Advanced Parallel Programming Exercise 3

Marvin Kaster

TECHNISCHE UNIVERSITÄT DARMSTADT

Summer term 2025
05.06.2025

Please solve the following tasks by 05.06.2025. The results are not graded, but a solution is discussed on 05.06.2025.

Task 1: Calculate the sum of a vector

Your task is to calculate the sum of a large vector. First, generate a random vector. Then, calculate the sum in serial. Finally, use `std::thread` to calculate the sum in parallel. One thread is responsible to calculate the sum of a subsection of the vector. As you cannot return the value directly as a return value, store it in a second vector where thread `i` will store its result at postion `i`. Then, add all partial sums to calculate the final overall sum. Make sure that your serial and parallel implementation calculate the same value and measure how long each implementation needs. Create a plot for your time measurements for different number of threads and sizes of vectors.

We added an utility function for your convience.

Hints:

- The constructor of std::thread will create copies of each passed parameter. Take a look at `std::ref` and `std::cref`[1] to avoid creating unnecessary copies of vectors.
- Take a look at `std::chrono`[2] for time measurements
- Take a look at `std::reduce`[3] for calculating a sum of a vector sequientally

```cpp
#pragma once

#include <random>
#include <vector>

template<std::integral T>
static std::vector<T> create_random_vector(const std::size_t size=1e5, const int seed = 42) {
    std::mt19937 mersenne_engine(seed);
    std::uniform_int_distribution<T> dist {1, 10};
    const auto gen = [&](){
        return dist(mersenne_engine);
    };

    std::vector<T> random_vector(size);
    std::generate(random_vector.begin(), random_vector.end(), gen);

    return random_vector;
}

template<typename T>
static void print_vector(const std::vector<T> &vector) {
    for (const auto &i: vector) {
        std::cout << i << " ";
    }
    std::cout << "\n";
```

[1] https://en.cppreference.com/w/cpp/utility/functional/ref
[2] https://en.cppreference.com/w/cpp/chrono/steady_clock/now
[3] https://en.cppreference.com/w/cpp/algorithm/reduce

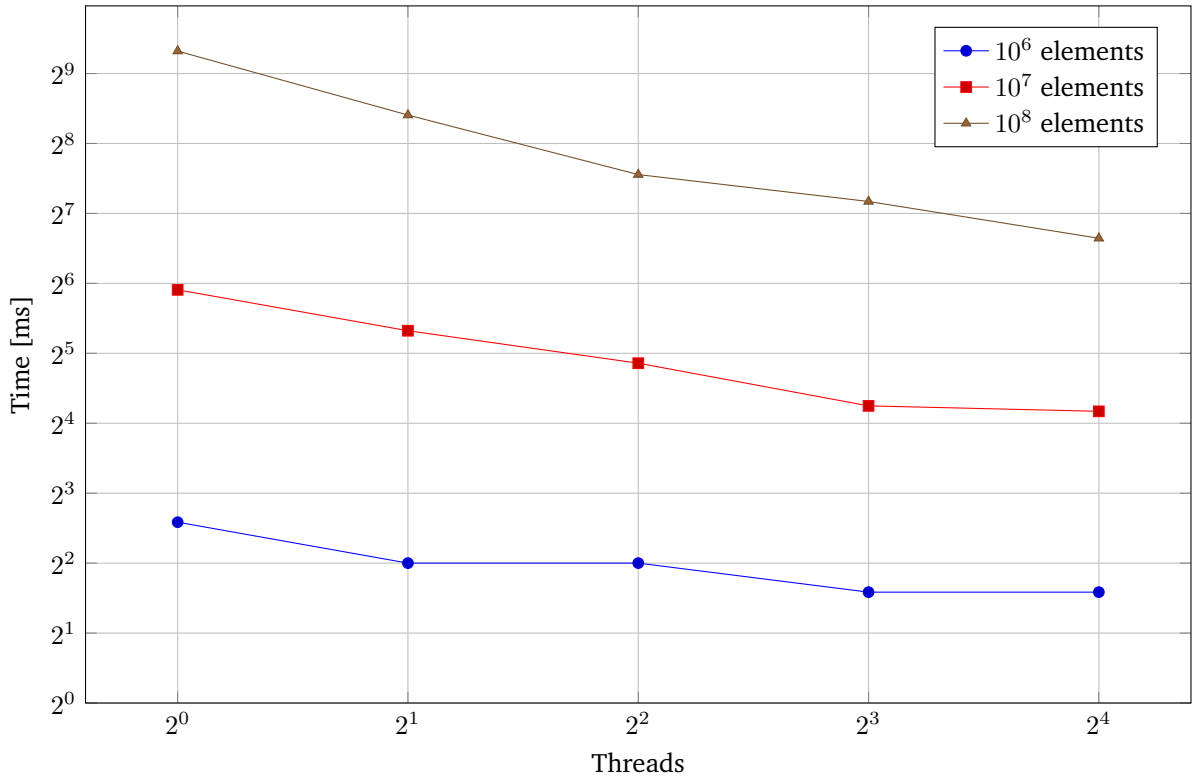

Figure 1: Execution time of the implementation (y-axis) vs number of threads used (x-axis) for different number of vector sizes.

```
}
```

Listing 1: Utility functions

Solution:

```
#include <algorithm>
#include <cassert>
#include <iostream>
#include <thread>
#include <vector>

#include "util.h"

using number_type = int;

void partial_sum(const std::vector<number_type> &input_vector, const std::size_t begin, const std::size_t end,
                 number_type &result_output) {
    assert(begin <= input_vector.size());
    assert(end <= input_vector.size());
    assert(begin <= end);
    const auto sum = std::reduce(std::next(input_vector.begin(), begin), std::next(input_vector.begin(), end), 0,
                                 std::plus<>());
    result_output = sum;
}


int main() {
    const std::size_t size = 1e7;
```

```
    const std::size_t number_threads = 4U;
    const auto vector = create_random_vector<number_type>(size);

    //Serial implementation
    const auto begin_serial = std::chrono::steady_clock::now();
    const auto serial_sum = std::reduce(vector.begin(), vector.end(), 0, std::plus<>());
    assert(size <= serial_sum && serial_sum <= size * 10U);
    const auto end_serial = std::chrono::steady_clock::now();
    const auto duration_serial = end_serial - begin_serial;
    std::cout << "Serial result " << serial_sum << '\n';
    std::cout << "Took " << std::chrono::duration_cast<std::chrono::milliseconds>(duration_serial).count()
     << "ms\n\n";


    //Parallel implementation
    const auto begin_parallel = std::chrono::steady_clock::now();
    std::vector<number_type> results(number_threads);
    std::vector<std::thread> threads;
    threads.reserve(number_threads);
    const auto number_numbers_per_thread = std::max(std::size_t(1U), size / number_threads);


    for (auto thread_id = 0U; thread_id < number_threads; thread_id++) {
        const std::size_t begin = std::min(thread_id * number_numbers_per_thread, size);
        const std::size_t end = std::min((thread_id + 1) * number_numbers_per_thread, size);
        auto thread = std::thread(partial_sum, std::cref(vector), begin, end, std::ref(results[thread_id])
    );
        threads.push_back(std::move(thread));
    }

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

    const auto parallel_sum = std::reduce(results.begin(), results.end(), 0, std::plus<>());
    const auto end_parallel = std::chrono::steady_clock::now();
    const auto duration_parallel = end_parallel - begin_parallel;
    std::cout << parallel_sum << '\n';
    std::cout << "parallel result " << parallel_sum << '\n';
    std::cout << "Took " << std::chrono::duration_cast<std::chrono::milliseconds>(duration_parallel).count
    ()
              << "ms\n\n";
    std::cout << "Used " << number_threads << " threads out of " << std::thread::hardware_concurrency() <<
     " cores\n";

    return 0;
}
```

Listing 2: Vector addition

Task 2: Implement a thread safe counter map

When we count the occurrences of unique objects in a list, we need some associative container that keeps track of the previous occurrences for each unique object. This could be implemented with a `std::unordered_map`. However, we would like to be able to go through large lists in parallel or through multiple lists in parallel. The implementation of `std::unordered_map` is not thread-safe, and therefore, we cannot share an `std::unordered_map` object with multiple threads. Your task is to implement a thread-safe counter. Internally, the data should be stored using a hash list to enable fast access to the elements.

The idea is to use the key's hash value to calculate the index in a fixed-size list. Note that this can result in the same index for different keys. Hence, we need a second list with a variable size for each index in the hash list, over which we have to iterate to find the entry with the matching key. Figure 2 shows a visualization of the data structure. Please implement the interface shown below.

```
template<typename KeyType, typename ValueType, typename HashType=std::hash<KeyType>>
class ThreadSafeCounterDict {
  public:
```

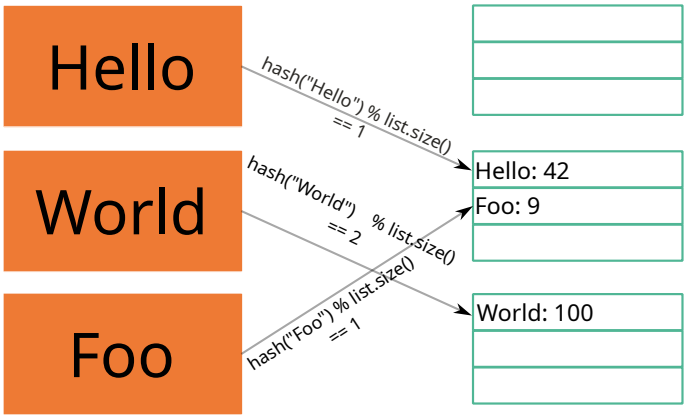

Figure 2: Visualization of a hash map. We calculate an index in our list by calculate the hash value of the key and apply a mudolo operation. At this index we have a nested list to which we can add the value. If two keys point to the same index, we need to check all elements in that nested list.

```
4   CounterDict(const ValueType &default_value, const size_t list_size);
5
6   ValueType get_value_for(const KeyType &key) const;
7
8   ValueType increase_value_for_key(const KeyType &key, const ValueType &increment);
9
10  std::vector<std::pair<KeyType, ValueType>> convert_to_pairs();
11
12 };
```

Solution:

Our counter dictionary interface has only three methods:

- Read the value of a key
- Increase the value of a key
- Return all key-value pairs

We follow simple thread-safety rules in our implementation. We avoid returning and storing references of objects to ensure modification from outside the class. As the bucket list has a fixed size and is not modified after the initialization in the constructor, we need no mutex for it. It is sufficient to use a single mutex per bucket. This mutex can use a `std::shared_lock` for read operations and a `std::unique_lock` for writing operations. The mutex must be locked at the beginning of the member function of the bucket before any reading or writing of the key-value pair list. It will be unlocked automatically after leaving the function.

If two threads want to add a value for the same key, a race condition could appear if it checks for the existence of the key first and then adds it without holding a lock the entire time. We avoid this problem by only providing a method to increase the value and provide a default value that is used when the key is not in the dictionary yet. In that way, we can hold a lock the entire time. Returning a default value in our read method solves the same problem in that we do not have to check if the key is in it.

Exceptions can only be raised in the `increase_value_for_key` function. The method `std::vector::emplace_back` could throw an exception (e.g., because of low memory), but it is exception-safe, leaving the vector in a valid state. Otherwise, exceptions can only be thrown in the operators from `ValueType` (e.g., the assignment operator, addition operator). It is up to the user to ensure thread safety for the provided type.

The `convert_to_pairs` method locks multiple mutexes and is the only point where a deadlock could occur. Hence, we must always lock the mutexes in the same order. We do it by locking the mutexes of the buckets with increasing indices.

```cpp
#pragma once

#include <functional>
#include <memory>
#include <mutex>
#include <shared_mutex>
#include <tuple>
#include <vector>

template<typename KeyType, typename ValueType, typename HashType=std::hash<KeyType>>
class ThreadSafeCounterDict {
  private:
  /**
  * Class that holds items for a single bucket
  */
  class Bucket {
    //The actual data in the bucket
    mutable std::vector<std::pair<KeyType, ValueType>> item_data;

    /**
    * Finds the entry for a certain key within this bucket
    * @param key The key
    * @return Iterator that point to the key-value pair or to the end of the vector when the key is not in the bucket
    */
    std::vector<std::pair<KeyType, ValueType>>::iterator find_entry_for_key(const KeyType &key) const {
      return std::find_if(item_data.begin(), item_data.end(),
      [&](const auto &item) { return item.first == key; });
    }

    public:
    //The mutex for the bucket
    mutable std::shared_mutex item_mutex;

    /**
    * Returns the current value for the key in this bucket
    * @param key The key of the key-value pair
    * @return The value belonging to the key
    */
    [[nodiscard]] ValueType get_value_for_key(const KeyType &key, const ValueType &initial_value) const {
      //We need a shared lock on this bucket while we read the value so that no thread can modify it
      //Other threads reading the same bucket is ok
      std::shared_lock<std::shared_mutex> lock(item_mutex);

      //Find the value belonging to the key
      const auto &it = find_entry_for_key(key);
      if (it != item_data.end()) {
        //Return a copy of the value
        return it->second;
      }
      //No value is assigned yet return the initial value
      return initial_value;
    }

    /**
    * Increase the value belonging to the key in this bucket
    * @param key The key of the key-value pair
    * @param increment The number of which we increase the value
    * @param initial_value The initial value belonging to the key when there is no value assigned to the key yet
    * @return The new value belonging to the key
    */
    ValueType
    increase_value_for_key(const KeyType &key, const ValueType &increment, const ValueType &initial_value) {
      //We need an unique lock on this bucket while we modify it
      //Other threads are not allowed to read the bucket while we modify it
      std::unique_lock<std::shared_mutex> lock(item_mutex);

      //Find the value belonging to the key
```

```
    auto it = find_entry_for_key(key);
    if (it != item_data.end()) {
      //Read the old value
      const auto old_value = it->second;
      //Add the increment
      const auto new_value = old_value + increment;
      //Store the new value
      it->second = new_value;
      //Return a copy of the new value
      return new_value;
    }

    //No value is assigned yet
    //Add the increment to the initial value
    const auto new_value = initial_value + increment;
    item_data.emplace_back(key, new_value);
    return new_value;
  }

  /**
  * Returns a list of all stored key-value pairs in this bucket
  * @return List of all key-value pairs
  */
  [[nodiscard]] std::vector<std::pair<KeyType, ValueType>> get_item_data() const {
    //We need a shared lock while we copy the data to prevent other threads from modifying it meanwhile
    std::shared_lock<std::shared_mutex> lock(item_mutex);

    return item_data;
  }
};

public:
/**
* Constructor
* @param init_value The initial object value for each key
* @param list_size The fixed size of the hash list
*/
ThreadSafeCounterDict(const ValueType &initial_value, const size_t list_size) : initial_value(
  initial_value),
buckets(list_size) {
  for (auto i = 0; i < list_size; i++) {
    buckets[i] = std::make_unique<Bucket>();
  }
}

ThreadSafeCounterDict() = delete;

ThreadSafeCounterDict(ThreadSafeCounterDict &other) = delete;

ThreadSafeCounterDict &operator=(const ThreadSafeCounterDict &other) = delete;

/**
* Returns the current value for the key
* @param key The key of the key-value pair
* @return The value belonging to the key
*/
[[nodiscard]] ValueType get_value_for_key(const KeyType &key) const {
  //Find the bucket belonging to the key
  const auto &bucket = get_bucket(key);

  //Return the value of the bucket. The bucket handles the mutex itself
  return bucket.get_value_for(key, initial_value);
}

/**
* Increase the value belonging to the key
* @param key The key of the key-value pair
* @param increment The number of which we increase the value
* @return The new value belonging to the key
*/
```

```
  ValueType increase_value_for_key(const KeyType &key, const ValueType &increment) {
    //Find the bucket belonging to the key
    auto &item_holder = get_bucket(key);

    //Increase the value by the increment and return it. The bucket handles the mutex itself
    return item_holder.increase_value_for_key(key, increment, initial_value);
  }

  /**
  * Returns a list of all stored key-value pairs
  * @return List of all key-value pairs
  */
  std::vector<std::pair<KeyType, ValueType>> convert_to_pairs() {
    //We need to lock all buckets at the beginning as we want to iterate over all of them
    std::vector<std::shared_lock<std::shared_mutex>> locks{};
    for (const auto &bucket: buckets) {
      locks.emplace_back(bucket->item_mutex);
    }

    //Copy the data from every bucket
    std::vector<std::pair<KeyType, ValueType>> all_pairs{};
    for (auto i = 0; i < buckets.size(); i++) {
      const auto &item_data = buckets[i]->get_item_data();
      //Copy each element of item data (key-value pairs) into the new vector
      all_pairs.insert(all_pairs.end(), item_data.begin(), item_data.end());
    }
    return all_pairs;
  }

  private:

  std::vector<std::unique_ptr<Bucket>> buckets{};
  HashType hash_function{};
  ValueType initial_value;

  Bucket &get_bucket(const KeyType &key) const {
    //Use hash function to determine the index of the bucket belonging to the key
    const auto bucket_index = hash_function(key) % buckets.size();
    return *buckets[bucket_index];
  }
};
```

Task 3: Tokenize text corpora in parallel

The web is full of unstructured human-readable text. Language models such as ChatGPT can only work on a list of words. While it seems straightforward to split a sentence represented as a string into a list of words (also strings) by just dividing them at the whitespace, the reality is much more complicated due to many exceptions. This process is called tokenization. We split an input sentence into a list of tokens (words). Your task is to tokenize multiple text corpora and count the occurrences of each token.

- Download the 20 Newsgroups data set[4].
- The data from each newsgroup is in a separate directory. Each directory contains text files that contain an entry in the newsgroup.
- Process the data from each newsgroup in parallel in a separate `std::thread`.
- In each thread: Tokenize each file from the newsgroup and count the occurrences of each token with your implementation of `ThreadSafeCounterDict`.
- Run the program in parallel.
- Print the 50 most used tokens in the entire dataset to the console

[4] `http://qwone.com/~jason/20Newsgroups/20news-18828.tar.gz`

- Implement and run the program in serial. You can use a `std::unordered_map` as a counter dictionary.
- Measure and compare the run time of the serial and parallel program
- Check the correctness of your program by comparing the results to the serial implementation
- Explain how you could further improve the performance of your program.

Take a look at `std::filesystem` for the browsing of directories, `std::ifstream` for the reading of a file, and `std::get_line` for the tokenization (separating by white spaces is sufficient). You can use the methods below to remove leading and trailing whitespaces from your token and convert them to lower case:

```
static inline void to_lower_case(std::string & s) {
  std::transform(s.begin(), s.end(), s.begin(),
  [](unsigned char c) { return std::tolower(c); });
}

static inline void ltrim(std::string &s) {
  s.erase(s.begin(), std::find_if(s.begin(), s.end(), [](unsigned char ch) {
    return !std::isspace(ch) && ch != '\n';
  }));
}

static inline void rtrim(std::string &s) {
  s.erase(std::find_if(s.rbegin(), s.rend(), [](unsigned char ch) {
    return !std::isspace(ch) && ch != '\n';
  }).base(), s.end());
}

static inline void trim(std::string &s) {
  rtrim(s);
  ltrim(s);
}

```

Solution:

When a lot of threads write for the same token, they access the same bucket and, therefore, need to wait. Collecting the counts in a simple `std::unordered_map` in each thread and combining the results of all threads at the end could improve the performance further, but that depends on the word-count distribution. Usually, text corpora have words that repeat very often (e.g., 'the', 'and'). Hence, our implementation could benefit from this change.

See the implementation below:

```
#include <cassert>
#include <filesystem>
#include <fstream>
#include <iostream>
#include <shared_mutex>
#include <thread>
#include <tuple>
#include <unordered_map>
#include <vector>

#include "ThreadSafeCounterDict.h"
#include "Utils.h"

/**
* Function that is executed by each thread in the parallel implementation of the token counter. Counts the tokens for a single newsgroup
* @param counter_dict Reference to the thread safe counter dict that is shared by every thread
* @param directory The root directory of the newsgroup
*/
void worker_func(ThreadSafeCounterDict<std::string, unsigned long> &counter_dict, const std::filesystem::path &directory) {
  //Iterate over each file from a single newsgroup
  for (const auto &entry: std::filesystem::directory_iterator(directory)) {
```

```
    if (entry.is_regular_file()) {
      std::ifstream file(entry.path());
      if (file.is_open()) {
        //Iterate over each token in the file (separated by whitespaces)
        std::string token;
        while (std::getline(file, token, ' ')) {
          //Trim newline and whitespace characters from the beginning and end of the token
          trim(token);
          //Convert the token to lower case
          to_lower_case(token);
          if (token.empty()) {
            //Ignore empty tokens
            continue;
          }
          //Count the token by increasing its value by 1
          counter_dict.increase_value_for_key(token, 1);
        }
        file.close();
      }
    }
  }
  std::cout << "A thread finished\n";
}

/**
* Parallel implementation of the token counter
* @param root_directory The root directory of the dataset
* @return List of token-count pairs
*/
std::vector<std::pair<std::string, unsigned long>> parallel(const std::filesystem::path &root_directory) {
  ThreadSafeCounterDict<std::string, unsigned long> counter_dict{0, 661};

  std::vector<std::thread> threads;

  //Start a thread for every newsgroup
  for (const auto &entry: std::filesystem::directory_iterator(root_directory)) {
    if (entry.is_directory()) {
      //Start thread with the worker function, the counter_dict reference and the newsgroup directory
      std::thread thread(worker_func, std::ref(counter_dict), entry);
      //Save thread object in list
      threads.push_back(std::move(thread));
    }
  }

  std::cout << "Started " << threads.size() << " threads\n";

  //Wait for all threads to finish
  for (auto &thread: threads) {
    thread.join();
  }

  //Return vector of token-count pairs from the counter dictionary
  return counter_dict.convert_to_pairs();
}

/**
* Serial implementation of the token counter
* @param root_directory The root directory of the dataset
* @return List of token-count pairs
*/
std::vector<std::pair<std::string, unsigned long>> serial(const std::filesystem::path &root_directory) {
  //Use an unordered_map for counting as it does not to be thread safe for the serial implementation
  std::unordered_map<std::string, unsigned long> counter_dict{};

  //Iterate over each file from a single newsgroup
  for (const auto &directory: std::filesystem::directory_iterator(root_directory)) {
    if (directory.is_directory()) {
      for (const auto &file: std::filesystem::directory_iterator(directory)) {
        if (file.is_regular_file()) {
          //Read file
```

```
          std::ifstream myfile(file.path());
          if (myfile.is_open()) {
            //Iterate over each token in the file (separated by whitespaces)
            std::string token;
            while (std::getline(myfile, token, ' ')) {
              //Trim newline and whitespace characters from the beginning and end of the token
              trim(token);
              //Convert the token to lower case
              to_lower_case(token);
              if (token.empty()) {
                //Ignore empty tokens
                continue;
              }
              //Count the token by increasing its value by 1
              counter_dict[token] = counter_dict.contains(token) ? counter_dict[token] + 1 : 1;
            }
            myfile.close();
          }
        }
      }
    }
  }

  //Return vector of token-count pairs from the counter dictionary
  return {counter_dict.begin(), counter_dict.end()};

}

int main() {
  std::filesystem::path root_directory = "20news-18828";

  auto comp = [](const auto &p1, const auto &p2) {
    if (p1.second == p2.second) return p1.first > p2.first;
    return p1.second > p2.second;
  };

  //Run the serial program
  auto begin = std::chrono::steady_clock::now();
  auto all_pairs_serial = serial(root_directory);
  auto end = std::chrono::steady_clock::now();
  std::sort(all_pairs_serial.begin(), all_pairs_serial.end(),
  comp);
  std::cout << "Elapsed time serial = " << std::chrono::duration_cast<std::chrono::seconds>(end - begin).
    count()
  << "[s]" << std::endl;

  //Run the parallel program
  begin = std::chrono::steady_clock::now();
  auto all_pairs_parallel = parallel(root_directory);
  end = std::chrono::steady_clock::now();
  std::sort(all_pairs_parallel.begin(), all_pairs_parallel.end(),
  comp);
  std::cout << "Elapsed time parallel = " << std::chrono::duration_cast<std::chrono::seconds>(end - begin)
    .count()
  << "[s]" << std::endl;

  assert(all_pairs_parallel == all_pairs_serial);

  //Print the 50 most used tokens
  for (auto i = 0; i < std::min(50ul, all_pairs_parallel.size()); i++) {
    const auto &[token, count] = all_pairs_parallel[i];
    const auto &[token_s, count_s] = all_pairs_serial[i];

    std::cout << token << "(" << token_s << ")" << ": " << count << "(" << count_s << ")" << "\n";
    assert(all_pairs_parallel[i] == all_pairs_serial[i]);
  }

  return 0;
}
```