C++ Programming

Lecture 11
Software Engineering Group

Philipp D. Schubert
Final lecture and introduction to the final project

- Static Analysis – Friday 17.07.2019
- Introduction to the Final Project – Friday 24.07.2019
Contents

1. High performance computing
2. High performance computing in C++
3. Example: matrix multiplication
   i. Important compiler flags
   ii. How to help the compiler
   iii. What the compiler can do for you

1. High performance computing
2. High performance computing in C++
3. Example: matrix multiplication
   i. Important compiler flags
   ii. How to help the compiler
   iii. What the compiler can do for you
Why should I care?

- Great demand for computation power
- Simulations
  - Weather forecast
  - Driving simulation
  - Design of drugs
  - Computer graphics
  - Power plants
- More computation power $\rightarrow$ more precise simulations
- Computer can achieve results much cheaper
- Parallel computation
- Implementation of a highly efficient program is time consuming and nerve wracking
Hard physical limits

- Clock rate is limited
- Power consumption
- Heat
- Signals can only travel at the speed of light
  - 3 GHz processor $\rightarrow$ time for one cycle 0,33 ns
  - Maximal distance a signal can travel in 0,33 ns:
    - Upper limit is speed of light in the vacuum: $0,3 \cdot 10^9 \frac{m}{s}$
    - $0,33 \cdot 10^{-9} \cdot 0,3 \cdot 10^9 \frac{m}{s} \approx 10 \text{ cm}$
    - Current chips: 200 – 400 $mm^2$
  - A signal must travel between two arbitrary position within one cycle
Processor development

- Moore's law
  - Complexity of integrated circuits doubles every ~year
    - Number of transistors
  - Clock rate is limited, use additional transistors for
    - Processor cores
    - Redundant ALU/ FPU
    - Registers
    - Cache memory
  - Build specialized hardware e.g. GPU’s
    - Implement programs in CUDA or OpenCL

Microprocessor Transistor Counts 1971-2011 & Moore's Law
Four levels of parallelism

1. Parallelism on bit-level
   - Early computers used a small number of bits for registers (word size)
   - Today: 32 bit, even more common: 64 bit architectures

2. Parallelism through pipelining
   - Divide instruction into sub-tasks (e.g. fetch, decode, execute, write back)

3. Parallelism through multiple function units (multiple ALUs, FPUs, load/store units, jump units)

4. Parallelism on process- or thread-level
   - Use a programming language that supports parallel execution
   - Help the compiler to produce faster code by specifying multiple execution threads
   - Usually our last resort, since pipelining has already reached its limits

In general:
   - Modern CPUs and GPUs become more and more complex
   - Only very few companies can manufacture them at all!
Flynn’s taxonomy

- Concepts and corresponding machines
  1. SISD
     - Classical von Neumann machine
  2. MISD
     - Makes no sense
  3. SIMD
     - Modern CPUs and Graphics processing units (GPUs)
  4. MIMD
     - Processors that can work asynchronously
Parallel processing

- Problems have to be decomposed in smaller independent computations
  - These can be computed on different processor cores
- But: data- and control-flow is not completely decoupled
- Correct execution of program has to be ensured by synchronization and information exchange
- Shared- and distributed memory space
  - **Shared memory parallelization**
    - Variables can be accessed and used for information exchange
    - Use different execution threads
  - Distributed memory parallelization
    - Information exchange through explicit message passing
      - Message-passing programming
    - Use different processes
Parallel processing

- Evaluation of parallel programs expressible in terms of
  - Speed-up
  - Efficiency (time, memory, …)
- Granularity is the average size of a subtask
  - Higher granularity is better
- Decision in which order a computation takes place is called scheduling
Problems with von Neumann’s concept

- What is wrong with our modern machines?
  1. Sequential execution of a program
  2. Every implementation of a Turing- or Random-access-machine has to deal with finite memory
  3. Memory is bottleneck: every processor cycle is much faster than a memory cycle
  4. Universal computation leads to inefficient execution of application programs
  5. Finite representation of values
  6. Reliability
  7. Input / output operation is done through processor, processor is blocked
  8. Computer security was never considered (only specialists could handle a machine anyway)
Does it pay off? Amdahl’s law

- Runtime of parallel program
  \[ T_p(n) \geq f \cdot T'(n) + \frac{(1-f) \cdot T'(n)}{p} \]
- Sequential part \( f \) + parallel part
- Maximal speed-up is then
  \[ S_p(n) = \frac{T_r(n)}{T_p(n)} = \frac{T_r(n)}{f \cdot T'(n) + \frac{(1-f) \cdot T'(n)}{p}} = \frac{1}{\frac{f}{p} + (1-f)} \]
- If \( f > 0 \) and \( p \to \infty \)
  \[ S_p(n) \leq \frac{1}{f} \]
Gustafson’s law

- Amdahl's law revisited

$$S_p(n) = \frac{1}{f + \frac{(1-f)}{p}} = \frac{1}{f_1 + \frac{1-p}{p} \cdot \frac{1}{1-f_1} + f_1}$$

$$= p \cdot (1 - f_1) + f_1$$

- Sequential part of program can be reduced
  - Through larger problem size
  - Through larger number of processors

- When to use what law?
  - Problem does not scale
    - Amdahl
  - Problem is perfectly scalable
    - Gustafson
How to?

- How to implement an algorithm efficiently?
  - Understand the algorithm in detail
  - Inspect algorithm
  - Understand your hardware
  - Check state-of-the-art techniques
  - Plan first, then implement!

- Still too slow?
  - Approximate solution
    - E.g. genetic algorithms
      - Guess a solution
      - Try to optimize according to some fitness function
  - Maybe a good solution is better than no solution at all
std::thread

- A data type that creates a separate execution thread (using shared memory)

```cpp
template< typename Function, typename... Args >
explicit thread( Function&& f, Args&&... args );
```

- A variable of type thread has to be constructed explicitly (no implicit conversion allowed)

- std::thread's constructor receives a 'callable' and some optional arguments

- Callable might be a ...
  - Function
    - Function pointer
    - Function object
    - Lambda function
  - Class implementing `operator()` (function call operator)

- A thread itself does not care about the return value
  - Cannot return data directly
std::thread

```cpp
#include <iostream>
#include <thread>
using namespace std;

class callable {
private:
    int i;
    int j;
public:
    callable(int a, int b) : i(a), j(b) {}
    void operator() () {
        cout << "t1: " << i + j << '\n';
    }
};

void func(int a, int b) {
    cout << "t2: " << a * b << '\n';
}

int main() {
    unsigned int n =
        thread::hardware_concurrency();
    cout << n << "hardware threads possible\n";
    thread t1(callable(10, 20));
    thread t2(func, 10, 20);
    thread t3([](){
        cout << "t3: " << 20
            / 10 << '\n';
    });
    t1.join();
    t2.join();
    t3.join();
    return 0;
}
```
```cpp
#include <array>
#include <iostream>
#include <thread>
#include <vector>

using namespace std;

array<int, 4> results;

void power(int id, int a) {
    results[id] = a*a;
}

int main() {
    vector<thread> threads;
    for (int i = 0; i < results.size(); ++i) {
        threads.push_back(thread(power, i, i+1));
    }
    for (auto& t : threads) { t.join(); }
    for (auto result : results) {
        cout << result << 'n';
    }
    return 0;
}
```

- Threads cannot return data directly
  - Use shared memory (global variables) for ...
    - storing results
    - communication between threads
  - Try to minimize usage of global variables!
std::thread

- What happens if two or more threads use a global variable at the same time?
- “Race condition”
  - You never ever want a race condition!
  - Hard to find and to fix
  - Even worse: not being aware of a race condition
- Lock critical code (e.g. with a mutex lock)
- Only one thread is allowed to execute locked code

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

using namespace std;

mutex results_mutex;
vector<int> results;

void power(int a) {
    int b = a * a;
    lock_guard<mutex> guard(results_mutex);
    results.push_back(b);
}

int main() {
    vector<thread> threads;
    for (int i = 1; i < 10; ++i) {
        threads.push_back(thread(power, i));
    }

    for (auto& t : threads) { t.join(); }

    for (auto i : results) {
        cout << i << "\n";
    }

    return 0;
}
```

What happens if two or more threads use a global variable at the same time?

“Race condition”
- You never ever want a race condition!
- Hard to find and to fix
- Even worse: not being aware of a race condition

Lock critical code (e.g. with a mutex lock)
- Only one thread is allowed to execute locked code
**std::atomic**

- If critical global data is small or a primitive
  - `std::atomic` can be used instead of a mutex
- Makes accessing a value (read and write) atomic
- “Lock-free programming”

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

using namespace std;

atomic<int> global_int(0);

void inc_global() { ++global_int; }

int main() {
    vector<thread> threads;
    for (int i = 0; i < 10; ++i) {
        threads.push_back(thread(inc_global));
    }

    for (auto& t : threads) { t.join(); }

    cout << global_int << '\n';
    return 0;
}
```
std::packaged_task

- Threads cannot return data directly
- Use a `packaged_task`
  - It uses a future to return a value
  - `future` is a very useful type
- How?
  - Create a `packaged_task`
  - Get future from it
  - Execute task
  - Obtain result from future
- This avoids using global variables!

```cpp
#include <iostream>
#include <thread>
#include <future>
#include <vector>
using namespace std;

int func(int a, int b) { return a * b; }

int main() {
    packaged_task<int(int,int)> task(func);
    future<int> result = task.get_future();

    thread t1(move(task), 2, 10);
    t1.join();

    cout << "task_thread: " << result.get() << 'n';
    return 0;
}
```

Threads cannot return data directly

Use a `packaged_task`

- It uses a future to return a value
- `future` is a very useful type

How?

- Create a `packaged_task`
- Get future from it
- Execute task
- Obtain result from future

This avoids using global variables!
std::async

- `async` is an elegant function
  - Starts an asynchronous task
  - Returns a future
- Use `async` if the problem is appropriate
- Generates new software or hardware threads
- Two policies are allowed
  - `launch::async` or `launch::deferred`

```cpp
#include <iostream>
#include <thread>
#include <future>
using namespace std;

int ret10() { return 10; }

int main() {
    future<int> f =
        async(launch::async, ret10);
    cout << f.get() << '\n';
    return 0;
}
```
std::async

- How to compute a bunch of tasks in parallel?
  - Use a vector of
    - futures
    - and loops!
- Caution
  - get() can only be called once on a given future

```cpp
#include <iostream>
#include <thread>
#include <future>
#include <vector>
using namespace std;

int retArg(int i) { return i; }

int main() {
  vector<future<int>> futures;
  for (int i = 0; i < 10; ++i) {
    futures.push_back(async(launch::async,
                             retArg, i));
  }
  vector<int> results;
  for (auto & future : futures) {
    results.push_back(future.get());
  }
  for (int i : results) { cout << i << '\n'; }
  return 0;
}
```
std::future and std::promise

- Compute even if values are missing?
- How does it work?
  - Create a promise
  - Promise will be fulfilled in the future
  - Prepare computation
  - Computation will start as soon as promise is fulfilled and value is ready to use
- As in real life
  - Always fulfill your promises
  - Otherwise
    A. broken_promise exception
    B. waiting for eternity

```cpp
#include <iostream>
#include <thread>
#include <future>
#include <vector>
#include <chrono>
using namespace std;

int power(future<int> f) {
    int a = f.get();
    return a * a;
}

int main() {
    promise<int> p;
    future<int> f = p.get_future();
    future<int> res = async(launch::async,
                             power, move(f));
    this_thread::sleep_for(chrono::seconds(10));
    p.set_value(10);
    int result = res.get();
    cout << result << '\n';
    return 0;
}
```
Make the most of your CPU cycles – matrix multiplication

- Testing different versions of a matrix multiplication
  - 3 000 x 3 000 * 3 000 x 3 000 matrix → 9 000 000 entries per matrix

- All tests run on my notebook
  - Intel® Core™ i7-5600U CPU @ 2.6 GHz
  - 2 hardware cores (+ hyper threading)
  - Using the g++ and clang++ compiler
    - Thread model: POSIX
    - g++ (Ubuntu 8.4.0-1ubuntu1~16.04.1) 8.4.0
    - clang++ version 10.0.0 ([https://github.com/llvm/llvm-project.git](https://github.com/llvm/llvm-project.git) x86_64-unonwn-linux-gnu)
  - Every test was run only once → poor measurement, but will still prove my point
A naive matrix multiplication (no additional compiler flags)

- Runtime: g++ 13m 18.250s
- Runtime: clang++ 13m 48.348s

```cpp
#include <iostream>
#include <vector>

struct mat {
    size_t rows;
    size_t cols;
    std::vector<std::vector<double>> data;

    mat(size_t rows, size_t cols, double ival = 0.0) :
        rows(rows), cols(cols), data(rows, std::vector<double>(cols, ival)) {}

    friend std::ostream &operator<<(std::ostream &os, const mat &m) {
        for (const auto &row : m.data) {
            for (const auto &entry : row) {
                os << entry << ' ';
            }
            os << '\n';
        }
        return os;
    }

friend mat operator* (const mat& lhs, const mat& rhs) {
    mat result(lhs.rows, rhs.cols, 0);
    for (size_t row = 0; row < lhs.rows; ++row) {
        for (size_t col = 0; col < rhs.cols; ++col) {
            for (size_t k = 0; k < lhs.cols; ++k) {
                result.data[row][col] += lhs.data[row][k] * rhs.data[k][col];
            }
        }
    }
    return result;
}

int main(int argc, char **argv) {
    size_t dim1 = atoi(argv[1]);
    size_t dim2 = atoi(argv[1]);
    mat a(dim1, dim2, 2);
    mat b(dim1, dim2, 3);
    mat result = a * b;
    std::cout << result.data[0][0] << '\n';
    return 0;
}
```
Turn on compiler optimizations -Ofast and -march=native

- Runtime: g++ 2m 30.203s (~ -10m 48s)
- Runtime: clang++ 2m 29.306 (~ -11m 19s)
- Same code as on the last slide
- -Ofast
  - Compiler performs every optimization (including the dark arts) it knows
- -march=native
  - Produce code that is optimized for the target processor
    - Compiled program is only usable on platforms with same processor
Use data locality (and -Ofast -march=native)

- Runtime: g++ 2m 19.923s (~ -10s)
- Runtime: clang++ 2m 17.214 (~ -12s)

```
#include <iostream>
#include <vector>

struct mat {
    size_t rows;
    size_t cols;
    std::vector<double> data;

    mat(size_t rows, size_t cols, double ival = 0.0) :
        rows(rows), cols(cols), data(rows * cols, ival) {}

    double &operator()(size_t row, size_t col) { return data[row * cols + col]; }
    const double &operator()(size_t row, size_t col) const { return data[row * cols + col]; }

    friend std::ostream &operator<<(std::ostream &os, const mat &m) {
        for (size_t row = 0; row < m.rows; ++row) {
            for (size_t col = 0; col < m.cols; ++col) {
                os << m(row, col) << ' '; 
            }
            os << '\n';
        }
        return os;
    }

friend mat operator*(const mat &lhs, const mat &rhs) {
    mat result(lhs.rows, rhs.cols, 0);
    for (size_t row = 0; row < lhs.rows; ++row) {
        for (size_t col = 0; col < rhs.cols; ++col) {
            for (size_t k = 0; k < lhs.cols; ++k) {
                result(row, col) += lhs(row, k) * rhs(k, col);
            }
        }
    }
    return result;
}

int main(int argc, char **argv) {
    size_t dim1 = atoi(argv[1]);
    size_t dim2 = atoi(argv[1]);
    mat a(dim1, dim2, 2);
    mat b(dim1, dim2, 3);
    mat result = a * b;
    std::cout << result(0, 0) << '\n';
    return 0;
}
```
Even more data locality (--Ofast --march=native)

- Runtime: g++ 19.920s (~ -2m)
- Runtime: clang++ 18.899s (~ -1m 59s)

```
#include <iostream>
#include <vector>

struct mat {
    size_t rows;
    size_t cols;
    std::vector<double> data;

    mat(size_t rows, size_t cols, double ival = 0.0)
        : rows(rows), cols(cols), data(rows * cols, ival) {}

    double &operator()(size_t row, size_t col) { return data[row * cols + col]; }
    const double &operator()(size_t row, size_t col) const { return data[row * cols + col]; }

    friend std::ostream &operator<<(std::ostream &os, const mat &m) {
        for (size_t row = 0; row < m.rows; ++row) {
            for (size_t col = 0; col < m.cols; ++col) {
                os << m(row, col) << ' '; }
            os << '\n'; }
        return os;
    }

    friend mat operator*(const mat &lhs, const mat &rhs) {
        mat result(lhs.rows, rhs.cols, 0);
        for (size_t row = 0; row < lhs.rows; ++row) {
            for (size_t k = 0; k < lhs.cols; ++k) {
                for (size_t col = 0; col < rhs.cols; ++col) {
                    result(row, col) += lhs(row, k) * rhs(k, col);
                }
            }
        return result;
    }

    int main(int argc, char **argv) {
        size_t dim1 = atoi(argv[1]);
        size_t dim2 = atoi(argv[1]);
        mat a(dim1, dim2, 2);
        mat b(dim1, dim2, 3);
        mat result = a * b;
        std::cout << result(0, 0) << '\n';
        return 0;
    }
```
Run it in parallel (and data locality and `-Ofast` `-march=native`)

- Runtime: `g++` 9.753s (~ -10s)
- Runtime: `clang++` 11.309 (~ -8s)

```cpp
#include <iostream>
#include <vector>

struct mat {
    size_t rows;
    size_t cols;
    std::vector<double> data;

    mat(size_t rows, size_t cols, double ival = 0.0) :
        rows(rows), cols(cols), data(rows * cols, ival) {}

    double &operator()(size_t row, size_t col) { return data[row * cols + col]; }
    const double &operator()(size_t row, size_t col) const { return data[row * cols + col]; }

    friend std::ostream &operator<<(std::ostream &os, const mat &m) {
        for (size_t row = 0; row < m.rows; ++row) {
            for (size_t col = 0; col < m.cols; ++col) {
                os << m(row, col) << ' ';
            }
            os << '
';
        }
        return os;
    }

    friend mat operator*(const mat &lhs, const mat &rhs) { 
        mat result(lhs.rows, rhs.cols, 0);
        size_t row, col, k;
        #pragma omp parallel for private(row, col, k) \
        shared(lhs, rhs, result) schedule(static)
        for (row = 0; row < lhs.rows; ++row) {
            for (k = 0; k < lhs.cols; ++k) {
                for (col = 0; col < rhs.cols; ++col) {
                    result(row, col) += lhs(row, k) * rhs(k, col);
                }
            }
        }
        return result;
    }

    int main(int argc, char **argv) {
        size_t dim1 = atoi(argv[1]);
        size_t dim2 = atoi(argv[1]);
        mat a(dim1, dim2, 2);
        mat b(dim1, dim2, 3);
        mat result = a * b;
        std::cout << result(0, 0) << '
';
        return 0;
    }
};
```

- My machine has only two cores
- It makes sense that we can cut the runtime in half

My machine has only two cores
It makes sense that we can cut the runtime in half
Make the most of your CPU cycles – matrix multiplication

- Testing different versions of a matrix multiplication
  - $3000 \times 3000 \times 3000$ matrix $\rightarrow 9,000,000$ entries per matrix
  - Using a clever implementation and compiler optimizations we could save $g++ \sim 13m 8s$ / $clang++ \sim 13m 37s$
    - Initial times $g++ 13m 18s$ / $clang++ 13m 48s$
    - Final times $g++ \sim 10s$ / $clang++ \sim 11s$
Most important optimization flags

- Use a modern style: `-std=c++XX`, where `XX` is >= 11
- `-Ox`, where `X` is 0, 1, 2 or 3
  - Meaning: off, on, some more, insane
- `-Ofast`
  - All `-O3` optimizations and invalidation of the standard-compliance
    - + use of `-ffast-math`, `-fno-protect-parens`, `-fstack-arrays`
- `-fmarch=native`
  - Generate architecture specific code
- `-DNDEBUG`
  - Do not use debug mode code
- `-fdata-sections`, `-ffunction-sections`
  - Place each data item and function into its own segment (might allows for better linker optimization)
- `-flto=full`
  - Use full link-time (whole-program) optimizations (caution: needs vast amounts of RAM)
Compiler Explorer and what to keep in mind

- Do not do what libraries and compilers can do!
  - Write readable code and express intent to help your colleagues and the compiler
    - The compiler will be able to see through your code (most of the time)
  - Checkout “Compiler Explorer”: https://godbolt.org
  - Corresponding talk: “What has my compiler done for me lately?” by Matt Godbolt
    - https://www.youtube.com/watch?v=bSkpMdDe4g4

- If performance matters (always) …
  - Never make assumptions based on your gut feeling! ➔ systems are way too complex
  - Test your code (is it still correct?)
  - Measure (has it become faster?)
  - Test different optimization flags (see last slide)
  - Test different compilers and compiler versions!
    - Prefer a stable compiler version over some development version
Compiler Explorer: check what the compiler can do for you

- Task: find the minimum of four integer values!
Wow!
Such minimum!
Much smart!
Very clever!
1 // find the minimum of four integers using hand-crafted function
2 int min(int a, int b, int c, int d) {
3     int m = a;
4     if (b < m) {
5         m = b;
6     }
7     if (c < m) {
8         m = c;
9     }
10    if (d < m) {
11        return d;
12    }
13    return m;
14 }

#include <algorithm>

1 // finding the minimum using the STL
2 int min(int a, int b, int c, int d) {
3     return std::min(a, b, c, d);
4 }

1 min(int, int, int, int):
2     cmp edx, ecx
3     mov eax, edi
4     cmovg edx, ecx
5     cmp edx, esi
6     cmovle esi, edx
7     cmovle esi, edi
8     cmp esi, edi
9     cmovle eax, esi
10    ret
Recap

- Why high performance computing matters
- Hard physical limits
- When does it pay off?
- Levels of parallelism
- Parallel programming in C++
- Optimizing compilers
- Compiler Explorer
- Express intent and do not trust your gut feeling!
Thank you for your attention

Questions?