C++ PROGRAMMING

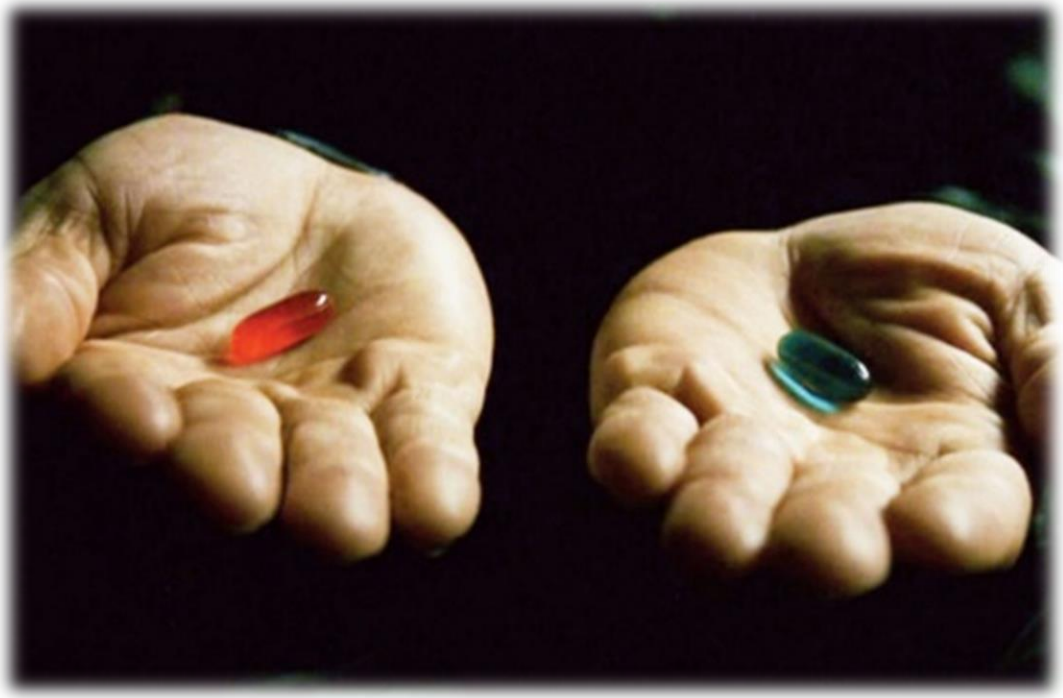
Lecture 11

Secure Software Engineering Group

Philipp Dominik Schubert

Final lecture and introduction to the final project

- Static Program Analysis – Friday 09.07.2021
- Introduction to the Final Project – Friday 16.07.2021



HEINZ NIXDORF INSTITUT UNIVERSITÄT PADERBORN
Fraunhofer IEM

CONTENTS

HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN

Fraunhofer
IEM

Why should I care?

- Great demand for computation power
- Simulations
  - Weather forecast
  - Driving simulation
  - Drug design
  - Computer graphics
  - Power plants
- More computation power ➔ more precise simulations
- Computer can achieve results much cheaper
- Parallel computation
- Implementation of highly efficient programs is time consuming and nerve wracking

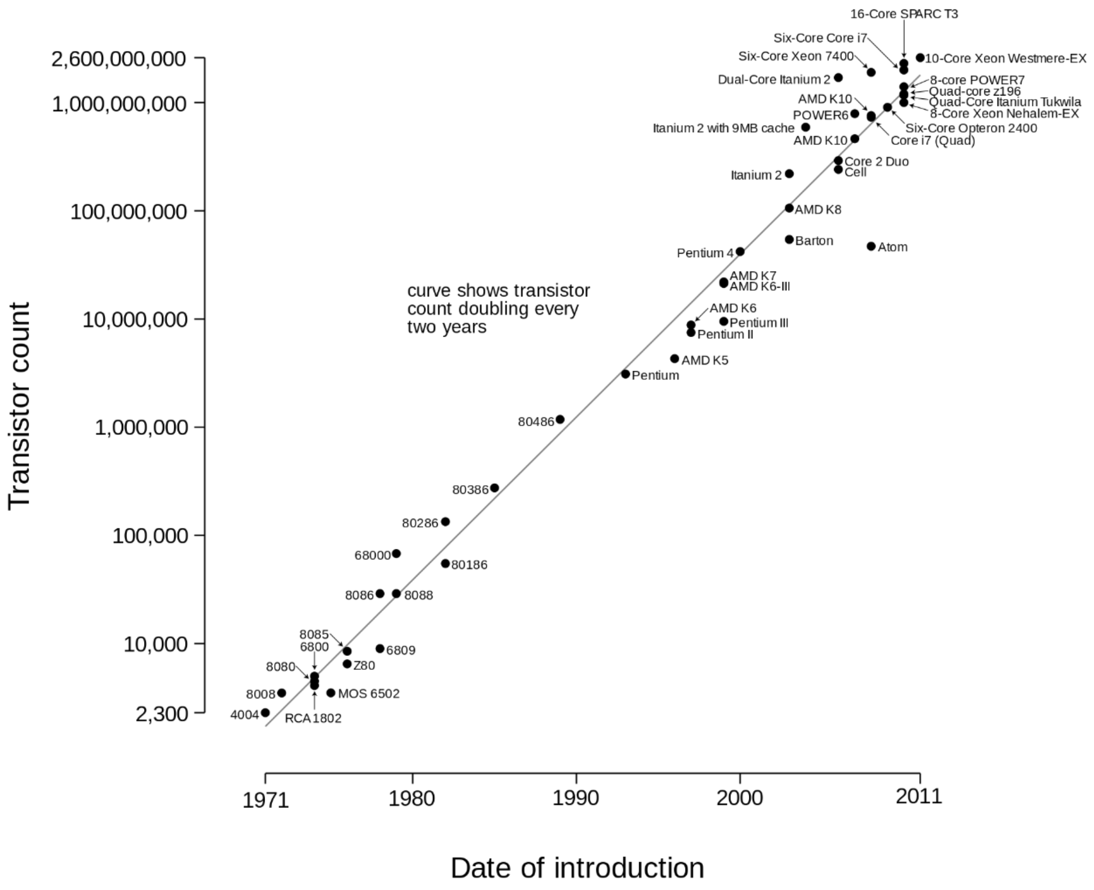
Hard physical limits

- Clock rate is limited
- Power consumption
- Heat
- Signals can only travel at the speed of light
  - 3 GHz processor ➔ 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: $0{,}3 \cdot 10^9 \frac{m}{s}$
    - $0{,}33 \cdot 10^{-9} s \cdot 0{,}3 \cdot 10^9 \frac{m}{s} \approx 10\ cm$
    - 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



HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN
Fraunhofer IEM

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

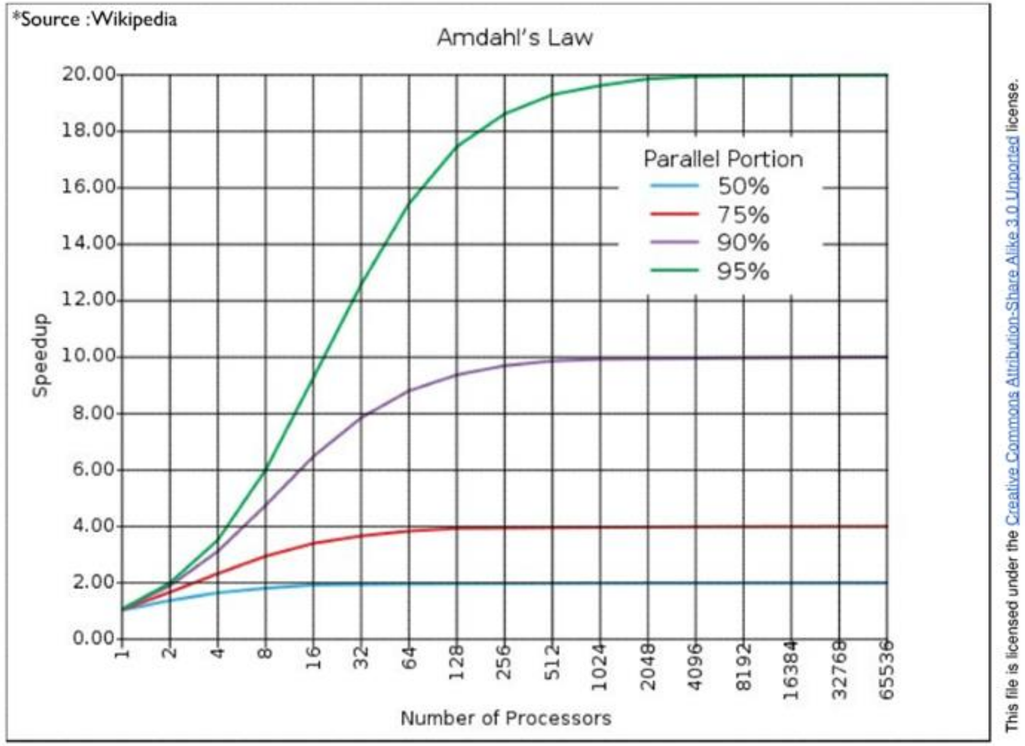
- Evaluation of parallel programs expressible in terms of
  - Speed-up
  - Efficiency (time, memory, …)
- Granularity is the average size of a subtask
  - Higher granularity is better
- Decision in which order a computation takes place is called scheduling

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
  - $S_p(n) = \frac{T'(n)}{T_p(n)} = \frac{T'(n)}{f \cdot T'(n) + \frac{(1-f) \cdot T'(n)}{p}} = \frac{1}{f + \frac{(1-f)}{p}}$
  - If $f > 0$ and $p \to \infty$
    - $S_p(n) \leq \frac{1}{f}$

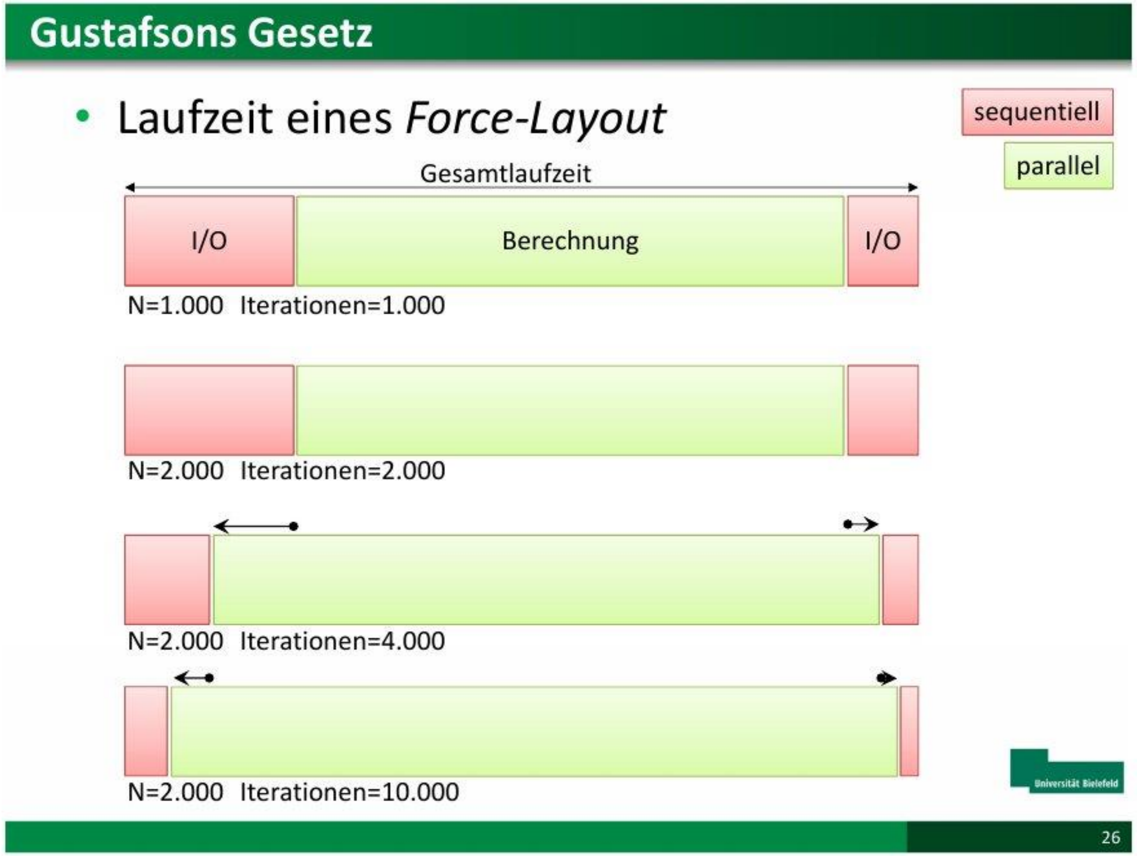


Amdahl’s Law

[Figure taken from http://arma.sourceforge.net/img/armadillo_logo.png]

Gustafson’s law

- Amdahl's law revisited
- $S_p(n) = \frac{1}{f+\frac{(1-f)}{p}} = \frac{1}{\frac{\frac{f_1}{p\cdot(1-f_1)+f_1}}{}+\frac{1-\frac{f_1}{p\cdot(1-f_1)+f_1}}{p}}$
  $= p \cdot (1 - f_1) + f_1$
- 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



[Slide taken from ‘Parallele Algorithmen und Datenverarbeitung’, Bielefeld University]

How to?

- How to implement an efficient algorithm?
  - 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)

```
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 pointer
  - Function object
  - Lambda function
  - … anything that can be “called”
- A thread itself does not care about the return value
  - Cannot return data directly

- `std::jthread`
- same general behavior as `std::thread`
- rejoins automatically on destruction
- can be cancelled/stopped in certain situations

std::thread

```
#include <iostream>
#include <thread>
class callable {
private:
  int i;
  int j;
public:
  callable(int a, int b) : i(a), j(b) {}
  void operator() () {
    std::cout << "t1: " << i + j << '\n';
  }
};
void func(int a, int b) {
  std::cout << "t2: " << a * b << '\n';
}
```

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

std::thread

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

```
#include <array>
#include <iostream>
#include <thread>
#include <vector>
std::array<int, 4> results;
void power(int id, int a) {
  results[id] = a*a;
}
int main() {
  std::vector<std::thread> threads;
  for (int i = 0; i < results.size(); ++i) {
    threads.push_back(std::thread(power, i, i+1));
  }
  for (auto& t : threads) { t.join(); }
  for (auto result : results) {
    std::cout << result << '\n';
  }
  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 a mutex lock)
- Only one thread is allowed to execute locked code

```
#include <iostream>
#include <thread>
#include <vector>
#include <mutex>
std::mutex results_mutex;
std::vector<int> results;
```

```
void power(int a) {
  int b = a * a;
  std::lock_guard<std::mutex> guard(results_mutex);
  results.push_back(b);
}
int main() {
  std::vector<std::thread> threads;
  for (int i = 1; i < 10; ++i) {
    threads.push_back(std::thread(power, i));
  }
  for (auto& t : threads) { t.join(); }
  for (auto i : results) {
    cout << i << '\n';
  }
  return 0;
}
```

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”

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

std::atomic<int> global_int(0);
void inc_global() { ++global_int; }
int main() {
  std::vector<std::thread> threads;
  for (int i = 0; i < 10; ++i) {
    threads.push_back(std::thread(inc_global));
  }
  for (auto& t : threads) { t.join(); }
  std::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 in the future
  - This avoids using global variables!

```cpp
#include <iostream>
#include <thread>
#include <future>
#include <vector>
int func(int a, int b) { return  a * b; }

int main() {
  std::packaged_task<int(int,int)> task(func);
  std::future<int> result = task.get_future();
  std::thread t1(std::move(task), 2, 10);
  t1.join();
  std::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`

```
#include <iostream>
#include <thread>
#include <future>

int ret10() { return 10; }

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


HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN
Fraunhofer IEM

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

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

std::future and std::promise

- “Computing with future values”
- 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

```
#include <iostream>
#include <thread>
#include <future>
#include <vector>
#include <chrono>
int power(future<int> f) {
  int a = f.get();
  return  a * a;
}
int main() {
  std::promise<int> p;
  std::future<int> f = p.get_future();
  std::future<int> res =
    std::async(std::launch::async,
               power, std::move(f));
  std::this_thread::sleep_for(std::chrono::seconds(10));
  p.set_value(10);
  int result = res.get();
  std::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 x86_64-unkonwn-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

```
#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 it knows (including the dark arts)
- `-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)

```
#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;
  }
```

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

```
  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) << '\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
  - Using a clever implementation and compiler optimizations we could save
    - g++ **~ 13m 8s**
    - clang++ **~ 13m 37s**
  - Initial times g++ ~ 13m 18s / clang++ ~ 13m 48s
  - Final times g++ ~ 10s / clang++ ~ 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 standard-compliance
    - + use of `-ffast-math`, `-fno-protect-parens`, `-fstack-arrays`
- `-fmarch=native`
  - Generate specific code for the target architecture
- `-DNDEBUG`
  - Do not use debug mode code
- `-fdata-sections`, `-ffunction-sections`
  - Place each data item and function into its own segment (might allow 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 for you!
  - 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

Find the minimum of four integer values!

Compiler Explorer | Editor | Diff View | More | Share | Other

C++ source #1

```cpp
// find the minimum of four integers using hand-crafted fu
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;
}
```

x86-64 gcc 7.2 (Editor #1, Compiler #1) C++

x86-64 gcc 7.2 | -O2

11010 | .LX0: | .text | // | \s+ | Intel | Demangle

```asm
min(int, int, int, int):
  cmp edx, ecx
  mov eax, edi
  cmovg edx, ecx
  cmp edx, esi
  cmovle esi, edx
  cmp esi, edi
  cmovle eax, esi
  ret
```

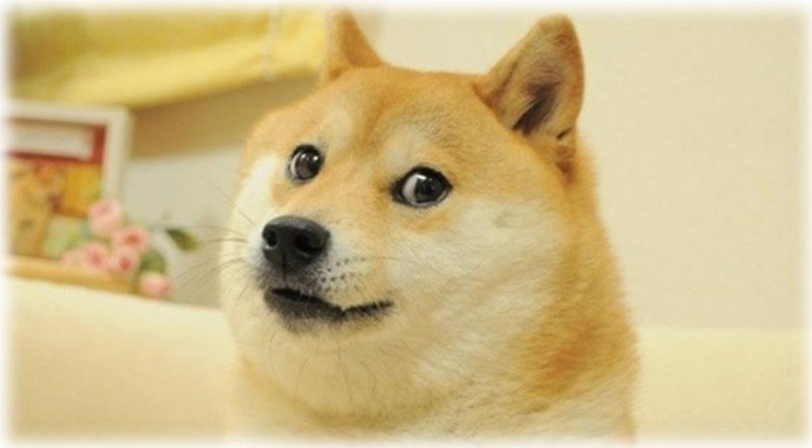
g++ (GCC-Explorer-Build) 7.2.0 - *cached*

Wow!

Such minimum!

Much smart!

Very clever!



Compiler Explorer Editor Diff View More▾ Share▾ Other▾

C++ source #1

C++

```cpp
// find the minimum of four integers using hand-crafted fu
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;
}
```

x86-64 gcc 7.2 (Editor #1, Compiler #1) C++

x86-64 gcc 7.2 -O2

11010 .LX0: .text // \s+ Intel Demangle

```asm
min(int, int, int, int):
  cmp edx, ecx
  mov eax, edi
  cmovg edx, ecx
  cmp edx, esi
  cmovle esi, edx
  cmp esi, edi
  cmovle eax, esi
  ret
```

g++ (GCC-Explorer-Build) 7.2.0 - *cached*

C++ source #2

C++

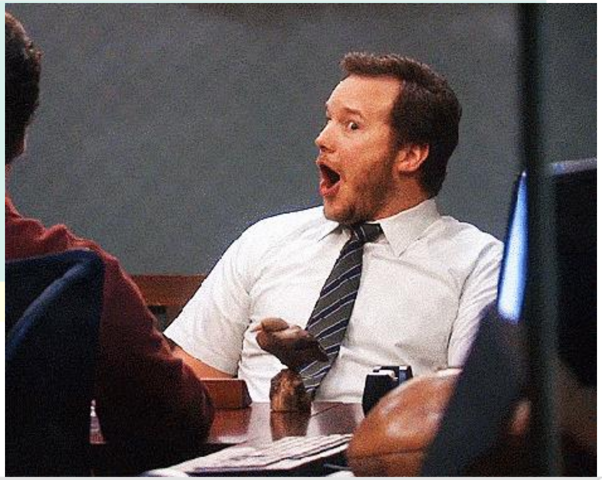
```cpp
#include <algorithm>
// finding the minimum using the STL
int min(int a, int b, int c, int d) {
    return std::min({a, b, c, d});
}
```

x86-64 gcc 7.2 (Editor #2, Compiler #2) C++

x86-64 gcc 7.2 -O2

11010 .LX0: .text // \s+ Intel Demangle

```asm
min(int, int, int, int):
  cmp edi, esi
  mov eax, ecx
  cmovg edi, esi
  cmp edi, edx
  cmovle edx, edi
  cmp edx, ecx
  cmovle eax, edx
  ret
```



g++ (GCC-Explorer-Build) 7.2.0 - *911ms*

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?