1. Informations about computer system used:

RAM SIZE: 16 GB (SO-DIMM DDR4, 2666MHz)

CPU: Intel® Core™ i5-8300H

NUMBER OF PHYSICAL CORRES: 4

NUMBER OF LOGICAL CORRES: 8

CLOCK RATE: 2300 - 4000 MHz

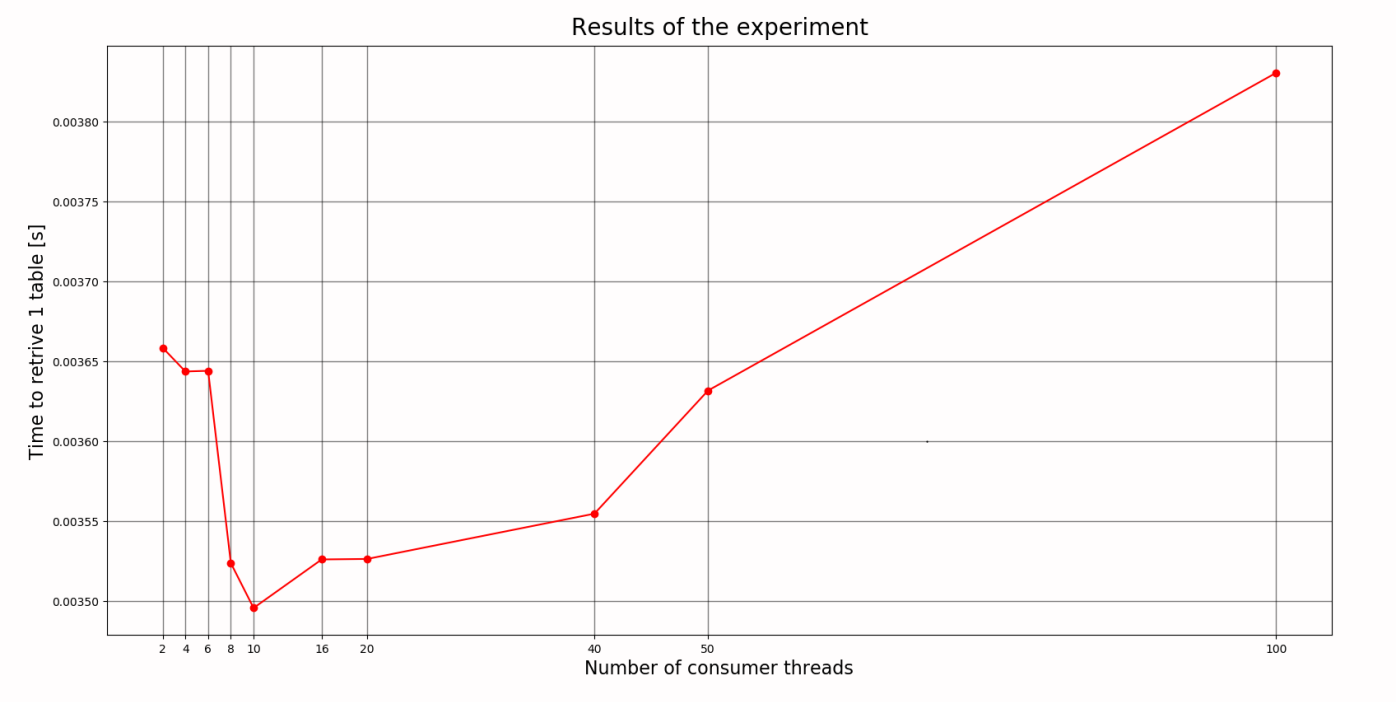
CPU CACHE: 8 MB

2. Parameters

Compiler parameters			Experiment parameters		
Version	**Mode**	**Architecture**	**Number of sorted arrays**	**Array size**	**Queue size**
Default C++14 for Visual Studio Community 2019 16.5.4	Relase	X64	5000	100000	1000

3. Comparison of the speed of retrieving arrays from the queue

Number of consumer threads	Execution time[s]	Time/1 table[s]
2	18.2917	0.00365834
4	18.2174	0.00364348
6	18.2196	000364392
8	17.6176	0.00352352
10	17.4771	0.00349542
16	17.6291	0.00352582
20	17.6306	0.00352612
40	17.7723	0.00355446
50	18.1577	0.00363154
100	19.1523	0.00383046



4. Commentary

- The presented results were obtained by the average of five measurements for given number of threads.
- The best time to sort single table has been achieved for 10 consumer threads.
- Method thread::hardware_concurrency() returns number of 8 logical cores, while the best time was achieved for 10 consumer threads. In connection with this it is beneficial to create number of threads not much larger than the number of logical cores.
- Creating a number of threads much larger than the number of cores has a negative impact on execution time.
- The largest decrease in execution time is noticeable between the number of threads corresponding to the number of physical and logical cores.

5. Example of single output

Number of logical cores: 8

Queue size is : 0

Check sum is: 49.5084

Check sum is: 49.4809

Check sum is: 49.5936

Check sum is: 49.5468

Check sum is: 49.4376

…

…

…

Producer has produced: 5000

Producer was destructed

Check sum is: 49.3699

Thread 13088 sorted: 455 tables

Consumer was destructed

Thread 8292 sorted: 508 tables

Consumer was destructed

Thread 22100 sorted: 523 tables

Consumer was destructed

Thread 21272 sorted: 470 tables

Consumer was destructed

Thread 12668 sorted: 484 tables

Consumer was destructed

Thread 15932 sorted: 497 tables

Consumer was destructed

Thread 22056 sorted: 496 tables

Consumer was destructed

Thread 4560 sorted: 492 tables

Consumer was destructed

Thread 4620 sorted: 529 tables

Consumer was destructed

Thread 8108 sorted: 546 tables

Consumer was destructed

Consumer was destructed

Producer sorted: 0

Producer was destructed

Queue size is : 0

Queue was destructed

Execution time: 17.7116s

6. **Source code**
   Output of source code was presented above

```
#include <iostream>
#include <thread>
#include <vector>
#include <queue>
#include <array>
#include <algorithm>
#include <string>
#include <mutex>
#include <windows.h>
#include <atomic>
#include <chrono>


#define ARRSIZE 100000
std::mutex mtx;
static std::atomic<int> atomic_global_variable;

template< typename T >

class Queue : public std::queue< T >
{
public:
    Queue(int size) : length(size)
    {
        std::cout << "Queue size is : " << std::queue< T >::size() << std::endl;
    }
    ~Queue()
    {
        std::cout << "Queue size is : " << std::queue< T >::size()<< std::endl;
        std::cout << "Queue was destructed" << std::endl;
    }

    void push(T &array )
    {
        if (current_length == length)
        {
            std::cout << "Queue is full!" << std::endl;
        }
        else
        {
        std::queue< T >::push(array);
        current_length++;
        }
    }

    void pop()
    {
        if (!std::queue< T >::empty())
        {
            std::queue< T >::pop();
            --current_length;
        }
    }

    const int length;
    int current_length = 0;
};

class Producer
{
public:
    Producer(int n, Queue<std::array<int, ARRSIZE>> &head) : num_of_arrays(n), queue(head) {}
    ~Producer()
    {
```

```
        std::cout << "Producer has produced: " <<push << std::endl;
        std::cout << "Producer was destructed" << std::endl;
    }
    void MakeArrays()
    {
        atomic_global_variable = num_of_arrays;

        while (push < num_of_arrays)
        {
            if (queue.current_length == queue.length)
            {
                std::this_thread::yield();
            }
            else
            {
                for (int i = 0; i < arrsize; i++)
                {
                    arr[i] = rand() % 100;
                }
                std::unique_lock<std::mutex> ul(mtx);
                queue.push(arr);
                push++;
            }
        }
    }

private:
    Queue<std::array<int, ARRSIZE>>& queue;
    std::array<int, ARRSIZE> arr;
    const int num_of_arrays = 0;
    const int arrsize = ARRSIZE;
    int push = 0;
};

class Consumer
{
public:
    Consumer(Queue<std::array<int, ARRSIZE>> &head) : queue(head) {}
    ~Consumer()
    {
        std::cout << "Consumer was destructed" << std::endl;
    }
    void FetchArrays()
    {
        while (!state)
        {
            if (atomic_global_variable == 0)
            {
                state = true;
                std::lock_guard<std::mutex> lock(mtx);
                break;
            }

            std::lock_guard<std::mutex> ul(mtx);
            if (!queue.empty())
            {
                //std::cout << "thread working" << std::endl;
                sum = 0;
                checksum = 0;
                std::array<int, ARRSIZE> arr;
                arr = queue.front();
                std::sort(arr.begin(), arr.end());
                for (int i = 0; i < arrsize; i++)
                {
                    sum += arr[i];
                }
                checksum = sum / arrsize;
                std::cout << "Check sum is: " << checksum << std::endl;
                tables_sorted++;
```

```
                atomic_global_variable--;
                queue.pop();
            }
            else
            {
                std::this_thread::yield();
            }
        }
        std::lock_guard<std::mutex> lock(mtx);
        std::cout << "Thread "<< std::this_thread::get_id() <<" sorted: "<< tables_sorted <<"
tables"<< std::endl;
    }

private:
    Queue<std::array<int, ARRSIZE>> &queue;
    const int arrsize = ARRSIZE;
    int tables_sorted = 0;
    float checksum=0;
    float sum=0;
    bool state = false;
};

void SeveralThreads(int thread_vector_number,Consumer &cons)
{
    std::vector<std::thread> thread_vector;
    thread_vector.reserve(thread_vector_number);
    for (int i = 0; i < thread_vector_number; i++)
    {
        thread_vector.push_back(std::thread(&Consumer::FetchArrays,cons));
    }
    for (auto &entry : thread_vector)
    {
        entry.join();
    }
 }

struct Timer
{
    std::chrono::time_point<std::chrono::steady_clock>start, end;
    std::chrono::duration<float>duration;

    Timer()
    {
        start = std::chrono::high_resolution_clock::now();
    }
    ~Timer()
    {
        end = std::chrono::high_resolution_clock::now();
        duration = end - start;
        std::cout <<"Execution time: "<< duration.count() <<'s'<< std::endl;
    }
};

int main(void)
{
    Timer time;
    std::cout << "Number of logical cores: " << std::thread::hardware_concurrency() << std::endl;
    Queue<std::array<int, ARRSIZE>> que(1000);
    Producer obj(5000, que);
    Consumer obj1(que);
    std::thread producer_thread(&Producer::MakeArrays, obj);
    SeveralThreads(10, obj1);
    producer_thread.join();

    return 0;
}
```