Project discussion / final lecture

- No lecture on Friday 19.07.2019
- Final lecture on Friday 26.07.2019
  - Room F1.110 HNI building
  - Discussion of the project
  - Hacks and miscellaneous
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
Why should I care?

- Demand for computation power
- Simulations
  - Weather forecast
  - Driving simulation
  - Design of drugs
  - Computer graphics
  - Power plants
- More computation power == more precise simulations
- Computer can achieve results much cheaper
- Parallel computation
- Implementation of a highly efficient program is time consuming and nerve wracking
  - In case of success, program can be executed much quicker
Hard physical limits

- Clock rate cannot grow arbitrary
- 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 light speed in 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 be able to travel between two arbitrary position in 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
    - More processor cores
    - Redundant ALU/FPU
    - Registers
    - More cache memory
  - Build specialized hardware e.g. GPU's
    - Use 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
   - 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
   - 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 asynchronous
Parallel processing

- Problems have to be decomposed in smaller independent computations
  - These can be computed on different processor cores
- BUT: Typically 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
    - 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
  - $S_p(n) = \frac{T'(n)}{T_p(n)} = \frac{T'(n)}{f \cdot T'(n) + \frac{(1-f) \cdot T'(n)}{p}} = \frac{1}{f + \frac{(1-f)}{p}}$
  - If $f > 0$ and $p \to \infty$
    - $S_p(n) \leq \frac{1}{f}$

Amdahl’s Law
Gustafson’s law

- Amdahl’s law revisited

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

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

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

(Slide taken from ‘Parallele Algorithmen und Datenverarbeitung’, Bielefeld University)
How to?

How to develop and implement an algorithm efficiently?
- Understand the algorithm in detail
- Inspect algorithm
- Understand your hardware
- Check current 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 type to create 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)

- It receives a ´callable´ and some optional arguments

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

- A thread itself does not care about the return type
  - Cannot return data directly
```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 << endl;
    }
};

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

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 << endl; });
  t1.join();
  t2.join();
  t3.join();
  return 0;
}
```
### std::thread

- Threads cannot return data directly
  - Use shared memory (global variables) for
    - Storing results
    - Communication with other threads
  - Try to minimize usage of global variables!

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

using namespace std;

int results[4];

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

int main() {
    vector<thread> threads;
    for (int i = 1; i < 5; ++i) {
        threads.push_back(thread(power, i-1, i));
    }
    for (auto& t : threads) t.join();
    for (unsigned i = 0; i < 4; ++i) {
        cout << results[i] << '
';
    }
    return 0;
}
```
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 mutex locks)
- 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 << '
';
    }
    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 mutex locks)
- 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 access (reading and writing) atomic

```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;
}
```
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() << '
';
    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 future
  - otherwise program crashes

```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 & std::promise

- Computing when values are still 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> fu = async(launch::async,
                            power, move(f));
    this_thread::sleep_for(chrono::seconds(10));
    p.set_value(10);
    int result = fu.get();
    cout << result << '\n';
    return 0;
}
```
Make the most of your CPU cycles – matrix multiplication

- Testing different variations of a matrix multiplication
  - 3 000 x 3 000 * 3 000 x 3 000 matrix → 9000 000 entries per matrix

- All tests run on my notebook
  - Intel® Core™ i7-5600U CPU @ 2.60GHz
  - 2 hardware cores (hyper threading)
  - Using the g++ compiler
    - Thread model: posix
    - gcc version 4.8.4 (Ubuntu 4.8.4-2ubuntu1~14.04.3)
  - Every test was run only once (poor measurement, but still proves the point)
A naive matrix multiplication (no additional compiler flags)

- Runtime: 14m 43.308s

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

struct mat {
    size_t rows;
    size_t cols;
    vector<vector<double>> entries;
    mat(size_t d1, size_t d2, double ival=0) :
        rows(d1),
        cols(d2),
        entries(rows, vector<double>(cols, ival)) {}
    friend ostream& operator<<(ostream& os, const mat& m) {
        for (auto row : m.entries) {
            for (auto entry : row)
                os << entry << " ";
            os << endl;
        }
        return os;
    }
};
friend mat operator* (const mat& lhs, const mat& rhs) {
    mat result(lhs.rows, rhs.cols, 0);
    for (size_t curr_row = 0; curr_row < lhs.rows; ++curr_row) {
        for (size_t curr_col = 0; curr_col < rhs.cols; ++curr_col) {
            result.entries[curr_row][curr_col] +=
                lhs.entries[curr_row][curr_entry] *
                rhs.entries[curr_entry][curr_col];
        }
    }
    return result;
}

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

- Runtime: 2m 34.889s (~ -12m 10s)
- Same code as on the last slide
  - `-Ofast`
    - Compiler performs every optimization it knows
  - `-march=native`
    - Produce code that is optimized for the target processor
      - Compiled program only usable on platforms with same processor
Use data locality (and -Ofast -march=native)

Runtime: 2m 4.468s (~30s)

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

struct mat {
    size_t rows;
    size_t cols;
    vector<double> entries;

    mat(size_t d1, size_t d2, double ival = 0) : rows(d1), cols(d2), entries(rows * cols, ival) {}

    friend ostream& operator<<(ostream& os, const mat& m) {
        for (size_t i = 0; i < m.rows; ++i) {
            for (size_t j = 0; j < m.cols; ++j)
                cout << m(i, j) << " ";
            cout << endl;
        }
        return os;
    }

    inline double& operator()(size_t r, size_t c) { return entries[r * cols + c]; }
    inline const double& operator()(size_t r, size_t c) const { return entries[r * cols + c]; }
    friend mat operator*(const mat& lhs, const mat& rhs) {
        mat result(lhs.rows, rhs.cols, 0);
        for (size_t curr_row = 0; curr_row < lhs.rows; ++curr_row) {
            for (size_t curr_col = 0; curr_col < rhs.cols; ++curr_col) {
                for (size_t curr_entry = 0; curr_entry < lhs.cols; ++curr_entry) {
                    result(curr_row, curr_col) += lhs(curr_row, curr_entry) * rhs(curr_entry, curr_col);
                }
            }
        }
        return result;
    }

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

- Runtime: 2m 4.468s (~30s)
Even more data locality (–Ofast –march=native)

- Runtime: 21.652s (~ 1m 44s)

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

struct mat {
    size_t rows;
    size_t cols;
    vector<double> entries;
    mat(size_t d1, size_t d2, double ival=0) : rows(d1), cols(d2), entries(rows*cols, ival) {}
    friend ostream& operator<< (ostream& os, const mat& m)
    {
        for (size_t i = 0; i < m.rows; ++i) {
            for (size_t j = 0; j < m.cols; ++j)
                cout << m(i, j) << " ";
            cout << endl;
        }
        return os;
    }
};

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

inline double& operator()(size_t r, size_t c) { return entries[r*cols+c]; }
inline const double& operator()(size_t r, size_t c) const { return entries[r*cols+c]; }
friend mat operator*(const mat& lhs, const mat& rhs)
{
    mat result(lhs.rows, rhs.cols, 0);
    for (size_t curr_row = 0; curr_row < lhs.rows; ++curr_row) {
        for (size_t curr_entry = 0; curr_entry < lhs.cols; ++curr_entry) {
            for (size_t curr_col = 0; curr_col < rhs.cols; ++curr_col) {
                result(curr_row,curr_col) += lhs(curr_row,curr_entry) *
                                          rhs(curr_entry,curr_col);
            }
        }
    }
    return result;
}
Run it in parallel (and data locality and `-Ofast -march=native`)

```
friend mat operator* (const mat& lhs, const mat& rhs) {
    size_t curr_row, curr_col, curr_entry;
    mat result(lhs.rows, rhs.cols, 0);
    #pragma omp parallel for private(curr_row, curr_col, curr_entry)
        shared(result, lhs, rhs) schedule(static)
    for (curr_row = 0; curr_row < lhs.rows; ++curr_row) {
        for (curr_entry = 0; curr_entry < lhs.cols; ++curr_entry) {
            for (curr_col = 0; curr_col < rhs.cols; ++curr_col) {
                result(curr_row, curr_col) += lhs(curr_row, curr_entry) * rhs(curr_entry, curr_col);
            }
        }
    }
    return result;
}
```
Make the most of your CPU cycles – matrix multiplication

- Testing different variations of a matrix multiplication
  - $3\,000 \times 3\,000 \times 3\,000$ matrix → 9,000,000 entries per matrix
  - Using a clever implementation and compiler optimizations we could save ~ **14m 32s** (initial time 14m 43s)
Most important optimization flags

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

- Do not do what libraries and compilers can do!
  - Checkout “Compiler Explorer”: [https://godbolt.org](https://godbolt.org)
  - Corresponding talk: [https://www.youtube.com/watch?v=bSkpMdDe4g4](https://www.youtube.com/watch?v=bSkpMdDe4g4)

- If performance matters (always) …
  - Never make assumptions based on your gut feeling! ➔ systems are 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! (Relevant for the final project)
Wow!
Such minimum!
Much smart!
Very clever!
```cpp
// find the minimum of four integers using hand-crafted function
int min(int a, int b, int c, int d) {
    int m = a;
    if (b < m)
        m = b;
    if (c < m)
        m = c;
    if (d < m)
        return d;
    return m;
}

#include <algorithm>

// finding the minimum using the STL
int min(int a, int b, int c, int d) {
    return std::min({a, b, c, d});
}
```
Recap

- Why high performance computing matters?
- Hard physical limits
- 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?