Verifying an Efficient Algorithm for Computing Bernoulli Numbers

Manuel Eberl
University of Innsbruck, Austria

Peter Lammich
University of Twente, The Netherlands

Abstract

The Bernoulli numbers B_k are a sequence of rational numbers that is ubiquitous in mathematics, but difficult to compute efficiently (compared to e.g. approximating π).

In 2008, Harvey gave the currently fastest known practical way for computing them: his algorithm computes $B_k \bmod p$ in time $O(p \log^{1+o(1)} p)$. By doing this for $O(k)$ many small primes p in parallel and then combining the results with the Chinese Remainder Theorem, one recovers the value of B_k as a rational number in $O(k^2 \log^{2+o(1)} k)$ time. One advantage of this approach is that the expensive part of the algorithm is highly parallelisable and has very low memory requirements. This algorithm still holds the world record with its computation of B_{10^8}.

We give a verified efficient LLVM implementation of this algorithm. This was achieved by formalising the necessary mathematical background theory in Isabelle/HOL, proving an abstract version of the algorithm correct, and refining this abstract version down to LLVM using Lammich's *Isabelle-LLVM* framework, including many low-level optimisations. The performance of the resulting LLVM code is comparable with Harvey's original unverified and hand-optimised C++ implementation.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases Bernoulli numbers, LLVM, verification, Isabelle, Chinese remainder theorem, modular arithmetic, Montgomery arithmetic

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Supplementary Material *Isabelle development on Zenodo*: 10.5281/zenodo.15749805

Acknowledgements The computational results presented here have been achieved (in part) using the LEO HPC infrastructure of the University of Innsbruck.

1 Introduction

The Bernoulli numbers B_k are a sequence of rational numbers that is ubiquitous in mathematics and has connections to, among other things: the closed-form expression for $\sum_{i=1}^{n} i^k$ (Faulhaber's formula), regular primes and cyclotomic fields, the combinatorics of alternating permutations, the Riemann zeta function, the Euler–Maclaurin summation formula, and the Maclaurin series of the tangent and cosecant functions.

Algorithmically, they are quite difficult to compute efficiently: Bernoulli himself only computed them up to $B_{10} = 5/66$ in the 17th century. Euler extended this up to $B_{30} = 8615841276005/14322$ in 1748, Adams up to B_{124} in 1877 [1], and Lehmer up to B_{220} in 1936 [22] – all by hand.

Interestingly, it is often claimed that the first non-trivial computer program ever written was Lovelace's *Note G*, published in 1843. This was an algorithm designed for Babbage's *Analytical Engine* (which was sadly never built) to compute Bernoulli numbers.[1] [9]

[1] Accounts differ on whether it was Lovelace or Babbage himself who authored the program, but the prevailing view seems to be that it was Lovelace.

The first actual use of a digital computer (that we are aware of) in computing Bernoulli numbers (and the first big step forward after Lehmer in 1936) was when Knuth and Buckholtz gave a table of the values up to B_{836} in 1967 [16] (the numerator of B_{836} already has 1421 decimal digits). The first large jump to higher indices happened in 1996, when Fee and Plouffe [10] computed the single value $B_{750{,}000}$ and a few others up to $B_{5{,}000{,}000}$ in the following years, using an entirely different method based on approximating the Riemann zeta function. Unlike previous methods, this method allowed computing a value B_k without knowing the preceding values $B_0, \ldots, B_{k-1}$.

In 2010, Harvey [14] computed the value of B_{10^8} using an entirely different method: He gave a fast algorithm to compute B_k modulo a prime p. This can then be run in parallel for many different primes, followed by an invocation of the Chinese Remainder Theorem to reconstruct the value of B_k as a rational number. While this approach is not asymptotically faster than the one based on the zeta function, it is much faster in practice, as Harvey demonstrated. To this day, Harvey's algorithm is still the state of the art for practical computation of a single Bernoulli number, and his record of B_{10^8} still stands. His algorithm takes time $O(k^2 \log^{2+o(1)} k)$ to compute B_k.[2]

In this paper, we verify Harvey's algorithm in Isabelle/HOL. Using the Isabelle Refinement Framework [21] and its parallel LLVM back end [20], our verification spans from a definition of Bernoulli numbers down to efficient LLVM code with performance comparable to Harvey's original algorithm. As a side product, we also verified several other interesting numerical algorithms and number-theoretic results, such as prime sieves and fast Chinese remaindering.

Our trusted code base consists of Isabelle, the Isabelle LLVM code generator, and the LLVM compiler. Only in the reconstruction phase, we additionally rely on correct integer operations in the GNU Multiple Precision Arithmetic Library [12].

2 Mathematical Background

The Bernoulli numbers B_k are defined as the coefficients of the exponential generating function $B(z) = z/(\exp(z) - 1)$ or, following an alternative convention, $z/(1 - \exp(-z))$. The only difference between the two conventions is that the first one yields $B_1 = -\frac{1}{2}$, and the second one yields $B_1 = \frac{1}{2}$. We shall adopt the first convention. We use the Isabelle/HOL formalisation of Bernoulli numbers in the Archive of Formal Proofs (AFP) by Bulwahn and Eberl [4], which already contains many useful results on Bernoulli numbers.

▶ Remark 1 (Notation). We shall write N_k and D_k for the numerator and denominator of B_k, respectively.

Both in this presentation and in our Isabelle proofs, we use the notation $x \equiv_n y$. If x, y are integers, the meaning is that $x \bmod n = y \bmod n$ or, equivalently, $n \mid (x - y)$. The corresponding notation in Isabelle/HOL is $[x = y]\ (\text{mod}\ n)$.

We extend this definition to rational numbers in the natural way: We define $\frac{a}{b} \bmod n = ab^{-1} \bmod n$, where b^{-1} denotes a modular inverse of b modulo n. Note that this is only well-defined if the denominators are coprime to n. In Isabelle, we use the notation $x \operatorname{qmod} n$ for the infix operator 'qmod' and $[x = y]\ (\text{qmod}\ n)$ for the congruence relation.

The notation $\operatorname{ord}_n(x)$ denotes the multiplicative order of the element x in the ring $\mathbb{Z}/n\mathbb{Z}$, i.e. the smallest positive integer k such that $x^k \equiv_n 1$.

[2] For comparison, π has been approximated to trillions of digits and is much easier to compute ($O(n \log^3 n)$ time for n digits with Chudnovsky's formula and binary splitting). [26]

Lastly, as is common in number theory, any sum or product with an index variable p is meant to be interpreted with p being restricted to the prime numbers.

The sequence of Bernoulli numbers shows an obvious pattern: if $k > 1$ then $B_k = 0$ if k is odd, $B_k > 0$ if $k \equiv_4 2$, and $B_k < 0$ if $k \equiv_4 0$. The vanishing of B_k for odd $k > 1$ is easily proved from the formal power series given above. The result about the sign of B_k for even $k > 1$ is obtained from a well-known connection between B_k and the Riemann zeta function:

▶ **Theorem 2** (The connection between B_k and ζ). *If $k > 0$, we have:*

$$\zeta(2k) = \sum_{n\geq 1} n^{-2k} = \frac{(-1)^{k+1}(2\pi)^{2k}B_{2k}}{2(2k)!}$$

Proof. We apply the Residue Theorem to the integral $\oint_{R_m} B(z)/z^{2k+1}\,\mathrm{d}z$. Here, R_m is a rectangle of width $2m$ and height $2(2m+1)\pi$ centred around the origin. Letting $m \to \infty$ establishes our claim. ◀

The next important ingredient gives us an easy way to compute the denominator of B_k:

▶ **Theorem 3** (Von Staudt–Clausen theorem). *If $k \geq 2$ is even, then $D_k = \prod_{(p-1)\mid k} p$.*

We omit the proofs of this fact and of the following ones from this presentation for reasons of space. All results up to this point are part of the AFP entry by Bulwahn and Eberl [4].

The core of Harvey's algorithm are two closely related congruences involving Bernoulli numbers modulo a prime or a prime power:

▶ **Theorem 4** (Voronoi's congruence). *Let $k \geq 2$ be even and let $n > 0$ and a be coprime integers. Then:*

$$(a^k - 1)N_k \equiv_n k a^{k-1} D_k \sum_{1\leq m<n} m^{k-1} \left\lfloor \frac{ma}{n} \right\rfloor$$

▶ **Theorem 5** (Kummer's congruence). *Let p be a prime number and $e \geq 0$ and $k, k' \geq e+1$ be integers with k, k' even and $k \equiv_{p^{e-1}(p-1)} k'$ and $(p-1) \nmid k$. Then $B_k/k \equiv_{p^e} B_{k'}/k'$.*

Proof. The Isabelle proofs closely follow Cohen [5, Prop. 9.5.20, Cor. 9.5.25] and are relatively straightforward. ◀

These congruences were not previously available in the AFP and were formalised in the course of this project [7].

By setting $e := 1$ and $k' := k \bmod (p-1)$ in Kummer's congruence, we obtain the range-reduction congruence $B_k \equiv_p k/k' B_{k'}$. Thus we can w.l.o.g. assume that $2 \leq k \leq p-3$.

Voronoi's congruence gives us a closed-form expression for computing B_k mod p. Harvey tweaks this congruence to obtain the following:

▶ **Theorem 6** (Voronoi's congruence, Harvey's first version). *Let $p \geq 5$ be prime and $2 \leq k \leq p-3$ and $1 < c < p$ with $c^k \not\equiv_p 1$. Then:*

$$B_k \equiv_p \frac{k}{1-c^k} \sum_{1\leq m<p} m^{k-1} h(m) \qquad \text{where } h(m) = \frac{m - c((mc^{-1}) \bmod p)}{p} + \frac{c-1}{2} \in \mathbb{Q}$$

Here, c^{-1} denotes a modular inverse of c modulo p. Note that $h(m)$ is either an integer or a half-integer, depending on whether c is even or odd.

We will later pick a convenient value for the parameter c in this theorem.

3 Computing N_k Modulo a Prime

The main part of Harvey's algorithm is to gather 'modular information', i.e. to compute $B_k \bmod p$ for many different primes p. Note that since $p \mid D_k$ if $(p-1) \mid k$, the term $B_k \bmod p$ is actually undefined if $(p-1) \mid k$, which is why Harvey does not use such primes.

We on the other hand decided to instead compute $N_k \bmod p$, which is well-defined for all primes. Since $D_k \bmod p$ is easy to compute via von Staudt–Clausen, it is easy to convert between $N_k \bmod p$ and $B_k \bmod p$ if the latter is well-defined.

Depending on p and k, we use one of three different methods to compute $N_k \bmod p$:

Case 1: If $(p-1) \nmid k$, we use a version of Theorem 6 to compute $B_k \bmod p$ and then multiply the result with $D_k \bmod p$ to obtain $N_k \bmod p$. To avoid big-integer arithmetic, we do the latter part by simply multiplying with $q \bmod p$ for every prime q with $(q-1) \mid k$. There are two sub-cases for how we compute $B_k \bmod p$:

- **Case 1.1** If $2^k \not\equiv_p 1$, we use a highly optimised version of Theorem 6 which we refer to as `harvey_fastsum` in our Isabelle formalisation.
- **Case 1.2** If $2^k \equiv_p 1$, we fall back to Harvey's *Algorithm 1*, which is a much less optimised version of Theorem 6. In our Isabelle formalisation, we refer to this as `harvey_slowsum`.

Case 2 If $(p-1) \mid k$, we recall that $B_k \bmod p$ is not well-defined. However, it is easy to show that in this case we have $N_k \equiv_p -D_k/p$, which is easy to compute.

It would in principle be possible to only use the primes that fall into Case 1.1 in order to keep the algorithm simpler and make it slightly faster. However, little is known about their distribution in practice, which makes it impossible to find an a-priori bound for how many primes we will need. Omitting Case 2 primes on the other hand, as Harvey does, is easy to do but brings no real benefit.

Since Case 1.1 is by far the most frequent and most optimised one, we dedicate the remainder of this section to it. For Case 1.2, we refer the reader to Harvey's paper.

Harvey derives the congruence behind Case 1.1 from Theorem 6 by choosing $c = \frac{1}{2} \bmod p$. He then picks a generator g of $\mathbb{Z}/p\mathbb{Z}$ (also referred to as a primitive root modulo p) and, with some elementary algebra and clever reindexing, derives the following:

▶ **Theorem 7** (The `harvey_fastsum` congruence). *Let $p \geq 5$ be prime with $2^k \not\equiv_p 1$ and $2 \leq k \leq p-3$ and g a generator of $\mathbb{Z}/p\mathbb{Z}$. Let $n = \mathrm{ord}_p(2)/2$ if $\mathrm{ord}_p(2)$ is even and $n = \mathrm{ord}_p(2)$ otherwise. Let $m = (p-1)/(2n)$. Then:*

$$B_k \equiv_p \frac{k}{2(2^{-k}-1)} \sum_{0 \leq i < m} g^{i(k-1)} \sum_{0 \leq j < n} (2^{k-1})^j \sigma(p, 2^j g^i) \quad \text{where } \sigma(p,x) = \mathrm{sgn}(p - 2(x \bmod p))$$

Empirically, m is small for most primes.[3] Thus, just like Harvey, we focus on optimising the inner sum. We will process the inner sum in chunks of w' blocks of size w. Currently, we pick $w = w' = 8$ for hardware architecture reasons that will become clear later.

Concretely, generalise 2^{k-1} to x and g^i to s in the inner sum in Theorem 7 and let $n_1 = \lfloor n/(ww') \rfloor$ and $n_2 = n \bmod (ww')$. Then:

$$\sum_{0 \leq j < n}^{n-1} x^j \sigma(p, 2^j s) = \left(\sum_{0 \leq j < n_1 ww'} x^j \sigma(p, 2^j s) \right) + x^{n_1 ww'} \left(\sum_{0 \leq j < n_2} x^j \sigma(p, 2^{j + n_1 ww'} s) \right)$$

That is, we split the sum into a 'chunk-aligned' part, and a remainder.

[3] The mean value of m for the first 10^6 odd primes is approximately 9.80. Less than 3% of these primes have $m \geq 30$.

3.1 The Inner Sum in General

For presentation purposes, we start by describing the algorithm for the remainder, which works for arbitrary n. The algorithm for chunk-aligned n uses the same implementation techniques with the addition of a tabulation optimisation, which we describe in Section 3.2.

3.1.1 Abstract Algorithm

We compute the sum $\sum_{0\leq j<n} x^j\sigma(p,2^j s)$ for arbitrary n with a simple loop with two local variables z and s' that hold the current values of $x^j \bmod p$ and $2^j s \bmod p$, respectively.

We have $\sigma(p,2^j s) = \sigma(p, 2^j s \bmod p) = \sigma(p,s')$ and also $\sigma(p,s') = -1$ if $p \leq 2s'$ and $\sigma(p,s') = 1$ otherwise. This leads us to the following simple algorithm:

```
harvey_restsum1 p x s n = doN {
  (_,_,acc) ← for 0 n (λj (z,s',acc). doN {
    assert (s' ∈ {0..<p}); let s' = 2*s';
    if s'<p then return (z*x,s',acc+z) else return (z*x,s'-p,acc-z)
  }) (1,s,0);
  return acc }
```

This algorithm is phrased in the nondeterminism-error monad of the Isabelle Refinement Framework (IRF) [21]. It iterates over the values $j = 0, 1, \ldots, n-1$, maintaining the state (z, s', acc) with the invariants $z = x^j$ and $s' = 2^j s \bmod p$ and $acc \equiv_p \sum_{0\leq j'<j} x^{j'}\sigma(p,2^{j'}s)$.

Given the ideas above, we can prove the algorithm correct:

```
lemma harvey_restsum1_correct:
  assumes "s∈{0..<p}" "odd p"
  shows "harvey_restsum1 p x s n ≤ return ((∑j<n. x^j * σ p (2^j * s)))"
  (* proof elided *)
```

In words: assuming that $0 \leq s < p$ and that p is odd, our algorithm refines the program that just returns the desired sum.

If m and m' are programs, the refinement relation $m \leq m'$ means that every possible result of program m is also a possible result of program m', or m' fails. Moreover, m can only fail if m' fails. In the above lemma, m' is a deterministic program that does not fail, thus our algorithm does not fail and has either no results or the desired result. The case of having no results will only be excluded later, when we subsequently refine our algorithm to *Isabelle-LLVM*, which provably cannot have an empty set of results.

An assertion fails if the asserted predicate does not hold. In the context of stepwise refinement, assertions are a valuable tool to organise proof obligations: during the proof of the algorithm, we know that $s' < p$ (it follows from the invariant $s' = 2^j s \bmod p$). Making this knowledge explicit as an assertion allows us to use it later on, when we further refine the algorithm: since a failed algorithm is refined by any program, we can assume that the algorithm to be refined does not fail, and thus that the assertions hold. Also, as in unverified programming, assertions are an engineering tool, making flawed proof attempts fail early. For the sake of readability, we will often elide assertions from presented listings.

For this paper, we will elide most actual Isabelle proofs and rather describe their ideas. The Isabelle listings shown are slightly edited for presentation purposes. Lemma and function names coincide with the actual formalisation.

```
harvey_restsum2 W p' p x s n = doN {
  z←to_mont1 W p p' 1;
  (_,_,acc) ← for 0 n (λ_ (z,s',acc). doN {
    pl ← cast_u.op (2*W) p; s' ← cast_u.op (2*W) s'; s' ← mul_uuu.op (2*W) 2 s';
    z' ← mont_times1 W p p' z x;
    if s'<pl then doN {
      acc ← mont_add_relaxed1 W acc z; s' ← cast_u.op W s'; return (z',s',acc)
    } else doN {
      acc ← mont_diff_relaxed1 W p acc z;
      s' ← sub_uuu.op (2*W) s' pl; s' ← cast_u.op W s';
      return (z',s',acc) }
  }) (z,s,mont_zero_relaxed1 W);
  return acc }
```

Figure 1 Computing the inner sum using Montgomery form

3.1.2 Introducing Montgomery Form

In a next step, we use the Montgomery arithmetic (also known as REDC arithmetic) [23] for the accumulator and x^j. We also insert annotations for type casting. The algorithm is shown in Fig. 1. Here, W is the bit width for numeric operations (in our case 32). The accumulator is maintained as a double-width word that will not overflow during the loop. Thus, we can defer the reduction modulo p until after the loop.

Moreover, the code contains annotations for type casts, bit widths, and the types of operations. For example, the operation `mul_uuu.op (2*W) a b` multiplies its unsigned double-width ($2W$) arguments a and b, asserting that there is no overflow. While these annotations are not strictly necessary, they make the next refinement step simpler (cf. Sec. 3.1.3).

Some operations for the Montgomery form also require extra parameters apart from the bit width W. For example, the multiplication requires both the modulus p and p', its inverse modulo 2^W. The inverse p' is pre-computed once and passed as an extra parameter to the function. We show that this function is a refinement of the above `harvey_restsum1`:

```
lemma harvey_restsum2_refine:
  assumes "mont_ctxt_refine' W p p' ctxt" "n<2^W" "(xi,x)∈mont_rel ctxt"
  shows "harvey_restsum2 W p' p xi s n
    ≤ ⇓(mont_rel_relaxed ctxt (n+1)) (harvey_restsum1 p x s n)"
```

The first assumption states that `ctxt` is a Montgomery refinement context[4] for bit width W, modulus p and its inverse p'. The lemma then states that if n is in bounds, and xi is the Montgomery form of x, then `harvey_restsum2 W p' p xi s n` refines `harvey_restsum1 p x s n` wrt. the relation `mont_rel_relaxed ctxt (n+1)`. This relation indicates that the left value modulo p is the Montgomery form of the right value, and that the left value is less than $(n+1)p$. The counter $(n+1)$ is used to keep track of an upper bound for the accumulator, while we keep adding values $< p$ to it.

Note that we can easily prove the non-overflow assertions on this level. The assertion $s' < p$ from the previous refinement level helps here, as it allows us to assume $s' < p$.

[4] The formalisation of Montgomery form uses these contexts, while our algorithms use explicit parameters. These are independent design choices, which slightly clash here.

3.1.3 Refinement to LLVM

The last step uses the *sepref* [18] tool to refine to (an Isabelle model of) LLVM code.

```
sepref_def harvey_restsum_impl is "harvey_restsum2 32"
  :: "u32A * u32A * u32A * u32A * u64A → u64A"
  unfolding harvey_restsum2_def for_by_while
  apply (annot_unat_const "TYPE(64)", annot_uint_const "TYPE(32)")
  by sepref
```

The already proved bounds annotations make this proof easy: after unfolding the definition and unfolding the for loop into a while loop (which *sepref* can process), we annotate the number literals with their respective bit-width, and then invoke *sepref*. This generates LLVM code and a refinement lemma, stating that the parameters p, p', xi, s are implemented by unsigned 32-bit words (`u32A`), the parameter n is implemented by an unsigned 64-bit word (`u64A`), and the result is returned as an unsigned 64-bit word.

3.2 Further Optimisation of the Inner Sum

For the aligned part of the summation, we apply another crucial optimisation that is also described by Harvey. Instead of iterating through the sum in single steps, we iterate in chunks of size ww'. Note that $\sigma(p, 2^j s)$ being 1 or -1 is equivalent to the jth digit of the binary expansion of s/p being 0 or 1, which allows us to determine the values of $\sigma(p, 2^j s)$ for ww' consecutive values of j in a single step by computing the corresponding ww' bits of the binary expansion of s/p. For this we define

```
modf b x p i = (b^i * x) mod p
digf b x p i = (b * modf b x p i) div p
dsgn d = if d ≠ 0 then -1 else 1
blbits i j = digf (2 ^ (w * w')) s p i div 2 ^ (j * w) mod 2 ^ w
```

and rewrite the inner sum for multiples of ww' as

$$\sum_{0\le j<nww'} x^j \sigma(p, 2^j s) = \sum_{0\le j<nww'} x^j \mathrm{dsgn}(\mathrm{digf}(2, s, p, j))$$

We split the sum into chunks and blocks, and introduce the tabulation

$$= \sum_{0\le k<w} x^k \sum_{0\le j<w'} x^{jw} \sum_{0\le i<n} x^{iww'} \mathrm{dsgn}(\mathrm{blbits}(i,j) \mathbin{!} k)$$

$$= \sum_{0\le b<2^w} \left(\left(\sum_{0\le k<w} x^k b_k \right) \cdot \left(\sum_{0\le j<w'} x^{jw} \mathrm{tab}(j,b) \right) \right)$$

where $\cdot \mathbin{!} k$ selects the kth bit of its argument, such that $\mathrm{blbits}(i,j) \mathbin{!} k = \mathrm{digf}(2, iww' + jw + k)$. The table is defined as

$$\mathrm{tab}(j,b) = \sum_{0\le i<n} \begin{cases} x^{iww'} & \text{if } b = \mathrm{blbits}(i,j) \\ 0 & \text{otherwise} \end{cases}$$

Since n is typically much larger than w and 2^w, most of the computational work goes into computing the table, which our optimisations make cheap: in an outer loop, we obtain chunks of ww' bits of the binary expansion and incrementally maintain $x^{iww'}$. The inner loop only bit-shifts and masks these bits, indexes into the table, and adds $x^{iww'}$ to table

```
harvey_tab_mod2 invp p p' x s n = doN {
  let curs = s; curx ← to_mont1 32 p p' 1; xww' ← mont_exp 32 p p' x (w*w');
  let tab = 0; (* Initialise table to all 0s *)
  (curs,curx,tab) ← for 0 n (λi (curs,curx,tab). doN {
    (cbits,curs) ← invp_push_div_mod1 invp curs p; (* Get 64 more bits of s/p *)
    (_,tab) ← for 0 w' (λj (cbits,tab). doN {
      let bbits = cbits AND (2^w-1); cbits = cbits>>w; (* Obtain block bits *)
      (* Update table *)
      v ← mop_tab_lookup tab j bbits; v ← mont_add_relaxed1 32 v curx;
      tab ← mop_tab_upd tab j bbits v;
      return (cbits,tab)
    }) (cbits,tab);
    curx ← mont_times1 32 p p' curx xww'; (* Maintain curx *)
    return (curs,curx,tab)
  }) (curs,curx,tab);
  return tab }
```

Figure 2 Computation of the inner-sum table

entries. Additionally, the implementation makes similar optimisations to those we described in Sec. 3. In particular, we use the relaxed Montgomery form for the table entries to allow for a mostly branch free implementation of the inner loop.[5] Moreover, the size of the table ($w'2^w$ double-width machine words, i.e. 16 KiB in our current setting) is chosen such that it fits comfortably inside the L1d cache of a typical modern CPU.

To incrementally obtain the bits of the binary expansion, we maintain the value of modf, and observe that $\mathrm{modf}(b, x, p, i + 1) = (b \cdot \mathrm{modf}(b, x, p, i)) \bmod p$. However, instead of performing an expensive division by p, we apply a further optimisation: we define $p^- = \lfloor 2^{128}/p \rfloor$. Then, for $p < 2^{32}$ and $x < p$, we have $\lfloor 2^{64}x/p \rfloor \in \{\lfloor p^- x/2^{64} \rfloor, \lfloor p^- x/2^{64} \rfloor + 1\}$. Using this, we can reduce the division and modulo operation to a multiplication operation. To be able to detect the +1 case, we limit $p < 2^{31}$, and define:

```
invp_push_div_mod invp x p =
  let z = (invp*x)>>64; x' = (-z*p) AND (2^32-1) in
  if x'<p then (z,x') else ((z+1) AND (2^64-1),(x'-p) AND (2^32-1))
```

Here, `>>` denotes the left shift operation and `AND` the bitwise 'and'. With this, we get:

```
lemma compute_next_bits:
  assumes "invp = 2 ^ 128 div p" "curs = modf (2 ^ 64) s p i"
  assumes "0 ≤ s" "s < p" "p<2^31"
  shows "invp_push_div_mod 32 64 128 invp curs p
    = (digf (2 ^ 64) s p i, modf (2 ^ 64) s p (i+1))"
```

This shows how to obtain the next digit and maintain the value of modf. Note that $\lfloor 2^{128}/p \rfloor$ is precomputed outside the loop. Figure 2 displays the algorithm for computing the table. After some initialisation, the outer loop iterates over the chunks. In each iteration of the outer loop, the binary expansion of s/p is maintained and another 64 bits of digits are

[5] The computation of the binary expansion of s/p we describe below does use a branch to correct a possible 'off-by-one' error. However, this happens so rarely that due to branch prediction, the cost of this branch is negligible.

```
bernoulli_num_harvey_wrapper2 dfs p g op2 k = doN {
  p' ← mk_mont_N1' p; (* Compute p' *)
  r ← if p-1 dvd k then doN { (* Case 2: p-1 | k *)
    t ← prod_list_exc2 p p' p dfs; mont_diff1 p (mont_zero1) t
  } else doN { (* Case 1 *)
    t ← if k < p - 1 then harvey_select2 p' p op2 k g
    else doN { (* Apply range reduction *)
      let k' = k mod (p - 1); t ← harvey_select2 p' p op2 k' g;
      t2 ← to_mont1_relaxed p p' k; t ← mont_times1 p p' t t2;
      kinv' ← to_mont1 p p' k'; t3 ← mont_inv1 p p' kinv'; mont_times1 p p' t t3
    };
    t' ← prod_list2 p p' dfs; mont_times1 p p' t t' }; (* Multiply with D_k *)
  r ← of_mont1 p p' r; return r } (* Convert to normal numbers *)
```

Figure 3 Computation of a single remainder (slightly simplified)

obtained in `cbits`. The inner loop iterates over the blocks, obtains the current block bits, and updates the table. After the inner loop, the outer loop maintains the value `curx` of $x^{iww'}$.

3.3 Putting Things Together

Finally, we assemble the aligned inner sum, the computation of the remainder, and the fallback to the slower algorithm when $2^k \equiv_p 0$ (i.e. when $k \bmod \mathrm{ord}_p(2) = 0$) to obtain the algorithm `harvey_select2` with the following correctness theorem:

```
lemma harvey_select2_correct:
  assumes "mont_ctxt_refine' W p p' ctxt"
    and "even k" "¬(p-1) dvd k" "k ∈ {2..p - 1}" "prime p" "5≤p"
    and "op2 = ord p 2" "g < p" "residue_primroot p g"
  shows "harvey_select2 p' p op2 k g
      ≤ ⇓(mont_rel ctxt) (SPEC (λr. [r = bernoulli_rat k] (qmod p)))"
```

That is, under the listed preconditions, `harvey_select2` will return the Montgomery form of an integer r with $r \equiv_p B_k$.

Next, we address some of the preconditions. First, we compute $p' = p^{-1} \bmod 2^W$ via Hensel lifting. Next, as mentioned before, when $(p-1) \mid k$ we have $N_k \equiv_p -D_k/p$, which allows us to easily determine $N_k \bmod p$ directly. Moreover, when $k \geq p-1$, we apply the previously-mentioned range reduction based on Kummer's congruence. Finally, we multiply the result with D_k to obtain $N_k \bmod p$, and convert the result from Montgomery form to normal numbers.

The resulting algorithm is displayed in Fig. 3. In addition to the parameters p, g, *op2*, and k, it also takes the prime factorisation *dfs* of D_k. The function `prod_list_exc2` computes D_k/p by multiplying all prime factors except p. The correctness theorem is:

```
lemma bernoulli_num_harvey_wrapper2_correct:
  assumes "even k" "k≠0" "prime p" "3≤p" "p<2^31"
  assumes "dfs = denom_factors k" "residue_primroot p g" "g<p" "op2 = ord p 2"
  shows "bernoulli_num_harvey_wrapper2 dfs p g op2 k
          ≤ SPEC (λr. r = bernoulli_num k mod p)"
```

In words: the algorithm returns $N_k \bmod p$ for any even natural number $k \neq 0$ and any odd prime less than 2^{31}.

```
compute_primes_mods_est t d k = doN {
  (ok,Y) ← estimate_check_a_priori_bound k; (* A-priori bound *)
  if ¬ok then return (False, (0,[],[])) (* Out-of-bounds *)
  else doN {
    (fc,pc) ← mk_factor_prime_cache (max (k+2) Y); (* Prime sieve *)
    dfs ← filter_denom_factors pc k; (* Get factorisation of D_k *)
    (_,pc) ← adjust_primes2 dfs k pc; (* Precise bound *)
    pgos ← compute_harvey_pgos pc fc; (* Compute generator and ord 2 p *)
    result ← pm.pmap_par_array (max 2 t) d (k,dfs) pgos (length pgos);
    return (True, (length result,result,dfs)) } } (* Ret result and D_k factors *)
```

Figure 4 Parallel computation of the prime numbers and remainders

4 Computing B_k as a Rational Number

The full algorithm consists of a preprocessing phase to create a large array of independent 'tasks', each being represented with a triple $(p, g_p, \mathrm{ord}_p(2))$, then the main part of computing $B_k \bmod p$ for each of these, and finally a modular reconstruction phase to determine B_k as a rational number. Figure 4 shows a high-level view of the algorithm without the modular reconstruction phase.

First, we compute a rough a-priori upper bound Y for the largest prime that will be needed to gather enough modular information to reconstruct B_k. We also check that k is small enough to not cause overflows later, and otherwise return an error. Next, we use a simple sieving algorithm to compute all primes up to Y and a factor map fc that maps every composite number to a prime factor. Next, we compute the prime factors of D_k using the von Staudt–Clausen theorem. We then use this information to determine a good upper bound for $\log_2 |N_k|$ and drop the unnecessary primes from the list. For each remaining prime p, we then compute $\mathrm{ord}_p(2)$ and find a generator g_p of $\mathbb{Z}/p\mathbb{Z}$. We refer to the list of triples $(p, g_p, \mathrm{ord}_p(2))$ as the 'task list' (or, in the Isabelle formalisation, the 'pgo list'). Finally, we compute the modular information for each of the primes in parallel and return the result as well as the prime factorisation of D_k.

In the following sections, we will explain each of these steps as well as the modular reconstruction step in some detail.

4.1 A-priori bound

The function `estimate_check_a_priori_bound k` (cf. Fig. 4) returns a pair $(ok, Y + 1)$. If ok = True, then Y is an upper bound for the largest prime number that will be needed to reconstruct N_k. If ok = False, we cannot guarantee that our 32-bit arithmetic will not overflow, and we abort the algorithm. The first ingredient to compute Y is a bound on B_k:

▶ **Theorem 8** (Upper bound for B_k). *Let $k \geq 4$ be an even integer. Then*

$$\log_2 |B_k| \leq (k + \tfrac{1}{2}) \log_2 k - c_1 k + c_2$$

where $c_1 = \frac{1}{\ln 2} + \log_2 \pi + 1 \approx 4.0942$ and $c_2 = \frac{3}{2} + \frac{9}{2} \log_2 \pi - \log_2 90 + \frac{1}{48 \ln 2} \approx 2.4699$.

Proof. We use Theorem 2 to express $|B_k|$ in terms of $\zeta(k)$. The monotonicity of $\zeta(x)$ for real $x > 1$ implies $\zeta(k) \leq \zeta(4) = \pi^4/90 \approx 1.082$. Combining this with Stirling's inequality $\ln(k!) \leq \frac{1}{2}\ln(2\pi k) + k \ln k - k + \frac{1}{12k}$ and noting that $\frac{1}{12k} \leq \frac{1}{48}$ yields the result. ◀

We can convert this into a bound on N_k by combining it with the von Staudt–Clausen theorem: it implies $D_k \mid 2(2^k - 1)$ and therefore $\log_2 D_k < k + 1$. Lastly, a weak (but non-asymptotic) version of the Prime Number Theorem states that $\sum_{p \leq x} \ln p \geq 0.82x$ for all $x \geq 97$ [6]. This was also formalised specifically for this project. Combining all of these bounds, we obtain the following:

▶ **Theorem 9** (A-priori prime bound)**.** *Let $k > 0$ be an integer. Then $|N_k| < \prod_{p \leq Y} p$ where*

$$Y = \max(97, c_1(k + \tfrac{1}{2})\lceil \log_2 k \rceil - c_2 k + c_3)$$

for constants $c_1 = \frac{625}{738} \approx 0.8469$, $c_2 = \frac{509417923851000000}{194404081163304960} \approx 2.6204$, $c_3 = \frac{23688125}{8060928} \approx 2.9386$.

In our algorithm, we use the following slightly weaker upper bound:

$$Y = \max\left(97, \left\lceil \frac{217k\lceil \log_2 k \rceil}{256} \right\rceil + \left\lceil \frac{217\lceil \log_2 k \rceil}{512} \right\rceil + 3 - \left\lfloor \frac{335k}{128} \right\rfloor\right)$$

If $Y \geq 2^{31}$, we return $ok = \text{False}$. We prove that Y will be in bounds for $k \leq 105{,}946{,}388$. We leave it to future work to delay the check until after we have pruned the list of primes with the more precise bound.

Note that our bound Y is lower than Harvey's since, first, we can make use of all primes $\leq Y$ because we compute N_k instead of B_k, and secondly because we use a somewhat sharper inequality for $\prod_{p \leq x} p$.

4.2 Sieving

The prime sieving algorithm `mk_factor_prime_cache K` returns a tuple (fc, pc), where pc ("prime cache") is the list of primes p with $2 < p < K$, and fc ("factor cache") encodes a mapping that maps any integer between 0 and K to its smallest prime factor (if it is composite) or to 0 (if it is prime or ≤ 1). The sieving algorithm starts by mapping all numbers to 0 and then proceeds in the usual fashion. When sieving is finished, the list of primes is extracted from the factor cache.

The advantage of the factor cache is that it allows us to not only quickly list prime numbers and determine whether a number is prime, but also to factor numbers efficiently. This speeds up the `compute_harvey_pgos` algorithm (cf. Sec 4.4).

4.3 Computing the Denominator and Precise Bounding

We can now determine the prime factorisation of D_k by simply taking all primes $p \leq k$ with $(p-1) \mid k$. The algorithm `adjust_primes2 dfs k pc` sums up $\log_2 p$ for all of these to get an approximation of $\log_2 D_k$ that easily fits in a machine integer. Together with our estimate for $\log_2 |B_k|$, we can now compute a relatively precise estimate of $\log_2 |N_k|$.

We then choose a prefix P of the sieved primes such that $\sum_{p \in P} \log_2(p) \geq \log_2 |N_k|$. For this, we use a fixed-point approximation of $\log_2 p$. This avoids using floating-point numbers or arbitrary precision integers and is still precise enough in practice. Note that we do not actually store the prime 2 in the sieve or pruned list of primes, but still consider it for the bound. Before reconstructing $|N_k|$ (cf. Sec. 4.6), we do add the congruence $N_k \equiv_2 1$ (which holds for any even k) to our modular information.

4.4 Preparing the Task List

The algorithm `compute_harvey_pgos pc fc` takes the pruned prime list and the factor map and computes a list of task entries of the form $(p, g, o, 0)$. Here, p is the prime itself, g is a

```
lemma pm.pmap_par_array_correct:
  assumes "n=length pgos" "1<t" "∀pgo∈set pgos. bernpre (k,dfs) pgo"
  shows "pm.pmap_par_array t d (k,dfs) pgos n
    ≤ SPEC (λpgos'. list_all2 (bernspec (k,dfs)) pgos pgos')"
```

where the pre- and postconditions are defined as:

```
bernpre (k,dfs) (p,g,op2,rX) =
    prime p ∧ p∈{3..<2^(num_len-1)} ∧ residue_primroot p g
  ∧ g < p ∧ op2 = ord p 2 ∧ dfs = denom_factors k
bernspec (k,dfs) (p,g,op2,rX) (p',g',op2',r) =
  (p',g',op2') = (p,g,op2) ∧ r = bernoulli_num k mod (int p)
```

Figure 5 Correctness lemma for parallel computation of the modular information. The input list contains tuples of the form $(p, g, op2, rX)$, where the last element rX is the not yet specified result. In the result list, the last element is replaced by the correct result, and the other elements are unchanged. Here, `list_all2` is the natural relator on lists, relating lists of the same length element-wise. Note that the parameters t and d are used to control the level of parallelisation and do not affect the result.

generator of $\mathbb{Z}/p\mathbb{Z}$, o is the multiplicative order of 2 in $\mathbb{Z}/p\mathbb{Z}$, and 0 is a placeholder that will be filled in by Harvey's algorithm.

A number g is a generator of $\mathbb{Z}/p\mathbb{Z}$ iff it has maximal order, i.e. if $\text{ord}_p(g) = p - 1$. We can determine whether g is a generator by checking that $g^{(p-1)/q} \not\equiv_p 1$ for all prime factors q of $p - 1$. We find the smallest generator g_p by simply checking $g = 2, 3, \dots, p - 1$ and taking the first one that works.[6]

As for computing the order: $\text{ord}_p(2)$ is defined as the smallest positive integer n such that $2^n \equiv_p 1$. Lagrange's theorem tells us that $\text{ord}_p(2) \mid p - 1$. Therefore, every factor q in the prime factorisation of $\text{ord}_p(2)$ must also be a factor of $p - 1$, and we can determine its multiplicity $\nu_q(\text{ord}_p(2))$ by letting $i \leftarrow 0$, $x \leftarrow 2^{(p-1)/q^e} \bmod p$ (where $e = \nu_q(p - 1)$) and then repeating $i \leftarrow i + 1$, $x \leftarrow x^q \bmod p$ until $x = 1$, at which point $i = \nu_q(\text{ord}_p(2))$ as desired.

We are not aware of better algorithms for computing the order and finding a generator; however, even the worst-case running time of these is negligible compared to the main part of our algorithm.

4.5 Parallel Map

To compute the modular information in parallel, we instantiate a generic parallel map combinator with `bernoulli_num_harvey_wrapper2` (cf. Sec. 3.3). We elide the boilerplate code for the instantiation, and only display the resulting correctness lemma in Fig. 5.

4.6 Postprocessing

Figure 6 displays our Isabelle formalisation of Harvey's full algorithm. After handling some special cases for $k = 0$, $k = 1$, and odd k, it computes the modular information and a prime

[6] Empirically, g_p is almost always quite small. For the first 10^6 primes, the mean and maximum of g_p are 4.91 and 94, with $g_p \in \{2, 3\}$ in 60 % of the cases. It is known that $g_p \in O(p^{\frac{1}{4}+\varepsilon})$ and the generalised Riemann hypothesis implies $g_p \in O(\log^6 p)$.

```
"bern_crt t d k = doN {
  if k=0 then return (True,1,1) else if k=1 then return (True,-1,2)
  else if odd k then return (True,0,1)
  else doN { (* Regular case: even k, k>1 *)
    (ok, (n,pgors,dfs)) ← compute_primes_mods_est t d k; (* Compute mod. info *)
    if ¬ok then return (False,0,0) (* Out-of-bounds *)
    else doN {
      (n,ams) ← map_to_ams2 n pgors; (* Map to (r,p)-pairs, add (1,2) *)
      (num,m) ← crt n ams; (* Chinese remaindering *)
      let num = (if 4 dvd k then num-m else num); (* Sign adjustment *)
      let denom = (∏x←dfs. x); (* Compute D_k *)
      return (True,num,denom)
    } } }"
```

Figure 6 The abstract version of Harvey's algorithm in Isabelle

factorisation of the denominator. If the *ok* flag is false, we abort the algorithm. Otherwise, the (p, g, o, r) tuples in *pgors* are converted to (r, p) pairs in *ams*. We also add the pair $(1, 2)$ since N_k is always odd. At this point, we know that for each pair $(r, p) \in \mathit{ams}$, we have $N_k \bmod p = r$. Moreover, the primes P in these pairs are disjoint, and their product is greater than N_k. Chinese remaindering gives us a pair (num, m) with $\mathit{num} = N_k \bmod m$ and $m = \prod_{p \in P} p$. Thus, $N_k = \mathit{num}$ if $N_k \geq 0$ and $N_k = \mathit{num} - m$ if $N_k < 0$, and we know that $N_k < 0$ iff $4 \mid k$. Finally, we compute the denominator $\mathit{denom} = D_k$ from its prime factorisation.

4.6.1 Fast Chinese Remaindering

The Chinese Remainder Theorem (CRT) allows us to determine $N_k \bmod (\prod_{p \in P} p)$ from the values of $N_k \bmod p$ for every $p \in P$.

A naïve implementation of the CRT is too slow for numbers as big as ours. We therefore follow the approach called *remainder trees* outlined in the classic textbook by von zur Gathen and Gerhard [24]. The basic data structure is a binary tree where every leaf has a pair of modular data and modulus of the form $(y \bmod p,\ p)$ attached to it, and every internal node has a modulus attached to it that is the product of the moduli of its children. Given this tree t, we can compute our desired result $y \bmod (\prod_{p \in P} p)$ as $f(\prod_{p \in P} p,\ t)$, where f is the following simple recursive function:

$$\begin{aligned} f(M, \mathrm{Leaf}(x,\ p)) &= (x \cdot ((M \bmod m^2)/p)^{-1}) \bmod p \\ f(M, \mathrm{Node}(l,\ m,\ r)) &= f(M \bmod m^2,\ l) \cdot \mathrm{root}(r)\ +\ f(M \bmod m^2,\ r) \cdot \mathrm{root}(l) \end{aligned}$$

This can also be parallelised easily since different branches of the tree are independent.

While the abstract version of our algorithm uses a tree data structure, which is initially created from the list of input pairs, we later refine this to use the list of input pairs directly for the leafs, and another array for the inner nodes. This implementation is slightly more memory-efficient than using a data structure with explicit pointers. However, in practice, we expect memory usage to be dominated by the space for the integers.

4.6.2 Arbitrary Precision Integers

While all other parts of our algorithm use machine words, CRT and the computation of D_k require large integers. We take a pragmatic approach and import the GNU Multiprecision

Library (GMP) [12] into our formalisation. We declare an assertion `mpzA` for big integers, and the GMP library functions and use Isabelle's specification mechanism to specify the Hoare-triples for them. We then instruct Isabelle LLVM to translate calls to the specified functions to the corresponding symbols from the GMP library.

The specification mechanism ensures that we do not know more about the constants than what was specified. Any such knowledge is dangerous, as it could be used to prove behaviour that is not exhibited by the GMP library. At the same time, the specification mechanism requires us to prove that a model for the specification exists. This is an important sanity check, as it prevents us from accidentally specifying contradictory statements.

Note that a verified big integer library is orthogonal to this work: it could easily replace our use of GMP, with little effect on the rest of the formalization.

5 Code Export and Final Correctness Theorem

Finally, we use *sepref* to refine `bern_crt`(cf. 4.6) to Isabelle LLVM and prove:

```
bern_crt_impl is bern_crt :: "sn64A * sn64A * u32A → u1A × mpzA × mpzA"
```

Note that Isabelle LLVM uses non-negative signed integers in various places, which are related to natural numbers by the `sn64A` refinement assertion. To make the algorithm usable from C++, we define the wrapper function `bern_crt_impl_wrapper` that returns void and uses pointer parameters to pass the results instead. Combining the refinement lemmas yields the following correctness theorem:

```
theorem bern_crt_impl_wrapper_correct: "llvm_htriple (
    ll_pto okX okp * ll_pto numX nump * ll_pto denomX denomp
  * sn64A t ti * sn64A d di * u32A k ki)
  (bern_crt_impl_wrapper okp nump denomp ti di ki)
  (λ_. EXS ok numi denomi num denom.
      ll_pto ok okp * ll_pto numi nump * ll_pto denomi denomp
    * mpzA num numi * mpzA denom denomi
    * (k≤105946388 ⟶ ok≠0) * (ok≠0 ⟶ num = N k ∧ denom = D k)"
```

The preconditions of this Hoare-triple are that `okp`, `nump`, and `denomp` point to valid memory, that `ti` and `di` are 64-bit non-negative signed integers, and that `ki` is a 32-bit unsigned with value `k`. Then, after calling `bern_crt_impl_wrapper okp nump denomp ti di ki`, the pointers `okp`, `nump`, and `denomp` will point to the values `ok`, `numi`, and `denomi`; `numi` and `denomi` will be valid GMP numbers representing the integers `num`, and `denom`; and if $k \leq 105{,}946{,}388$, `ok` will be non-zero, and if `ok` is non-zero, `num` and `denom` represent the Bernoulli number B_k. Note that we explicitly specified N_k and D_k here, as this also guarantees that `num` and `denom` have no common divisors.

This correctness theorem only depends on the Isabelle LLVM model, the formalisation of separation logic, our specification of GMP, and the definition of Bernoulli numbers. All the intermediate refinement steps need not to be trusted.

Finally, we use Isabelle LLVM to export the function to actual LLVM text and to generate a header file to interface with the generated code from C/C++.

```
export_llvm bern_crt_impl_wrapper is "void bern_crt (
    uint8_t *, gmp_mpz_struct**, gmp_mpz_struct**, sint64_t, sint64_t, uint32_t)"
  file  "../../llvm_export/bern_crt.ll"
```

We then implement a command line interface in C++, with which we run our experiments to generate actual Bernoulli numbers. For the more detailed timing measurements, we

Input k	10^5	$\lceil 10^{5.5} \rceil$	10^6	$\lceil 10^{6.5} \rceil$	10^7	$\lfloor 10^{7.5} \rfloor$	10^8
time elapsed (s)	1.12	1.41	5.75	44.6	432.3	4,215	45,458
CPU time (s)	12.56	61.45	430.11	4,318.58	46,960	475,160	5,330,806
CPU factor	12.1	44.0	74.9	96.9	108.6	112.7	117.3
memory (MB)	48.9	124.4	378.8	1220	3,948	11,775	36,769
sieving	0.4%	1.3%	1.1%	0.7%	0.3%	0.1%	0.0%
generator/ord(2)	6.5%	18.9%	15.6%	7.3%	2.7%	0.9%	0.3%
main phase	25.8%	61.5%	64.6%	81.2%	92.4%	96.9%	98.9%
reconstruction	30.5%	8.4%	8.2%	4.5%	1.9%	0.7%	0.3%
writing result	2.1%	8.7%	10.0%	6.2%	2.7%	1.3%	0.5%

Table 1 Upper half: Performance statistics of the exported LLVM code as given by the GNU `time` command. The CPU factor is an indicator for the amount of actual parallelism (best possible would be 128); Lower half: Percentage of elapsed time spent in each part of the algorithm. Note that the "writing" phase is conducted by an (unverified) GMP routine and contains both the conversion of the GMP integer into a base-10 string and the actual writing of that string to the result file.

instrumented the LLVM code to invoke callbacks into our C++ program when certain phases of the algorithm are completed.

6 Benchmarks

Table 1 shows the performance data of the exported LLVM code, compiled with Clang 18.1.8 and running on a single "standard" node of the LEO5 cluster operated by the University of Innsbruck with two 32-core 2.60 GHz Intel Xeon Platinum 8358 CPUs with hyperthreading.

Among other things, the table shows the percentage of elapsed time spent in the various parts of the algorithm. It can be seen that especially for larger inputs, the main phase (i.e. computing the modular information B_k mod p for each prime p) dominates the running time so that optimisation of the other parts will not result in a noticeable speed-up.

Figure 7 shows the relative performance of our verified LLVM implementation and Harvey's unverified C++ implementation on different machines: a 12-core desktop computer, a 16-core server, and a cluster node with two 32-core server CPUs. For the desktop computer, we also show the single-core performance. Figure 8 gives a closer look at the single-core performance on the desktop computer, which is less noisy and thus allows a better comparison.

The data suggests that our implementation is significantly slower than Harvey's for small inputs but competitive on the more interesting larger inputs (roughly $k \geq 3 \cdot 10^5$). Interestingly, our implementation slightly outperforms Harvey's on the cluster in the range $3 \cdot 10^5 \leq k \leq 5 \cdot 10^6$. This may be due to the different implementation of parallelisation.

Analyzing where and why our implementation is slower than Harvey's will require thorough microbenchmarking. Also note that the results shown in this section are not a statistically rigorous performance analysis, but merely some experiments to establish that the performance of our implementation is roughly comparable to that of Harvey's.

7 Related Work

We are not aware of any other work on the verification of algorithms involving Bernoulli numbers. As far as proof assistants are concerned, we are only aware of two others that

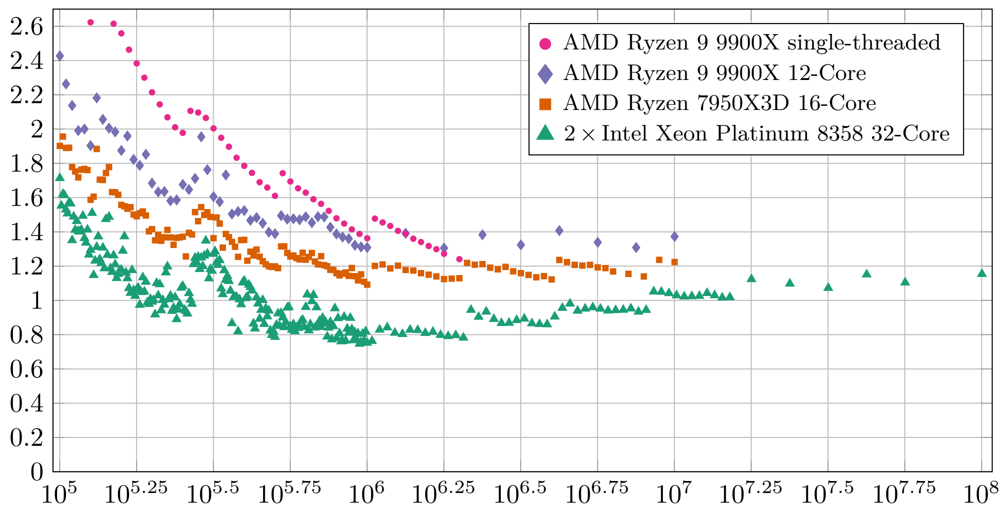

Figure 7 The ratio of the elapsed time of our verified implementation divided by that of Harvey's, for a range of input indices k and on several different machines.

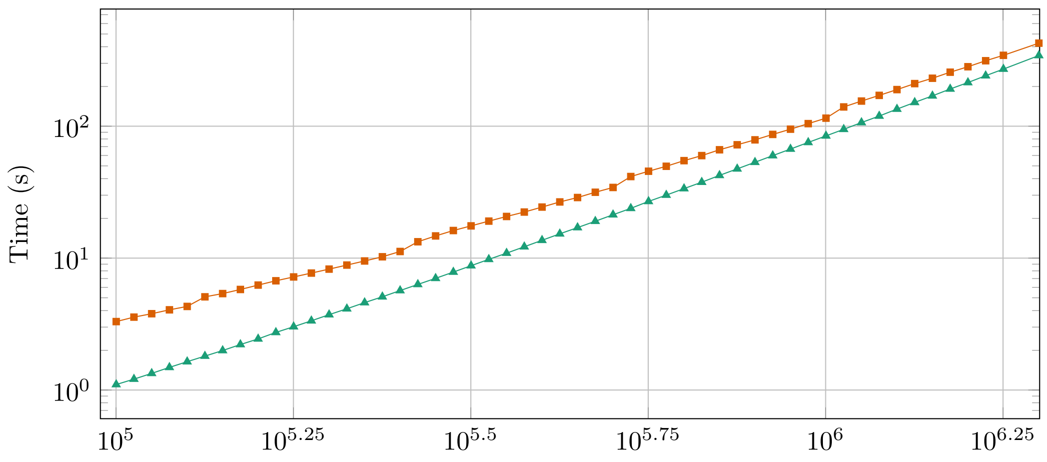

Figure 8 Single-core performance of Harvey's implementation against ours (AMD Ryzen 9 9900X)

even have a *definition* of Bernoulli numbers, namely HOL Light and Lean. However, neither system's library seems to have an algorithm for computing them other than using the recurrence that follows directly from their definition. We are not aware of any verification efforts for Bernoulli number algorithms outside proof assistants either.

As for unverified algorithms, there are various approaches apart from Harvey's 2010 algorithm (which we formalised in this paper) that are more efficient than the naïve approach:

- as the Akiyama–Tanigawa transform of the sequence $(\frac{1}{k+1})_{k\in\mathbb{N}}$ [2, 15]
- by computing D_k and approximating $\zeta(k)$ to sufficient precision
- by performing the division $\sin(z)/\cos(z)$ as formal power series, either directly or via Kronecker substitution (cf. e.g. Brent and Harvey [3])

Harvey discusses some of these in some more detail in his paper. The first of them is already available in Isabelle [4], but it is quite inefficient for even moderate values of k. The last two have the same asymptotic running time as the algorithm we verify, namely $O(k^2 \log^{2+o(1)} k)$, but for various reasons discussed in Harvey's paper, they seem to perform much worse

Component	LOC
Voronoi/Kummer congruences [7]	2,397
Prime bounds [6]	1,814
Fixed-point approximation of $\log_2 n$	892
Prime sieving, $\mathrm{ord}_p(x)$, generators	2,765
Fast Chinese remaindering	3,801
Montgomery arithmetic	2,317
Binary fraction expansion of $1/p$	968
Miscellaneous	1,373

Component (cont'd)	LOC
GMP bindings for *Isabelle-LLVM*	1,749
Additions to *sepref*/*Isabelle-LLVM*	4,365
Main algorithm (abstract)	1,569
Main algorithm (refinement)	8,568
Total	32,578

Table 2 Size summary of the different parts of our Isabelle contributions

in practice. Note however that the third algorithm computes not only B_k but the entire sequence $B_0, \ldots, B_k$.

In 2012, Harvey also published another multi-modular algorithm for computing B_k which achieves subquadratic running time by computing B_k modulo prime powers instead of simply primes. Asymptotically, this is a large improvement over all previously known algorithms; however, to date, there is still no implementation, and it is not clear how well it would perform for feasible input sizes in practice. In fact, it seems that the only fast algorithms that have actual implementations are the ones based on approximating $\zeta(k)$ and Harvey's 2010 algorithm.

The Isabelle LLVM framework has been used for verifying efficient implementations of several tools and algorithms, mainly for SAT solving [11, 19] and model checking [17, 13]. Its parallel extension has only been used for sorting algorithms [20] so far.

Verified implementations of Montgomery arithmetic exist in VeriFun by Walther [25] or in Coq as part of the Fiat Cryptography library by Erbsen et al. [8].

8 Conclusion

We fully verified a challenging state-of-the-art mathematical algorithm to compute Bernoulli numbers, achieving performance comparable to highly optimised hand-written C++ code. A large amount of machinery had to be developed to achieve this, including:

Mathematical background: bounds for Bernoulli numbers, the Voronoi and Kummer congruences [7], bounds on the distribution of primes [6]

Algorithms, abstractly and verified down to LLVM: prime sieving, Chinese remaindering via remainder trees, computing $\lceil \log_2 n \rceil$, fixed-point approximations of $\log_2 n$, efficient computation of the binary fraction expansion of a rational number, Hensel lifting, the extended Euclidean algorithm, Montgomery ('REDC') arithmetic, computing $\mathrm{ord}_n(x)$, finding generators in $\mathbb{Z}/n\mathbb{Z}$

Extending the Isabelle/LLVM framework: axiomatisation of GMP integers, destructive parallel maps over an array, low-level bit arithmetic

We expect many of these developments to be useful for other applications.

The total development comprises about 32 kLOC in Isabelle (see Table 2 for more details). The mathematical core of the development is relatively small (about 6 kLOC or 20 %), which matches our experience that proofs about algorithms tend to be bulkier than proofs about mathematics.

References

1 J. C. Adams. On the calculation of Bernoulli's numbers up to B_{62} by means of Staudt's theorem. In *Report of the meeting of the British Association for the Advancement of Science*, Rep. Brit. Assn., pages 8–15, 1877. URL: `https://gallica.bnf.fr/ark:/12148/bpt6k78160g`.

2 Shigeki Akiyama and Yoshio Tanigawa. Multiple zeta values at non-positive integers. *The Ramanujan Journal*, 5(4):327–351, 2001. `doi:10.1023/A:1013981102941`.

3 Richard P. Brent and David Harvey. *Fast Computation of Bernoulli, Tangent and Secant Numbers*, pages 127–142. Springer New York, 2013. `doi:10.1007/978-1-4614-7621-4_8`.

4 Lukas Bulwahn and Manuel Eberl. Bernoulli numbers. *Archive of Formal Proofs*, January 2017. `https://isa-afp.org/entries/Bernoulli.html`, Formal proof development.

5 Henri Cohen. *Number Theory: Volume II: Analytic and Modern Tools*. Graduate Texts in Mathematics. Springer New York, 2007.

6 Manuel Eberl. Concrete bounds for Chebyshev's prime counting functions. *Archive of Formal Proofs*, September 2024. `https://isa-afp.org/entries/Chebyshev_Prime_Bounds.html`, Formal proof development.

7 Manuel Eberl. Kummer's congruence. *Archive of Formal Proofs*, March 2024. `https://isa-afp.org/entries/Kummer_Congruence.html`, Formal proof development.

8 Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic – with proofs, without compromises. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1202–1219, 2019. `doi:10.1109/SP.2019.00005`.

9 Eugene Eric Kim and Betty Alexandra Toole. Ada and the first computer. *Scientific American*, 280(5):76–81, May 1999. `doi:10.1038/scientificamerican0599-76`.

10 Greg Fee and Simon Plouffe. An efficient algorithm for the computation of Bernoulli numbers, 2007. URL: `https://arxiv.org/abs/math/0702300`, `arXiv:math/0702300`.

11 Mathias Fleury and Peter Lammich. A more pragmatic CDCL for isasat and targetting LLVM (short paper). In Brigitte Pientka and Cesare Tinelli, editors, *Automated Deduction - CADE 29 - 29th International Conference on Automated Deduction, Rome, Italy, July 1-4, 2023, Proceedings*, volume 14132 of *Lecture Notes in Computer Science*, pages 207–219. Springer, 2023. `doi:10.1007/978-3-031-38499-8_12`.

12 The GNU multiple precision arithmetic library. URL: `https://gmplib.org/`.

13 Arnd Hartmanns, Bram Kohlen, and Peter Lammich. Efficient formally verified maximal end component decomposition for mdps. In André Platzer, Kristin Yvonne Rozier, Matteo Pradella, and Matteo Rossi, editors, *Formal Methods - 26th International Symposium, FM 2024, Milan, Italy, September 9-13, 2024, Proceedings, Part I*, volume 14933 of *Lecture Notes in Computer Science*, pages 206–225. Springer, 2024. `doi:10.1007/978-3-031-71162-6_11`.

14 David Harvey. A multimodular algorithm for computing Bernoulli numbers. *Math. Comput.*, 79(272):2361–2370, 2010. `doi:10.1090/S0025-5718-2010-02367-1`.

15 M. Kaneko. The Akiyama–Tanigawa algorithm for Bernoulli numbers. *Journal of Integer Sequences*, 3, 2000.

16 Donald E. Knuth and Thomas J. Buckholtz. Computation of tangent, Euler, and Bernoulli numbers. *Mathematics of Computation*, 21(100):663–688, 1967. `doi:10.1090/s0025-5718-1967-0221735-9`.

17 Bram Kohlen, Maximilian Schäffeler, Mohammad Abdulaziz, Arnd Hartmanns, and Peter Lammich. A formally verified IEEE 754 floating-point implementation of interval iteration for mdps. *CoRR*, abs/2501.10127, 2025. `arXiv:2501.10127`, `doi:10.48550/ARXIV.2501.10127`.

18 Peter Lammich. Generating verified LLVM from isabelle/hol. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*, volume 141 of *LIPIcs*, pages 22:1–22:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. `doi:10.4230/LIPIcs.ITP.2019.22`.

19 Peter Lammich. Fast and verified UNSAT certificate checking. In Christoph Benzmüller, Marijn J. H. Heule, and Renate A. Schmidt, editors, *Automated Reasoning - 12th International Joint Conference, IJCAR 2024, Nancy, France, July 3-6, 2024, Proceedings, Part I*, volume 14739 of *Lecture Notes in Computer Science*, pages 439–457. Springer, 2024. `doi:10.1007/978-3-031-63498-7_26`.

20 Peter Lammich. Refinement of parallel algorithms down to LLVM: applied to practically efficient parallel sorting. *J. Autom. Reason.*, 68(3):14, 2024. `doi:10.1007/S10817-024-09701-W`.

21 Peter Lammich and Thomas Tuerk. Applying data refinement for monadic programs to Hopcroft's algorithm. In *Proc. of ITP*, volume 7406 of *LNCS*, pages 166–182. Springer, 2012.

22 D. H. Lehmer. An extension of the table of Bernoulli numbers. *Duke Mathematical Journal*, 2(3), September 1936. `doi:10.1215/s0012-7094-36-00238-7`.

23 Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985. `doi:10.1090/s0025-5718-1985-0777282-x`.

24 Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, 3rd edition, 2013.

25 Christoph Walther. Formally verified Montgomery multiplication. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 505–522. Springer, 2018. `doi:10.1007/978-3-319-96142-2_30`.

26 Alexander Yee and Shigeru Kondo. 10 trillion digits of pi: A case study of summing hypergeometric series to high precision on multicore systems. Technical report, Siebel School of Computing and Data Science, 2011. URL: `https://hdl.handle.net/2142/28348`.