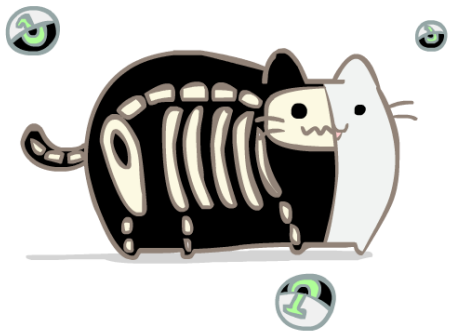
High Performance Programming, Lecture 1

Course introduction

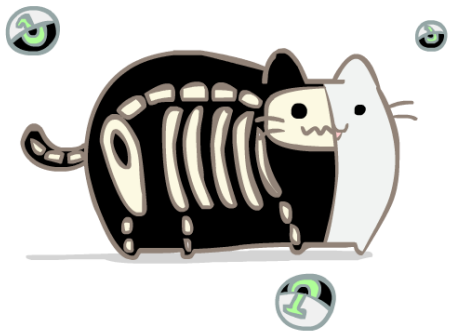
Personal introductions

Lectures 1-8:
Teacher: **Anastasia Kruchinina**
PhD Student at IT department
Room: ITC 2348
anastasia.kruchinina@it.uu.se



Personal introductions

Lectures 1-8:
Teacher: **Anastasia Kruchinina**
PhD Student at IT department
Room: ITC 2348
anastasia.kruchinina@it.uu.se



Lectures 9-14:
Teacher: **Jarmo Rantakokko**
Senior lecturer at IT department
Room: ITC 2435
jarmo.rantakokko@it.uu.se

First of all

- Make sure you have a working Linux account and can log on to the systems (rooms 1515, 2516, 1549 and 2315)
- Make sure you have access to the **Student Portal** (Studentportalen)
- All students should **web-register** for the course in the Student Portal.
- Course homepage also found in the Student Portal
- Schedule also found via the Student Portal / TimeEdit
- Check if your email in Student Portal is correct. You should be able to receive my emails!

If there are problems with course registration, contact IT-kansliet (the Student Office): `it-kansli@it.uu.se`

Course layout

The course consists of the following parts, in this order:

- Introduction
- C programming basics
- Optimizing serial code
- Parallelization using Pthreads
- Parallelization using OpenMP
- A few extra things

Questions

Please, ask questions!

Serial optimization aspects

"Serial optimization" aspects covered:

- Optimisations that do not involve modifying the source code
- Optimisations that do involve modifying the source code

- Reducing instructions (how to do less work)
- Efficient memory usage (how to wait less, how to use less space)
- Instruction-level parallelism (how to do work faster)

Parallelization aspects

This course includes **shared-memory parallelization** approaches. For distributed-memory parallelization (e.g. MPI) you need to take other courses.

Parallelization topics covered:

- POSIX Threads (Pthreads)
- Threading using OpenMP
- GPU programming using CUDA (extra part)

Course structure

Each week there are two lectures and one lab covering both lectures.

7 assignments, handed in via Student Portal.

Socrative STUDENT

In lectures, the **Socrative STUDENT** smartphone app will be used. It is available in App Store and Google Play.

If you can, please install it. If you do not have a supported smartphone, or have trouble installing the app for some reason, don't worry. You can work in groups of 2 students, work together with someone who has the app.

We will use it already today!

Labs

For each lab you get a pdf file with instructions, and a package of source code files for that lab.

Instructions and other files for each lab will be uploaded 24 hours before the lab so you can look at the lab beforehand if you want. Files available under DOCUMENT in the Student Portal.

Work with the labs individually or in groups of 2-3 students. If working in a group, make sure that you still **understand things individually**.

Work on lab computers or on your own laptop, or both.

Labs

Labs are mandatory. Check that they get marked as completed in the Progress function in the Student Portal.

At each lab we keep a list of student names, if you are there and working actively with the lab we check you on the list.

If you miss a lab you must submit a small report for that lab instead, no later than 3 days after the lab. If you are too late with the report then you may not pass the course in time.

Submit report for the lab using Student portal under ASSIGNMENTS $\longrightarrow$ " Extra reports" or send me by email (first option is preferable)

Assignments

	Topic
Assignment 1	Intro/Makefiles
Assignment 2	C programming
Assignment 3	Serial optimization
Assignment 4	More serial opt.
Assignment 5	Pthreads
Assignment 6	OpenMP
Individual assignment	(More info from Jarmo)

For Assignments 2-6 you are expected to work in pairs, "groups" with two students/group, using the "group division" function in the Student Portal.

In assignments 3-6 you work on the same problem and completing the same report.

Assignments

Assignments done in pairs

Write in the report who in your group did what.

When you are in a group, you can access the files for Assignment 2.

If you miss deadline for submitting an assignment then you may not pass the course in time.

If you do not have a teammate, send email to Anastasia about it, then I can pair you up with someone.

If you really, really, *really* want to work alone and/or have special reasons why you need to work alone, then explain that to me. In special cases working alone may be allowed.

Assignments

Use version control system!

For Assignment 3 (and assignments 4, 5, 6 later) it is highly recommended that you use some kind of version control system, e.g. **svn** or **git**. (But you don't have to)

Websites like GitHub/Bitbucket/GitLab give free access to some of their services for students, for small projects.

Please do not put your assignment codes in public repositories — that makes it too easy for other students to copy your code.

Grading

How grading will work:

During the course, lectures, labs, assignments, etc: no question is too stupid, nothing will affect grades. All we do here is about learning, *you do not need to worry about grading here*.

Labs and assignments 1-6 are only pass/fail. You just need to pass, the final grade is not affected.

After feedback on the assignment 6 you submit the final report and all needed codes **for grading**.

Your grade is determined as
40% grade of the group assignment + 60% grade of the individual project (so both parts are important!)

Do not copy code from other students!

You are of course allowed, and encouraged, to talk to other students about the course. However: **do not copy another student's code**.

It is very important that you learn to write and understand your own code. Talking to others to understand things better is very good, but copying their code is not good. Write your own code, using your own names for your variables and functions, etc.

So, please do not copy code from other students, and do not give your code to other students. If they ask for it, say no. Teachers are required to report plagiarism.

Which computer(s) to use?

For labs and assignments (and the final miniproject) you are allowed to use almost any computer you want.

- You can use lab computers, available in lab rooms 1515, 1549, 2507, 2510, 2516.
- You are allowed (and encouraged!) to use your own computer, preferably Linux/Mac, Windows also possible (see next slide).
- If using a Windows computer, get a Linux/unix-like environment using e.g. Cygwin or VirtualBox or VMware (but be aware that virtualization may reduce performance).

However, for assignments *your code must be portable* to the University servers so that your teachers can test it.

Which computer(s) to use?

If you want to use your own computer and it is a Mac or Windows computer, see additional info in the Student Portal, under CONTENT $\longrightarrow$ "Notes for Mac and Windows users"

Which computer(s) to use?

You can login remotely using ssh (and copy files using scp) to university computers:

Linux hosts:
`http://www.it.uu.se/datordrift/maskinpark/linux`

Solaris (Unix) hosts:
`http://www.it.uu.se/datordrift/faq/unixinloggning`

See small tutorial under CONTENT $\longrightarrow$ "Linux command line basics" in the Student Portal.

Reading

Two books:

Agner Fog
Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms
Technical University of Denmark
`http://www.agner.org/optimize/optimizing_cpp.pdf`
(updated 2018-08-18)

Peter Pacheco
An Introduction to Parallel Programming
Burlington: Elsevier Science, 2011.
Available as e-book via UU library

Plus: Additional material and links throughout the course

Additional course material

During the course, material will be added under DOCUMENT (Document collections) in the Student Portal:

Lab files: before each lab session, lab instructions and other files for that lab will be uploaded here.

Lecture slides: after each lecture, a pdf with lecture slides will be available here. Sometimes also extra files related to that lecture.

Example programs: some small example programs will be uploaded here, e.g. demonstrating certain optimization approaches.

See also CONTENT $\longrightarrow$ "Course material overview" in the Student Portal.

Questions?

Questions about formal things, examination, course material, etc?

Why C?

We want high performance.

Compiled languages generally allow us to obtain higher performance than languages that are interpreted or use an intermediate code, like Java. See Fog book, pages 8-10.

Most well-known compiled languages:

- C
- C++
- Pascal
- Fortran

We choose C because it is simple and extremely well established. What we learn for C is also directly applicable for C++.

How to increase computational power?

Nowadays, processor clock frequency no longer increases.

$\longrightarrow$ to get performance improvements we **need optimization** and/or **parallelization** of the code.

Improvement compared to previous systems lies in **larger number of cores**.

`www.top500.org`
IBM POWER9 22C 3.07GHz : 2,397,824 cores

Serial and parallel programming

In this course, we focus first on other optimization aspects, then on parallelization.

Parallel programming is important, but does not solve everything.

In massively parallel computations the **performance is often limited by the serial execution time for certain critical parts**.

We want to **combine serial optimization techniques with parallelization** to achieve the best performance on modern hardware.

Optimization important

$\longrightarrow$ **optimization of serial code remains very important**, and (perhaps increasingly) challenging.

Remember:

- Before attempting to optimize anything, find out which the critical parts are
- Try different compiler optimization flags
- Different optimization techniques have different effect depending on hardware
- Always check if your "optimization" helps!

Use scientific approach

Reproducibility

Reproducibility is important.

Examples:
When measuring timings for your code, check the variability.
Reported timings should be reproducible.
In written reports, make sure to **include all relevant details** so that the described results can be reproduced:

- Details about CPU: model name/number
- Compiler and compiler version

Imagine yourself in the position of the reader of your report: is all necessary information included?

Priorities

Priorities:

1. Correctness
2. Find out where the time goes
3. Improving algorithm usually more important than optimizations
4. Try different compiler optimization flags
5. Do your own optimizations

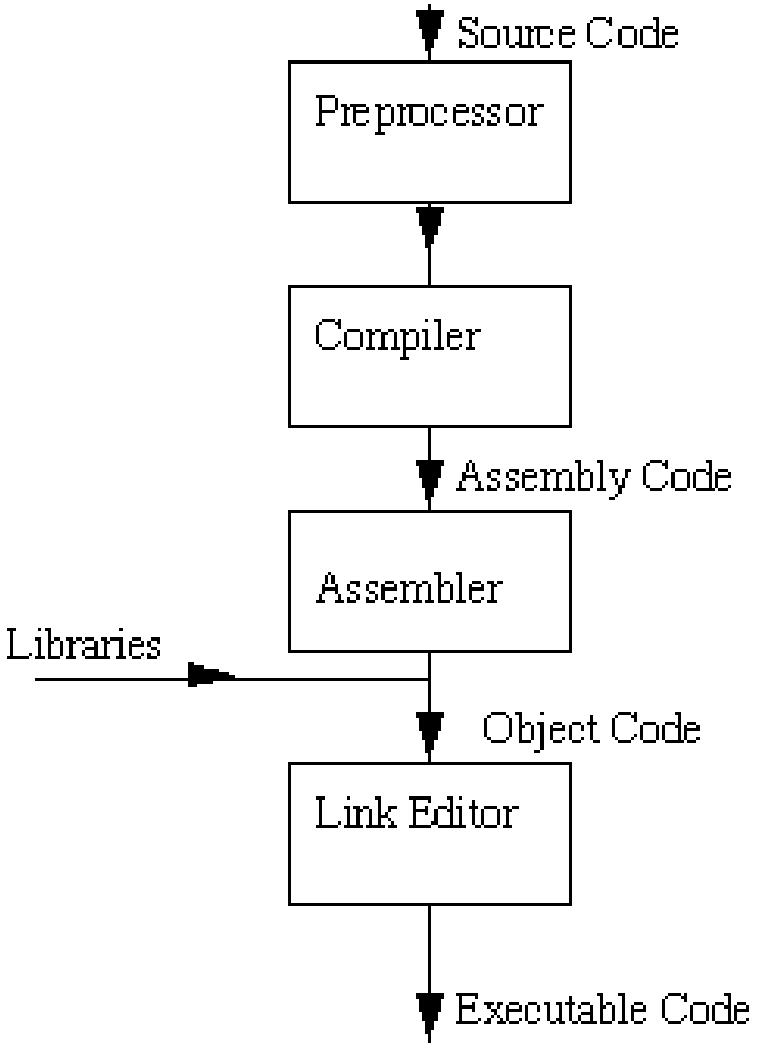
When to parallelize???

Remember:
Premature optimization is the root of all evil. – Donald Knuth

An overview of the build process for C/C++.

The build process is a process of transforming a written program into an executable.

In Linux/Unix it takes 4 distinct steps.



Preprocessing

The **preprocessor** takes your C code as input and generates another C code as output.

Preprocessing is used to combine different source files together, introduce code from an external source (e.g. a library), and to customise a code so it suits system or user requirements.

Preprocessing

- `#include` inserts code into a file. Typically, a C program code is separated into two sets of files:
  – header files (ending in .h) contain function prototypes
  – source files (ending in .c) contain the function definitions
- `#define` defines a macro.
  Examples: "`#define PI 3.14`" or "`#define SQR(x) ((x)*(x))`". The preprocessor will look for PI or SQR(...) and replace those tokens with 3.14 or ((...)*(...)), respectively.
- `#if, #ifdef, #ifndef, #endif` are used to add or remove code depending either on a boolean condition or whether a keyword is defined.

Preprocessing

Example

Check `https://gcc.gnu.org/onlinedocs/cpp/Preprocessor-Output.html`

Useful predefined macros, handy for debug prints or to annotate output:

- `__LINE__` Line of the current file
- `__FILE__` Name of the current file
- `__DATE__` Current date
- `__TIME__` Current time

Example: `gcc -E -o myprog.i myprog.c`

Preprocessing

Example

myprog.c:

```c
#include "myprog.h"

#define TWO 2

int main()
{
    int two = TWO;
    int sum = add(two, __LINE__);
    function_we_dont_use();
}

int add(int a, int b)
{
    return a + b;
}
```

myprog.h:

```c
#ifndef MYPROG_H
#define MYPROG_H

int add(int, int);

void function_we_dont_use();

#endif
```

myprog.i:

```c
....
# 1 "myprog.h" 1

int add(int, int);

void function_we_dont_use();
# 2 "myprog.c" 2

int main()
{
    int two = 2;
    int sum = add(two, 8);
    function_we_dont_use();
}

int add(int a, int b)
{
    return a + b;
}
```

Preprocessing

Be careful when using preprocessor macros

When defining a preprocessor macro like this:

```
#define SQR(x) x*x
```

Remember that the preprocessor is "dumb", it does not understand or care about what the code is intended to do, it will only replace `SQR(x)` $\longrightarrow$ `x*x`.

Example:
`SQR(a+b)` $\longrightarrow$ `a+b*a+b`

To avoid such problems, use parentheses when defining the macro:

```
#define SQR(x) ((x)*(x))
```

Compilation

The compiler takes C code as input and generates assembly code as output.

You can ask gcc to generate assembly code with `gcc -S *.c`, and it will output a .s file for every .c file.

Can be useful if we suspect that the compiler has a bug in it.

Compilation

Useful basic compiler options, warnings, etc

- `-Imyincludedir` Add myincludedir to search path for include files.
- `-Wall` Turn on all warnings. Make this one a habit!
- `-g` Produce debugging information that a debugger (like `gdb`) can use.
- `-O` Optimisation settings.
  - `-O0` Turn off all optimisation (default). Use this when debugging!
  - `-O` or `-O1` Turn on optimisations to reduce code size and execution time that don't take much compilation time.
  - `-O2` Optimize even more. Perform nearly all optimisations that don't involve a space-speed tradeoff.
  - `-O3` Turn on maximum optimisation. You'll experiment with these optimisation levels in the next lab.
  - `-Os` Optimise for size.

Assembly

The **assembler** takes assembly code as input and generates object files as output.

Compile all the .c files with the command `gcc -c *.c`. Each source file will result in a .o file.

The `nm` program can parse an object file, library, or executable and report on its contents.
Example: `nm myprog.o`

```
0000000000000032 T add
                 U function_we_dont_use
0000000000000000 T main
```

Linking

Static vs dynamic libraries

The **linker** combines object files and external libraries and generates an executable or library as output.

Example:

```
myprog.c:(.text+0x27): undefined reference to'
'function_we_dont_use'
```

Two kinds of libraries:

- Static libraries (`libstaticfunction.a`)
- Dynamic, also called *shared* libraries (`libdynfunction.so`)

The *static library* is just an archive of .o files.
– use it by linking it into your executable in the same way as your own .o files.
– the linker only loads the functions of the library that your code references.

Linking

Static vs dynamic libraries

The **dynamic library** is prepared differently.
– the linker just puts references in the executable
– when the running executable encounters function that is linked in, the function loads dynamically
– the dynamic linker tends to load a large portion of the library, even if only a few functions are used

For performance reasons it is usually better to use static libraries. See See section 14.11 in Fog book.

Where to specify compiler optimization flags?

```
gcc -c help_fns.c
gcc -c main.c
gcc -O3 main.o help_fns.o -o prog
```

What is wrong here? Is the `-O3` flag really used?

Note: compiler optimization flags like `-O3` need to be used for the **compilation** step, not for the **linking** step!

Where to specify compiler optimization flags?

Here is how we should do it, to properly use the -O3 flag:

```
gcc -O3 -c help_fns.c
gcc -O3 -c main.c
gcc main.o help_fns.o -o prog
```

Note: compiler optimization flags like -O3 need to be used for the **compilation** step, not for the **linking** step!
(For the linking step such flags have no effect, since the compilation is already done.)

Will be usefull when we will write makefiles!

Everything at once

We can compile a simple program with a single command.

```
gcc -o myprog myprog.c
```

The "-o myprog" flag tells the compiler to call the output file "myprog". The compiler then recognises the .c file ending, denoting a C source file, which it puts through the entire build process.

You can run the program by typing `./myprog`.

Question for you

Assume that the C programmer got a compiler error that some include files are missing. In which stage of the C building process broke the code?

Question for you

Assume that the C programmer got a compiler error that some include files are missing. In which stage of the C building process broke the code?

Answer: Preprocessing

Question for you

What is the output file generated by the linker?

Question for you

What is the output file generated by the linker?

Answer: linker generates the executable file.

Compiler warnings

Some compiler warnings are already turned on by default.
Compiler warnings are good because they help us find bugs already at compile-time!

To turn on "all" warnings, use the `-Wall` compiler option.

The `-Wpedantic` option gives warnings for even more things than `-Wall`.
Note that having all compiler warnings turned on is good because the warnings help us when **developing** our code.

However, if you are releasing some kind of software package that you have developed, and other people are supposed to compile it, then it is best to remove the `-Wall` flag in the makefile for your released code. Compiler warnings are of interest for developers, but probably not for the end users of your code.

What happens now?

* **Assignment 1** is available! (deadline 1/2)

* **Lecture 2** on Thursday

* **Lab 1** on Friday (you can see it on Thursday in Student portal)
Look at the document CONTENT $\longrightarrow$ "Linux command line basics" in the Student Portal (see next slide).

NOTE: come to 1515 first

* **Assignment 2** available on Monday 28/1. (deadline 8/2)

Preparation for the first lab

If you are not registered in Student Portal and cannot see files mentioned in the lecture, please, write me an email!

You need to know the basics of the Linux/UNIX command line!

Look at the document CONTENT $\longrightarrow$ "Linux command line basics" in the Student Portal.

It contains short Linux command line tutorial.

In particular, the **tar** command will be useful later when you are going to submit code and/or reports for the assignments in this course. Then you will be asked to package your submission into a single .tar.gz and submit that in the Student Portal.

If you get stuck – ask for help!

If you get stuck with something, ask your teachers for help.

Communication is very important; do not suffer in silence, tell us what it is you find difficult!

That's all

Please, answer questions in the Socrative app!