The Journal of Supercomputing manuscript No.
(will be inserted by the editor)

Convergence verification of the Collatz problem

David Barina

Received: date / Accepted: date

Abstract This article presents a new algorithmic approach for computational convergence verification of the Collatz problem. The main contribution of the paper is the replacement of huge precomputed tables containing $O(2^N)$ entries with small look-up tables comprising just $O(N)$ elements. Our single-threaded CPU implementation can verify 4.2×10^9 128-bit numbers per second on Intel Xeon Gold 5218 CPU computer and our parallel OpenCL implementation reaches the speed of 2.2×10^{11} 128-bit numbers per second on NVIDIA GeForce RTX 2080. Besides the convergence verification, our program also checks for path records during the convergence test.

Keywords Collatz conjecture · Software optimization · Parallel computing · Number theory

1 Introduction

One of the most famous problems in mathematics that remains unsolved is the Collatz conjecture, which asserts that, for arbitrary positive integer n, a sequence defined by repeatedly applying the function $C(n) = 3n+1$ if n is odd, or $C(n) = n/2$ if n is even will always converge to the cycle passing through the number 1. The terms of such sequence typically rise and fall repeatedly, oscillate wildly, and grow at a dizzying pace. The conjecture has never been proven. There is however experimental evidence and heuristic arguments that support it. As of 2020, the conjecture has been checked by computer for all starting values up to 10^{20} [1]. There is also an extensive literature, [2, 3], on this question.

Centre of Excellence IT4Innovations
Faculty of Information Technology
Brno University of Technology
Bozetechova 1/2, Brno, Czech Republic
E-mail: ibarina@fit.vutbr.cz

The most striking thing about the Collatz conjecture is that it would shed a light on the relation between the prime factorizations of n and $n+1$. The Collatz function consists of two multiplicative operations and adding 1 that has a huge effect on the factorization. Note that this problem has led directly to theoretical work showing that very similar questions are formally undecidable [4]. Another interesting relation is that the Collatz problem can be encoded as a simple Emil Post's tag system [5].

The competitive (past and other ongoing) projects verifying the convergence of the Collatz problem use huge pre-computed sieves and lookup tables to calculate multiple iterates in a single step. For k steps, the tables have a size of 2^k entries, the entry comprises usually two 64, 96, or 128-bit numbers. Our approach is fundamentally different. We have realized that the additive step in the Collatz function can be technically avoided. Rather than tracking the trajectory directly on n, we track the same trajectory on $n+1$. The trick is that, when calculating the function iterates, we switch between n and $n+1$ domains in such a way that we always use only multiplicative operations. Considering the binary representation of the n, we only use the ctz (count trailing zeros) operation, right shift, and a small lookup table with precomputed powers of three (tens of bytes in total).

The rest of the paper is organized as follows. Section 2 reviews related work, especially competitive projects and the results achieved so far. Section 3 presents a new algorithm for computing iterates of the Collatz function. Section 4 describes optimization techniques used in conjunction with this algorithm. Section 5 presents performance evaluation and results achieved using the algorithm. Finally, Section 6 concludes the paper.

2 Related Work

Several past or ongoing projects are trying to verify or disprove the Collatz conjecture. These projects can be divided into two groups according to the algorithm they use: (1) The first group checks for the convergence of the problem for all numbers up to some upper bound. The bottom line is that they calculate the Collatz function iterates, starting from the initial number n, and stopping once an iterate drops below n. This is also known as computing the stopping time [6], or glide. (2) The second group also checks all numbers up to some upper bound but searches for the highest number of iterates (steps) before reaching 1. This is known as computing the total stopping time [6], or delay. Importantly, algorithms used for this second group are at least one order of magnitude slower compared to the first group. Our work targets the first group. The question is how fast (in terms of numbers per second) are state-of-the-art methods in both these groups. The current upper bound under which the problem is verified is $2^{66.4}$ [1].

The ongoing project of Eric Roosendaal[1] asserts that their problem can check about $2^{27.3}$ numbers per second. By examining his program on our AMD

[1] `http://www.ericr.nl/wondrous/`

Ryzen Threadripper 2990WX we have however found out the speed of $2^{25.7}$ numbers per second. The algorithm belongs to the second group. All numbers up to $2^{60} \approx 10^{18}$ have been checked for convergence.

We are also aware of the ongoing BOINC project.[2] But we are unable to find how fast their program is, and which algorithm uses. Based on our personal correspondence with Eric Roosendaal, we found that this ongoing BOINC project is meant to disprove the Collatz conjecture by trying to find a counter-example. The project started off at 2^{71}. It looks like they have reached roughly $2^{72.3}$. No information can be found regarding whether all numbers up to that limit have indeed be checked.

In 2017, the yoyo@home project [1] checked for convergence all numbers up to $10^{20} \approx 2^{66.4}$. The work of Honda et al. [7] claims that they can check $2^{40.25}$ numbers per second for the convergence (the first group), or $2^{29.9}$ numbers per second for the delay, both on GPU. Their programs are however only able to verify 64-bit numbers (which have been known to converge due to the yoyo@home project). The paper by Tomás Oliveira e Silva [8] from 2010 claims that the author verified in 2009 the conjecture up to $2^{62.3} \approx 5.76 \times 10^{18}$. According to information published on his website, the speed of his program was about 2.25×10^9 numbers per second on computers of that time. Earlier, in 2008, Tomás Oliveira e Silva[3] tested all numbers below 19×2^{58}. Much earlier, in 1992, Leavens and Vermeulen [9] verified the convergence for all numbers below $5.6 \times 10^{13} \approx 2^{45.67}$. And as the first tracked record, in 1973, Dunn [10] verified the convergence below ca. $2^{24.78}$. According to [8], there exist other unpublished records before the year 1992.

3 New Approach

Recall that the Collatz conjecture asserts that a sequence defined by repeatedly applying the function

$$C(n) = \begin{cases} 3n+1 & \text{if } n \text{ is odd, or} \\ n/2 & \text{if } n \text{ is even} \end{cases} \tag{1}$$

will always converge to the cycle passing through the number 1 for arbitrary positive integer n. Note that the outcome of the odd branch in (1) is always even, and thus the next iteration must go through the even branch. Thus, the modified formulation

$$T(n) = \begin{cases} (3n+1)/2 & \text{if } n \equiv 1 \pmod 2, \text{ or} \\ n/2 & \text{if } n \equiv 0 \pmod 2 \end{cases} \tag{2}$$

is often [2] used. Multiplying by 3 and factoring out a power of 2 have only a small effect on the prime factorization of n. The question here is how does the prime factorization of n affect the prime factorization of $n+1$.

[2] `https://boinc.thesonntags.com/collatz/`

[3] `http://sweet.ua.pt/tos/3x+1.html`

We have realized that the additive step in the $T(n)$ can be technically avoided when computing the function iterates. Rather than defining the $T(n)$ as in (2), and tracking the trajectory directly on n, we can track the same trajectory on $n+1$ with the auxiliary function

$$T_1(n) = \begin{cases} (n+1)/2 & \text{if } n \equiv 1 \pmod 2, \\ 3n/2 & \text{if } n \equiv 0 \pmod 2. \end{cases} \tag{3}$$

Thus the multiplying by 3 just moved to the even branch. The trick is that, when calculating the function iterates, we switch between n and $n+1$ in such a way that we always use only the even branch of either T or T_1. Therefore, the above functions can be expressed as

$$T(n) = \begin{cases} T_1(n+1) - 1 & \text{if } n \equiv 1 \pmod 2, \\ n/2 & \text{if } n \equiv 0 \pmod 2, \end{cases} \tag{4}$$

and

$$T_1(n) = \begin{cases} T(n-1) + 1 & \text{if } n \equiv 1 \pmod 2, \\ 3n/2 & \text{if } n \equiv 0 \pmod 2. \end{cases} \tag{5}$$

There is seemingly still an additive operation in each step. However, considering the binary representation of the n, these additive operations can be almost avoided by merging several even steps into a single one. In other words, we use operation which counts the number of trailing zero bits following the least significant non-zero bit (ctz operation) and then perform multiple divisions by two (right shifts) at once. This also include performing multiple multiplications by three at once. However, the powers of three can be precomputed in a small look-up table and also these multiplications can be performed using a single one. The size of the small look-up table can be arbitrarily small and correspond to the number of steps performed at once, thus the space complexity is $O(N)$, where the N is the number of steps performed in a single step.

Algorithm 1 Convergence test

Require: n_0 is positive integer
1: $n \leftarrow n_0$
2: **repeat**
3: $\quad n \leftarrow n + 1$
4: $\quad \alpha \leftarrow \operatorname{ctz}(n)$
5: $\quad n \leftarrow n \times 3^\alpha / 2^\alpha$
6: $\quad n \leftarrow n - 1$
7: $\quad \beta \leftarrow \operatorname{ctz}(n)$
8: $\quad n \leftarrow n / 2^\beta$
9: **until** $n < n_0$

Algorithm 2 Convergence test

Require: n is positive integer
1: **repeat**
2: $n \leftarrow n + 1$
3: $\alpha \leftarrow \text{ctz}(n)$
4: $n \leftarrow n \times 3^{\alpha}/2^{\alpha}$
5: $n \leftarrow n - 1$
6: $\beta \leftarrow \text{ctz}(n)$
7: $n \leftarrow n/2^{\beta}$
8: **until** $n = 1$

Now we can formulate two convergence verification algorithms, in Algorithm 1 and 2, according to the division in Section 2. The first algorithm checks for the convergence, maximum value reached during the progression, and glide. The second one is above that able to check for the delay. Delay is computed as the sum of all alphas and betas before reaching number 1. Given that $\text{Max}(n)$ is maximum value reached during the progression $n, T(n), T^2(n), \ldots, 1$, a positive integer m is called a path record if for all $n < m$ the inequality $\text{Max}(m) > \text{Max}(n)$ holds. We check for the convergence of the problem for all numbers starting from 1 up to some upper bound. Under this assumption, both above algorithms can check for path records (note that for Algorithm 1 the path record always occurs before $n < n_0$).

4 Sieve

The general form [11] of $T^k(n)$ is

$$T^k(2^k n_H + n_L) = 3^{\text{odd}(n_L)} n_H + T^k(n_L), \tag{6}$$

where $\text{odd}(n_L)$ is the number of odd steps of $T(n)$ that were taken in the computation of $T^k(n_L)$. Competitive programs use this equation to perform k steps at once (the tables have the size of 2^k entries, indices correspond to n_L). The difference between this and competitive algorithms lies in the fact that the competitive algorithms compute a fixed number of iterates in a single step (using the equation above). On the contrary, the number of steps in our algorithm depends on the specific number tested. One can verify that for odd n, the average number of iterates computed in a single step for both Algorithm 1 and 2 is 4. Thus, using $k > 4$ in (6) leads to a higher number of iterations calculated in one step of the algorithm.

However, even more important acceleration technique of the convergence test is the usage of a sieve (the sieve has the size of 2^k entries). Using the sieve we test only those numbers that do not either converge or join[4] the path of a lower number in k steps. The acceleration obtained from this method is significant. The disadvantage is a huge memory footprint of such sieves. For example, the sieve having the size of 2^{34} occupies 2^{34} bits which is exactly 2

[4] enhancement proposed by Eric Roosendaal

gigabytes. We have however found that these convergence sieves (considering values stored in bits) are formed by constantly repeated bit patterns. Specifically, the 2^{34} sieve may have a memory footprint of 256 megabytes. The reason is that this sieve is formed by only fifty constantly repeated 64-bit patterns, so we can store only indices into a small look-up table. Here we consider 8-bit indices. Similarly, the 2^{24} sieve has a size of 256 kilobytes. The compression ratio can reach the value around 1:10 (64 bits represented by 6-bit index).

We experimented with many sieve sizes and came to the conclusion that the sieve size 2^{34} is optimal for our CPU implementation, whereas the sieve size 2^{24} is optimal for GPUs. We are aware that other authors have used even larger sieves (and therefore have reached higher performance), e.g., the sieve of the size 2^{37} in [7]. However, such a sieve is absolutely impractical since it occupies 16 gigabytes of memory.

To speed up the convergence verification even further, our CPU program verifies 2^{40-34} numbers of the same congruence class modulo 2^{34} concurrently. This particularly means that the program verifies the work units having the size of 2^{40} numbers and solves the lowest 34 bits at once. Then the code paths diverge, resulting in the verification of individual numbers up to 2^{40}. We are aware that Eric Roosendaal used a similar "interlaced" technique in his convergence algorithm. Note that the size of congruence class 2^{34} exactly matches the sieve size.

5 Performance Evaluation

Our CPU implementation (written in C), as well as GPU implementation (OpenCL), can verify work units of 2^{40} 128-bit numbers. If the 128-bit arithmetic is not sufficient, the program switches to multi-precision arithmetic for the necessary amount of time. Both of these programs implement Algorithm 1. Partial (per work unit) path records are stored during the verification. Additionally, the program sums all the αs for all n inside a particular work unit as proof of work so that the results can be independently verified. This data was not collected in [7], whereas our program is focused on practical use. A comparison of our program with competing programs is given in Table 1. Note that all other programs can process only 64-bit numbers while our program natively operates on 128-bit arithmetic. Note also that the comparison is made on different hardware. Finally note that all other programs require tables of the size $O(2^N)$, whereas our program only requires a small table of the size $O(N)$. To allow other developers and scientists to benefit from this work and build on it, the programs used in this article have been released as open-source software.[5]

The program presented in this paper runs as a part of a distributed computing project to check the convergence of the Collatz problem. From September 2019 to May 2020, the project managed to verify this conjecture for all

[5] `https://github.com/xbarin02/collatz/`

authors	sieve	numbers	speed	hardware
Honda et al.	2^{37}	64-bit	1.31×10^{12}	NVIDIA GeForce GTX TITAN X
Honda et al.	2^{37}	64-bit	5.25×10^{9}	Intel Core i7-4790
Roosendaal	2^{32}	64-bit	4.63×10^{8}	contemporary CPUs
Oliveira et al.	2^{46}	64-bit	2.25×10^{9}	CPUs of the 2004–2009 era
this paper	2^{34}	128-bit	4.21×10^{9}	Intel Xeon Gold 5218
this paper	2^{24}	128-bit	2.20×10^{11}	NVIDIA GeForce RTX 2080

Table 1 Comparison with competitive programs. The speed is given in numbers per second.

numbers below 2^{68}. Define $t(n)$ the highest number occurring in the sequence starting at n. The n is called the path record if for all $m < n$ the inequality $t(m) < t(n)$ holds. Lagarias and Weiss [12] predicted using the large deviation theory for random walks that

$$\limsup_{n\to\infty} \frac{\log t(n)}{\log n} = 2. \tag{7}$$

In other words, the highest number occurring in the sequence for a path record n grows like n^2. The results recorded up to 2^{68} confirm this prediction. The largest know path record below 2^{68} occurs for the starting value $n = 274133054632352106267$ (previously unpublished).

6 Conclusion

This article presents a new method for computing iterates of the Collatz function. The advantage over existing approaches is that it only requires a table of the size $O(N)$ to compute N steps at once, whereas other approaches require tables of the size $O(2^N)$ to do the same. In addition, the article presents a new memory-saving method for representing a sieve that further accelerates the convergence test. Our programs can process 128-bit numbers, whereas competitive programs can only process 64-bit numbers. The programs used in this work have been released as open-source software.

Acknowledgement

Computational resources were supplied by the project "e-Infrastruktura CZ" (e-INFRA LM2018140) provided within the program Projects of Large Research, Development and Innovations Infrastructures. This work was supported by The Ministry of Education, Youth and Sports from the Large Infrastructures for Research, Experimental Development and Innovations project "IT4Innovations National Supercomputing Center – LM2015070".

References

1. Hercher C (2018) Über die Länge nicht-trivialer Collatz-Zyklen. Die Wurzel 6 and 7
2. Lagarias JC (2003) The $3x+1$ problem: An annotated bibliography (1963–1999) (sorted by author). arXiv:math/0309224
3. Lagarias JC (2006) The $3x + 1$ problem: An annotated bibliography, II (2000-2009). arXiv:math/0608208
4. Conway JH (1972) Unpredictable iterations. In: Proceedings of the 1972 Number Theory Conference, pp 49–52
5. Mol LD (2008) Tag systems and Collatz-like functions. Theoretical Computer Science 390(1):92–101, DOI 10.1016/j.tcs.2007.10.020
6. Lagarias JC (1985) The 3x + 1 problem and its generalizations. The American Mathematical Monthly 92(1):3–23, DOI 10.2307/2322189
7. Honda T, Ito Y, Nakano K (2017) GPU-accelerated exhaustive verification of the Collatz conjecture. International Journal of Networking and Computing 7(1):69–85
8. Oliveira e Silva T (2010) Empirical verification of the 3x+1 and related conjectures. In: Lagarias JC (ed) The Ultimate Challenge: The 3x+1 Problem, American Mathematical Society, pp 189–207
9. Leavens GT, Vermeulen M (1992) 3x+1 search programs. Computers & Mathematics with Applications 24(11):79–99, DOI 10.1016/0898-1221(92)90034-F
10. Dunn R (1973) On Ulam's problem. Tech. rep., University of Colorado at Boulder
11. Oliveira e Silva T (1999) Maximum excursion and stopping time record-holders for the $3x + 1$ problem: Computational results. Mathematics of Computation 68(225):371–384, DOI 10.1090/S0025-5718-99-01031-5
12. Lagarias JC, Weiss A (1992) The $3x + 1$ problem: Two stochastic models. Annals of Applied Probability 2(1):229–261, DOI 10.1214/aoap/1177005779