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*/
```

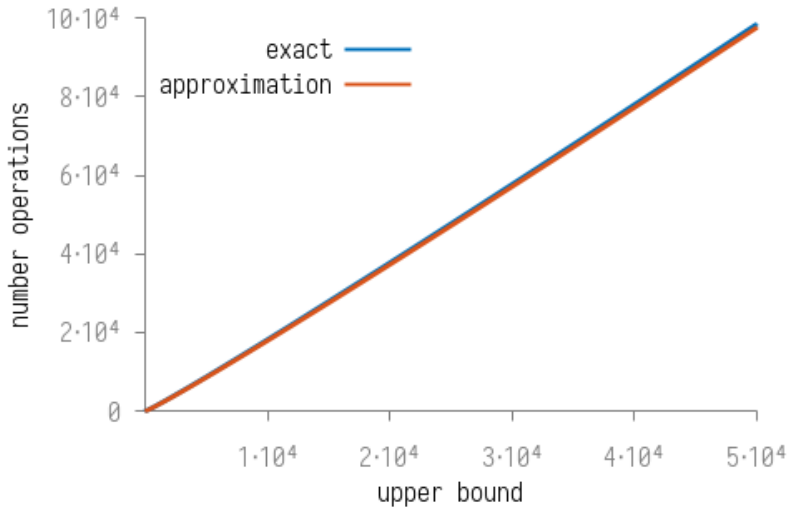
Let us now estimate the total number of cross-out operations. For an upper bound n, the total number of cross-out operations is $\sum_{p\leq n} 1 + \sum_{p\leq\sqrt{n}} \lfloor (n-p^2)/p \rfloor + 1$, where sums run over prime numbers p. The first sum is just the number of primes not exceeding n which 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.

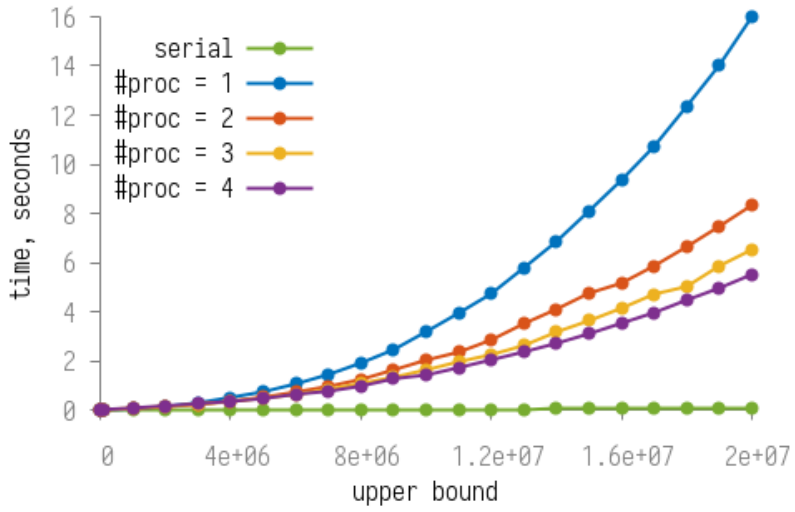
2. Analysis

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
}
```

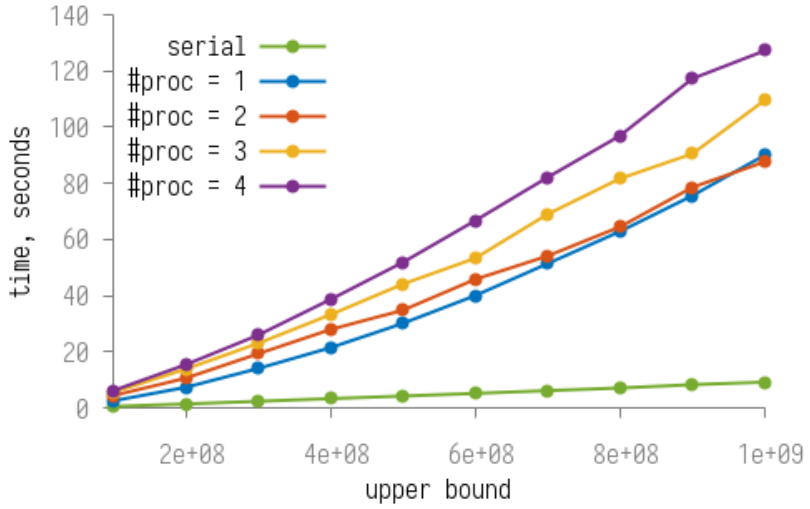
Measurements for a 4-core Intel Core i5-8625U with 6MB of L3 cache are shown on the following figure:



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

As requested in the assignment, the following subsections contain copy-pasted code. It is suggested, however, Github repository `https://github.com/twesterhout/parallel-algorithms-course` is used to browse the code and all helper scripts.

A.1. include/ex17.h

```
#include <stdbool.h>
#include <stdint.h>

#if defined(_MSC_VER) || defined(WIN32) || defined(_WIN32)
#    define EX17_EXPORT __declspec(dllexport)
#    define EX17_NOINLINE __declspec(noinline)
#    define EX17_FORCEINLINE __forceinline inline
#    define EX17_LIKELY(cond) (cond)
#    define EX17_UNLIKELY(cond) (cond)
#    define EX17_CURRENT_FUNCTION __FUNCTION__
#else
#    define EX17_EXPORT __attribute__((visibility("default")))
#    define EX17_NOINLINE __attribute__((noinline))
#    define EX17_FORCEINLINE __attribute__((always_inline)) inline
```

```
#    define EX17_LIKELY(cond) __builtin_expect(!!(cond), 1)
#    define EX17_UNLIKELY(cond) __builtin_expect(!!(cond), 0)
#    define EX17_CURRENT_FUNCTION __PRETTY_FUNCTION__
#endif

typedef uint64_t ex17_prime_t;

typedef struct ex17_result {
    int           status;
    ex17_prime_t* primes;
    uint64_t      size;
} ex17_result_t;

ex17_result_t ex17_generate_primes_serial(ex17_prime_t upper_bound);
ex17_result_t ex17_generate_primes_parallel(ex17_prime_t upper_bound,
                                            uint64_t     block_size);

void ex17_filter_twins(uint64_t numbers[static 1], uint64_t* size);

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

#if defined(EX17_COUNTERS)
void     ex17_reset_scratch_counter(void);
uint64_t ex17_get_scratch_counter(void);
#endif
```

A.2. src/ex17.c

```
#include "ex17.h"

#include <assert.h>  //
#include <errno.h>   // POSIX error codes
#include <stdbool.h> //
#include <stddef.h>  // offset_of
#include <stdio.h>
#include <stdlib.h> // malloc, free
#include <string.h> // memset

#include <mpi.h>

// A hack from Linux kernel. Google if you're not familiar with it
#define container_of(ptr, type, member)                                 \
    ((type*)((char*)(ptr)-offsetof(type, member)))

#if defined(EX17_DEBUG)
#    include <stdio.h>
#    define EX17_TRACE(fmt, ...)                                        \
        do {                                                            \
            int _local_rank;                                            \
            MPI_Comm_rank(MPI_COMM_WORLD, &_local_rank);                \
            fprintf(stderr,                                             \
```

```
                    "[%i] \x1b[1m\x1b[97m%s:%i:%s:\x1b[0m "                    \
                    "\x1b[90mtrace:\x1b[0m " fmt,                              \
                    _local_rank, __FILE__, __LINE__, __FUNCTION__,             \
                    __VA_ARGS__);                                              \
        } while (false)
#else
#    define EX17_TRACE(fmt, ...) (void)0
#endif

// ==== Forward declarations ==== {{{
typedef ex17_prime_t prime_t;

typedef struct v_u64 v_u64_t;
static inline void   v_u64_construct(v_u64_t* self);
static inline void   v_u64_destruct(v_u64_t* self);
static inline int    v_u64_reserve(v_u64_t* self, uint64_t size);
static inline int    v_u64_push_back(v_u64_t* self, uint64_t x);

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

int     ex17_bb_construct(ex17_bb_t* self, prime_t first, uint64_t size);
void    ex17_bb_destruct(ex17_bb_t* self);
prime_t ex17_bb_find_next(ex17_bb_t const* self, prime_t current);
void    ex17_bb_scratch(ex17_bb_t* self, prime_t current);

typedef struct bb_list bb_list_t;
static inline void     ex17_bb_list_pop_front(ex17_bb_t** block);
ex17_bb_t*             ex17_make_blocks(uint64_t const min, uint64_t const max,
                                        uint64_t block_size);
void                   ex17_delete_blocks(ex17_bb_t** block);

typedef struct ex17_ws ex17_ws_t;
static int             ex17_ws_construct(ex17_ws_t* ws, uint64_t upper_bound,
                                         uint64_t block_size);
static void            ex17_ws_destruct(ex17_ws_t* ws);

static int ex17_worker_task(ex17_ws_t* ws, bool* done);
static int ex17_master_task(ex17_ws_t* ws, bool* done);

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;

#if defined(EX17_COUNTERS)
```

```c
static inline uint64_t* ex17_scratch_counter(void);
#endif
// ==== Forward declarations ==== }}}

// ==== v_u64 ==== {{{
typedef struct v_u64 {
    uint64_t* data;
    uint64_t  size;
    uint64_t  capacity;
} v_u64_t;

static inline void v_u64_construct(v_u64_t* self)
{
    assert(self != NULL);
    self->data     = NULL;
    self->size     = 0;
    self->capacity = 0;
}

static inline void v_u64_destruct(v_u64_t* self)
{
    assert(self != NULL);
    if (self->data != NULL) { free(self->data); }
    self->data     = NULL;
    self->size     = 0;
    self->capacity = 0;
}

static inline int v_u64_reserve(v_u64_t* self, uint64_t const size)
{
    assert(self != NULL);
    if (self->capacity >= size) { return 0; }
    void* p = realloc(self->data, size * sizeof(uint64_t));
    if (p == NULL) { return ENOMEM; }
    self->data     = p;
    self->capacity = size;
    return 0;
}

static inline int v_u64_push_back(v_u64_t* self, uint64_t const x)
{
    assert(self != NULL);
    if (self->size == self->capacity) {
        // Growth policy
        uint64_t new_capacity = self->capacity + 1048576 / sizeof(uint64_t);
        int      status       = v_u64_reserve(self, new_capacity);
        if (status != 0) { return status; }
    }
    self->data[self->size++] = x;
    return 0;
}
// ==== v_u64 ==== }}}
```

```c
// ==== counters ==== {{{
#if defined(EX17_COUNTERS)
static inline uint64_t* ex17_scratch_counter(void)
{
    static _Thread_local uint64_t counter = 0;
    return &counter;
}

void ex17_reset_scratch_counter(void) { *ex17_scratch_counter() = 0; }

uint64_t ex17_get_scratch_counter(void) { return *ex17_scratch_counter(); }
#endif
// ==== counters ==== }}}

// ==== ex17_bb_t ==== {{{
int ex17_bb_construct(ex17_bb_t* self, prime_t const first, uint64_t const size)
{
    assert(self != NULL);
    uint64_t const capacity = (size + 64 - 1) / 64;
    self->data              = malloc(capacity * sizeof(uint64_t));
    if (EX17_UNLIKELY(self->data == NULL)) { return ENOMEM; }
    memset(self->data, 0xFF, capacity * sizeof(uint64_t));
    if (size % 64 != 0) {
        self->data[capacity - 1] = (~0ULL) >> (64 - size % 64);
    }
    self->first = first;
    self->size  = size;
    assert(self->size % 64 == 0
           || (((~0ULL) << (self->size % 64))
               & self->data[(self->size + 64 - 1) / 64 - 1])
                  == 0);
    return 0;
}

void ex17_bb_destruct(ex17_bb_t* self)
{
    assert(self != NULL && self->data != NULL);
    if (EX17_LIKELY(self != NULL)) {
        free(self->data);
        self->size = 0;
    }
}

#if 0
static void ex17_bb_print(ex17_bb_t const* block)
{
    assert(block != NULL);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("[%i] buffer_block@%p(%p, %lu, %lu): [", rank, (void const*)block,
           (void const*)block->data, block->first, block->size);
```

```
    if (block->size > 1) {
        for (uint64_t i = 0; i < block->size - 1; ++i) {
            printf("%lu, ", (block->data[i / 64] >> (i % 64)) & 0x01);
        }
    }
    if (block->size != 0) {
        printf("%lu ",
               (block->data[(block->size - 1) / 64] >> ((block->size - 1) % 64))
                   & 0x01);
    }
    printf("]\n");
}
#endif

prime_t ex17_bb_find_next(ex17_bb_t const* self, prime_t const current)
{
    EX17_TRACE("self=%p, current=%lu\n", (void const*)self, current);
    assert(self != NULL);
    assert(self->first <= current && current < self->first + self->size
           && "current is out of range for this block");
    assert((self->size == 0 && self->data == NULL)
           || (self->size != 0 && self->data != NULL));
    assert(self->size % 64 == 0
           || (((~0ULL) << (self->size % 64))
               & self->data[(self->size + 64 - 1) / 64 - 1])
                  == 0);
    uint64_t const offset     = (current - self->first) / 64;
    uint64_t const num_chunks = (self->size + 64 - 1) / 64;
    for (uint64_t i = offset; i < num_chunks; ++i) {
        if (self->data[i] != 0) {
            return self->first + 64 * i
                   + (uint64_t)__builtin_ctzl(self->data[i]);
        }
    }
    return self->first + self->size;
    // uint64_t const offset = current - self->first;
    // uint8_t const* p = memchr(self->data + offset, 0xFF, self->size - offset);
    // if (p == NULL) { return self->first + self->size; }
    // return self->first + (uint64_t)(p - self->data);
}

static inline void _clear_bit(uint64_t* data, uint64_t const i)
{
    uint64_t const chunk = i / 64;
    uint64_t const rest  = i % 64;
    data[chunk] &= ~(1ULL << rest);
#if defined(EX17_COUNTERS)
    ++(*ex17_scratch_counter());
#endif
}

void ex17_bb_scratch(ex17_bb_t* self, prime_t const current)
```

```
{
    assert(self != NULL && current != 0);
    if (current >= self->first + self->size) { return; }
    uint64_t i;
    if (current >= self->first) {
        // self->data[current - self->first] = 0;
        _clear_bit(self->data, current - self->first);
        i = current * current - self->first;
    }
    else {
        i = current * ((self->first + current - 1) / current) - self->first;
    }

    if (current <= self->size) {
        for (; i < self->size; i += current) {
            _clear_bit(self->data, i);
            // self->data[i] = 0;
        }
    }
    else if (i < self->size) {
        _clear_bit(self->data, i);
        // self->data[i] = 0;
    }
}
// ==== ex17_bb_t ==== }}}

EX17_EXPORT ex17_result_t ex17_generate_primes_serial(prime_t const upper_bound)
{
    ex17_result_t result = {.status = 0, .primes = NULL, .size = 0};
    if (upper_bound < 2) { return result; }

    v_u64_t primes;
    v_u64_construct(&primes);
    // result.status = v_u64_reserve(&primes, upper_bound + 1);
    // if (result.status != 0) { goto malloc_primes_failed; }

    prime_t   current = 2;
    ex17_bb_t block;
    result.status =
        ex17_bb_construct(&block, current, upper_bound + 1 - current);
    if (result.status != 0) { goto construct_block_failed; }

#if defined(EX17_COUNTERS)
    ex17_reset_scratch_counter();
#endif
    for (;;) {
        current = ex17_bb_find_next(&block, current);
        if (current > upper_bound) { break; }
        result.status = v_u64_push_back(&primes, current);
        if (result.status != 0) { goto push_back_failed; }
        ex17_bb_scratch(&block, current);
    }
```

```
    result.primes = primes.data;
    result.size   = primes.size;
    primes.data   = NULL;

push_back_failed:
    ex17_bb_destruct(&block);
construct_block_failed:
malloc_primes_failed:
    v_u64_destruct(&primes);
    return result;
}

// ==== bb_list_t ==== {{{
typedef struct bb_list bb_list_t;
struct bb_list {
    ex17_bb_t  current;
    bb_list_t* next;
};

/**
 * \brief Appends an elements to the list.
 *
 * \pre `self != NULL` and `self->next == NULL`. If preconditions are violated
 *      `EINVAL` is returned.
 */
static inline int bb_list_emplace_back(bb_list_t* self, prime_t const first,
                                       uint64_t const size)
{
    assert(self != NULL && self->next == NULL);
    if (self == NULL || self->next != NULL) { return EINVAL; }
    bb_list_t* node = malloc(sizeof(bb_list_t));
    if (node == NULL) { return ENOMEM; }
    int status = ex17_bb_construct(&node->current, first, size);
    if (status != 0) {
        free(node);
        return status;
    }
    node->next = NULL;
    self->next = node;
    return 0;
}

#if 0
static inline int ex17_bb_list_emplace_back(ex17_bb_t*     block,
                                            prime_t const  first,
                                            uint64_t const size)
{
    assert(block != NULL);
    if (block == NULL) { return EINVAL; }
    bb_list_t* node = container_of(block, bb_list_t, current);
    return bb_list_emplace_back(node, first, size);
```

```
}
#endif

static inline void ex17_bb_list_pop_front(ex17_bb_t** block)
{
    assert(block != NULL && *block != NULL);
    bb_list_t* node = container_of(*block, bb_list_t, current);
    *block          = (node->next != NULL) ? &node->next->current : NULL;
    ex17_bb_destruct(&node->current);
    free(node);
}

ex17_bb_t* ex17_make_blocks(uint64_t const min, uint64_t const max,
                            uint64_t const block_size)
{
    assert(min <= max);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    int block_index = 0;
    // `head` is a stack-allocated dummy node. Its `current` member is
    // never initialised.
    bb_list_t  head = {.next = NULL};
    bb_list_t* node = &head;
    uint64_t   i;
    for (i = min; i + block_size <= max; i += block_size, ++block_index) {
        if (block_index % size == rank) {
            if (EX17_UNLIKELY(bb_list_emplace_back(node, i, block_size) != 0)) {
                goto emplace_failed;
            }
            node = node->next;
        }
    }
    if (i < max && block_index % size == rank) {
        if (EX17_UNLIKELY(bb_list_emplace_back(node, i, max - i) != 0)) {
            goto emplace_failed;
        }
    }
    return &head.next->current;

emplace_failed:
    if (head.next != NULL) {
        ex17_bb_t* block = &head.next->current;
        do {
            ex17_bb_list_pop_front(&block);
        } while (block != NULL);
    }
    return NULL;
}

void ex17_delete_blocks(ex17_bb_t** block)
{
```

```
    assert(block != NULL);
    while (*block != NULL) {
        ex17_bb_list_pop_front(block);
    }
}
// ==== bb_list_t ==== }}}

// ==== MPI ex17_msg_t datatype ==== {{{
static void ex17_msg_dtype_construct(MPI_Datatype* datatype)
{
    assert(datatype != NULL);
    int          lengths[2]       = {1, 1};
    MPI_Aint     displacements[2] = {offsetof(ex17_msg_t, type),
                                     offsetof(ex17_msg_t, payload)};
    MPI_Datatype types[2]         = {MPI_INT, MPI_UNSIGNED_LONG_LONG};

    int status =
        MPI_Type_create_struct(2, lengths, displacements, types, datatype);
    assert(status == MPI_SUCCESS);
    status = MPI_Type_commit(datatype);
    assert(status == MPI_SUCCESS);
}

static void ex17_msg_dtype_destruct(MPI_Datatype* datatype)
{
    assert(datatype != NULL);
    int status = MPI_Type_free(datatype);
    assert(status == MPI_SUCCESS);
}
// ==== MPI ex17_msg_t datatype ==== }}}

// ==== ex17_ws_t ==== {{{
typedef struct ex17_ws {
    ex17_bb_t*   block;
    v_u64_t      primes;
    prime_t      upper_bound;
    int          id;
    int          next_id;
    int          master_id;
    MPI_Datatype msg_dtype;
    bool         last;
} ex17_ws_t;

static inline prime_t ex17_bb_list_upper_bound(ex17_bb_t const* block)
{
    assert(block != NULL);
    prime_t    max  = block->first + block->size;
    bb_list_t* node = container_of(block, bb_list_t, current)->next;
    while (node != NULL) {
        max  = node->current.first + node->current.size;
        node = node->next;
    }
```

```
    return max;
}

static int ex17_ws_construct(ex17_ws_t* ws, uint64_t upper_bound,
                             uint64_t block_size)
{
    assert(ws != NULL);
    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &ws->id);
    ws->next_id     = (ws->id + 1) % size;
    ws->master_id   = 0;
    ws->upper_bound = upper_bound;
    v_u64_construct(&ws->primes);
    ws->block = ex17_make_blocks(2, upper_bound + 1, block_size);
    if (EX17_UNLIKELY(ws->block == NULL)) { return ENOMEM; }
    ws->last = ex17_bb_list_upper_bound(ws->block) >= upper_bound;
    ex17_msg_dtype_construct(&ws->msg_dtype);
    return 0;
}

static void ex17_ws_destruct(ex17_ws_t* ws)
{
    assert(ws != NULL);
    ex17_delete_blocks(&ws->block);
    v_u64_destruct(&ws->primes);
    ex17_msg_dtype_destruct(&ws->msg_dtype);
}
// ==== ex17_ws_t ==== }}}

static int worker_scratch(ex17_ws_t* ws, prime_t const current)
{
    assert(ws != NULL && current != 0);
    EX17_TRACE("%s", "started...\n");
    int status = v_u64_push_back(&ws->primes, current);
    if (EX17_UNLIKELY(status != 0)) {
        EX17_TRACE("v_u64_push_back failed with status %i\n", status);
        return status;
    }
    // We know that `current` can only increase, so we can throw away all blocks
    // with upper bounds smaller than `current`.
    while (ws->block != NULL && current >= ws->block->first + ws->block->size) {
        ex17_bb_list_pop_front(&ws->block);
    }
    // Iterate over all remaining blocks and scratch out `current`.
    if (ws->block != NULL) {
        for (bb_list_t* node    = container_of(ws->block, bb_list_t, current);
             node != NULL; node = node->next) {
            ex17_bb_scratch(&node->current, current);
        }
    }
    EX17_TRACE("finished! is ws->block NULL? %i\n", ws->block == NULL);
```

```
    return 0;
}

static inline void broadcast(ex17_ws_t const* ws, ex17_msg_t* msg,
                             MPI_Request* request)
{
    int status = MPI_Ibcast(
        /*buffer=*/msg, /*count=*/1, /*datatype=*/ws->msg_dtype,
        /*root=*/ws->master_id, /*comm=*/MPI_COMM_WORLD, /*request=*/request);
    assert(status == MPI_SUCCESS);
}

/**
 * \brief Task executed by worker processes.
 *
 * \return `true` if the algorithm is complete, `false` if we have to keep
 * computing.
 */
static int ex17_worker_task(ex17_ws_t* ws, bool* done)
{
    assert(ws != NULL);
    MPI_Request request;
    ex17_msg_t  msg_buffer;
    broadcast(ws, &msg_buffer, &request);

    int status = 0;
    for (;;) {
        status = MPI_Wait(&request, MPI_STATUS_IGNORE);
        assert(status == MPI_SUCCESS);
        ex17_msg_t msg = msg_buffer;
        EX17_TRACE("message: type=%i payload=%lu\n", msg.type, msg.payload);
        switch (msg.type) {
        case EX17_SCRATCH: {
            broadcast(ws, &msg_buffer, &request);
            status = worker_scratch(ws, msg.payload);
            if (status != 0) { return status; }
            break;
        }
        case EX17_SWITCH_MASTER: {
            ws->master_id = (int)msg.payload;
            if (ws->master_id == ws->id) {
                *done = false;
                return 0;
            }
            broadcast(ws, &msg_buffer, &request);
            break;
        }
        case EX17_STOP: *done = true; return 0;
        } // end switch
    }
}
```

```c
// ==== message_queue_t ==== {{{
#define EX17_MESSAGE_QUEUE_CAPACITY 16

typedef struct ex17_msg_queue {
    ex17_msg_t  messages[EX17_MESSAGE_QUEUE_CAPACITY];
    MPI_Request requests[EX17_MESSAGE_QUEUE_CAPACITY];
    unsigned    first;
    unsigned    size;
} ex17_msg_queue_t;

static inline void ex17_msg_queue_construct(ex17_msg_queue_t* queue)
{
    assert(queue != NULL);
    queue->first = 0;
    queue->size  = 0;
}

static inline void ex17_msg_queue_destruct(ex17_msg_queue_t* queue)
{
    // There's no need to wait for all the messages, so we just reset `first`
    // and `size`.
    assert(queue != NULL);
    queue->first = 0;
    queue->size  = 0;
}

static inline void ex17_msg_queue_pop(ex17_msg_queue_t* queue)
{
    assert(queue != NULL);
    assert(queue->size > 0);
    MPI_Wait(&queue->requests[queue->first], MPI_STATUS_IGNORE);
    queue->first = (queue->first + 1) % EX17_MESSAGE_QUEUE_CAPACITY;
    --(queue->size);
}

static inline void ex17_msg_queue_push(ex17_msg_queue_t* queue, ex17_msg_t msg,
                                       MPI_Datatype datatype)
{
    if (queue->size == EX17_MESSAGE_QUEUE_CAPACITY) {
        ex17_msg_queue_pop(queue);
    }
    unsigned i = (queue->first + queue->size) % EX17_MESSAGE_QUEUE_CAPACITY;
    queue->messages[i] = msg;
    int root, status;
    status = MPI_Comm_rank(MPI_COMM_WORLD, &root);
    assert(status == MPI_SUCCESS);
    status = MPI_Ibcast(
        /*buffer=*/&queue->messages[i], /*count=*/1,
        /*datatype=*/datatype,
        /*root=*/root, /*comm=*/MPI_COMM_WORLD,
        /*request=*/&queue->requests[i]);
    assert(status == MPI_SUCCESS);
```

```
    ++(queue->size);
}

#if 0
static void message_queue_clear(message_queue_t* queue)
{
    while (queue->size != 0) {
        message_queue_pop(queue);
    }
}
#endif
// ==== message_queue_t ==== }}}

static inline int _master_switch(ex17_ws_t* ws, ex17_msg_queue_t* queue,
                                 bool* done)
{
    ws->master_id  = ws->next_id;
    ex17_msg_t msg = {EX17_SWITCH_MASTER, (uint64_t)ws->next_id};
    ex17_msg_queue_push(queue, msg, ws->msg_dtype);
    ex17_msg_queue_destruct(queue);
    *done = false;
    return 0;
}

static inline bool _master_stop(ex17_ws_t* ws, ex17_msg_queue_t* queue,
                                bool* done)
{
    ex17_msg_t msg = {EX17_STOP, ~0UL};
    ex17_msg_queue_push(queue, msg, ws->msg_dtype);
    ex17_msg_queue_destruct(queue);
    *done = true;
    return 0;
}

static int ex17_master_task(ex17_ws_t* ws, bool* done)
{
    // fprintf(stderr, "%i: %s(%p)...\n", ws->id, __FUNCTION__, (void const*)ws);
    assert(ws != NULL);
    ex17_msg_queue_t queue;
    ex17_msg_queue_construct(&queue);
    prime_t n = ws->block->first;
    if (ws->primes.size != 0) {
        uint64_t const last = ws->primes.data[ws->primes.size - 1];
        if (last > n) { n = last; }
    }
    for (;;) {
        if (ws->block == NULL) {
            EX17_TRACE("%s\n", ws->last ? "stop" : "switch");
            return ws->last ? _master_stop(ws, &queue, done)
                            : _master_switch(ws, &queue, done);
        }
        n = ex17_bb_find_next(ws->block, n);
```

```
        if (n == ws->block->first + ws->block->size) {
            ex17_bb_list_pop_front(&ws->block);
            EX17_TRACE("%s\n", n > ws->upper_bound ? "stop" : "switch");
            return (n > ws->upper_bound) ? _master_stop(ws, &queue, done)
                                         : _master_switch(ws, &queue, done);
        }
        ex17_msg_t msg = {EX17_SCRATCH, n};
        ex17_msg_queue_push(&queue, msg, ws->msg_dtype);
        int status = worker_scratch(ws, n);
        if (status != 0) {
            EX17_TRACE("%s\n", "worker scratch failed");
            ex17_msg_queue_destruct(&queue);
            return status;
        }
    }
}

EX17_EXPORT ex17_result_t ex17_generate_primes_parallel(prime_t  upper_bound,
                                                        uint64_t block_size)
{
    ex17_result_t result = {.status = 0, .primes = NULL, .size = 0};

    ex17_ws_t ws;
    result.status = ex17_ws_construct(&ws, upper_bound, block_size);
    if (result.status != 0) { goto workspace_construction_failed; }

    bool done = false;
    do {
        result.status = ws.id == ws.master_id ? ex17_master_task(&ws, &done)
                                              : ex17_worker_task(&ws, &done);
        if (result.status != 0) { MPI_Abort(MPI_COMM_WORLD, result.status); }
    } while (!done);

    result.primes  = ws.primes.data;
    result.size    = ws.primes.size;
    ws.primes.data = NULL;
    ex17_ws_destruct(&ws);
workspace_construction_failed:
    return result;
}

EX17_EXPORT void ex17_filter_twins(uint64_t numbers[static 1], uint64_t* size)
{
    assert(numbers != NULL && size != NULL);
    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);
}

EX17_EXPORT int ex17_goldbach(uint64_t min, uint64_t max, bool* result)
{
    if (min > max) { return EINVAL; }
    else if (min == max) {
        *result = true;
        return 0;
    }
    min += min % 2; // Make sure min is even

    ex17_result_t r = ex17_generate_primes_serial(max - 2);
    if (r.status != 0) { return r.status; }

    ex17_bb_t block;
    int       status = ex17_bb_construct(&block, min, (max - min) / 2);
    if (status != 0) { goto block_construction_failed; }

    for (uint64_t i = 0; i < r.size && r.primes[i] <= max / 2; ++i) {
        for (uint64_t j = i; j < r.size; ++j) {
            uint64_t sum = r.primes[i] + r.primes[j];
            if (sum >= max) { break; }
            if (sum >= min) { _clear_bit(block.data, (sum - min) / 2); }
        }
    }

    *result =
        ex17_bb_find_next(&block, block.first) == block.first + block.size;
    ex17_bb_destruct(&block);

block_construction_failed:
    if (r.primes != NULL) { free(r.primes); }
    return status;
}
```

A.3. src/generate.c

```
#include "ex17.h"

#include <mpi.h>

#include <getopt.h>
#include <unistd.h>

#include <assert.h>
#include <ctype.h> // isprint
#include <errno.h>
#include <stdbool.h>
#include <stdio.h>
```

```c
#include <stdlib.h>
#include <string.h>

static inline int _parse(int argc, char* argv[])
{
    return getopt(argc, argv, "hqtb:");
}

static void help(int argc, char* argv[])
{
    (void)argc;
    // clang-format off
    fprintf(stderr,
        "Usage: %s [-q] [-t] [-b block_size] upper_bound\n"
        "\n"
        "Generates all prime numbers up to upper_bound and prints them to stdout.\n"
        "If block_size is given, parallel algorithm is used, otherwise primes are\n"
        "generated using a serial implementation.\n"
        "\n"
        "-q flag causes output to be supressed. This is useful for benchmarking.\n"
        "-t flag causes the program to generate twin prime numbers.\n", argv[0]);
    // clang-format on
}

int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int status = 0;

    // Make sure getopt doesn't try to print error messages from all the
    // processes
    opterr               = 0;
    bool  help_flag      = false;
    bool  quiet_flag     = false;
    bool  twins_flag     = false;
    char* block_size_str = NULL;
    for (int c = _parse(argc, argv); c != -1; c = _parse(argc, argv)) {
        switch (c) {
        case 'b': block_size_str = optarg; break;
        case 'q': quiet_flag = true; break;
        case 't': twins_flag = true; break;
        case 'h': help_flag = true; break;
        case '?':
            if (rank == 0) {
                if (optopt == 'b') {
                    fprintf(stderr, "Option -%c requires an argument.\n",
                            optopt);
                }
                else if (isprint(optopt)) {
```

```
                fprintf(stderr,
                        "Unknown option '-%c'. Use -h flag to see "
                        "available options\n",
                        optopt);
            }
            else {
                fprintf(stderr,
                        "Unknown option character '\\x%x'. Use -h flag to "
                        "see available options\n",
                        optopt);
            }
        }
        status = 1;
        goto cleanup;
    } // end switch
}
if (help_flag) {
    if (rank == 0) { help(argc, argv); }
    goto cleanup;
}

if (optind + 1 != argc) {
    if (rank == 0) {
        fprintf(stderr, "%s: expected a single argument upper_bound\n",
                argv[0]);
    }
    status = 1;
    goto cleanup;
}
uint64_t upper_bound;
{
    char* end   = NULL;
    upper_bound = strtoull(argv[optind], &end, /*base=*/10);
    if (errno == ERANGE || (upper_bound == 0 && *end != '\0')) {
        if (rank == 0) {
            fprintf(stderr, "%s: invalid upper_bound: '%s'\n", argv[0],
                    argv[optind]);
        }
        status = 1;
        goto cleanup;
    }
}

ex17_result_t result;
if (block_size_str != NULL) {
    char*    end        = NULL;
    uint64_t block_size = strtoull(block_size_str, &end, /*base=*/10);
    if (errno == ERANGE || block_size == 0 || *end != '\0') {
        if (rank == 0) {
            fprintf(stderr, "%s: invalid block_size: '%s'\n", argv[0],
                    block_size_str);
        }
```

```
            status = 1;
            goto cleanup;
        }
        result = ex17_generate_primes_parallel(upper_bound, block_size);
    }
    else {
        result = ex17_generate_primes_serial(upper_bound);
    }

    if (result.status != 0) {
        if (rank == 0) {
            fprintf(stderr, "ex17_generate_primes() failed: %s\n",
                    strerror(result.status));
        }
        status = result.status;
        MPI_Abort(MPI_COMM_WORLD, status);
        goto cleanup;
    }
    if (!quiet_flag && rank == 0) {
        if (twins_flag) {
            ex17_filter_twins(result.primes, &result.size);
            assert(result.size % 2 == 0);
            for (uint64_t i = 0; i < result.size; i += 2) {
                printf("%lu %lu\n", result.primes[i], result.primes[i + 1]);
            }
        }
        else {
            for (uint64_t i = 0; i < result.size; ++i) {
                printf("%lu\n", result.primes[i]);
            }
        }
    }
    if (result.primes != NULL) { free(result.primes); }

cleanup:
    MPI_Finalize();
    return status;
}
```