Algorithm Engineering

Course Introduction and Getting Started With OpenMP

Mark Blacher

Friedrich Schiller University Jena

Winter Semester of 2022/23

Organization

- **One lecture per week** (asynchronously)
- **Slides**, **code** and **videos** are available here:
  https://cloud.uni-jena.de/s/HxGiTB576jqaW59
- **Wednesdays** from **10:15 to 11:45** in **LinuxPool 2** (Room 3410; Ernst-Abbe-Platz 2; from October 26, 2022 on) you can **work on** the **assignments**, **projects**, **ask** me **questions** (**optional**)
- If you have **questions**, you can also **write** me an **e-mail**
  mark.blacher@uni-jena.de
- If something comes up that is relevant to everyone, I will write to everyone

Grading

No **oral exam**. **No** **written exam**.

The **following things determine** your **final grade**:

1. **Written answers** to the **Exam Assignments** and your **solutions** to **some small coding exercises**
2. **C++ Project** (If you know someone you want to do the project together, please do so, otherwise do it alone)
3. **Four page paper** about your **project** (I provide the double column LaTeX template)

Grade = *part 1* (40%) + *part 2* (40%) + *part 3* (20%)

You have time to fulfill everything until March 1, 2023.

Don't worry about the details of the individual parts for the time being, you will receive detailed information about them in the course of the lectures.

Create Your Own Repository

- You use <u>**one**</u> **repository for the entire course**
- Please **make three folders** in your repository (`answers`, `project`, `paper`)
- The **repository** is your **flagship**, make it **clean** and **clear**
- **By March 1, 2023**, email me the **content of your repository** (if it is not too big)
- You can also **add me to** the **repository** with the following email: `mark.blacher@uni-jena.de` (**give me** the **appropriate access rights**)
- Please **choose one** of the following repository providers:
  - https://git.uni-jena.de/
  - https://gitlab.com/
  - https://github.com/

Course Mantra

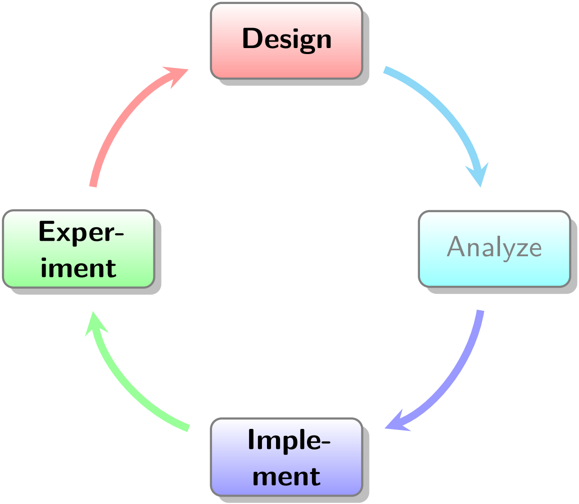
I hear and I forget.
I see and I remember.
I do and I understand.

Confucius

Motivation

Algorithm theory lacks the connection to real hardware. In addition to algorithmic complexity with its hidden constant factors, **hardware-specific implementation details determine the execution time of algorithms**. To close this gap between theory and practice, the field of algorithm engineering emerged. The challenge to make software faster is, in the sense of Algorithm Engineering, to **develop and implement algorithms that use the hardware efficiently**. Advances in computer architecture such as **multi-core processors, CPU caches, and SIMD instructions** opened up new opportunities for improving the performance of algorithms. In this course we learn how to exploit the hardware efficiently with C++ to speed up algorithms.

Algorithm Engineering Cycle (This Course)



Course Overview

- Parallelization with **OpenMP**
- Compiling programs with the help of **CMake**
- Exploiting vector registers (**vectorization**)
- Writing **cache-friendly code**
- **Debugging** and **profiling**
- Using **Cython** to integrate C++ code in Python
- Writing **efficient code for SSDs**

Recommended Literature

- http://chryswoods.com/beginning_c++/
- http://chryswoods.com/beginning_git/
- https://www.youtube.com/playlist?list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG
- https://www.agner.org/optimize/optimizing_cpp.pdf
- https://github.com/Kobzol/hardware-effects
- Computer Systems: A Programmer's Perspective (3rd Edition)
- Parallel Programming Concepts and Practice
- C++ High Performance: Master the art of optimizing the functioning of your C++ code (2nd Edition)
- http://chryswoods.com/beginning_python/
- http://chryswoods.com/intermediate_python/
- Cython: A Guide for Python Programmers
- Writing for Computer Science (3rd Edition)

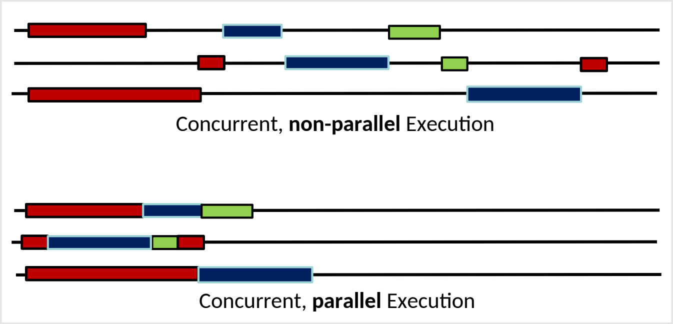
Why Parallel Computing?

The Free Lunch Is Over.

- The **number of transistors** on a microchip **doubles about every two years**. (Moore's Law)
- $\mathit{power} = \mathit{performance}^{1.74}$
- To keep up with the "doubling", we need multiple cores per CPU chip

Concurrency vs. **Parallelism**

A system is **concurrent** if it can support two or more actions *in progress* at the same time. A system is **parallel** if it can support two or more actions executing simultaneously.



Parallelism is a subset of concurrency

Parallel Programming with **OpenMP**

OpenMP is . . .

- a set of **compiler directives**, **library routines** and **environment variables**
- easy to learn
- powerful
- **included in most C/C++ compilers** (and Fortran)

Hello World with OpenMP

hello.cpp

```cpp
#include <iostream>
#include <omp.h> // required for library routines

using namespace std;

int main() {
#pragma omp parallel // compiler directive
  {
    // omp_get_thread_num() is an OpenMP library routine
    auto thread_id = omp_get_thread_num();
    cout << "Hello from thread " << thread_id << endl;
  }
}
```

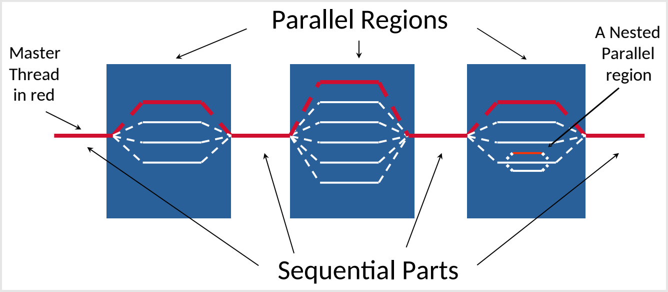
Terminal

```
$ g++ -fopenmp hello.cpp -o hello
$ OMP_NUM_THREADS=4 ./hello # environment variable
```

OpenMP Programming Model

Fork-Join Parallelism

Master thread spawns a **team of threads**. At a **subsequent point threads are joined** and the program resumes executing sequentially.



What the OpenMP Compiler Does

OpenMP code

```cpp
#include <iostream>

int main() {
#pragma omp parallel num_threads(4)
  { std::cout << "Hello World!" << std::endl; }
}
```

OpenMP compiler generates logically similar code to this

```cpp
#include <iostream>
#include <thread>

void thunk() { std::cout << "Hello World!" << std::endl; }

int main() {
  std::thread threads[4];
  for (int i = 1; i < 4; ++i)
    threads[i] = std::thread(thunk); // fork threads
  thunk(); // master thread
  for (int i = 1; i < 4; ++i)
    threads[i].join(); // join threads
}
```

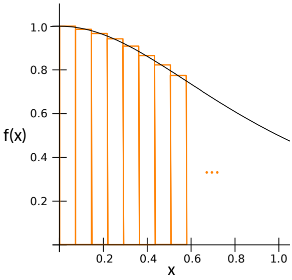
Estimating π

Mathematically, we know that:

$$\int_0^1 \frac{1}{1+x^2}dx = \frac{\pi}{4}$$

(see `http://mb-soft.com/public3/pi.html`)

We can **approximate the integral** as a **sum of rectangles**.



Estimating π with Numerical Integration

pi_numerical_integration.cpp

```cpp
#include <iomanip>
#include <iostream>
#include <omp.h>

using namespace std;

int main() {
  int num_steps = 100000000; // amount of rectangles
  double width = 1.0 / double(num_steps); // width of a rectangle
  double sum = 0.0; // for summing up all heights of rectangles

  double start_time = omp_get_wtime(); // wall clock time in seconds
  for (int i = 0; i < num_steps; i++) {
    double x = (i + 0.5) * width; // midpoint
    sum = sum + (1.0 / (1.0 + x * x)); // add new height of a rectangle
  }
  double pi = sum * 4 * width; // compute pi
  double run_time = omp_get_wtime() - start_time;

  cout << "pi with " << num_steps << " steps is " << setprecision(17)
       << pi << " in " << setprecision(6) << run_time << " seconds\n";
}
```

LinuxPool 1

You can **compile your programs with** g++

Terminal

```
$ g++ -fopenmp hello.cpp -o hello
$ ./hello
```

You can **log on** to a PC **remotely** in the university network

Terminal

```
$ ssh -X xy34abc@ppc819.mirz.uni-jena.de # replace
  xy34abc with your ID, ppc819.mirz is the machine
$ caja # start file manager
$ geany # text editor to write code
$ cat /proc/cpuinfo # show CPU information
$ exit # log out
```

For **remote file transfer** use git

Compiler, IDE Recommendations

You **need** a **compiler** that **supports OpenMP**

- **Linux:** GNU C++ Compiler, `Install on Ubuntu`
- **Windows:** MinGW 17.1 Distro from `https://nuwen.net/mingw.html`
- **Mac:** GNU compiler collection, `Install Homebrew` , `Install gcc`

The best you can get for free as a student is **Intel oneAPI Base & HPC Toolkit**, `Click here for installation and more information`

IDE recommendation: `Clion`, You can apply `here` for a free license

Coding Warmup

1. **Compile and run** the programs of **slides 13 and 15**
2. **Create a parallel version** of the π program (**slide 17**)

In addition to `#pragma omp parallel`, you will need the runtime library routines:

- `omp_get_num_threads() // number of threads`
- `omp_get_thread_num() // thread id`
- `omp_get_wtime() // wall-clock time in seconds`

3. **Watch** the **times** your implementation needs with **different numbers of threads**
4. Try compiling with **optimization flag `-Ofast`**

Possible Solution for the Warmup π Program

pi_openmp_v1.cpp

```cpp
#include <iomanip>
#include <iostream>
#include <omp.h>

using namespace std;

int main() {
  int num_steps = 100000000; // amount of rectangles
  double width = 1.0 / double(num_steps); // width of a rectangle
  double sum = 0.0; // for summing up all heights of rectangles
  double start_time = omp_get_wtime(); // wall clock time in seconds
  omp_set_num_threads(4); // setting the desired number of threads in the parallel region
#pragma omp parallel // parallel region starts here
  { // idea for parallelisation: split up loop iterations between threads
    int num_threads = omp_get_num_threads(); // used as increment in for loop
    int thread_id = omp_get_thread_num();
    double sum_local = 0.0; // for summing up heights locally
    for (int i = thread_id; i < num_steps; i += num_threads) {
      double x = (i + 0.5) * width; // midpoint
      sum_local = sum_local + (1.0 / (1.0 + x * x)); // add new height
    }
#pragma omp atomic // only one thread performs the update of the sum
    sum += sum_local;
  } // parallel region ends here
  double pi = sum * 4 * width; // compute pi
  double run_time = omp_get_wtime() - start_time;
  cout << "pi with " << num_steps << " steps is " << setprecision(17) << pi
       << " in " << setprecision(6) << run_time << " seconds\n";
}
```

Estimating π using Monte Carlo

pi_monte_carlo.cpp

```
#include <iostream>
#include <omp.h>
#include <random>

using namespace std;

int main() {
  int seed = 0;
  default_random_engine re{seed};
  uniform_real_distribution<double> zero_to_one{0.0, 1.0};

  int n = 100000000; // amount of points to generate
  int counter = 0; // counter for points in the first quarter of a unit circle
  auto start_time = omp_get_wtime(); // omp_get_wtime() is an OpenMP library routine

  // compute n points and test if they lie within the first quadrant of a unit circle
  for (int i = 0; i < n; ++i) {
    auto x = zero_to_one(re); // generate random number between 0.0 and 1.0
    auto y = zero_to_one(re); // generate random number between 0.0 and 1.0
    if (x * x + y * y <= 1.0) { // if the point lies in the first quadrant of a unit circle
      ++counter;
    }
  }

  auto run_time = omp_get_wtime() - start_time;
  auto pi = 4 * (double(counter) / n);

  cout << "pi: " << pi << endl;
  cout << "run_time: " << run_time << " s" << endl;
  cout << "n: " << n << endl; }
```

Exam Assignments

- Describe how **parallelism** differs from **concurrency**.
- What is **fork-join parallelism**?
- **Read Chapter 1** from Computer Systems: A Programmer's Perspective.

  http://csapp.cs.cmu.edu/2e/ch1-preview.pdf

  - Discuss one thing you find particularly interesting. (google it to find more information)
- Read the paper **There's plenty of room at the Top: What will drive computer performance after Moore's law?**

  Click here to download the paper

  - Explain in detail the figure ***Performance gains after Moore's law ends.*** (on the first page)
- Do the **coding warmup** on **slide 20**. **Parallelize the program** of **slide 22**.