Exercise 1.7

Tom Westerhout

October 9, 2019

1. Method

1.1. Serial algorithm

To establish a baseline, let us start by implementing a sequential version of the algorithm. We will use `uint64_t`s to represent prime numbers since 64 bits allows us to work with numbers up to $\approx 10^{19}$, and we only need to generate primes up to 10^7:

```
typedef uint64_t ex17_prime_t;
```

The core data structure we will use is a block:

```
typedef struct ex17_bb {
    ex17_prime_t first;
    uint64_t*    data;
    uint64_t     size;
} ex17_bb_t;
```

It represents a range of integers `[first, first + size)` which we can scratch out. Each bit in `data` indicates whether a particular number is still present. For example, `(data[1] >> 5) & 1 == 1` if and only if `first + 64 + 5` has not yet been scratched out.

There are two main operations we need to implement in order to build the sieve of Erastosthenes:

- search for the smallest remaining number, and
- scratching out all larger multiples of a number.

To find the smallest remaining number, we perform a linear search in the block. See the code of `ex17_bb_find_next` function for implementation details.

Scratching out numbers is equivalent to setting corresponding bits of the block to zeros. We use a trick here: if q is a prime, we start scratching at q^2. Every multiple of q smaller than q^2 is equal to nq for some $n < q$. If n is a prime, then we have alredy considered it and thus also scratched out all multiples of it. If n is not prime, then it is a product of primes and we again see that nq is already scratched out. See the code of `ex17_bb_scratch` function for implementation details.

With these components, the main loop of the algorithm looks as follows:

```
/*implementation detail*/ primes;

ex17_prime_t current = 2;
ex17_bb_t    block;
```

```
... initialises the block ...

for (;;) {
    current = ex17_bb_find_next(&block, current);
    if (current > upper_bound) { break; }
    ... adds current to primes ...
    ex17_bb_scratch(&block, current);
}

/*primes now contains all prime numbers up to upper_bound*/
```

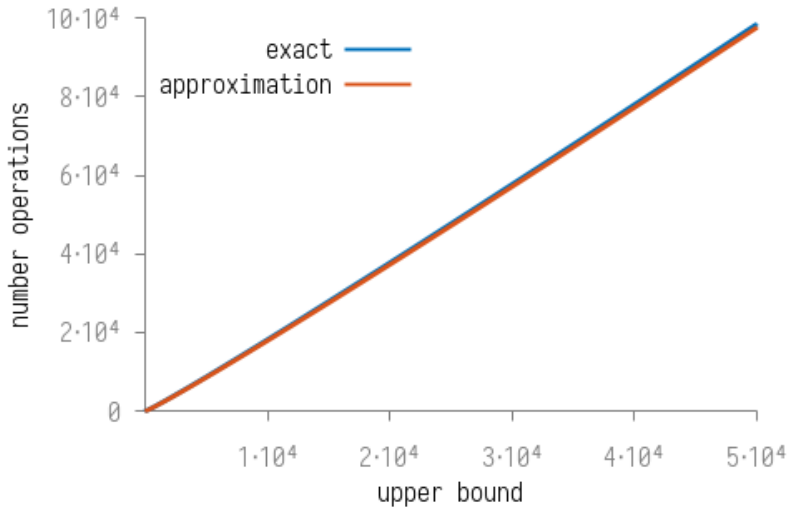
Let us now estimate the total number of cross-out operations. For an upper bound n, the total number of cross-out operations is $\sum_{p\leq n} 1 + \sum_{p\leq\sqrt{n}} \lfloor (n-p^2)/p \rfloor + 1$, where sums run over prime numbers p. The first sum is just the number of primes not exceeding n which is by definition $\pi(n)$ and is approximately $n/\log(n)$ for big n. The second sum consists of three terms:

$$\begin{aligned}
\sum_{p\leq\sqrt{n}} &= \pi(\sqrt{n}) \approx \frac{2\sqrt{n}}{\log(n)}, \\
n\sum_{p\leq\sqrt{n}} &\approx n(\log(\log(\sqrt{n})) + M) \approx n(\log(\log(n)) + 0.261497212 - \log(2)\dots), \\
-\sum_{p\leq\sqrt{n}} &\approx -\frac{n}{\log(n)}
\end{aligned}$$

For the total number of cross-out operations we thus get

$$n\left(\log(\log(n)) + 0.261497212 - \log(2) + \frac{2}{\sqrt{n}\log(n)}\right)$$

Quality of the approximation is shown on the following figure



1.2. Parallel algorithm

We can also reuse `ex17_bb_t` datastructure and operations on it to design a parallel version of the sieve of Eratosthenes. We will use an appoach called “task-based programming” which has been gaining

popularity in recent years.

The only operation we can parallelise is scratching out numbers. If every process (or thread) has its own `ex17_bb_t`, then given a prime number `q`, processes can run `ex17_bb_scratch` in parallel. Note that processes which have blocks which contain numbers greater than `q*q` will do useful work. To ensure this remains the case throughout program execution we will use block-cyclic distribution of numbers between processes. `ex17_make_blocks` implements this:

```
ex17_bb_t* ex17_make_blocks(uint64_t const min, uint64_t const max,
                            uint64_t const block_size)
```

It distributes integers `[min, max)` between all MPI processes using `ex17_bb_t` of size `block_size`. After this function completes every process will have a

```
typedef struct bb_list bb_list_t;
struct bb_list {
    ex17_bb_t  current;
    bb_list_t* next;
};
```

i.e. a singly-linked list of blocks. Then given a prime number, a process calls `ex17_bb_scratch` for every block it owns. `worker_scratch` function implements this idea.

The only remaining issue is how to find the prime `q`. Process which owns the block with the lowest lower bound should run `ex17_bb_find_next` and propagate the result to other processes. This is a very well known master-worker pattern. We implement it by passing around messages:

```
enum ex17_msg_type { EX17_SWITCH_MASTER, EX17_SCRATCH, EX17_STOP };
typedef enum ex17_msg_type ex17_msg_type_t;

typedef struct ex17_msg {
    ex17_msg_type_t type;
    uint64_t        payload;
} ex17_msg_t;
```

There are three types of messages:

- `EX17_SCRATCH` tells a process to run `ex17_scratch` with the prime number `q` stored in `payload`.
- `EX17_SWITCH_MASTER` a vague analogue of context switching. It tells processes that a different process is now the master. Process id of the new master is stored in `payload`.
- `EX17_STOP` just tells processes to stop execution, because the upper bound has been reached.

The rest of the algorithm looks very similar to the serial version. Some care must be taken, however, to correctly handle all the details. See `ex17_generate_primes_parallel` function for an implementation.

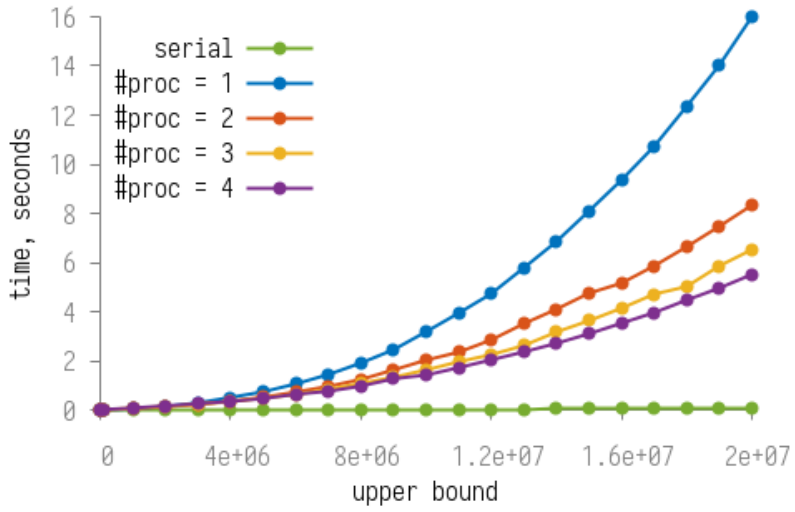
2. Analysis

We can use a simple Bash script to measure the execution time of the algorithm.

```
main()
{
    declare -a BOUNDS=(
        1000 10000 100000 1000000 2000000
        3000000 4000000 5000000 6000000 7000000
```

```
        8000000 9000000 10000000 11000000 12000000
        13000000 14000000 15000000 16000000 17000000
        18000000 19000000 20000000
    )
    declare -a PROCS=(1 2 3 4)
    declare -r block_size=10000
    declare -r PARALLEL=1
    for upper_bound in ${BOUNDS[@]}; do
        if [ $PARALLEL -eq 1 ]; then
            printf '%lu\t' $upper_bound
            for num_procs in ${PROCS[@]}; do
                { time -p mpirun -n $num_procs \
                    "$PREFIX/generate" -b $block_size -q $upper_bound; } 2>&1 \
                        | grep real | cut -d' ' -f 2
            done | tr '\n' '\t'
            printf '\n'
        else
            printf '%lu\t' $upper_bound
            { time -p "$PREFIX/generate" -q $upper_bound; } 2>&1 \
                    | grep real | cut -d' ' -f 2
        fi
    done
}
```

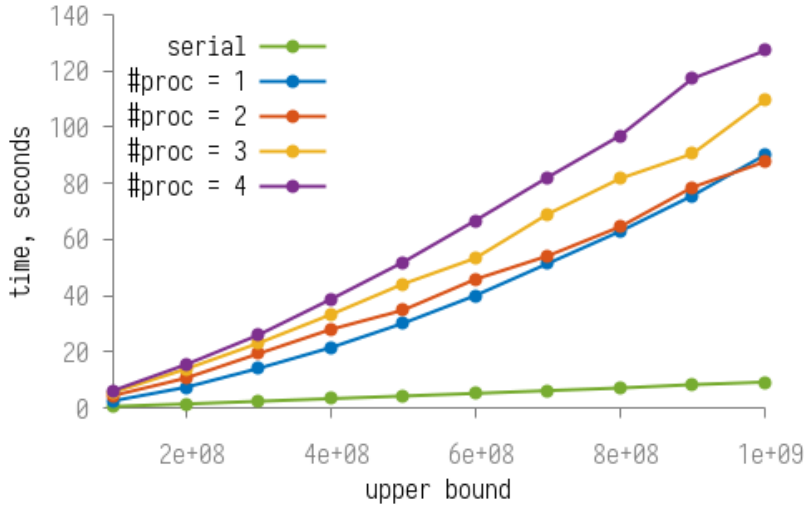
Measurements for a 4-core Intel Core i5-8625U with 6MB of L3 cache are shown on the following figure:



We explicitly used a very small block size (10000), because a large number of blocks allows one to see some benefits of parallelisation. However, using a small block size degrades the overall performance compared to a serial implementation.

The following figure shows the performance for relatively big upper bounds and block sizes:

We see that performance difference is smaller, but still very big. Since we are always sending the exact same very small amount of data around, we can conclude that communication latency outweighs the benefits of parallelisation.



In the serial algorithm, for each prime $q \leq \sqrt{n}$, we had to perform $2 + \lfloor (n - q^2)/q \rfloor$ cross-out operations. In the parallel version, for relatively small block sizes, each process now needs to perform p times fewer operations (where p is the number of processes). However, now each processor also needs to send/receive a message per prime number which incurs additional overhead. This explains why we still see linear scaling but with a bigger proportionality constant.

3. Twin primes

After we learned how to generate prime numbers, filtering out twin primes can be achieved using `ex17_filter_twins` function:

```
void ex17_filter_twins(uint64_t numbers[static 1], uint64_t* size)
{
    uint64_t* first = numbers;
    uint64_t* next  = numbers;
    while (next != numbers + *size) {
        uint64_t const n = *(next++);
        if (n + 2 == *next) {
            *(first++) = n;
            *(first++) = *next;
        }
    }
    *size = (uint64_t)(first - numbers);
}
```

Time complexity of this function is $\mathcal{O}(\text{size})$, so the overall complexity of the algorithm remains the same.

4. Goldbach conjecture

Given a range of numbers `[min, max)` we can check the Goldbach conjecture for this range by generating all prime numbers up to `max - 2` and the computing all possible sums of prime numbers. Function `ex17_goldbach` implements this:

```
int ex17_goldbach(uint64_t min, uint64_t max, bool* result);
```

Using it, we can easily verify the conjecture up to millions.

5. Possible improvements

There are two main points how the algorithm can be improved (which I did not have time to implement):

- `ex17_bb_t` does not need to store flags for even numbers. Since division by 2 is just a right shift on modern computers, memory usage of the algorithm can be reduced by a factor of 2 without sacrificing performance (actually, performance will improve since for larger upper bounds all bits will still fit in cache).
- Since the density of primes is approx $1/\log(n)$ for big ns, which is close to a constant, there will be quite a few prime numbers between q and q^2. This means that we don't have to handle prime numbers one by one, but can instead run `ex17_bb_find_next` a few times before starting scratching out. This allows us to reduce the number of broadcast operations. The amount of data transferred will remain the same, but since currently the main problem is latency rather than throughput, sending a bunch of numbers at once will help a lot.

A. Code