Concepts and Models for Parallel and Data-centric Programming, Summer 2021

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?
Assumed that there is N processors minimum runtime = (0.05 + 0.95/N)*100s and speedup = 1/(0.05 + 0.95/N)

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?
as long as # N increases infinitely, the efficiency is getting close to 0 -> worse than not to do

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?
that the increasing # processor cannot always enhance the efficiency T(N) also increases such that the time from the parallelized process can enormously slower than the T(1)

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
Two threads (with integer ids 0 and 1) want to access a resource in a critical region using a `lock(int tid)` and an `unlock(int tid)` routine. Thereby, thread 0 gets in a critical region twice, whereby thread 1 needs access only once:

```
1  int main() {
2      // first thread gets integer id 0
3      int id = 0;
4      std::thread t0([id]
5      {
6          lock(id);
7          std::cout << "Thread 0 is in first critical region" << std::endl;
8          unlock(id);
9          lock(id);
10         std::cout << "Thread 0 is in second critical region" << std::endl;
11         unlock(id);
12     });
13     // second thread gets integer id 1
14     id = 1;
15     std::thread t1([id]
16     {
17         lock(id);
18         std::cout << "Thread 1 is in critical region" << std::endl;
19         unlock(id);
20     });
21
22     t0.join(); t1.join();
23
24     return 0;
25 }
```

In the following two tasks, two different implementations of the `lock()` and `unlock()` functions are proposed.

Task 3.1. Deadlock with Naive Lock Implementation
Consider the following lock implementation:

lockone.cpp

```
1  bool flag[2] = {false, false}; // global array
2
3  void lock(int i) {
4      int j = (i + 1) % 2; // calculates the other thread id
5      flag[i] = true;
6      while (flag[j]) {};          if the value of flag[j] is changed as true simultaneously
7  }
8
9  void unlock(int i) {
10     flag[i] = false;
11 }
```

For this lock implementation, construct a scenario that a deadlock may occur. You can write down a table with two columns ("Thread 0" and "Thread 1") and depict the interleaving of the executed code lines that leads to a deadlock. You also might want to check your expectations by putting in some delays in the code like:

```
std::this_thread::sleep_for(std::chrono::milliseconds(x)));
```

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 (see code above) only and only the case of one or two thread shall be considered in this task.

locktwo.cpp

```
1  int victim; // global variable
2
3  void lock(int i) {
4      victim = i;
5      while (victim == i) {}
6  }
7
8  void unlock(int i) {
9  }
```

3.2no, no clear restriction for using a shared resource
3.3 if the value of victim changes
the two thread can use the resource at the same time
there is no restriction to possess the resource
there is no waiting process
the resource can be easily obeyed

Does this approach implement mutual exclusion? Justify your answer.

Task 3.3. Another naive lock implementation cont'd
Under which condition does the implementation "locktwo.cpp" 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 parameter `tid` in the access functions represents the thread identifier in the range $0, ..., n-1$ for a number of threads n. The first thread gets the value 0, the second threads gets the value 1, and so on.

access.cpp

```
1  int A[100];   /* global variable */
2
3  void access_one(int tid)
4  {
5     A[tid] = rand();
6  }
7
8  void access_two(int tid)
9  {
10    A[tid % 10] = rand();
11 }
12
13 void access_three(int tid)
14 {
15    A[rand() % 100] = tid;
16 }
```

more than 2 threads access a shared resource concurrently
and there is no limit or control these

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()
{
   std::lock_guard<std::mutex> lk(mut);
   return data.empty();
}
```

it checks if the queue is empty
in case that some modifications occurs
to avoid chaos

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

```
1 bool threadsafe_queue::try_pop(T& value)
2 {
3    // TODO //
4 }
```

Discussion of this exercise on May 06th, 2021.