Algorithm Engineering

False Sharing, Race Conditions, and Schedules

Mark Blacher

Friedrich Schiller University Jena

Winter Semester of 2022/23

Your Project (Best Case)

- version control with **Git**
- the use of **CMake** to build the project
- correct, efficient, parallel, vectorized, cache-friendly, well organized, **commented**, (portable) C++/C code
- If you write a **library**, the **API** should be **clear** and **well documented**
- If you create a **standalone executable**, **document well** the **usage** and the **available command line options** (command line parameters can be parsed with `getopt`)
- **unit tests** (for example with **Catch2**)

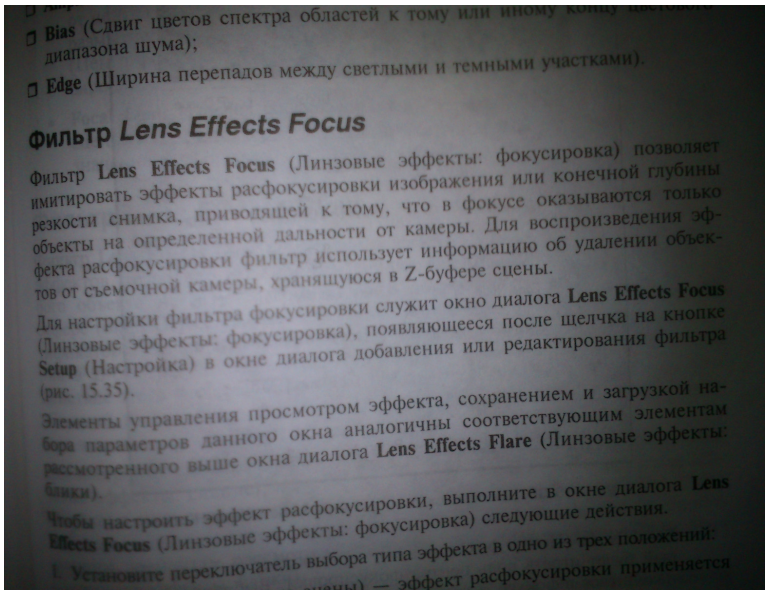
Project 1: **Enhancer** for **Scanned Images**

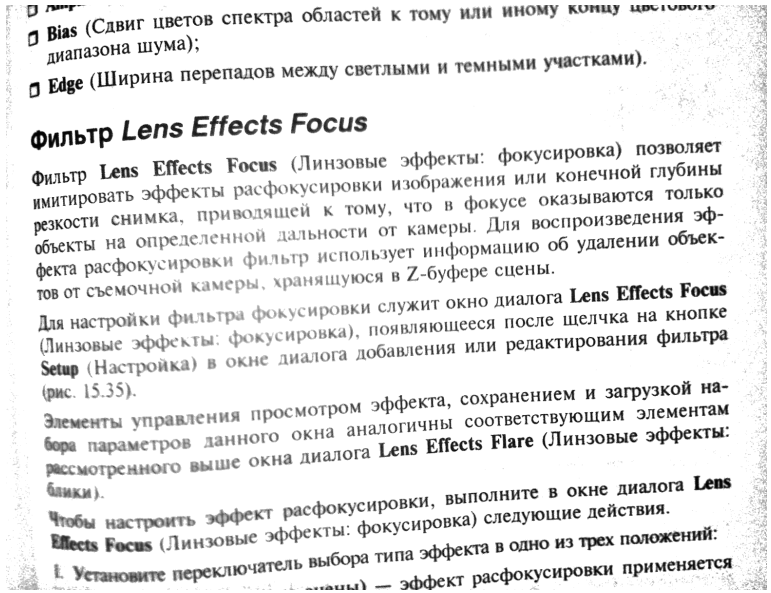
Input: portable pix map (ppm) color image(s) in ASCII-format

Output: cleaned ppm image(s) in ASCII-format

Motivation: free tool (**executable**) for improving the quality of scanned documents before printing (or sending) them

Keywords: noise reduction, background removal, image filter





Project 2: **Generating 2D Datasets** With **Similar Statistical Properties**

Input: a 2D point cloud

Output: a 2D point cloud with same statistical properties (for a given shape)

Motivation: the current approach is slow (written in Python); perhaps it can be done in a more general way than through hordecoded shapes in the program

Keywords: simulated annealing, genetic algorithms, mean, standard deviation, correlation

For more information see:
https://www.autodesk.com/research/publications/same-stats-different-graphs

Project 3: Parallel In-Place Partitioning, Quickselect and Quicksort 

Input: an array of any type of data

Output: a correctly modified array (depends on the called function)

Motivation: learn advanced parallelization techniques

Keywords: partitioning, quickselect, quicksort, atomics (fetch-and-add), header-only library, templates, skewed distributions

Remark: Provided Functionality should be similar to `std::partition`, `std::nth_element`, `std::sort` (easy to test the implementation)

For more information see:

`A Simple, Fast Parallel Implementation of Quicksort`

`How to use templates for sorting`

`Parallel implementations for benchmarking`

Your Own Project?

- You can choose one of three mentioned projects
- But, it is also possible to come up with your own project
- If you have an **own project idea**, **describe it to me in an e-mail**
- If your own project is suitable for the course, then you can do it

Send me an **e-mail until November 18rd** with a **description of your own project**, **or**, with the **information which of the three presented projects** you would like to do.

It might be beneficial for you to **add me to your repository**.

mark.blacher@uni-jena.de

Avoid False Sharing Among Threads

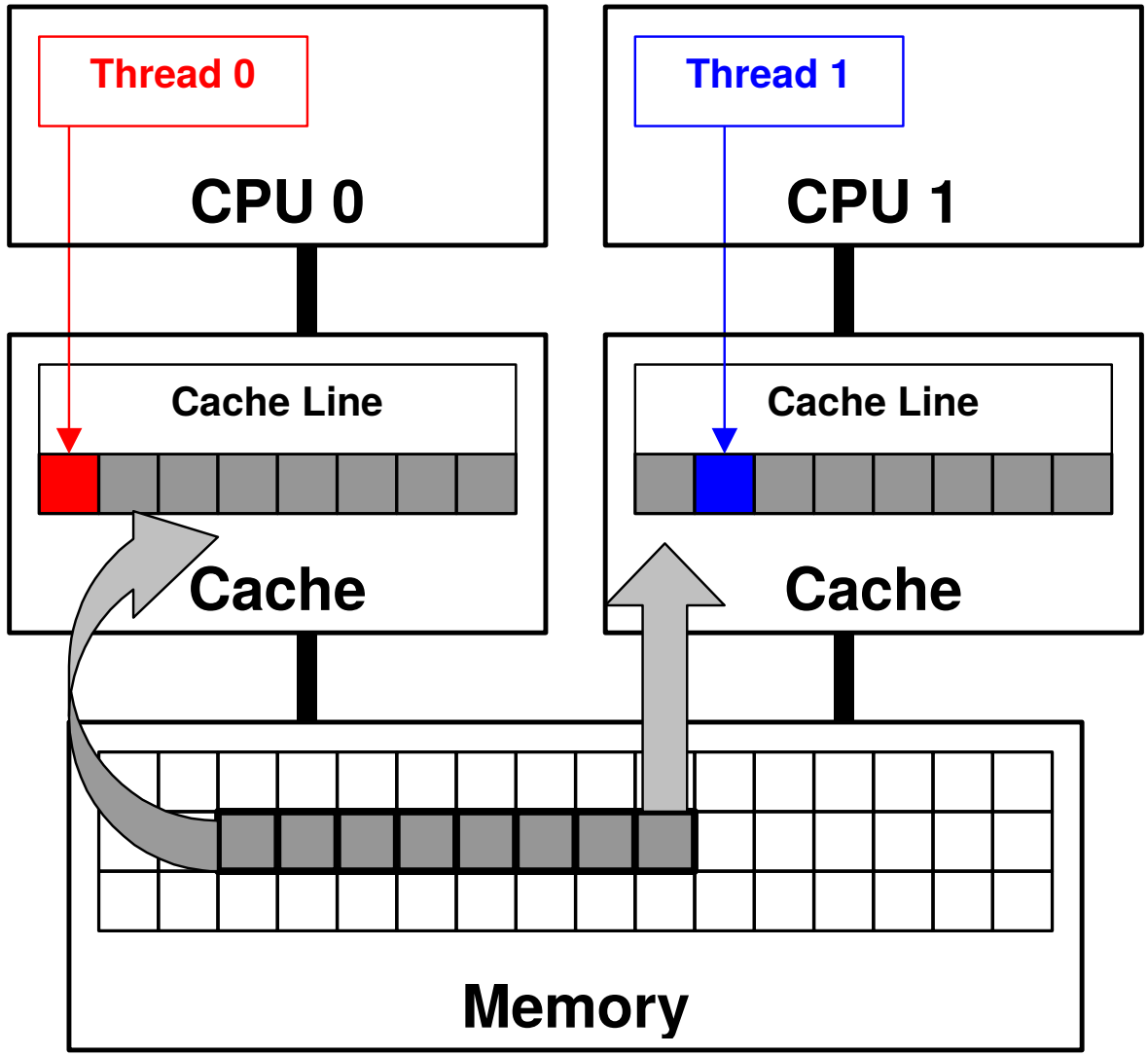
Cache Line

The smallest unit of memory that can be transferred between the main memory and the cache (**typically 64 bytes**).

False Sharing

False sharing occurs when **threads on different processors modify variables that reside on the same cache line**. This invalidates the cache line and forces a memory update to maintain cache coherency.

False Sharing



Click here for more information

Code Example **With False Sharing**

false_sharing.cpp

```
1  #include <iostream>
2  #include <omp.h>
3
4  using namespace std;
5
6  int main() {
7    const int threads = 4; // number of threads
8    volatile long arr[threads]; // no compiler optimizations on arr
9    double start_time = omp_get_wtime(); // wall clock time in seconds
10 #pragma omp parallel num_threads(threads)
11   {
12     const int thread_id = omp_get_thread_num();
13     for (long i = 0; i < 1000000000; ++i) {
14       // each thread uses (probably) the same cache line
15       ++arr[thread_id];
16     }
17   }
18   // cout the byte where arr starts in the cache line
19   cout <<  ((long) &arr[0]) % 64 << endl;
20   cout << omp_get_wtime() - start_time << " seconds\n";
21 }
```

Code Example **Without False Sharing**

no_false_sharing.cpp

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

using namespace std;

int main() {
  const int threads = 4; // number of threads
  const int PAD = 8; // padding value to avoid false sharing
  volatile long arr[threads * PAD]; // no compiler optimizations on arr
  double start_time = omp_get_wtime(); // wall clock time in seconds
#pragma omp parallel num_threads(threads)
  {
    const int thread_id = omp_get_thread_num();
    for (long i = 0; i < 1000000000; ++i) {
      // each thread increments a value on a different cache line
      ++arr[thread_id * PAD];
    }
  }
  cout << omp_get_wtime() - start_time << " seconds\n";
}
```

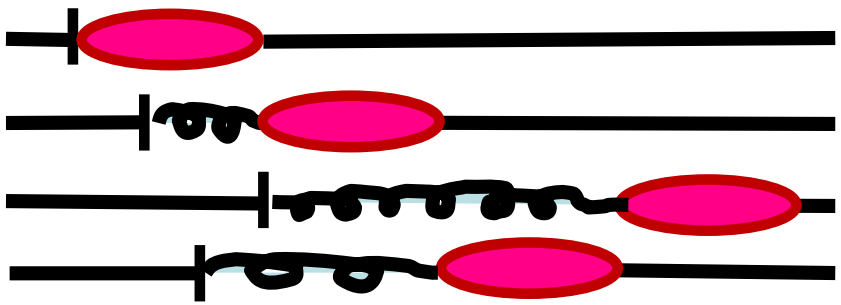
Prevent Race Conditions With Mutual Exclusion

Race Condition

A race condition occurs when **two or more threads try to change shared data** at the same time.

Mutual Exclusion

A form of synchronization in which **only one thread** at a time **can execute a block of code**.



Mutual Exclusion Constructs

- `#pragma omp critical` *// critical block runs on one thread at a time*
- `std::mutex` *// an instance of std::mutex should be shared between threads*
- `#pragma omp atomic` *// updates a memory location atomically*
- ...

Usually a **thread** that reaches a mutual exclusion construct **waits until it gains exclusive access**.

Can You Spot the **Race Condition Bug**?

race_condition.cpp

```
  double pi = 0.0;
#pragma omp parallel num_threads(4)
  {
    int num_threads = omp_get_num_threads();
    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
    }
    pi += sum_local * 4 * width;
  }
```

Why Is This **Code Slow**?

pi_slow.cpp

```
1   double pi = 0.0;
2 #pragma omp parallel num_threads(4)
3   {
4     int num_threads = omp_get_num_threads();
5     int thread_id = omp_get_thread_num();
6     for (int i = thread_id; i < num_steps; i += num_threads) {
7       double x = (i + 0.5) * width; // midpoint
8 #pragma omp atomic
9       pi = pi + (4.0 / (1.0 + x * x)) * width;
10    }
11  }
```

The #pragma omp for Construct

#pragma omp for splits up a for loop within a parallel region among the threads in a team

pragma_omp_for.cpp

```
1  image *denoise_image(image *in) { // mock denoise
2    image *out = nullptr;
3    for (volatile int i = 0; i < 20000000; ++i) {}
4    return out;
5  }
6
7  int main() {
8    int amount_images = 64;
9    vector<image *> images(amount_images, nullptr); // mock images
10   vector<image *> denoised_images(amount_images); // mock output
11   double start = omp_get_wtime();
12 #pragma omp parallel
13   {
14 #pragma omp for // denoise a bunch of images
15     for (int i = 0; i < amount_images; ++i) {
16       denoised_images[i] = denoise_image(images[i]);
17     }
18   }
19   cout << omp_get_wtime() - start << " seconds" << endl;
20 }
```

Unbalanced Workload

unbalanced_workload.cpp

```cpp
image *denoise_image(image *in, int scale) { // mock denoise
  image *out = nullptr;
  const int loop_until = 20000 * scale; // unbalanced workload
  for (volatile int i = 0; i < loop_until; ++i) {}
  return out;
}

int main() {
  int amount_images = 64;
  vector<image *> images(amount_images, nullptr); // mock images
  vector<image *> denoised_images(amount_images); // mock output
  double start = omp_get_wtime();

  // #pragma omp parallel for is an "OpenMP shortcut"
#pragma omp parallel for num_threads(4) // denoise a bunch of images
  for (int i = amount_images - 1; i > -1; --i) {
    denoised_images[i] = denoise_image(images[i], i * i);
  }

  cout << omp_get_wtime() - start << " seconds" << endl;
}
```

The Schedule Clause

The schedule clause affects **how loop iterations are mapped onto threads when using** `#pragma omp for`

By far the **most important schedule clauses** are:

- **static** (we make the scheduling decision at compile time)
- **dynamic** (deciding at runtime, how to break up work between the threads)

Usage:

- `schedule(static [,chunk])`
  Deal-out blocks of iterations of size "chunk" to each thread.
- `schedule(dynamic [,chunk])`
  Each thread grabs "chunk" iterations off a queue until all iterations have been handled.

Schedule Clause Demo

schedule_clause.cpp

```
int main() {
#pragma omp parallel num_threads(4)
  {
    // each thread has its own stringstream instance
    stringstream thread_iterations;
    // task for you: experiment with different schedules
#pragma omp for schedule(static, 8)
    for (int i = 0; i < 32; ++i) {
      // write the iterations of a thread to the stream
      thread_iterations << i << " ";
    }
    // only one thread is allowed to output to the console
#pragma omp critical
      cout << "thread " << omp_get_thread_num()
           << " computed the iterations: "
           << thread_iterations.str() << endl;
  }
}
```

Choosing the Right Schedule Clause

If each iteration in the for loop needs the **same workload**, make the **schedule static**.

If the **workload** is **unbalanced** between iterations, make the **schedule dynamic**.

How to Stop a Parallel For Loop

- Unfortunately, **you can't *easily* just break out** of a **parallel for loop** in OpenMP
- But there's a **workaround**

Don't Stop a Parallel For Loop, Just Avoid Work

continue_hack.cpp

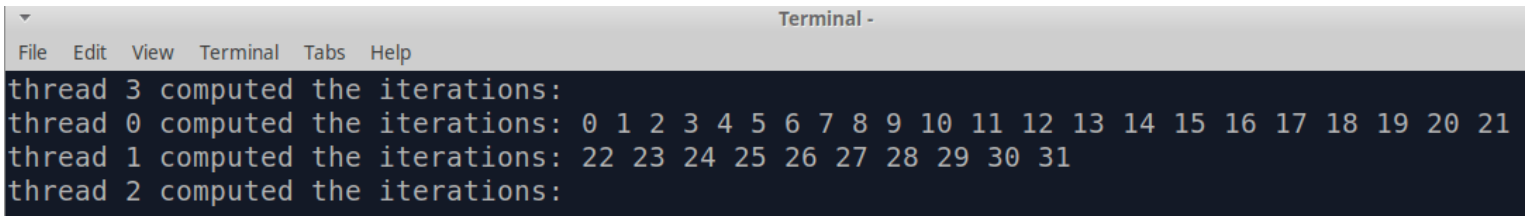
```cpp
bool is_solution(int number) {  // test if number solves the problem
  for (volatile int i = 10000000; i--;) {}  // mock computation
  return number > 133 && number < 140;
}

int main() {
  constexpr int biggest_possible_number = 10000;
  // To avoid undefined code, it is necessary to use std::atomic for variables
  // where race conditions may happen.
  // https://databasearchitects.blogspot.com/2020/10/c-concurrency-model-on-x86-for-dummies.html
  atomic<bool> solution_found(false);  // if true than we found the solution
  atomic<int> final_solution(INT32_MAX);
  const double start = omp_get_wtime();

#pragma omp parallel for schedule(dynamic)  // start parallel region
  for (int i = 0; i < biggest_possible_number; ++i) {
    if (solution_found)  // we found the solution, just continue iterating
      continue;
    if (is_solution(i)) {     // find some solution, not necessary the smallest
      solution_found = true;
      final_solution = i;
    }
  }  // end parallel region
  // check if we've found a solution at all is omitted, you can add the check
  cout << "The solution is: " << final_solution << endl;
  cout << omp_get_wtime() - start << " seconds" << endl;
}
```

Coding Warmup

1. Fix the **race condition bug** on page 13 with a `std::mutex`.
2. **Reduce** the **runtime** of the image denoising program on slide 16 by adding an **appropriate schedule**.
3. What **schedule** on slide 18 produces the **following (bad) pattern**?



4. **Modify** the **program** on slide 21 so that it **always** finds the **smallest possible solution**.

Solution for the Warmup *Smallest* Program (Lock)

continue_hack_min_solution_lock.cpp

```cpp
bool is_solution(int number) { // test if number solves the problem
  for (volatile int i = 10000000; i--;) {} // mock computation
  return number > 133 && number < 140;
}

int main() {
  constexpr int biggest_possible_number = 10000;
  atomic<int> final_solution(INT32_MAX);
  const double start = omp_get_wtime();
  mutex m; // mutual exclusion construct for updating final_solution

#pragma omp parallel for schedule(dynamic) // we could also use schedule(static, 1)
  for (int i = 0; i < biggest_possible_number; ++i) {
    if (final_solution < i) // we found a smaller solution, just continue iterating
      continue;
    if (is_solution(i)) {    // one solution found, update final_solution if it's smaller
      lock_guard<mutex> lg(m); // constructs a lock_guard, locking the given mutex
      final_solution = min(i, final_solution.load()); // only one thread computes min
    }
  }
  // end parallel region
  // check if we've found a solution at all is omitted, you can add the check
  cout << "The smallest solution is: " << final_solution << endl;
  cout << omp_get_wtime() - start << " seconds" << endl;
}
```

Solution for the Warmup *Smallest* Program (Atomic)

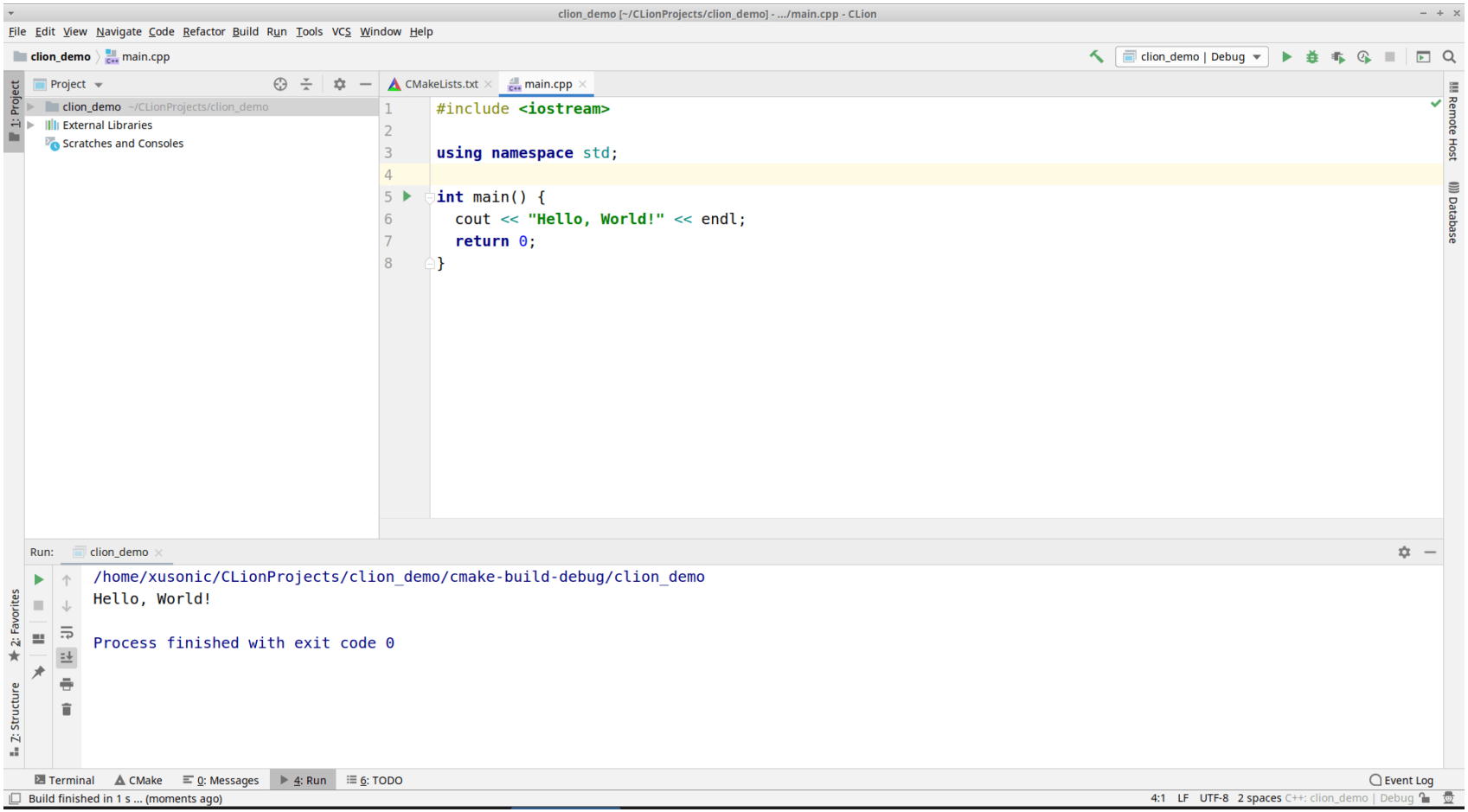
continue_hack_min_solution_atomic.cpp

```cpp
bool is_solution(int number) { // test if number solves the problem
  for (volatile int i = 10000000; i--;) {} // mock computation
  return number > 133 && number < 140;
}

int main() {
  constexpr int biggest_possible_number = 10000;
  atomic<int> final_solution(INT32_MAX); // now we use an atomic
  const double start = omp_get_wtime();

#pragma omp parallel for schedule(dynamic) // start parallel region
  for (int i = 0; i < biggest_possible_number; ++i) {
    if (final_solution < i) // we found a smaller solution, just continue iterating
      continue;
    if (is_solution(i)) { // one solution found, update final_solution if it's smaller
      int previous = final_solution.load();
      while (previous > i && !final_solution.compare_exchange_weak(previous, i)) {}
      // http://www.cplusplus.com/reference/atomic/atomic/compare_exchange_weak/
      // atomic.compare_exchange_weak(T& expected, T desired);
      // 1. Compare a given value expected with the value stored in atomic.
      // 2. If both values stored in expected and atomic coincide then set atomic to a
      // given value desired, otherwise write the actual value stored in atomic to expected.
      // 3. Return true if the swap in 2. was successful, otherwise return false.
    }
  } // end parallel region
  // check if we've found a solution at all is omitted, you can add the check
  cout << "The solution is: " << final_solution << endl;
  cout << omp_get_wtime() - start << " seconds" << endl;
}
```

Clion Demo


clion_demo [~/CLionProjects/clion_demo] - .../main.cpp - CLion
File Edit View Navigate Code Refactor Build Run Tools VCS Window Help
clion_demo main.cpp
clion_demo | Debug
Project
clion_demo ~/CLionProjects/clion_demo
External Libraries
Scratches and Consoles
CMakeLists.txt
main.cpp
#include <iostream>
using namespace std;
int main() {
cout << "Hello, World!" << endl;
return 0;
}
Run: clion_demo
/home/xusonic/CLionProjects/clion_demo/cmake-build-debug/clion_demo
Hello, World!
Process finished with exit code 0
Terminal CMake 0: Messages 4: Run 6: TODO Event Log
Build finished in 1 s ... (moments ago)
4:1 LF UTF-8 2 spaces C++: clion_demo | Debug
1: Project
2: Favorites
7: Structure
Remote Host
Database

Exam Assignments

- What causes **false sharing**?
- How do **mutual exclusion constructs** prevent **race conditions**?
- Explain the differences between **static** and **dynamic schedules in OpenMP**.
- What can we do if we've **found a solution** while running a **parallel for loop** in OpenMP, but still have **many iterations left**?
- Do the **coding warmup** on **slide 22**. Explain in your own words how `std::atomic::compare_exchange_weak` work.
- **Rewrite** the **parallel program** for estimating π from the last exercise 📌 to use the construct `#pragma omp for`.