MINISTRY FOR EDUCATION AND SCIENCE OF RUSSIA

____

SAINT PETERSBURG ELECTROTECHNICAL UNIVERSITY «LETI»

____

Laboratory work № 2
Computational Systems

Student
Group 4300
Bathaie N.

Teacher
PhD, associate prof.
of CE department
Paznikov A.A.

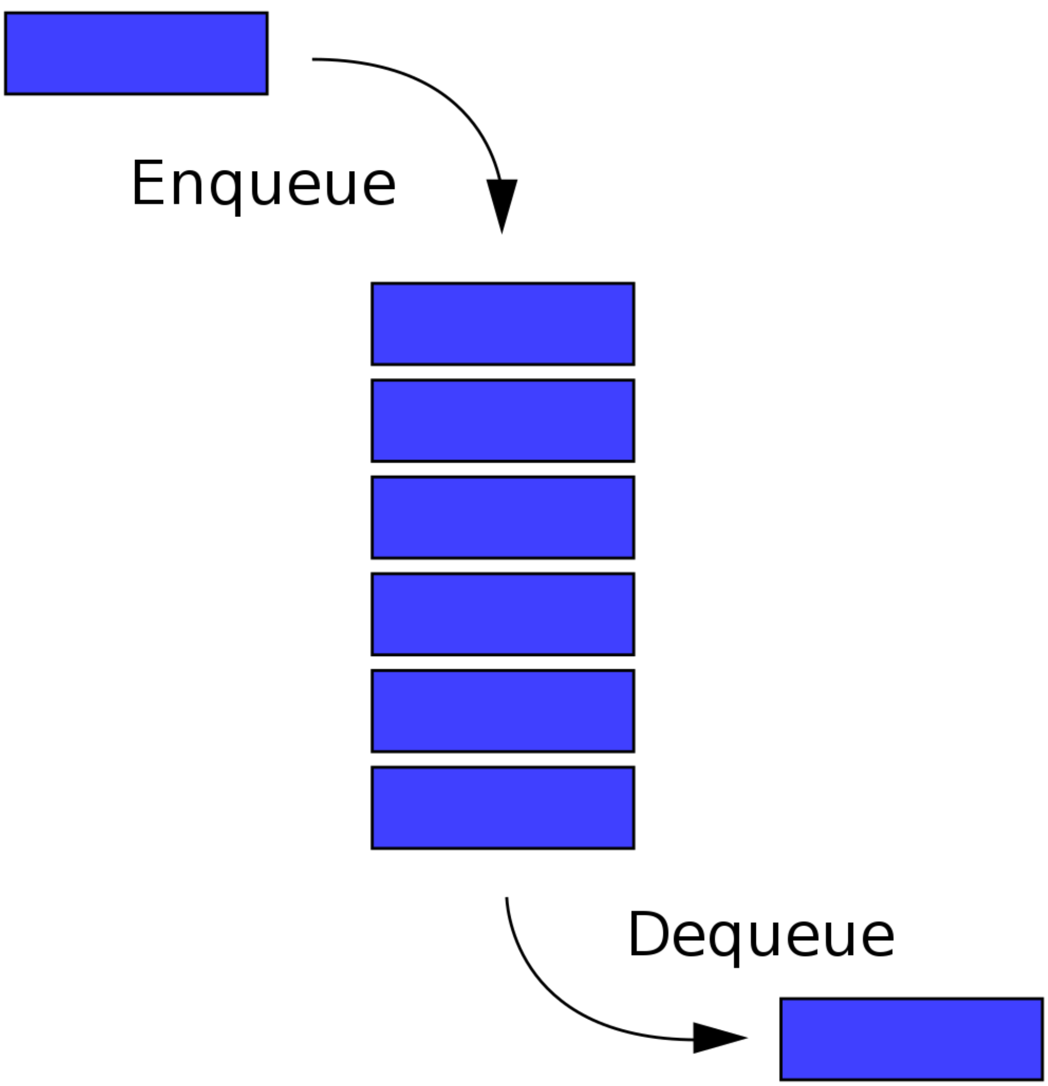
Saint Petersburg
ETU «LETI»
2018

IMPLEMENTATION OF CONCURRENT SINGLE-ENDED QUEUE

1. Introduction to queue

Queue is a data structure designed to operate in FIFO (First in First out) context. In queue elements are inserted from rear end and get removed from front end.

Queue class is container adapter. Container is an objects that hold data of same type. Queue can be created from different sequence containers. Container adapters do not support iterators therefore we cannot use them for data manipulation. However they support push() and pop() member functions for data insertion and deletion respectively.



2. Queue class:

To handle pop() function when the queue is empty a condition variable used.

Through this way the thread who wants to pop but the queue is empty waits until another thread pushes something in the queue.

```
class concurrent_queue
{
public:
    concurrent_queue(unsigned _capacity =
concurrent_queue::def_capacity):
        capacity{_capacity}, items{new TItem[_capacity]}
    {}
    ~concurrent_queue()
    {
        delete[] items;
    }
    TItem pop()
      {
            std::unique_lock<TLock> mlock(lock);
            while (queue_.empty())
            {
                  cond_.wait(mlock);
            }
            auto item = queue_.front();
            queue_.pop();
            return item;
      }
    void push(TItem &item)
      {
            std::unique_lock<TLock> mlock(lock);
            queue_.push(item);
            mlock.unlock();
            cond_.notify_one();
      }
    void print()
      {
            for (auto i = 0u; i < capacity; i++)
                  std::cout << items[i] << " ";
            std::cout << std::endl;
      }
      unsigned get_capacity()
    {
        return capacity;
    }
private:
    static const unsigned def_capacity = nopers * 10;
    unsigned capacity = def_capacity;
    TLock lock;
    TItem *items;
    std::queue<TItem> queue_;
    std::condition_variable cond_;
};
```

3. Code

```
#include <iostream>
#include <random>
#include <algorithm>
#include <thread>
#include <mutex>
#include <chrono>
#include <fstream>
#include <queue>
#include <condition_variable>
const auto nruns = 10;
const auto nopers = 8000000;
class thread_raii
{
public:
    thread_raii(std::thread&& _t): t{std::move(_t)} { }
    thread_raii(thread_raii &&thr_raii): t{std::move(thr_raii.t)} {
}
    ~thread_raii()
    {
        if (t.joinable())
            t.join();
    }
    std::thread& get()
    {
        return t;
    }
    void join()
    {
        if (t.joinable())
            t.join();
    }
private:
    std::thread t;
};
///.................
template<typename TItem, typename TLock>
class concurrent_queue
{
public:
    concurrent_queue(unsigned _capacity =
      concurrent_queue::def_capacity):
        capacity{_capacity}, items{new TItem[_capacity]}{}
    ~concurrent_queue()
    {
        delete[] items;
    }
    TItem pop();
    void push(TItem &item);
    unsigned get_capacity()
    {
        return capacity;
    }
    void print();
private:
```

```
    static const unsigned def_capacity = nopers * 10;
    unsigned capacity = def_capacity;
    TLock lock;
    TItem *items;
    std::queue<TItem> queue_;
    std::condition_variable cond_;
};
///..................
template<typename TItem, typename TLock>
TItem concurrent_queue<TItem, TLock>::pop()
{
    std::unique_lock<TLock> mlock(lock);
    while (queue_.empty())
    {
        cond_.wait(mlock);
    }
    auto item = queue_.front();
    queue_.pop();
    return item;
}
template<typename TItem, typename TLock>
void concurrent_queue<TItem, TLock>::push(TItem &item)
{
    std::unique_lock<TLock> mlock(lock);
    queue_.push(item);
    mlock.unlock();
    cond_.notify_one();
}
template<typename TItem, typename TLock>
void concurrent_queue<TItem, TLock>::print()
{
    for (auto i = 0u; i < capacity; i++)
        std::cout << items[i] << " ";
    std::cout << std::endl;
}
static auto rand_gen()
{
    static const auto maxrand = 100;
    static std::random_device rnd_device;
    static std::mt19937 mersenne_engine{rnd_device()};
    static std::uniform_int_distribution<int> dist{1, maxrand};
    return dist(mersenne_engine);
}
int main(int argc, const char *argv[])
{
    try
    {
        const auto max_threads =
            std::thread::hardware_concurrency();
        std::ofstream speedupfile{"throughput"};
        if (!speedupfile.is_open())
        {
            std::cerr << "can't open file" << std::endl;
            return 1;
        }
        auto get_time = std::chrono::steady_clock::now;
```

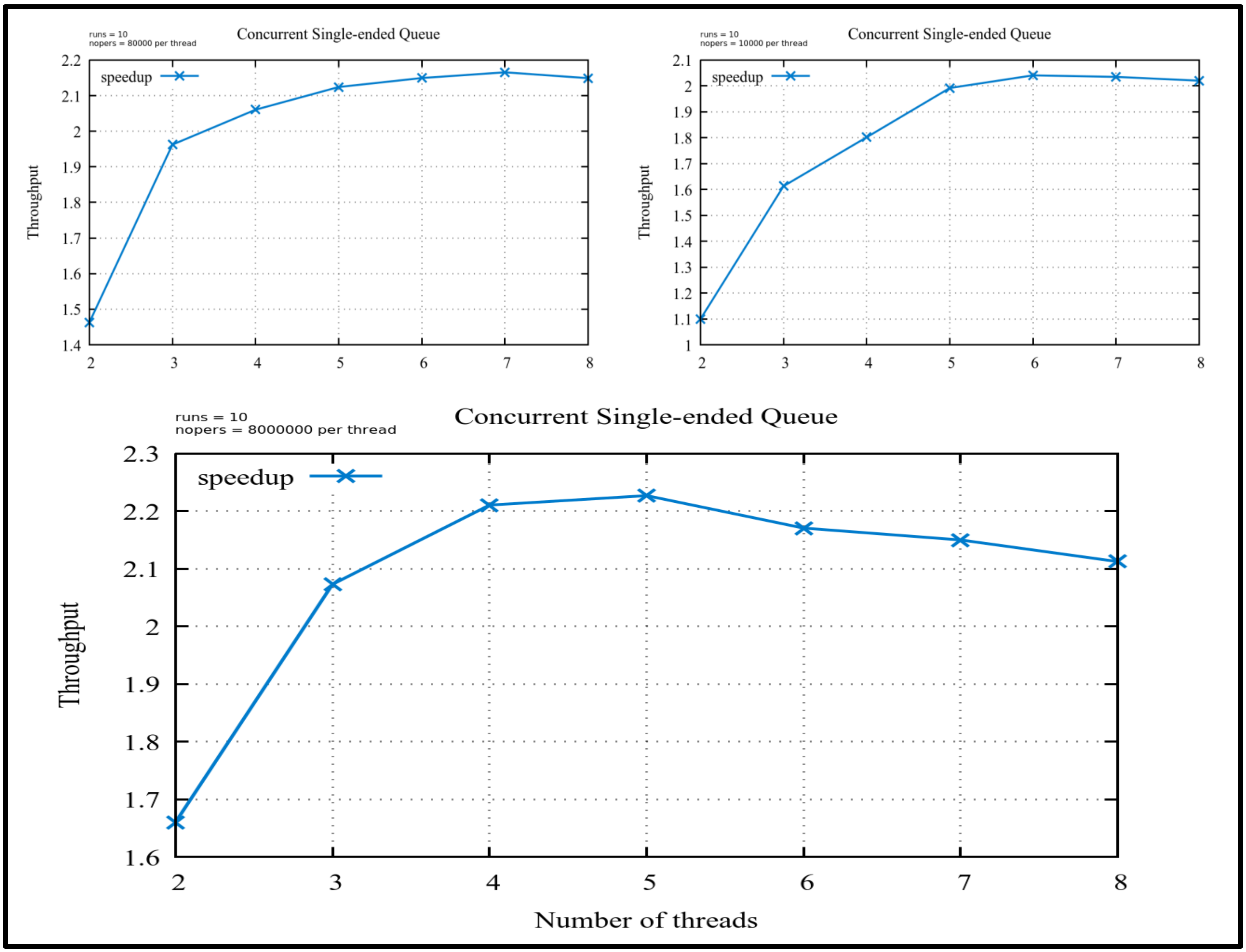
```
        decltype(get_time()) start, end;
        for (auto nthr = 2u; nthr <= max_threads; nthr++)
        {
            start = get_time();
            for (auto i = 0; i < nruns; i++)
            {
                std::vector<thread_raii> threads;
                concurrent_queue<int, std::mutex> cqueue;
                for (auto i = 0u; i < cqueue.get_capacity()/2; i++)
                {
                    auto item = rand_gen();
                    cqueue.push(item);
                }
                auto thread_func = [&cqueue, &nthr]()
                {
                    for (auto i = 0u; i < nopers/nthr; i++)
                    {
                        auto oper = rand_gen();
                        if (oper <= 50)
                        {
                            auto item = rand_gen();
                            cqueue.push(item);
                        }
                        else
                        {
                            cqueue.pop();
                        }
                    }
                };
                for (auto thr_id = 0u; thr_id < nthr; thr_id++)
                    threads.emplace_back(thread_raii
                {
                    std::thread{thread_func}});
                for (auto &thr: threads)
                    thr.join();
                 // cqueue.print();
            }
            end = get_time();
            const auto elapsed = std::chrono::duration_cast
                   <std::chrono::milliseconds>(end - start).count();
            const auto par_time = double(elapsed) / nruns;
            const auto throughput = nopers / (par_time * 1000);
            std::cout << "Threads: " << nthr << " elapsed time: "
                                  << par_time << " ms throughput: "
                                           << throughput <<std::endl;
            speedupfile << nthr << "\t" << throughput << std::endl;
        }
        speedupfile.close();
    }
    catch (std::runtime_error &e)
    {
        std::cerr << "Caught a runtime_error exception: "
                  << e.what () << std::endl;
    }
    return 0;
}
```

4. Conclusion

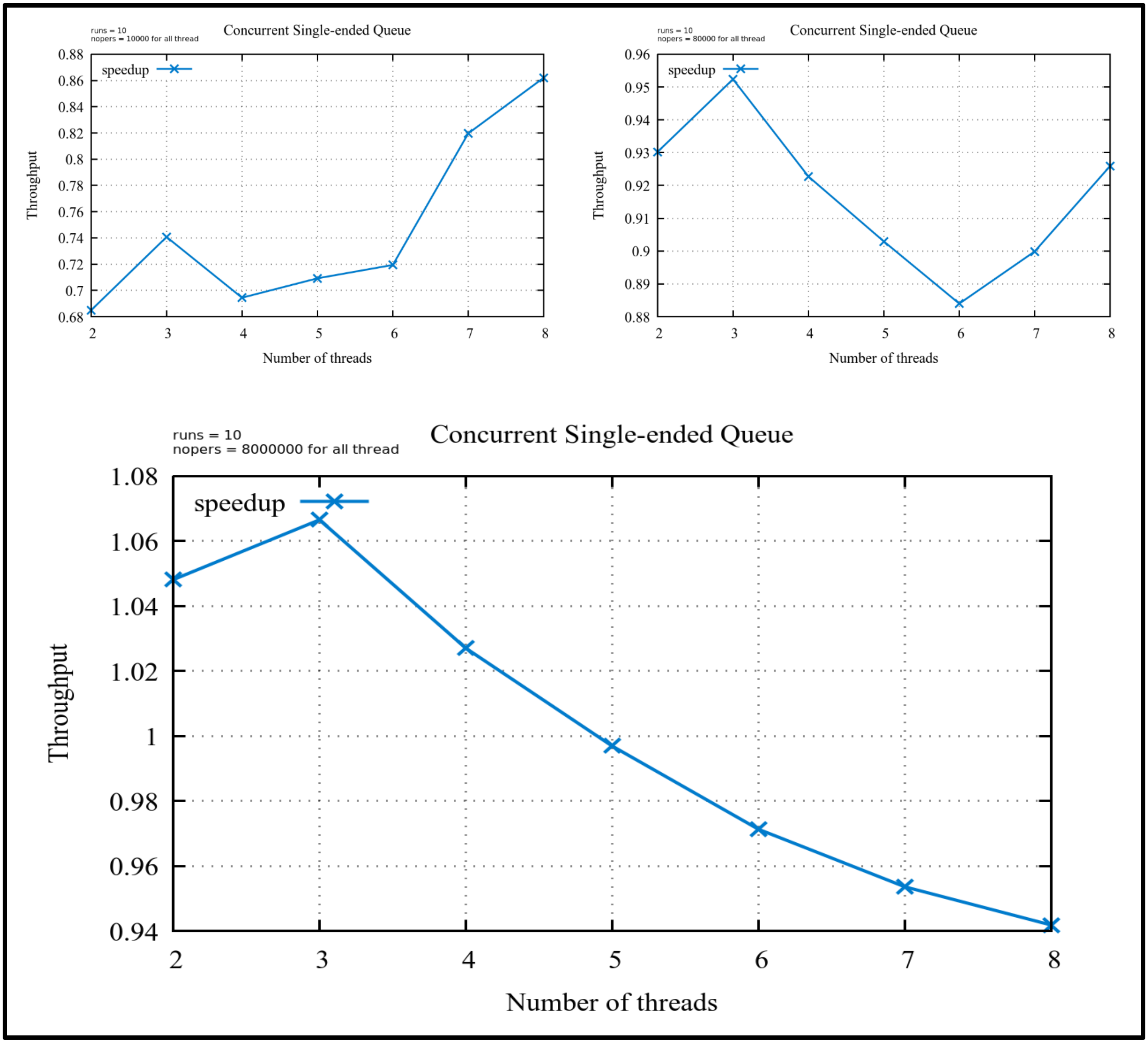
The analyzes of the output of this code carried out in 2 ways.

- First, each thread performs the same number of operation.
- In the second approach the number of operations are fixed but we carry them out using different number of thread.

The output graphs for these two situation are as bellow.



The Outpu when each thread has the same number of operations to carry out



The Output when a certain number of operations is divided between threads

The output of the first examples shows that the throughput increases when we use more threads; however, the output of the second approach shows throughput reduction when we have enough iterations!

By increasing the number of threads the performance also can be increased because each thread can be carried out in one core of the CPU (parallelism); however, this increment has a side effect that the probably of one or more threads being blocked by another one increases as well. This happens when two or more threads want to go to the critical section resulting that except one thread, all the other ones get blocked and have to wait.

Due to this trade off, the final result - when we have enough iterations - is highly depended on the number of operations the threads have to perform inside and outside of the critical section, where less operations in critical section and more operations outside critical section can lead to a higher throughput when more threads are being used.