Parallel Computing, 2023S

Assignment #1

ANALYZING A SET OF RANDOM NUMBERS

Given is a file (called "input.txt") of 2000000 (2 million) numbers using

```
std::uniform_int_distribution dist(0, 4000);
```

The serial code analyzes this set and displays statistics, such as the total number of prime and non-prime numbers in the set, the mean, and counts the numbers that end with different digits (0-9).

The output look as follows:

```
>./a1-serial --data input.txt
Primes: 276345
Nonprimes: 1723655
Mean: 2000.69
1: 199533
2: 199616
3: 199883
4: 200898
5: 199597
6: 200286
7: 199836
8: 200197
9: 200228
10: 199926
Elapsed time (1 threads): 10.18
```

```
std::vector<int> numbers_counts(10, 0);
```

Vector of 10 elements initialized to 0

SEQUENTIAL CODE: HOW IT WORKS

Data source

A file for the purpose of this exercise:
*Each line represents a number to be analyzed

```
while (!ifilestream.eof())
{
  int num;
  ifilestream >> num;
  ...
  // process
}
```

Reading

```
bool kernel(int number) {…}
```

*Returns true if the given number is a prime number

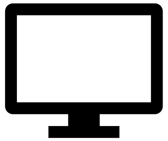
```
if (kernel(num))
  primes++;
else
  nonprimes++;

number_counts[num % 10]++;

count++;
sum += num;
```

Needed to calculate the mean

Processing



output

Writing to the output

GOAL: IMPLEMENTING AN EFFICIENT PARALLEL VERSION

Your task is to implement a parallel version of the provided serial version of the code using C++ multithreading in the following way:

1. **One “Master/Producer” thread**
   - Parses input data (integers) from the given file („`input.txt`“), and puts them in a queue (`std::queue`)
   - Returns the total number of numbers pushed into the queue (keeps count of produced items)

2. **And (one or more) “worker” threads, each doing the following:**
   - Take the next element from the queue and increment the `consumed_count` counter
   - Pass each number to the `kernel()` function to check if the number is a prime number
     - If the number is prime increment the prime counter, otherwise increment the nonprime counter
   - Based on the last digit increment the value in th the `number_counts` array
     - e.g., the number 42 ends with “2”, therefore `number_counts[2]++` increment will be performed
   - Repeat until there are no more items in the queue and no producer thread is adding more items

3. **The rest of the code calculates the mean and prints the results**

→ The goal is also to make master/producer and worker threads work simultaneously

→ The queue is simultaneously accessed by all threads, and these accesses should be protected using condition variables

ADDITIONAL REQUIREMENTS

A template for the parallel version is given as a starting point (a1-parallel.cpp). That code does not do anything useful at the moment, but it gives you a structure where you should insert your code and extend it to fulfill the requirements. In summary, you need to:

- Spawn the producer thread so it returns the number of items pushed to the queue (`produced_count`) - an integer value using `std::futures` and `std::async`.

- Extend the SafeQ class to support safe accesses by multiple threads

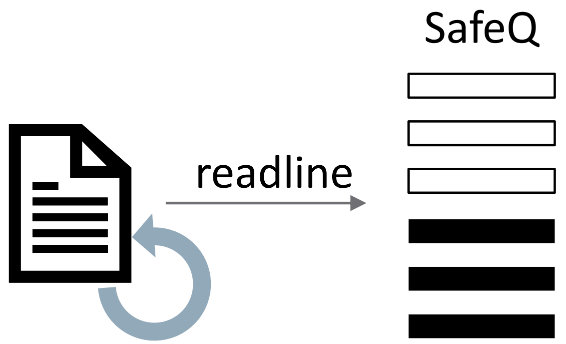
- Spawn worker threads using standard multithreading mechanisms in C++ (std::thread)
  - Make sure that access to shared data is synchronized (make two versions – one with the `worker()` function using only atomics and one where the `worker()` is using mutexes)
  - It's also ok to produce a third "best-effort" version that does things better, perhaps avoids some synchronizations, etc…

HOW IT NEEDS TO WORK

Data source

A file for the purpose of this exercise:

- Each line represents a number to be analyzed




```
while (!ifilestream.eof())
{
  int num;
  ifilestream >> num;
  q.push(num);
  // processing by workers
  produced_count++; // count
}
```

Reading

Worker Thread

```
if (kernel(num))
  primes++;
else
  nonprimes++;

number_counts[num % 10]++;

count++;
sum += num;
```

...

Worker Thread

```
if (kernel(num))
  primes++;
else
  nonprimes++;

number_counts[num % 10]++;

count++;
sum += num;
```

Processing

```
Primes: 276345
Nonprimes: 1723655
Mean: 2000.69
1: 199533
2: 199616
3: 199883
4: 200898
5: 199597
6: 200286
7: 199836
8: 200197
9: 200228
10: 199926
Elapsed time (1 threads): 10.18
```

Writing to the output

Reading and processing need to happen simultaneously!

*Note: If the number of “produced” numbers (pushed to the queue) is not the same as the number of “consumed” (popped from the queue), the program will print an error message

A SIMPLE, THREAD-SAFE QUEUE

On top of `std::queue`

- Producer *pushes* to the queue
- Worker *pops* from the queue
- Simplified for this example
  - No constructor, not copyable, …
  - Templated – you can use different types if needed
- Use condition variables

Simple methods:

- `push(...)` – add an element to the queue
- `pop(...)` – remove from the queue
- `wait_pop(...)` – remove from the queue
  - Returns a `shared_ptr` of the popped element
    - you can modify this as needed, but be careful
  - May wait when called
- `size()` – the number of elements in the queue
- `empty()` – is the queue empty?

```
template <typename T>
class SafeQ {
private:
    queue<T> q;
public:
    void push(T value) {
        q.push(value);          // safe?
    }
    void pop(T &value) {
        if (!q.empty())
        {
            value = q.front(); // safe?
            q.pop();
        }
    }
    shared_ptr<T> wait_pop() {
        if (!q.empty())
        {
            value = q.front(); // safe?
            q.pop();           // safe?
        }
    }
    size_t size() {
        return q.size();       // safe?
    }

    bool empty() {
        return q.empty();      // safe?
    }
};
```

SafeQ class

SERIAL VERSION AND A TEMPLATE FOR YOUR SOLUTION

`a1-serial.cpp, input.txt` (both in Moodle)

To compile on Alma:

```
/opt/global/gcc-11.2.0/bin/g++ -O2 -lpthread -std=c++20 -o a1 a1-sequential.cpp
```

compiler | Optimization | lib | standard | output | source code

The template for the parallel version (a1-parallel.cpp) does not do anything useful at the moment. It gives you a structure where you should insert your code. Notably, you need to:

- **Extend the SafeQ class to support safe accesses by multiple threads**
  - Try to keep the queue-related synchronization in the class
- **Spawn worker threads using standard threading mechanisms (e.g., std::thread)**
  - Make sure that access to shared data is synchronized (**make two versions** – one with the `worker()` function using only atomics and one where the `worker()` is using mutexes)
  - If you require mutexes for implementing the SafeQ – these are ok
- **Spawn the producer thread so it returns the number of items pushed to the queue (`produced_count`) - an integer value using `std::futures` and `std::async`.**

CODE OVERVIEW

```
int producer(std::string filename, SafeQ<int> &q);
```

- Opens the input file, reads the numbers and pushes elements to the queue
- May be modified as needed
- `SafeQ<int>` - the basic form (can also be something extended)

```
void worker(SafeQ<> &q,  int &primes, int &nonprimes, double &sum,
std::vector<int> &number_counts);
```

- Takes elements from the queue and processes them
- The function executed by a worker
- Has to be modified to work correctly in parallel
- Data access needs to be handled properly (e.g., with synchronization)

```
bool kernel(int number);
```

- Represents a computational kernel
- Take an integer as input, returns true if the given number is a prime number
- Not to be modified, just to be called by the *workers*

See the a1-parallel.cpp code for comments

TIME MEASUREMENTS

How to:

```
auto t1 = std::chrono::high_resolution_clock::now();
<...your code part...>
auto t2 = std::chrono::high_resolution_clock::now();

std::cout<< std::chrono::duration<double>(t2-t1).count() << std::endl;
```

*`auto` is actually `std::chrono::high_resolution_clock::time_point` type

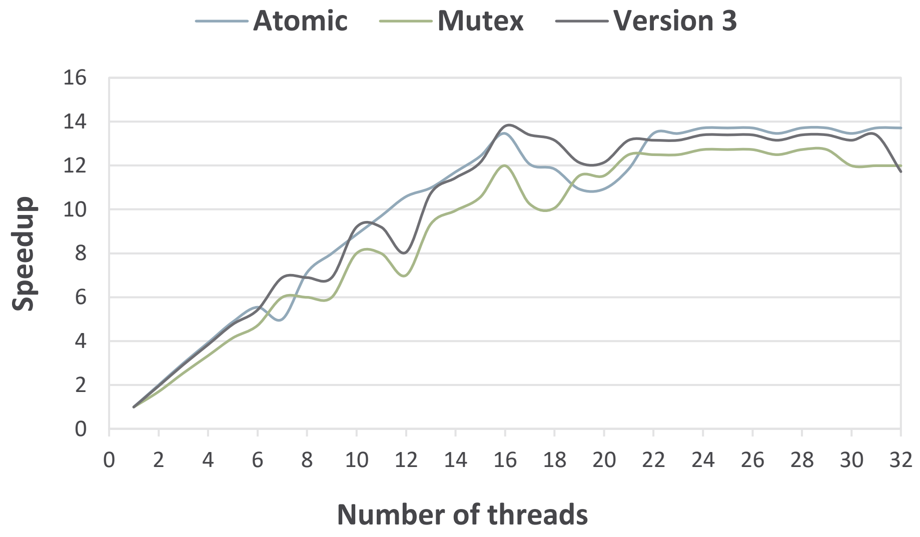
What measurement to include?

- Execution time of different code variants
  1. worker() function with atomics
  2. worker() function with mutexes

RESULTS

Plot Speedup Graphs

- x-axis: number of threads
- y-axis: speedup
- Plot speedup with respect to the sequential version
- Use lines



*not the actual data, just a possible example of how your graph could look like

EXECUTION, INPUT PARAMETERS, FILES

Source files (found in A1 in Moodle):

- `a1-sequential.cpp` The sequential version
- `a1-parallel.cpp` A template for the parallel version
- `a1-helpers.hpp` Helper functions such as printing and parsing arguments

Parameters (and how to run on Alma)

```
srun --nodes=1 ./a1 [options] file
```

Options:

`--help`	**Display this information.**
`--num-threads`	**Set the desired number of threads.**
`--only-exec-times`	**Outputs only the number of threads and the execution time Could be useful when collecting performance measurements**

EXPERIMENTS / SUBMISSION (1/2)

1. **Run the code on one of the nodes on Alma (not on the frontend node)**
   - **Measure the performance of different code versions**
     - The sequential version (~10s on alma nodes with "input.txt")
     - The parallel version(s) and execute using with 1 to 32 threads
       - Measure speedup with respect to the sequential version
       - Write the performance measurements in a table and make speedup graphs
     - A minimum required speedup is about 6, but you should aim for 10+ on Alma
   - Make **two versions** of the code – one where the `worker()` function is using mutexes and another where the `worker()` function uses atomics for synchronization
     - *start with one version, optimize, and then switch to another
2. **Correctness**
   - Your code needs to produce the output that matches the output of the sequential version!
3. **You can develop on your PC/laptop but performance measurements must be done on the Alma cluster**
   - Test early to avoid having to wait for the nodes to free up

Alma system: http://www.par.univie.ac.at/teach/doc/alma.html

EXPERIMENTS / SUBMISSION (2/2)

1. **Write a report**
   - Use the template provided in Moodle as a starting point, which you may extend if needed.

3. **Submit your solution to the Moodle**
   a. a1-parallel-atomic.cpp, a1-parallel-mutex.cpp (use these filenames!)
      - You can also attach additional variants (need to be explained in the report)

   b. A report (PDF document based on the provided template)
      - Upload everything to A1 entry in Moodle before the deadline

Note: **both the documented source code and the PDF report are required, i.e., your code without the report will not be graded positive!**