Exercise 1: Shared Memory

Task 1. Limits of Scalability

Task 1.1. Application of Amdahl's Law
Assume a parallel program with a serial runtime $t = 100s$ and a parallel fraction $p = 0.95$. What is the minimum runtime and maximum speedup that can be achieved with this program?

Task 1.2. Limit Value Consideration of Amdahl's Law
Perform a limit value consideration of S_p for $N \to \infty$, with N being the number of processors used. What does the result imply regarding the efficiency E_p?

Task 1.3. Limitations of Amdahl's Law
Which problems of real-world applications do also limit the achievable speedup, but are not taken into account by Amdahl's Law?

Task 2. C++ Threading
In order to work as intended, the following code is missing several Threading-related API calls.

helloworld.cpp

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


void hello()
{
    std::cout << "Hello " << std::endl;
}

void world()
{
    std::cout << "World. " << std::endl;
}

int main()
{
    // TODO: call with two new threads
    hello();
    world();

    // TODO: join the threads with the main thread


    return 0;
}
```

Task 2.1. Completion of a Program Skeleton
Complete the code shown above. Execute it with two threads and provide the program's output.

Task 3. Deadlocks and Races

Task 3.1. Deadlock with naive lock implementation
For the naive lock implementation (code on slide 15 from the third lecture), construct a scenario that a deadlock may occur.

Task 3.2. Another naive lock implementation
The following code also aims to implement a lock by giving the other thread preference. Again, the implementation is restricted to two threads only and only the case of one or two thread shall be considered in this task.
The function `make_logical_threadid()` maps the systems's thread to an integer in the range $0...n-1$ for a number of threads n.

locktwo.cpp

```
int victim;   /* global variable */

int make_logical_threadid(std::thread::id i);

void lock()
{
   int i = make_logical_threadid(std::this_thread::get_id());
   victim = i;
   while (victim == i) {}
}

void unlock()
{
}
```

Does this approach implement mutual exclusion? Justify your answer.

Task 3.3. Another naive lock implementation cont'd
Under which condition does this implementation stop making progress? And, why did we not call this a deadlock?

Task 3.4. Existence of a Data Race
The following code shows routines which all access the array `A`.
The function `make_logical_threadid()` maps the systems's thread to an integer in the range $0...n-1$ for a number of threads n.

access.cpp

```
int A[100];   /* global variable */

int make_logical_threadid(std::thread::id i);

void access_one()
{
   int i = make_logical_threadid(std::this_thread::get_id());
   A[i] = rand();
}

void access_two()
{
   int i = make_logical_threadid(std::this_thread::get_id()) % 10;
   A[i] = rand();
}

void access_three()
{
   int i = make_logical_threadid(std::this_thread::get_id());
   A[rand() % 100] = i;
}
```

For each of these routines: can a data race occur if they are executed concurrently by multiple threads? Justify your answer.

Task 4. Completion of the Queue Type
The second lecture contained the declaration of the `threadsafe_queue` type on slide 53 and parts of the implementation on slide 54, as shown below.

tsqueue.cpp

```
template<typenameT>
class threadsafe_queue
{
private:
   std::queue<T> data;
   std::mutex mut;
   std::condition_variable cond;

public:
   threadsafe_queue() {}
   threadsafe_queue(threadsafe_queue const& other)
   {
      std::lock_guard<std::mutex> lk(other.mut);
      data = other.data;
   }


   void push(T new_val)
   {
```

```
        std::lock_guard<std::mutex> lk(mut);
        data.push(new_value);
        cond.notify_one();
    }
    void wait_and_pop(T&value)
    {
        std::unique_lock<std::mutex> lk(mut);
        cond.wait(lk, [this]{return!data.empty() ;} );
        value = data.front();
        data.pop();
    }
};
```

Task 4.1. Implementation of the `empty()` member function
The member function `empty()` is implemented as follows:

tsqueue-empty.cpp

```
bool threadsafe_queue::empty() const
{
    std::lock_guard<std::mutex> lk(mut);
    return data_queue.empty();
}
```

Why is it required to acquire the lock in this member function?

Task 4.2. Implementation of the `try_pop()` member function
Implement the `try_pop()` member function declared as follows. It should return `false` if the queue is empty, or return `true` and provide the front element in the parameter `argument` otherwise.

tsqueue-trypop.cpp

```
bool threadsafe_queue::try_pop(T& value)
{
    // TODO //
}
```

Discussion of this exercise on May 6th, 2019.