Exercise 1: Solutions

Lecture, Summer 2019

Dr. Christian Terboven

Simon Schwitanski

Task 1

Task 1.1

- **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?**

- **Serial fraction: s = 1 – p = 0.05**
  → Fraction which cannot be parallelized
- **Minimum runtime: 5s**
  → Parallelized program's execution time: $\mathbf{T(N)} = (\boldsymbol{s} + \frac{\boldsymbol{p}}{\boldsymbol{N}}) \cdot \boldsymbol{T}(\mathbf{1})$
- **Maximum speedup: 100s / 5s = 20**

- **Speedup according to Amdahl's Law:**

$$\boldsymbol{S}_{0.95}(\boldsymbol{N}) = \frac{\boldsymbol{T}(\mathbf{1})}{\boldsymbol{T}(\boldsymbol{N})} = \frac{\mathbf{1}}{\boldsymbol{s} + \frac{\mathbf{1}-\boldsymbol{s}}{\boldsymbol{N}}} = \frac{\mathbf{1}}{\mathbf{0.05} + \frac{\mathbf{1}-\mathbf{0.05}}{\mathbf{N}}}$$

Task 1.2

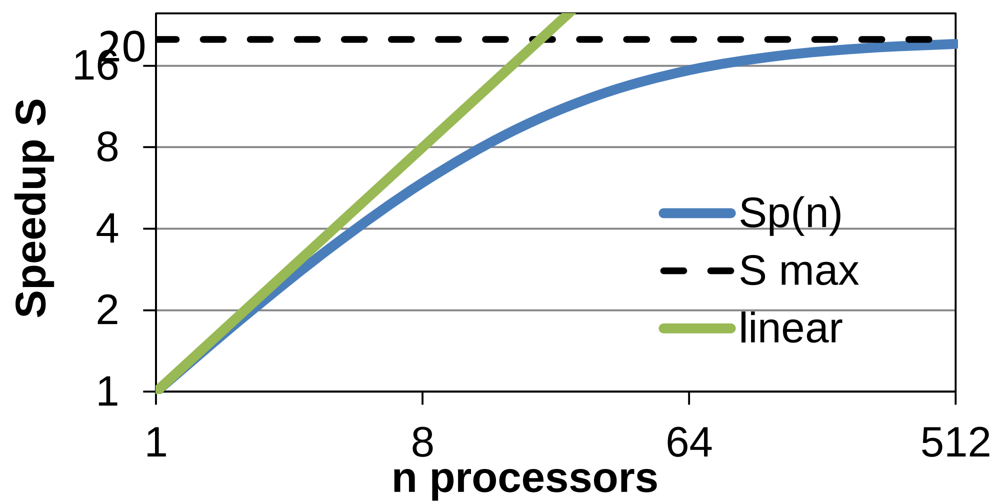
- **Perform a limit value consideration of Sp for N → ∞, with N being the number of processors used. What does the result imply regarding the efficiency Ep?**

$$S_p(N) = \frac{T(1)}{T(N)} = \frac{1}{\mathrm{s} + \frac{1-\mathrm{s}}{\mathrm{N}}}$$

$$S_{max} = \lim_{N\to\infty} S_p(N) = \lim_{N\to\infty} \frac{1}{s + \frac{1-s}{N}} = \frac{1}{s}$$

$$\lim_{N\to\infty} \varepsilon_p(N) = \lim_{N\to\infty} \frac{1}{s(N-1)+1} = 0$$

Example:
s=0.05



Task 1.3

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

- **Selection of possible answers:**
  - → Overhead of parallelization
    - → Creation of threads, Communication of information, ...
  - → Contention on shared resources
    - → Overhead of locks, ...
  - → Additional work with increasing number of processing elements
    - → Increasing effort of problem partitioning / load balancing, ...

Task 2

Task 2.1 / 1

Completion of the program skeleton

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

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

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

Task 2.1 / 2

```
int main()
{
    // call with two new threads
    std::thread t1( hello );
    std::thread t2( world );

    // join the threads with the main thread
    t1.join();
    t2.join();

    return 0;
}
```

Task 3

Task 3.1

- **Construct a scenario in which the naive lock impl. deadlocks**

```
bool flag[2];   // init. w/ false
void lock() {
    flag[i] = true;
    while (flag[j]) {}
}
```

Schedule:

Thread 0	Thread 1
`flag[0] = true`	
	`flag[1] = true`
	`while (flag[0]) {}`
`while (flag[1]) {}`	

→ Thread 0 first sets `flag[0]` to true and might be interrupted or just slower than Thread 1

→ Thread 1 sets `flag[1]` to `true` (faster than Thread 0) and observes `flag[0] == true`, so it has to wait

→ When thread 0 continues, it observes `flag[1] == true`, so it has to wait

→ Both threads wait for each other: Deadlock

Task 3.2

- **Does the other naive lock impl. provide mutual exclusion?**
  → In case of 2 threads: Yes. The variable **`victim`** is either 0* or 1* and it can never be the case that both threads pass the while condition.

  → In case of 3 or more threads: No. Only one thread can be the **`victim`**, therefore 2 or more threads can simultaneously enter the critical region.

  → ***But:** We have a data race on the variable **`victim`**. According to the C++ standard, this is *undefined behavior*. That means: *Anything* can happen during runtime (e.g., **`victim`** could be set to a random value or the program could crash). Thus, according to the C++ standard, the implementation obviously does not implement mutual exclusion.

    → Solution to avoid undefined behavior: Make accesses on **`victim`** atomic.

Task 3.3

- **Under which condition does this implementation stop making progress?**
  → Single-threaded execution / the other thread terminated

- **Why is this condition not a deadlock?**
  → A deadlock requires at least two processes / threads waiting for each other.
  Here, we have only one thread waiting for an event.

Task 3.4

- **Data Race in `access_one()`?**

  → No.

- **Data Race in `access_two()`?**

  → Yes, two threads potentially write to same `A[i]` simultaneously (at least 11 threads needed here)

- **Data Race in `access_three()`?**

  → Yes, two threads potentially write to same `A[i]` simultaneously (at least 2 threads needed here).

Task 4

Task 4.1

- **Why does the `empty()` function have to acquire a lock?**
  - → If not, the access to member variable **`data_queue`** is not safe.
  - → Problem: The implementation of **`std::queue::empty()`** is <u>not</u> guaranteed to be thread-safe (although it might be in some C++ STL implementations).

Task 4.1 – Additions

- **Note: The `conditional_variable::wait()` function cannot be used with a `lock_guard`, but instead `unique_lock` has to be used. The reason is that the `wait()` function has to `unlock()` and `lock()` a given mutex which is not supported by `lock_guard`.**

Task 4.2

- **Implementation of the `try_pop()` member function**

```
bool threadsafe_queue::try_pop(T& value)
{
    std::lock_guard<std::mutex> lk(mut);
    if (data_queue.empty())
        return false;
    value = data_queue.front();
    data_queue.pop();
    return true;
}
```

Task 4.2 – Additions

- **What happens if some thread calls `push()` with `cond.notify_one()` to notify a waiting thread j? Is it possible that another thread k gets the released mutex of thread i *before* thread j which has been woken up?**

  → Yes (see C++ standard on condition variables). But this is not a problem: Even if thread k modifies the queue, thread j can only continue execution after it has acquired the mutex. Therefore, it has to wait for thread k to release the lock. As soon as thread j can go on with execution, it first checks the predicate (`!queue.empty()`). If the queue is empty again, then thread j goes again to sleep and waits for a notification. Otherwise, thread j can get the element out of the queue.

  → Thus, the implementation is thread-safe.