data to processing unit is critical in general, and even more so on parallel computers.

- Memory hierarchies even deeper on parallel computers.
- In parallel computing, data often must be communicated through the network, which is slow. Again, locality is key.
- Computation should mostly be on local (cache-local, in DRAM, processor-local, disc-local) data



Outline

Why Use Version Control?

Slides adapted from Andreas Skielboe

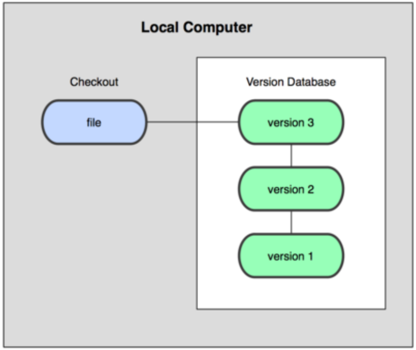
A Version Control System (VCS) is an integrated fool-proof framework for

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

... with minimal overhead.

Local Version Control Systems

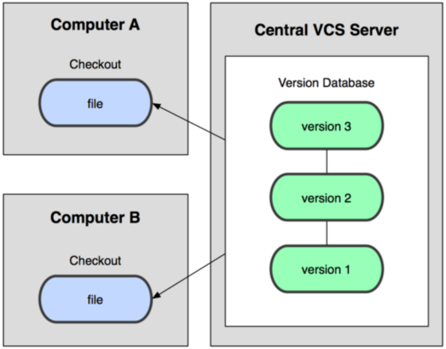
Conventional version control systems provides some of these features by making a local database with all changes made to files.



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

Centralized Version Control Systems

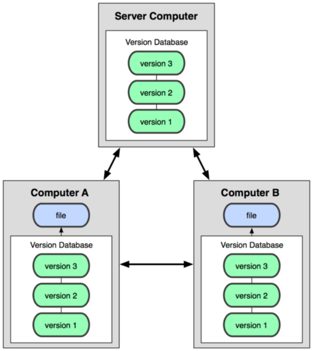
To enable synchronization and collaborative features the database is stored on a central VCS server, where everyone works in the same database.



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.

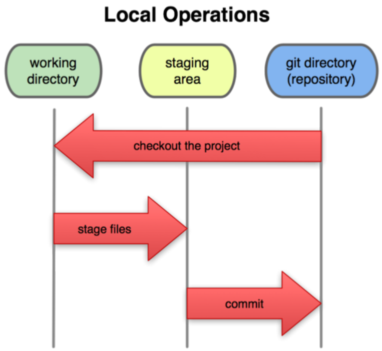
Git Basics - The Git Workflow

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*.

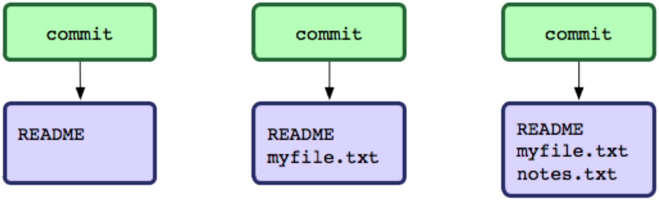
Git Basics - The Three States

The three basic states of files in your Git repository:



Git Basics - Commits

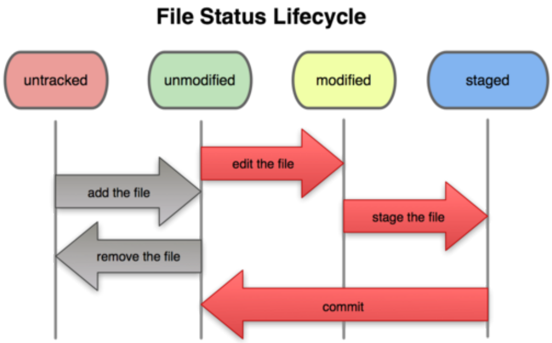
Each commit in the git directory holds a snapshot of the files that were staged and thus went into that commit, along with author information.



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

Git Basics - File Status Lifecycle

Files in your working directory can be in four different states in relation to the current commit.



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

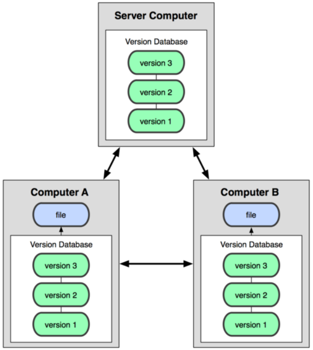
```
$ git config --list
```

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

Outline

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.

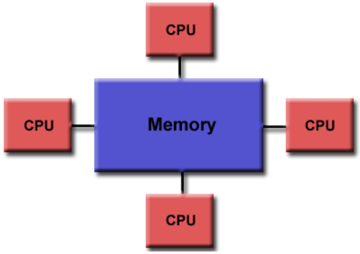
Programming model must reflect architecture

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?

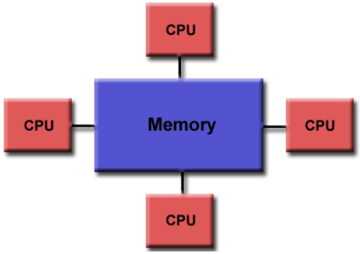
Shared memory programming model

- 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

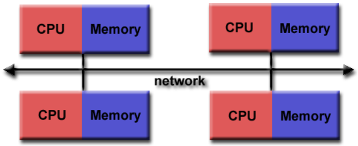
- Program is a collection of control threads, that are created dynamically
- Each thread has private and shared variables
- Threads can exchange data by reading/writing shared variables
- Danger: more than 1 processor core reads/writes to a memory location: race condition



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`

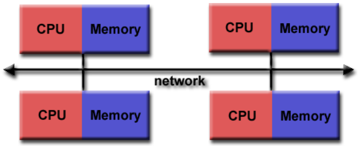
Distributed memory programming model

- 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).

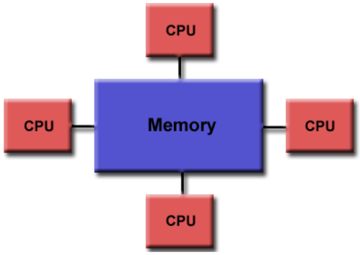
Other parallel programming approaches

- 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
- ...

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

- 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.

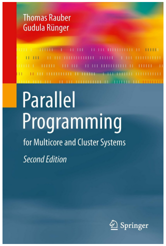
Shared memory programming—Literature

OpenMP standard online:
www.openmp.org

Very useful online course:
www.youtube.com/user/OpenMPARB/

Recommended reading:

Chapter 6 in



Chapter 7 in