Verifying an Efficient Algorithm to Compute Bernoulli Numbers

Manuel Eberl Peter Lammich

Faulhaber's formula

$$\sum_{i=0}^{n} i^0 = n + 1$$

Faulhaber's formula

$$\sum_{i=0}^{n} i^0 = n + 1$$

$$\sum_{i=0}^{n} i^1 = \frac{1}{2}(n^2 + n)$$

Faulhaber's formula

$$\sum_{i=0}^{n} i^0 = n + 1$$

$$\sum_{i=0}^{n} i^1 = \frac{1}{2}(n^2 + n)$$

$$\sum_{i=0}^{n} i^2 = \frac{1}{6}(2n^3 + 3n^2 + n)$$

Faulhaber's formula

$$\sum_{i=0}^{n} i^0 = n + 1$$

$$\sum_{i=0}^{n} i^1 = \tfrac{1}{2}(n^2 + n)$$

$$\sum_{i=0}^{n} i^2 = \tfrac{1}{6}(2n^3 + 3n^2 + n)$$

$$\sum_{i=0}^{n} i^k = \tfrac{1}{m+1}(P_{m+1}(n+1) - P_{m+1}(0)) \qquad \text{where } P_m(x) = \sum_{0 \le k \le m} \binom{m}{k} B_k x^{m-k}$$

Bernoulli Numbers

Definition

Definition (Bernoulli numbers)

The Bernoulli numbers are the sequence of rationals $(B_k)_{k\geq 0}$ with

$$\sum_{k\geq 0} B_k \frac{z^k}{k!} = \frac{z}{\exp(z)-1}$$

We write $B_k = N_k/D_k$.

Definition

Definition (Bernoulli numbers)

The Bernoulli numbers are the sequence of rationals $(B_k)_{k\geq 0}$ with

$$\sum_{k\geq 0} B_k \frac{z^k}{k!} = \frac{z}{\exp(z)-1}$$

We write $B_k = N_k / D_k$.

k	0	1	2	3	4	5	6	7	8	9	10
B_k	1	$-\frac{1}{2}$	$\frac{1}{6}$	0	$-\frac{1}{30}$	0	$\frac{1}{42}$	0	$-\frac{1}{30}$	0	$\frac{5}{66}$

Definition

Definition (Bernoulli numbers)

The Bernoulli numbers are the sequence of rationals $(B_k)_{k\geq 0}$ with

$$\sum_{k\geq 0} B_k \frac{z^k}{k!} = \frac{z}{\exp(z)-1}$$

We write $B_k = N_k / D_k$.

k	0	1	2	3	4	5	6	7	8	9	10
B_k	1	$-\frac{1}{2}$	$\frac{1}{6}$	0	$-\frac{1}{30}$	0	$\frac{1}{42}$	0	$-\frac{1}{30}$	0	$\frac{5}{66}$

$B_{60} \approx -1.2 \cdot 10^{42} \;/\; 56786730$ $\qquad$ $B_{110} \approx 7.6 \cdot 10^{93} \;/\; 1518$

Some more properties

- $B_k = 0$ for odd $k > 1$

Some more properties

- $B_k = 0$ for odd $k > 1$
- $B_2, B_4, B_6, \ldots$ positive; $B_4, B_8, B_{12}, \ldots$ negative

Some more properties

- $B_k = 0$ for odd $k > 1$
- $B_2, B_4, B_6, \ldots$ positive; $B_4, B_8, B_{12}, \ldots$ negative
- Through connection with ζ: $\log|B_{2k}|$ and $\log|N_{2k}|$ are of order $\Theta(k \log k)$

Some more properties

- $B_k = 0$ for odd $k > 1$
- $B_2, B_4, B_6, \ldots$ positive; $B_4, B_8, B_{12}, \ldots$ negative
- Through connection with ζ: $\log|B_{2k}|$ and $\log|N_{2k}|$ are of order $\Theta(k \log k)$

Theorem (Voronoi's congruence)

$$(a^k - 1)B_k \equiv ka^{k-1} \sum_{1 \le m < p} m^{k-1} \left\lfloor \frac{ma}{p} \right\rfloor \pmod{p}$$

Allows us to compute $B_k \bmod p$ *as a sum.*

Some more properties

- $B_k = 0$ for odd $k > 1$
- $B_2, B_4, B_6, \ldots$ positive; $B_4, B_8, B_{12}, \ldots$ negative
- Through connection with ζ: $\log|B_{2k}|$ and $\log|N_{2k}|$ are of order $\Theta(k \log k)$

Theorem (Voronoi's congruence)

$$(a^k - 1)B_k \equiv ka^{k-1} \sum_{1 \le m < p} m^{k-1} \left\lfloor \frac{ma}{p} \right\rfloor \pmod{p}$$

Allows us to compute $B_k \bmod p$ as a sum.

Theorem (Kummer's congruence)

$$B_k/k \equiv B_{k'}/k \pmod{p} \qquad \text{where } k' = k \bmod (p-1)$$

Thus we can w.l.o.g. assume that $k = 2, 4, \ldots, p-3$.

Computation

Algorithms

Slow algorithms:

- Using the recurrence that drops out of the definition: very inefficient.

Algorithms

Slow algorithms:

- Using the recurrence that drops out of the definition: very inefficient.
- Various cubic methods (some only use integer arithmetic)

Algorithms

Slow algorithms:

- Using the recurrence that drops out of the definition: very inefficient.
- Various cubic methods (some only use integer arithmetic)

Modern quadratic algorithms, roughly $O(k^2 \log^{1+o(1)} k)$:

Algorithms

Slow algorithms:

- Using the recurrence that drops out of the definition: very inefficient.
- Various cubic methods (some only use integer arithmetic)

Modern quadratic algorithms, roughly $O(k^2 \log^{1+o(1)} k)$:

- Compute $\sin(z)/\cos(z)$ as formal power series

Algorithms

Slow algorithms:

- Using the recurrence that drops out of the definition: very inefficient.
- Various cubic methods (some only use integer arithmetic)

Modern quadratic algorithms, roughly $O(k^2 \log^{1+o(1)} k)$:

- Compute $\sin(z)/\cos(z)$ as formal power series
- Compute B_{2k} by approximating $\zeta(2k)$

Algorithms

Slow algorithms:

- Using the recurrence that drops out of the definition: very inefficient.
- Various cubic methods (some only use integer arithmetic)

Modern quadratic algorithms, roughly $O(k^2 \log^{1+o(1)} k)$:

- Compute $\sin(z)/\cos(z)$ as formal power series
- Compute B_{2k} by approximating $\zeta(2k)$
  Allows computing B_{2k} without $B_2, \ldots, B_{2k-2}$.

Algorithms

Slow algorithms:

- Using the recurrence that drops out of the definition: very inefficient.
- Various cubic methods (some only use integer arithmetic)

Modern quadratic algorithms, roughly $O(k^2 \log^{1+o(1)} k)$:

- Compute $\sin(z)/\cos(z)$ as formal power series
- Compute B_{2k} by approximating $\zeta(2k)$
  Allows computing B_{2k} without $B_2, \ldots, B_{2k-2}$.
- Harvey (2010): Multimodular algorithm.

Algorithms

Slow algorithms:

- Using the recurrence that drops out of the definition: very inefficient.
- Various cubic methods (some only use integer arithmetic)

Modern quadratic algorithms, roughly $O(k^2 \log^{1+o(1)} k)$:

- Compute $\sin(z)/\cos(z)$ as formal power series
- Compute B_{2k} by approximating $\zeta(2k)$
  Allows computing B_{2k} without $B_2, \ldots, B_{2k-2}$.
- Harvey (2010): Multimodular algorithm.
  Compute B_k mod p for many primes p independently.

Algorithms

Slow algorithms:

- Using the recurrence that drops out of the definition: very inefficient.
- Various cubic methods (some only use integer arithmetic)

Modern quadratic algorithms, roughly $O(k^2 \log^{1+o(1)} k)$:

- Compute $\sin(z)/\cos(z)$ as formal power series
- Compute B_{2k} by approximating $\zeta(2k)$
  Allows computing B_{2k} without $B_2, \dots, B_{2k-2}$.
- Harvey (2010): Multimodular algorithm.
  Compute B_k mod p for many primes p independently.
  Combine results via *Chinese Remainder Theorem*.

Harvey: High-Level Algorithm

Given input k. compute $B_k = N_k / D_k$ as follows:

- Compute "enough" primes via prime sieve.

Harvey: High-Level Algorithm

Given input k. compute $B_k = N_k / D_k$ as follows:

- Compute "enough" primes via prime sieve.
- Compute denominator via *von Staudt–Clausen*: $D_k = \prod_{(p-1)|k} p$

Harvey: High-Level Algorithm

Given input k. compute $B_k = N_k / D_k$ as follows:

- Compute "enough" primes via prime sieve.
- Compute denominator via *von Staudt–Clausen*: $D_k = \prod_{(p-1)|k} p$
- Compute precise approximation of $\log_2 N_k$

Harvey: High-Level Algorithm

Given input k. compute $B_k = N_k / D_k$ as follows:

- Compute "enough" primes via prime sieve.
- Compute denominator via *von Staudt–Clausen*: $D_k = \prod_{(p-1)|k} p$
- Compute precise approximation of $\log_2 N_k$
- Compute set of primes P such that $\prod P > N_k$

Harvey: High-Level Algorithm

Given input k. compute $B_k = N_k / D_k$ as follows:

- Compute "enough" primes via prime sieve.
- Compute denominator via *von Staudt–Clausen*: $D_k = \prod_{(p-1)|k} p$
- Compute precise approximation of $\log_2 N_k$
- Compute set of primes P such that $\prod P > N_k$
- Compute N_k mod p for each $p \in P$

Harvey: High-Level Algorithm

Given input k. compute $B_k = N_k / D_k$ as follows:

- Compute "enough" primes via prime sieve.
- Compute denominator via *von Staudt–Clausen*: $D_k = \prod_{(p-1)|k} p$
- Compute precise approximation of $\log_2 N_k$
- Compute set of primes P such that $\prod P > N_k$
- Compute N_k mod p for each $p \in P$
- Use CRT to compute N_k mod $\prod P$

Harvey: High-Level Algorithm

Given input k. compute $B_k = N_k / D_k$ as follows:

- Compute “enough” primes via prime sieve.
- Compute denominator via *von Staudt–Clausen*: $D_k = \prod_{(p-1)|k} p$
- Compute precise approximation of $\log_2 N_k$
- Compute set of primes P such that $\prod P > N_k$
- Compute N_k mod p for each $p \in P$
- Use CRT to compute N_k mod $\prod P$
- Read off N_k

Harvey: Computation modulo p

What dictates the performance: Computing B_k mod p for “nice” primes p (where $2^k \not\equiv_p 1$).

Harvey: Computation modulo p

What dictates the performance: Computing B_k mod p for “nice” primes p (where $2^k \not\equiv_p 1$).

Each such computation boils down to:

- An outer for-loop with few iterations

Harvey: Computation modulo p

What dictates the performance: Computing B_k mod p for “nice” primes p (where $2^k \not\equiv_p 1$).

Each such computation boils down to:

- An outer for-loop with few iterations
- An inner for-loop with many iterations

Harvey: Computation modulo p

What dictates the performance: Computing B_k mod p for "nice" primes p (where $2^k \not\equiv_p 1$).

Each such computation boils down to:

- An outer for-loop with few iterations
- An inner for-loop with many iterations
- In the loop body:

Harvey: Computation modulo p

What dictates the performance: Computing $B_k \bmod p$ for “nice” primes p (where $2^k \not\equiv_p 1$).

Each such computation boils down to:

- An outer for-loop with few iterations
- An inner for-loop with many iterations
- In the loop body:
  - Compute next 64 bits of binary expansion of $1/p$

Harvey: Computation modulo p

What dictates the performance: Computing B_k mod p for “nice” primes p (where $2^k \not\equiv_p 1$).

Each such computation boils down to:

- An outer for-loop with few iterations
- An inner for-loop with many iterations
- In the loop body:
  - Compute next 64 bits of binary expansion of $1/p$
  - For each 8-bit block $b_0, \ldots, b_7$, add some value to a table entry $t[i, b_i]$

Harvey: Computation modulo p

What dictates the performance: Computing B_k mod p for “nice” primes p (where $2^k \not\equiv_p 1$).

Each such computation boils down to:

- An outer for-loop with few iterations
- An inner for-loop with many iterations
- In the loop body:
  - Compute next 64 bits of binary expansion of $1/p$
  - For each 8-bit block $b_0, \ldots, b_7$, add some value to a table entry $t[i, b_i]$
- Afterwards: determine final result as a weighted sum of the table entries

Harvey: Computation modulo p

What dictates the performance: Computing $B_k \bmod p$ for "nice" primes p (where $2^k \not\equiv_p 1$).

Each such computation boils down to:

- An outer for-loop with few iterations
- An inner for-loop with many iterations
- In the loop body:
  - Compute next 64 bits of binary expansion of $1/p$
  - For each 8-bit block $b_0, \ldots, b_7$, add some value to a table entry $t[i, b_i]$
- Afterwards: determine final result as a weighted sum of the table entries

Important: Inner loop body as cheap as possible.

Harvey: Computation modulo p

What dictates the performance: Computing B_k mod p for "nice" primes p (where $2^k \not\equiv_p 1$).

Each such computation boils down to:

- An outer for-loop with few iterations
- An inner for-loop with many iterations
- In the loop body:
  - Compute next 64 bits of binary expansion of $1/p$
  - For each 8-bit block $b_0, \ldots, b_7$, add some value to a table entry $t[i, b_i]$
- Afterwards: determine final result as a weighted sum of the table entries

Important: Inner loop body as cheap as possible.

Size of blocks chosen so that table fits into L1d cache.

Refinement

Refinement

Isabelle Refinement Framework and Isabelle-LLVM by Lammich:

- Start with a high-level view of the algorithm:
  - `nat`, `int` instead of `uint32_t`, `int32_t` etc.
  - Use abstract mathematical notions like “smallest prime factor of n”, “$\mathrm{ord}_{\mathbb{Z}/p\mathbb{Z}}(n)$” without worrying about how to compute them

Refinement

Isabelle Refinement Framework and Isabelle-LLVM by Lammich:

- Start with a high-level view of the algorithm:
  - `nat`, `int` instead of `uint32_t`, `int32_t` etc.
  - Use abstract mathematical notions like "smallest prime factor of n", "$\mathrm{ord}_{\mathbb{Z}/p\mathbb{Z}}(n)$" without worrying about how to compute them
- Refine down to more concrete implementations, e.g. for/while loops to compute prime sieve

Refinement

Isabelle Refinement Framework and Isabelle-LLVM by Lammich:

- Start with a high-level view of the algorithm:
  - `nat`, `int` instead of `uint32_t`, `int32_t` etc.
  - Use abstract mathematical notions like “smallest prime factor of n”, “$\mathrm{ord}_{\mathbb{Z}/p\mathbb{Z}}(n)$” without worrying about how to compute them
- Refine down to more concrete implementations, e.g. for/while loops to compute prime sieve
- Data refinement to fixed-width machine words (adding assumptions as needed)

Refinement

Isabelle Refinement Framework and Isabelle-LLVM by Lammich:

- Start with a high-level view of the algorithm:
  - `nat`, `int` instead of `uint32_t`, `int32_t` etc.
  - Use abstract mathematical notions like "smallest prime factor of n", "$\mathrm{ord}_{\mathbb{Z}/p\mathbb{Z}}(n)$" without worrying about how to compute them
- Refine down to more concrete implementations, e.g. for/while loops to compute prime sieve
- Data refinement to fixed-width machine words (adding assumptions as needed)

Example applications:

- IsaSAT by Fleury: fully verified SAT solver
- Lammich: verified sorting algorithms on par with C++ standard library

Refinement example: Factor Cache

On the abstract level:

```
definition "smallest_divisor n =
  (if n < 2 ∨ prime n then 0 else LEAST d. d ≠ 1 ∧ d dvd n)"
```

Refinement example: Factor Cache

On the abstract level:

```
definition "smallest_divisor n =
  (if n < 2 ∨ prime n then 0 else LEAST d. d ≠ 1 ∧ d dvd n)"

definition factor_cache_impl :: "nat ⇒ (nat ⇒ nat) nres" where ...
```

Refinement example: Factor Cache

On the abstract level:

```
definition "smallest_divisor n =
  (if n < 2 ∨ prime n then 0 else LEAST d. d ≠ 1 ∧ d dvd n)"

definition factor_cache_impl :: "nat ⇒ (nat ⇒ nat) nres" where ...

lemma "factor_cache_impl N ≤ SPEC (λa. ∀k<N. a k = smallest_divisor k)"
```

Refinement example: Factor Cache

On the abstract level:

```
definition "smallest_divisor n =
  (if n < 2 ∨ prime n then 0 else LEAST d. d ≠ 1 ∧ d dvd n)"

definition factor_cache_impl :: "nat ⇒ (nat ⇒ nat) nres" where ...

lemma "factor_cache_impl N ≤ SPEC (λa. ∀k<N. a k = smallest_divisor k)"
```

First refinement: Replace function with list; record only entries for odd indices

```
definition factor_cache_impl2 :: "nat ⇒ nat list nres" where ...
```

Refinement example: Factor Cache

On the abstract level:

definition "smallest_divisor n =
(if n < 2 ∨ prime n then 0 else LEAST d. d ≠ 1 ∧ d dvd n)"

definition factor_cache_impl :: "nat ⇒ (nat ⇒ nat) nres" **where** ...

lemma "factor_cache_impl N ≤ SPEC (λa. ∀k<N. a k = smallest_divisor k)"

First refinement: Replace function with list; record only entries for odd indices

definition factor_cache_impl2 :: "nat ⇒ nat list nres" **where** ...

lemma "3 ≤ N ⟹ factor_cache_impl2 N ≤⇓(fc'_rel N) (factor_cache_impl N)"

Refinement example: Factor Cache

On the abstract level:

definition "smallest_divisor n =
  (if n < 2 ∨ prime n then 0 else LEAST d. d ≠ 1 ∧ d dvd n)"

definition factor_cache_impl :: "nat ⇒ (nat ⇒ nat) nres" **where** ...

lemma "factor_cache_impl N ≤ SPEC (λa. ∀k<N. a k = smallest_divisor k)"

First refinement: Replace function with list; record only entries for odd indices

definition factor_cache_impl2 :: "nat ⇒ nat list nres" **where** ...

lemma "3 ≤ N ⟹ factor_cache_impl2 N ≤⇓(fc'_rel N) (factor_cache_impl N)"

Second refinement: Replace lists with arrays and *nat* with *word*:

sepref_def factor_cache_ll_impl :: "'w word ⇒ ('w word ptr × 'w word) llM" **where** ...

Refinement example: Factor Cache

On the abstract level:

definition "smallest_divisor n =
  (if n < 2 ∨ prime n then 0 else LEAST d. d ≠ 1 ∧ d dvd n)"

definition factor_cache_impl :: "nat ⇒ (nat ⇒ nat) nres" **where** ...

lemma "factor_cache_impl N ≤ SPEC (λa. ∀k<N. a k = smallest_divisor k)"

First refinement: Replace function with list; record only entries for odd indices

definition factor_cache_impl2 :: "nat ⇒ nat list nres" **where** ...

lemma "3 ≤ N ⟹ factor_cache_impl2 N ≤⇓(fc'_rel N) (factor_cache_impl N)"

Second refinement: Replace lists with arrays and *nat* with *word*:

sepref_def factor_cache_ll_impl :: "'w word ⇒ ('w word ptr × 'w word) llM" **where** ...

lemma "(factor_cache_ll_impl, factor_cache_impl2)
  ∈ unat_assn^k $\rightarrow_a$ array_assn unat_assn $\times_a$ unat_assn"

Final correctness theorem

```
llvm_htriple
(
  ↿ll_pto okX okp ∧* ↿ll_pto numX nump ∧* ↿ll_pto denomX denomp ∧*
  ssize_assn thr thri ∧* ssize_assn depth depthi ∧* numnat_assn k ki
)

(bern_crt_impl_wrapper okp nump denomp thri depthi ki)

(λ_. EXS oki numi denomi num denom.
  ↿ll_pto oki okp ∧* ↿ll_pto numi nump ∧* ↿ll_pto denomi denomp ∧*
  mpzb_assn num numi ∧* mpzb_assn denom denomi ∧*
  ↑((oki ≠ 0 ⟶ denom = int (bernoulli_denom k) ∧ num = bernoulli_num k) ∧
    (k ≤ 105946388 ⟶ oki ≠ 0))
)
```

Final correctness theorem

```
llvm_htriple
(
  ↿ll_pto okX okp ∧* ↿ll_pto numX nump ∧* ↿ll_pto denomX denomp ∧*
  ssize_assn thr thri ∧* ssize_assn depth depthi ∧* numnat_assn k ki
)

(bern_crt_impl_wrapper okp nump denomp thri depthi ki)

(λ_. EXS oki numi denomi num denom.
  ↿ll_pto oki okp ∧* ↿ll_pto numi nump ∧* ↿ll_pto denomi denomp ∧*
  mpzb_assn num numi ∧* mpzb_assn denom denomi ∧*
  ↑((oki ≠ 0 ⟶ denom = int (bernoulli_denom k) ∧ num = bernoulli_num k) ∧
    (k ≤ 105946388 ⟶ oki ≠ 0))
)
```

Challenges

Lots of algorithmic components

- Prime sieve, factoring integers

Lots of algorithmic components

- Prime sieve, factoring integers Done.

Lots of algorithmic components

- Prime sieve, factoring integers Done.
- Group-theoretic computations in $\mathbb{Z}/p\mathbb{Z}$: find generators, compute $\mathrm{ord}(x)$

Lots of algorithmic components

- Prime sieve, factoring integers Done.
- Group-theoretic computations in $\mathbb{Z}/p\mathbb{Z}$: find generators, compute $\mathrm{ord}(x)$ Done. (using factorisation)

Lots of algorithmic components

- Prime sieve, factoring integers Done.
- Group-theoretic computations in $\mathbb{Z}/p\mathbb{Z}$: find generators, compute $\mathrm{ord}(x)$ Done. (using factorisation)
- Efficient computations modulo p (e.g. Montgomery form)

Lots of algorithmic components

- Prime sieve, factoring integers Done.
- Group-theoretic computations in $\mathbb{Z}/p\mathbb{Z}$: find generators, compute $\mathrm{ord}(x)$ Done. (using factorisation)
- Efficient computations modulo p (e.g. Montgomery form) Done.

Lots of algorithmic components

- Prime sieve, factoring integers Done.
- Group-theoretic computations in $\mathbb{Z}/p\mathbb{Z}$: find generators, compute $\operatorname{ord}(x)$ Done. (using factorisation)
- Efficient computations modulo p (e.g. Montgomery form) Done.
- Floating-point computations for bounds etc.

Lots of algorithmic components

- Prime sieve, factoring integers Done.
- Group-theoretic computations in $\mathbb{Z}/p\mathbb{Z}$: find generators, compute $\mathrm{ord}(x)$
  Done. (using factorisation)
- Efficient computations modulo p (e.g. Montgomery form) Done.
- Floating-point computations for bounds etc.
  Replaced with fixed-point.

Lots of algorithmic components

- Prime sieve, factoring integers Done.
- Group-theoretic computations in $\mathbb{Z}/p\mathbb{Z}$: find generators, compute $\mathrm{ord}(x)$
  Done. (using factorisation)
- Efficient computations modulo p (e.g. Montgomery form) Done.
- Floating-point computations for bounds etc.
  Replaced with fixed-point.
- Computing the binary fraction expansion of $1/p$

Lots of algorithmic components

- Prime sieve, factoring integers Done.
- Group-theoretic computations in $\mathbb{Z}/p\mathbb{Z}$: find generators, compute $\mathrm{ord}(x)$
  Done. (using factorisation)
- Efficient computations modulo p (e.g. Montgomery form) Done.
- Floating-point computations for bounds etc.
  Replaced with fixed-point.
- Computing the binary fraction expansion of $1/p$ Done.

Lots of algorithmic components

- Prime sieve, factoring integers Done.
- Group-theoretic computations in $\mathbb{Z}/p\mathbb{Z}$: find generators, compute $\mathrm{ord}(x)$
  Done. (using factorisation)
- Efficient computations modulo p (e.g. Montgomery form) Done.
- Floating-point computations for bounds etc.
  Replaced with fixed-point.
- Computing the binary fraction expansion of $1/p$ Done.
- Fast Chinese Remaindering

Lots of algorithmic components

- Prime sieve, factoring integers Done.
- Group-theoretic computations in $\mathbb{Z}/p\mathbb{Z}$: find generators, compute $\mathrm{ord}(x)$
  Done. (using factorisation)
- Efficient computations modulo p (e.g. Montgomery form) Done.
- Floating-point computations for bounds etc.
  Replaced with fixed-point.
- Computing the binary fraction expansion of $1/p$ Done.
- Fast Chinese Remaindering
  Done. Using remainder trees.

Lots of algorithmic components

- Prime sieve, factoring integers Done.
- Group-theoretic computations in $\mathbb{Z}/p\mathbb{Z}$: find generators, compute $\mathrm{ord}(x)$ Done. (using factorisation)
- Efficient computations modulo p (e.g. Montgomery form) Done.
- Floating-point computations for bounds etc. Replaced with fixed-point.
- Computing the binary fraction expansion of $1/p$ Done.
- Fast Chinese Remaindering Done. Using remainder trees.
- Arbitrary-precision integers

Lots of algorithmic components

- Prime sieve, factoring integers Done.
- Group-theoretic computations in $\mathbb{Z}/p\mathbb{Z}$: find generators, compute $\mathrm{ord}(x)$
  Done. (using factorisation)
- Efficient computations modulo p (e.g. Montgomery form) Done.
- Floating-point computations for bounds etc.
  Replaced with fixed-point.
- Computing the binary fraction expansion of $1/p$ Done.
- Fast Chinese Remaindering
  Done. Using remainder trees.
- Arbitrary-precision integers
  Out of scope. We use GMP as trusted component.

Lots of algorithmic components

- Prime sieve, factoring integers Done.
- Group-theoretic computations in $\mathbb{Z}/p\mathbb{Z}$: find generators, compute $\mathrm{ord}(x)$
  Done. (using factorisation)
- Efficient computations modulo p (e.g. Montgomery form) Done.
- Floating-point computations for bounds etc.
  Replaced with fixed-point.
- Computing the binary fraction expansion of $1/p$ Done.
- Fast Chinese Remaindering
  Done. Using remainder trees.
- Arbitrary-precision integers
  Out of scope. We use GMP as trusted component. Future work?

Lots of mathematical background

- Definition of Bernoulli numbers and basic properties

Lots of mathematical background

- Definition of Bernoulli numbers and basic properties Already there.

Lots of mathematical background

- Definition of Bernoulli numbers and basic properties Already there.
- Bounds for Bernoulli numbers

Lots of mathematical background

- Definition of Bernoulli numbers and basic properties Already there.
- Bounds for Bernoulli numbers Easy.

Lots of mathematical background

- Definition of Bernoulli numbers and basic properties Already there.
- Bounds for Bernoulli numbers Easy.
- Kummer/Voronoi congruence and Harvey's tweaks

Lots of mathematical background

- Definition of Bernoulli numbers and basic properties Already there.
- Bounds for Bernoulli numbers Easy.
- Kummer/Voronoi congruence and Harvey's tweaks Done.

Lots of mathematical background

- Definition of Bernoulli numbers and basic properties Already there.
- Bounds for Bernoulli numbers Easy.
- Kummer/Voronoi congruence and Harvey's tweaks Done.
- Concrete bounds for the Chebyshev ϑ function:

$$\vartheta(x) = \sum_{p \le x} \ln p \ge 0.82x \quad \text{for } x \ge 97$$

Lots of mathematical background

- Definition of Bernoulli numbers and basic properties Already there.
- Bounds for Bernoulli numbers Easy.
- Kummer/Voronoi congruence and Harvey's tweaks Done.
- Concrete bounds for the Chebyshev ϑ function:

$$\vartheta(x) = \sum_{p \le x} \ln p \ge 0.82x \quad \text{for } x \ge 97$$

Needed for our a-priori estimate of how many primes to sieve.

Lots of mathematical background

- Definition of Bernoulli numbers and basic properties Already there.
- Bounds for Bernoulli numbers Easy.
- Kummer/Voronoi congruence and Harvey's tweaks Done.
- Concrete bounds for the Chebyshev ϑ function:

$$\vartheta(x) = \sum_{p \leq x} \ln p \geq 0.82x \quad \text{for } x \geq 97$$

  Needed for our a-priori estimate of how many primes to sieve.
  Done.

Lines of Code

Component	LOC
Voronoi/Kummer	2300
Prime bounds	1800

Lines of Code

Component	LOC
Voronoi/Kummer	2300
Prime bounds	1800
Fixed-point $\log_2$	1000
Binary fraction expansion	900
Montgomery multiplication	2300
Prime sieve, order, generators	2700
Fast Chinese Remaindering	3800
Other	900

Lines of Code

Component	LOC
Voronoi/Kummer	2300
Prime bounds	1800
Fixed-point $\log_2$	1000
Binary fraction expansion	900
Montgomery multiplication	2300
Prime sieve, order, generators	2700
Fast Chinese Remaindering	3800
Other	900

Component	LOC
Additions to `sepref` package	4300
GMP bindings	1700

Lines of Code

Component	LOC
Voronoi/Kummer	2300
Prime bounds	1800
Fixed-point $\log_2$	1000
Binary fraction expansion	900
Montgomery multiplication	2300
Prime sieve, order, generators	2700
Fast Chinese Remaindering	3800
Other	900

Component	LOC
Additions to `sepref` package	4300
GMP bindings	1700
Abstract algorithm	1500
Concrete algorithm	8500
Total	31700

Low-level optimisations

We can compute the binary fraction expansion of $1/p$ in 64-bit chunks by letting `bitbuf = 1` and then repeating

```
output ((bitbuf << 64) / p)
bitbuf = ((bitbuf << 64) % p)
```

Low-level optimisations

We can compute the binary fraction expansion of $1/p$ in 64-bit chunks by letting `bitbuf = 1` and then repeating

```
output ((bitbuf << 64) / p)
bitbuf = ((bitbuf << 64) % p)
```

But: Division is expensive. Therefore, we instead precompute a 128-bit fixed-point approximation `invp` of $1/p$ and compute

- quotient `(bitbuf << 64) / p` via `bitbuf_new = (invp * bitbuf) >> 64`

Low-level optimisations

We can compute the binary fraction expansion of $1/p$ in 64-bit chunks by letting `bitbuf = 1` and then repeating

```
output ((bitbuf << 64) / p)
bitbuf = ((bitbuf << 64) % p)
```

But: Division is expensive. Therefore, we instead precompute a 128-bit fixed-point approximation `invp` of $1/p$ and compute

- quotient `(bitbuf << 64) / p` via `bitbuf_new = (invp * bitbuf) >> 64`
- remainder `(bitbuf << 64) % p` via `-p * bitbuf_new`.

Low-level optimisations

We can compute the binary fraction expansion of $1/p$ in 64-bit chunks by letting `bitbuf = 1` and then repeating

```
output ((bitbuf << 64) / p)
bitbuf = ((bitbuf << 64) % p)
```

But: Division is expensive. Therefore, we instead precompute a 128-bit fixed-point approximation `invp` of $1/p$ and compute

- quotient `(bitbuf << 64) / p` via `bitbuf_new = (invp * bitbuf) >> 64`
- remainder `(bitbuf << 64) % p` via `-p * bitbuf_new`.

There is a small chance that the result is off-by-one, which we have to detect and correct accordingly.

Low-level optimisations

```
// bitbuf := 2 ^ 64 * x / p,   x := 2 ^ 64 * x % p
uint64_t bitbuf = (invp * (__uint128_t) x) >> 64;
x = -p * bitbuf;

// There is a small chance that our quotient is actually too small by 1; we detect this here.
if (__builtin_expect(x >= p, 0)) [[unlikely]] {
    bitbuf++;
    x -= p;
}
```

Low-level optimisations

Proving such low-level code correct requires

- understanding what the right high-level model is

Low-level optimisations

Proving such low-level code correct requires

- understanding what the right high-level model is
- figuring out preconditions, both abstractly and regarding overflow etc.

Low-level optimisations

Proving such low-level code correct requires

- understanding what the right high-level model is
- figuring out preconditions, both abstractly and regarding overflow etc.

Lessons learnt:

- Proving absence of overflow can be painful.

Low-level optimisations

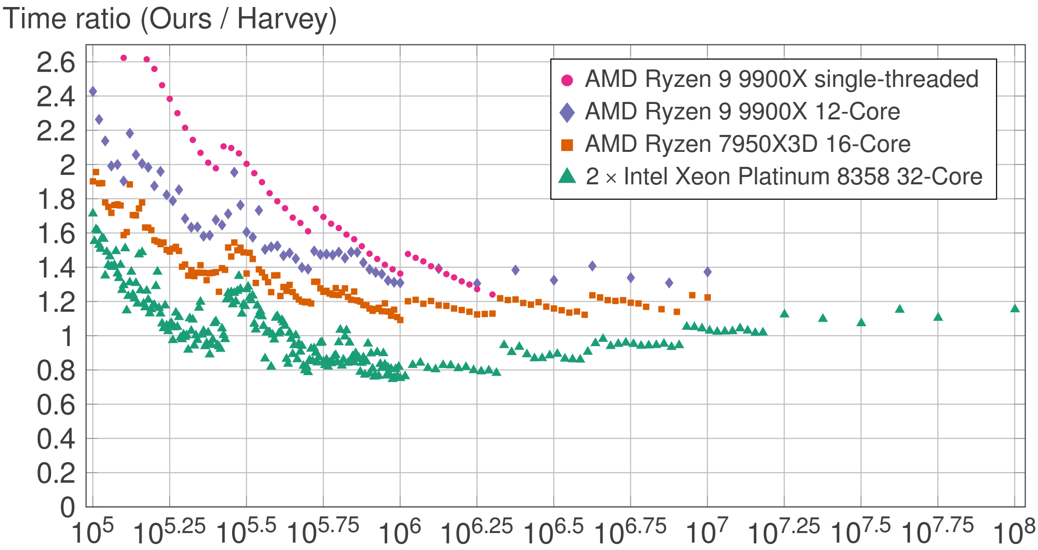
Proving such low-level code correct requires

- understanding what the right high-level model is
- figuring out preconditions, both abstractly and regarding overflow etc.

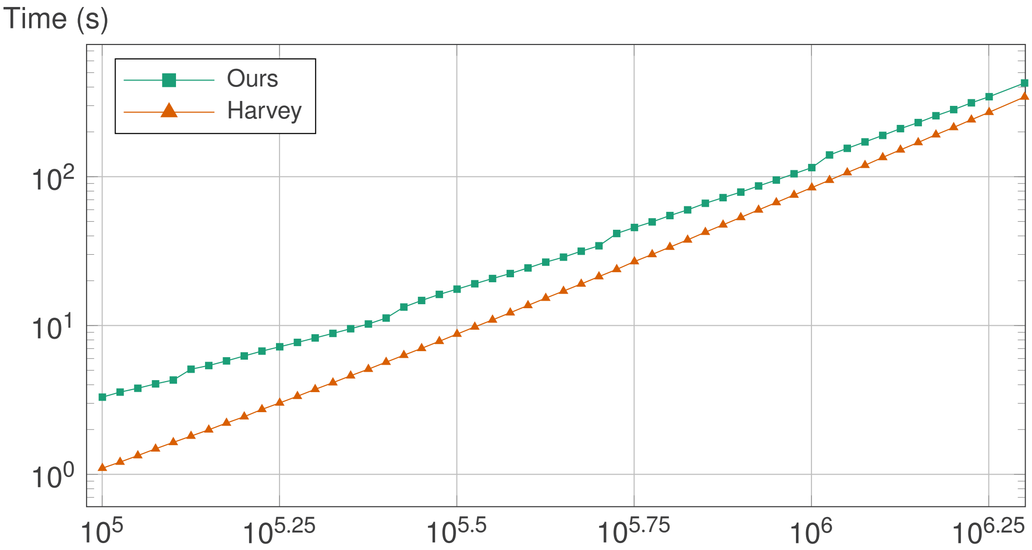
Lessons learnt:

- Proving absence of overflow can be painful.
- Advantage: Using an ITP helps you figure out the range in which the algorithm does not produce overflow.

Evaluation



universität innsbruck UNIVERSITY OF TWENTE.


Time (s)
Ours
Harvey
10^2
10^1
10^0
10^5
$10^{5.25}$
$10^{5.5}$
$10^{5.75}$
10^6
$10^{6.25}$

Conclusion

Conclusion

We verified a complex and challenging mathematical algorithm all the way down to LLVM code.

Conclusion

We verified a complex and challenging mathematical algorithm all the way down to LLVM code.

Made various additions to Isabelle-LLVM; exposed some weak points (e.g. sharing read-only access among parallel threads).

Conclusion

We verified a complex and challenging mathematical algorithm all the way down to LLVM code.

Made various additions to Isabelle-LLVM; exposed some weak points (e.g. sharing read-only access among parallel threads).

Performance of resulting LLVM code not quite on par with Harvey's unverified C++ code, but quite close (especially for large inputs).

Conclusion

We verified a complex and challenging mathematical algorithm all the way down to LLVM code.

Made various additions to Isabelle-LLVM; exposed some weak points (e.g. sharing read-only access among parallel threads).

Performance of resulting LLVM code not quite on par with Harvey's unverified C++ code, but quite close (especially for large inputs).

Closing the gap would require in-depth microbenchmarking.

Conclusion

We verified a complex and challenging mathematical algorithm all the way down to LLVM code.

Made various additions to Isabelle-LLVM; exposed some weak points (e.g. sharing read-only access among parallel threads).

Performance of resulting LLVM code not quite on par with Harvey's unverified C++ code, but quite close (especially for large inputs).

Closing the gap would require in-depth microbenchmarking.

We are already *much* faster than Mathematica's `BernoulliB` algorithm!