The Journal of Supercomputing (2025) 81:810
https://doi.org/10.1007/s11227-025-07337-0

Improved verification limit for the convergence of the Collatz conjecture

David Barina[1]

Accepted: 21 April 2025

Abstract
This article presents our project, which aims to verify the Collatz conjecture computationally. As a main point of the article, we introduce a new result that pushes the limit for which the conjecture is verified up to 2^{71}. We present our baseline algorithm and then several sub-algorithms that enhance acceleration. The total acceleration from the first algorithm we used on the CPU to our best algorithm on the GPU is 1 335×. We further distribute individual tasks to thousands of parallel workers running on several European supercomputers. Besides the convergence verification, our program also checks for path records during the convergence test. We found four new path records.

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

1 Introduction

Collatz conjecture is one of the simplest unsolved problems in mathematics. It deals with iterations of the number theoretic function, which takes an odd integer n to $3n + 1$ and an even n to $n/2$. The conjecture asserts that iterations of this function starting at arbitrary positive number n always lead to the number 1. Jeffrey Lagarias, the famous mathematician, stated that the Collatz conjecture "is an extraordinarily difficult problem, completely out of reach of present-day mathematics" [1]. There is an extensive literature on this conjecture. Preprints [2] and [3] provide an overview of such papers.

✉ David Barina
ibarina@fit.vutbr.cz

[1] Faculty of Information Technology, Brno University of Technology, Bozetechova 1/2, Brno, Czech Republic

Published online: 02 May 2025

Because the output of $3n + 1$ branch is even for odd n and therefore $n/2$ step must follow, we can replace this branch by $(3n + 1)/2$. The Collatz conjecture now asserts that a sequence defined by repeatedly applying the function

$$T(n) = \begin{cases} (3n+1)/2, & \text{for odd } n, \\ n/2, & \text{for even } n, \end{cases} \tag{1}$$

always converges to the cycle passing through the number 1 for arbitrary positive integer n. The conjecture has never been proven. There is, however, experimental evidence and heuristic arguments that support it. At the time of writing this article, all starting values up to 2^{68} were computer-checked [4]. This paper presents a result that pushes this limit to 2^{71}.

We have created a distributed heterogeneous computing system that sequentially tests all integers and thus pushes the boundary beyond which the Collatz conjecture is explored. The system runs on supercomputers in Europe and consists of thousands of parallel workers to which work is centrally assigned. Workers compute their tasks either on the CPU or on the GPU. This paper describes the architecture and development of this system.

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 explains all the algorithms we used in the convergence verification. Section 4 deals with the distribution of individual tasks to thousands of parallel workers. Section 5 provides a performance evaluation of the algorithms from Sect. 3. Section 6 presents the achieved results. Finally, Sect. 7 concludes the paper.

2 Related work

Several current or past projects seek to disprove or verify the Collatz conjecture. Convergence addresses the question of whether all initial numbers tested will eventually reach the value 1. We focus on the projects that check for the convergence of the conjecture for all numbers up to some upper bound. The path records are starting values that set new records for the highest point of trajectory before reaching 1. The programs that check for convergence usually also collect path records. The question is how far the Collatz conjecture has been computationally verified and how fast (numbers per second) the methods in such projects are. The measure numbers per second means the number of initial numbers verified per second, which does include numbers that are not checked at all.

As the first tracked record, in 1973, Dunn [5] verified the convergence below approximately $2^{24.78}$. In 1992, Leavens and Vermeulen [6] verified the convergence for all numbers below approximately $2^{45.67}$.

Before 2012, Eric Roosendaal [7] checked all numbers up to 2^{60} on the CPUs. Meanwhile, Tomás Oliveira e Silva [8] checked the numbers up to 5×2^{60} using a different software/hardware. Up to that moment, Tomás Oliveira e Silva and Eric Roosendaal alternated in the lead. According to information published on Oliveira e Silva's website, the speed of his program was about 2^{31} numbers per second on

computers at that time. Eric Roosendaal estimates the theoretical speed of his algorithm on GPU as approximately $2^{38.86}$ numbers per second. However, this is a projected speed; no such program exists.

In 2017, the yoyo@home project [9] checked for convergence of all numbers up to 87×2^{60}. Their CPU code checked about $2^{35.48}$ numbers per second and core. 1 000 volunteers participated in the project, and each number was independently checked twice.

Also, in 2017, Honda et al. [10] claimed they can check $2^{40.25}$ numbers per second on the GPU. Their program is, however, only able to verify 64-bit numbers.

From 2019 to 2021, our project [4] verified the convergence of all numbers below 2^{68}. We used CPUs as well as GPUs along with the power of supercomputers. Our CPU implementation achieved a speedup to $2^{31.97}$ numbers/second, whereas the speed on GPUs reached the limit of $2^{37.68}$ numbers/second. Both implementations can check 128-bit numbers. This article is a continuation of our previous work.

Unlike [4], the implementation of sieves in this paper is much more sophisticated (and thus faster). Unlike [4], we compared many sieve sizes and different settings, and the results are given in Sect. 5. The contributions of this paper include (a) introduction of 3^k sieves and optimized code for the sieve of the size of 3^2 (Sect. 3.2), (b) extending the algorithm to a distributed environment (Sect. 4), (c) comparison of different algorithms under different settings (Sect. 5), (d) publishing the result where the project verified all numbers below 2^{71} (Sect. 6). My motivation was to push the limit to which the Collatz conjecture is verified, and possibly find a counterexample.

3 Algorithms

This section will explain all the algorithms we used in the convergence testing. Primarily, the same algorithms are used on both CPU and GPU. Another issue is distributing convergence testing across multiple computers, which we will cover in the next section.

3.1 Baseline testing

The baseline testing is an algorithm by which the iterations of the Collatz function are calculated. The first thing that comes to mind is to use Eq. (1) directly. Indeed, we can formulate a naive convergence verification algorithm, as listed in Algorithm 1. The $\langle n \rangle_2$ is the result of n modulo 2 operator. The algorithm tests the numbers in ascending order. This way, we can stop testing when n drops below the initial value n_0 (since all numbers below were already tested). In more detail, the algorithm tests a single number and thus has to be repeated on each number independently, from the smallest to the largest one.

Algorithm 1 Naive algorithm

Require: n_0 is a positive integer
1: $n \leftarrow n_0$
2: **repeat**
3: **if** $\langle n \rangle_2 = 1$ **then**
4: $n \leftarrow (3n+1)/2$
5: **else**
6: $n \leftarrow n/2$
7: **end if**
8: **until** $n < n_0$

In [4], we tracked $T(n)$-trajectory on $n+1$ rather than directly on n. The trick is that when the function calculating $T(n)$ iterates, we switch between n and $n+1$ to always use only multiplication with no additive operation. For example, the trajectory on n of 12 is (6, 3, 5, 8, 4, 2, 1), whereas on $n+1$, the same trajectory is (7, 4, 6, 9, 5, 3, 2). We use the ctz operation, which counts the number of trailing zero bits following the least significant nonzero bit, and then perform multiple divisions by two and multiple multiplications by three at once (for details see [4]). Thus, we merge several even as well as odd steps into single ones. Moreover, the powers of three can be precomputed in a small look-up table, and these multiplications can be performed using a single multiplication. In other words, combining a number of k consecutive even steps leads to $a2^k \longrightarrow a$, and combining a number of k consecutive odd steps leads to $a2^k - 1 \longrightarrow a3^k - 1$. Thus, if the binary representation of a number ends in k zeros, we have the first case, and if it ends in k ones, the second one. The procedure is listed in Algorithm 2. One can verify that for odd n, the average number of iterates computed in a single step for Algorithm 2 is 4.

For those for which it is not obvious that a number with k trailing zeros in its binary representation will result in k consecutive odd steps, we offer another perspective on the matter. Rather than defining the $T(n)$ as in (1), and tracking the trajectory directly on n, we can track the same trajectory on $n+1$ with the auxiliary function

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

Thus, the multiplying by 3 just moved to the even branch. The trick is that when calculating the function iterates, our algorithm switches 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

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

and

$$T_1(n) = \begin{cases} T(n-1)+1 & \text{if } n \equiv 1 \pmod 2, \\ 3n/2 & \text{if } n \equiv 0 \pmod 2. \end{cases} \tag{4}$$

So, by suitably switching these two equations, we can always use the ctz operation and perform several steps at once. Therefore, we can perform several $(3n+1)$ steps at once.

Algorithm 2 New algorithm used in [4]

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

Comparison with the baseline testing algorithm, where the time is given per a work unit of the size of 2^{30}, gives us 8.8976 s for Algorithm 1 and 5.7024 s for Algorithm 2. This quick comparison reveals that Algorithm 2 is about 36 % faster. The results were obtained on a 3.0 GHz CPU (AMD Ryzen Threadripper 2990WX). For this reason, we decided to optimize further only Algorithm 2.

3.2 Sieve 3^k

Imagine the following step on numbers of the form $2n+1$ (odd numbers).

$$2n+1 \rightarrow \frac{3(2n+1)+1}{2} = 3n+2 \tag{5}$$

Notice that $3n+2 > 2n+1$, and the algorithm tests the numbers in ascending order. This step tells us that there is no need to test numbers of the form $3n+2$ because it was already tested in the $2n+1$ test. This will allow us to reduce the number of tested values by 33.33 %. We call this optimization modulo 3 sieving. The same procedure can be used for the initial numbers of the general form $3^k n + m$. For example, using 3^2 sieve, the percentage of eliminated initial numbers is 44.44 %. However, higher 3^k sieves bring almost no acceleration (for $3^6 = 729$ the percentage is only 45.95 %). Table 1 summarizes the percentage of sieves up to 3^6. Note that there is no improvement between 3^2 and 3^3 sieves.

Using a 3^k sieve is a time-critical operation. Therefore, much optimization was devoted to it. For example, the following piece of code tests whether an initial number passes the 3^2 sieving. It is based on the identity $2^{60} \equiv 1 \pmod 9$. The problem is how to calculate modulo 9 from the 128-bit data type using only the 64-bit type. Since we know that $2^{60} \equiv 1 \pmod 9$, we can shift the number 60 bits to

Table 1 Residues modulo 3^k and percentage of initial values eliminated by this sieve

Sieve size	Residues	Eliminated (%)
3^1	$\{2\} \pmod 3$	33.3333
3^2	$\{2, 4, 5, 8\} \pmod 9$	44.4444
3^3	$\{2, 4, 5, 8, 11, 13, 14, 17, 20, 22, 23, 26\} \pmod{27}$	44.4444
3^4	–	45.6790
3^5	–	45.6790
3^6	–	45.9533

the right (from position 2^{60} to position 1). This is how we break a 128-bit data type into several 64-bit ones. We then simply call the modulo operation on a 64-bit data type.

```
static int is_live_in_sieve9(uint128_t n)
{
        uint64_t r = 0;

        r += (uint64_t)(n)         & 0xffffffffffffffff;
        r += (uint64_t)(n >>  60) & 0xffffffffffffffff;
        r += (uint64_t)(n >> 120);

        r = r % 9;

        return r != 2 && r != 4 && r != 5 && r != 8;
}
```

3.3 Sieve 2^k

Let T^k denote the kth iteration of the function T. 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 $T^k(n_L)$. We basically split n as $n = 2^k n_H + n_L$, i.e., k lowest significant bits (least significant bits) along with the rest of the high significant bits. Competitive programs use this equation to perform k steps at once (two tables have the size of 2^k entries, and indices correspond to n_L). However, we used this acceleration technique to build a large sieve (the size of the sieve is 2^k bits). Using the sieve, we test only those numbers that (a) do not converge or (b) join the path of a lower number in k steps. This is the reason why the entry of such a sieve is just one bit (2^k bits in total). As might be expected, the acceleration obtained from this method is significant.

For those who find the above explanation unclear, we offer a slightly simpler view of the matter. Look at Equation (6). It gives you a way to calculate k iterations of a $T(n)$ function at once. On the left side, you see $T^k(2^k n_H + n_L)$, which means that to

calculate k iterations, we have to split the number into two parts $n = 2^k n_H + n_L$. For all n_L we precalculate (a) $\text{odd}(n_L)$ and (b) $T^k(n_L)$. Now we can jump k steps forward just using two pre-calculated tables. It holds that $0 \leq n_L < 2^k$. Now the question is whether the initial value is greater than the value on the right side of (6), formally $n_H 3^{\text{odd}(n_L)} + T^k(n_L) < 2^k n_H + n_L$ for all $n_H > 0$. If the answer is yes, the ending number drops below the starting number, and we can end the verification. This is the principle of the 2^k sieve.

3.4 Solving congruence classes concurrently

Our CPU and GPU programs can concurrently verify 2^k (k matches the sieve size) numbers of the same congruence class. This mainly means that the program verifies the work units having the size of $2^{\text{task size}}$ numbers and solves the lowest k bits at once, where $k <$ task size. After k least significant bits, the code paths diverge. This results in a verification of individual numbers up to $2^{\text{task size}}$.

For example, consider Algorithm 2. The concurrent algorithm solves a class of 2^k numbers having k least significant bit in common. The 2^k sieve is applied before the concurrent algorithm starts (note that the lowest k bits are the index into the 2^k sieve). In contrast, the 3^k sieve is applied when solving bits greater than k.

4 Distributed computing

This section deals with distributing individual tasks to thousands of concurrent workers. All the algorithms described in the previous section are implemented for the CPU and GPU and referred to as the worker. The GPU code is written in OpenCL (both NVIDIA and AMD are supported), whereas the CPU code is written in C. The code uses 128-bit integer compiler extensions. The CPU worker uses the GMP library to resolve computation above 128 bits (for arbitrary precision arithmetic).

All αs (see Algorithm 2) that occurred during the convergence test of the range are summed together to raise the checksum (proof of work). These checksums are recorded on the server. The maximum value of $n_{\max}$ that occurred during the convergence test for a given interval is also detected and recorded on the server. Our implementation can verify work units of $2^{\text{task size}} = 2^{40}$ 128-bit numbers. We experimented with several values for task size. The selected value turned out to be small enough to be processed in a maximum of a few hours on conventional CPUs and large enough to allow such units to be managed by a regular server. The server manages 2^{32} work units, which allows up to 2^{72} numbers to be explored. A record of each work unit occupies 322 bits on the server (161 gigabytes for the whole 2^{32} space). The data is stored in these 322 bits comprising a calculation time, number of 128-bit overflows, offset of maximum value encountered in the calculation, checksum, client ID, and some bit flags.

Table 2 provides examples of ranges that are calculated by every worker from task id.

Table 2 Ranges of numbers for workers depending on task id. Ranges are expressed using hexadecimal numbers

Task id	Range of numbers
0	0x0 – 0x10000000000
1	0x10000000000 – 0x20000000000
1000000000	0x3b9aca000000000000 – 0x3b9aca010000000000
2175961363	0x81b289130000000000 – 0x81b289140000000000

The architecture of our project is shown in Fig. 1. It consists of three components—the server, clients, and workers. The server runs in a single instance in our university's infrastructure. Clients are run by job scheduling systems (PBS, TORQUE, SGE, Slurm) in the infrastructure of various supercomputers in Europe. A single computing node can host a single client or several clients. Clients are multi-threaded and can serve several CPU or GPU workers. Clients spawn workers and correspond to individual CPU cores or GPUs.

The server and clients run on different computers. They communicate with each other using the TCP/IP protocol. The communication protocol has been specially designed to have low latency (all client-to-server requests precede server-to-client responses). Individual requests that the server can serve include Request Assignment, Request Lowest Incomplete Assignment, Return Assignment, Interrupt Assignment, Request Multiple Assignments, and others.

Conversely, the client and workers run on the same computer. They communicate with each other via the worker's standard output (stdout). For these purposes, we have developed an extensive text protocol. Messages in this log include the time elapsed while solving the work unit, checksum, the initial value for the maximum value reached during the solution of the work unit, the sequence number of the work unit (task id), and others.

This whole system can serve thousands of workers working simultaneously. It uses various supercomputers for this. MetaCentrum operates and manages distributed computing infrastructure consisting of computing resources owned by CESNET (an association of universities of the Czech Republic and the Czech Academy of Sciences) in the Czech Republic. IT4Innovations National Supercomputing Center operates the most powerful supercomputing systems in the Czech Republic. At present, IT4Innovations runs three supercomputers, including the Czech most powerful supercomputer, Karolina. The LUMI supercomputer is the most powerful supercomputing system in Europe (and, at the time we were solving our project, third most powerful in the world). A few examples of how powerful the system can be are shown in Table 3. MetaCentrum supercomputer provides up to 2 000 CPUs cores,

Fig. 1 An architecture of the distributed computing project. The numbers indicate the multiplicity of individual boxes. The server always runs in a single instance

Table 3 Supercomputers used in the computational verification of the Collatz conjecture. WU/sec stands for work units per second. Measured at the end of May 2023

Computing resources involved	CPU workers	GPU workers	WU/sec
MetaCentrum	1 949	0	7.7097
MetaCentrum, IT4I (16 nodes)	3 989	0	11.6041
MetaCentrum, IT4I (32 nodes)	6 043	0	15.8773
MetaCentrum, IT4I (32 nodes), LUMI (16 nodes)	6 048	128	39.6292
MetaCentrum, LUMI (24 nodes)	1 944	192	43.4859
MetaCentrum, LUMI (52 nodes)	1 445	416	66.0541

whereas Karolina provides 128 cores per node (AMD Zen 2 EPYC 7H12), and finally LUMI supercomputer provides eight AMD MI250X GPUs per node.

Notice in Table 3 that the resulting performance is massively higher when GPUs are included compared to the CPU-only situation. During our verification, we consumed 12 395 CPU-years (average time per solving a single work unit is 6 min and 25 s) and 159 GPU-years (average time is 7 s). This includes even very slow machines. The typical average duration at the time of writing is much lower.

Someone might be interested in how individual assignments are distributed to workers. Each client asks the server to allocate a range to search. The server allocates the smallest unassigned range and sends it to the client. The client then passes the range to the worker (workers can occupy one CPU core or one GPU). Since multiple workers can run in parallel (multiple threads, or multiple GPUs) on a single computing node, each client actually requests several ranges and distributes them to multiple workers.

5 Evaluation

This section provides a performance evaluation of the algorithms from Sect. 3. The comparison is performed on AMD Ryzen Threadripper 2990WX and NVIDIA GeForce RTX 2080 Ti. The evaluation starts with comparing 2^k sieve sizes and 2^k sieve compression. Then, we evaluate the benefit of 3^k sieves. Finally, we deal with processing congruence classes concurrently. In all cases, we use a work unit of the size of 2^{40} (the time is given per 2^{40} numbers). The only evaluation metric is time given in seconds. At the end of the section, we also mention the speed-up factor. The paper evaluates these techniques: using a 2^k sieve, solving congruence classes concurrently, and using a 3^k sieve.

We experimented with many sieve sizes and concluded that the sieve size 2^{34} is optimal for the CPU implementation, while the size 2^{24} is optimal for GPUs. See Table 4a and b. Note that 2^{34} bits equals 2 gigabytes. So, the consequence is a huge memory footprint. However, we have found that these sieves are formed by several constantly repeated bit patterns. More precisely, such sieves are formed by only fifty constantly repeated 64-bit patterns (which can be stored using

Table 4 Comparison of sieve sizes. The best result is in bold

Sieve size [bits]	Time [secs]
(a) Several promising sieve sizes used on the CPU. Sieve 3^2 was used.	
2^{30}	264.4728
2^{32}	231.8417
2^{34}	**223.2903**
2^{36}	243.2161
Sieve size [bits]	Time [secs]
(b) Several promising sieve sizes used on the GPU. No 3^k sieve was used.	
2^{16}	6.0532
2^{24}	**5.0139**
2^{32}	5.2693

indices in a small look-up table). In our project, we use 8-bit indices. However, the compression ratio can reach the value of around 1:10 if 64-bit patterns are represented by a 6-bit index.

Now, we compare the benefits of the 2^k sieve compression using a look-up table. The comparison is shown in Table 5a and b. On the CPU, we can see that as the sieve size increases, the compression becomes less and less effective. In the case of 2^{24} sieve, the speedup is about 3 %, whereas in the case of 2^{34} sieve the speedup is zero. However, we must realize that compression also saves disk space (1:8 ratio) and allows files to be copied to computing nodes faster. Without compression, large sieves could not be distributed to computing nodes. On the GPU, the compression provides speedup using 2^{32} sieve. However, 2^{24} sieve is faster; compression does not help here.

Table 5 Speed comparison of an enabled/disabled sieve compression. The best result is in bold

Sieve	Compression disabled [secs]	Compression enabled [secs]
(a) Compression on the CPU. All experiments are made with a 3^2 sieve and concurrently solve 2^k congruence classes.		
2^{24}	385.8868	375.5922
2^{32}	237.6361	231.3289
2^{34}	**219.9232**	220.7683
Sieve	Compression disabled [secs]	Compression enabled [secs]
(b) Compression on the GPU. All experiments are made with a 3^2 sieve.		
2^{16}	6.2159	6.6444
2^{24}	**4.4690**	4.7300
2^{32}	5.6153	5.2906

Table 6 Efficiency comparison of 3^k sieves. 2^k sieve compression was disabled in all cases. The best result is in bold

Sieve 3^k	Time on CPU [secs]	Time on GPU [secs]
not used	317.5541	5.1053
3^1	245.0160	**4.0477**
3^2	**214.8214**	4.2992

Table 7 Speed comparison of an enabled/disabled concurrently solving congruence classes. The best result is in bold

Modulo	Algorithm disabled [secs]	Algorithm enabled [secs]
(a) Speed comparison on the CPU.		
2^{24}	1 008.7780	387.0207
2^{32}	757.4297	233.4601
2^{34}	711.5311	**217.5859**
2^{36}	686.7216	234.2653
Modulo	Algorithm disabled [secs]	Algorithm enabled [secs]
(b) Speed comparison on the GPU.		
2^{24}	**4.4866**	overflow
2^{32}	5.7281	9.2971

Without compression, it is possible to distribute the 2^{32} sieve to the CPU clients. The compression enables the distribution of the 2^{34} sieve. From Table 4a, the speedup is about 4 %.

We also compare the efficiency of 3^k sieves. See Table 6. 2^{34} sieve is used on the CPU, while 2^{24} sieve is used on the GPU. The CPU computes the same congruence classes concurrently. We can see that 3^k sieves bring some acceleration. On the CPU, it is purely about the number of reduced initial numbers. On the GPU, a 3^2 sieve seems to be breaking thread convergence; 3^1 is slightly faster.

As another evaluation point, we compare the efficiency of computing the same congruence classes concurrently. The comparison is given in Table 7a and b. The time is given per a work unit of the size of 2^{40}, and sieve 3^2 is used. Overflow means the `ulong` data type (OpenCL) is insufficient for intermediate calculations. It can be seen that, on the CPU, this algorithm brings a massive speedup (e.g., 3.27× for sieve 2^{34}). The situation on GPUs is the opposite; the algorithm does not bring any acceleration.

We now look at how much cumulative speedup each algorithm brings. See corresponding Tables 8 and 9. The baseline algorithm also computes the checksum and maximum value reached during the computation. The baseline algorithm is

Table 8 Overall comparison on the CPU

Algorithm	Time [secs]	Cumm. speedup
baseline algorithm	5 643.9640	1.00×
using 2^{34} sieve	949.9505	5.94×
solving congruence classes concurrently	320.4502	17.61×
using 3^2 sieve	220.0285	25.65×

Table 9 Overall comparison on the GPU

Algorithm	Time [secs]	Cumm. speedup
baseline algorithm	12.8870	1.00×
using 2^{24} sieve	5.0535	2.55×
using 3^1 sieve	4.2248	3.05×

Table 10 Timeline of our project verifying the convergence of the Collatz conjecture

Date	Status
2019-09-04	Started the project
2020-05-07	The convergence of all numbers below 2^{68} is verified
2021-12-10	The convergence of all numbers below 2^{69} is verified
2023-07-09	The convergence of all numbers below 2^{70} is verified
2023-11-03	The convergence of all numbers below 1.5×2^{70} is verified
2025-01-15	The convergence of all numbers below 2^{71} is verified

Algorithm 2. Showing the speedup over Algorithm 1 would not be fair because it is not implemented in an optimized form. The tables mentioned above show that the best GPU implementation is 52.1× faster than the best CPU implementation. The total speedup from the first algorithm we started with on the CPU to our best algorithm on the GPU is 1 335.9×.

6 Results

At the time of writing this article, we have managed to verify the convergence of the Collatz conjecture for all numbers up to the limit of 2^{71} (which is equal to $2\,048 \times 2^{60}$). This is the moment when the length of a non-trivial cycle rises to 355 504 839 929 [12]. See Table 10 for a timeline from the start of our project.

The n is called the path record if, for all $m < n$, the inequality $t(m) < t(n)$ holds, where $t(n)$ is the highest number occurring in the sequence starting at n. During the verification, we found five new path records (sequence A006884 in The On-Line Encyclopedia of Integer Sequences). The initial values for path records are

– 274 133 054 632 352 106 267,
– 1 378 299 700 343 633 691 495,
– 1 735 519 168 865 914 451 271,
– 1 765 856 170 146 672 440 559, and
– 2 358 909 599 867 980 429 759.

Lagarias and Weiss [13] predicted using the large deviation theory for random walks that the highest number occurring in the sequence for a path record n grows like n^2. The results recorded up to 2^{71} agree with this prediction.

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.[1]

7 Conclusion

This article presents our project aiming to computationally verify the convergence of the Collatz conjecture. In our previous article, in 2020, we verified the convergence up to the limit 2^{68}. This paper presents a result that further pushes this limit to 2^{71}.

We presented several sub-algorithms that bring some acceleration to our baseline algorithm. The paper further dealt with the distribution of individual tasks to thousands of parallel workers. While doing so, our system runs on several supercomputers in Europe. The workers compute their tasks either on the CPU or on the GPU. On GPU, the code is written in OpenCL, and both NVIDIA and AMD cards are supported.

We further evaluated our algorithms and found that our best GPU implementation is 52× faster than our best CPU implementation. The total speedup from the first algorithm we used several years ago on the CPU to our best algorithm today on the GPU is 1 335×.

Acknowledgements Computational resources were provided by the e-INFRA CZ project (ID:90254), supported by the Ministry of Education, Youth and Sports of the Czech Republic.

Funding Open access publishing supported by the institutions participating in the CzechELib Transformative Agreement.

[1] https://github.com/xbarin02/collatz, commit 53c2a06.

References

1. Lagarias JC (ed) (2010) The Ultimate Challenge: The 3x + 1 Problem. American Mathematical Society
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. Barina D (2021) Convergence verification of the Collatz problem. J Super Comput 77(3):2681–2688. https://doi.org/10.1007/s11227-020-03368-x
5. Dunn R (1973) On Ulam's Problem. University of Colorado at Boulder, Tech. rep
6. Leavens GT, Vermeulen M (1992) 3x+1 Search programs. Comput Math Appl 24(11):79–99. https://doi.org/10.1016/0898-1221(92)90034-F
7. Roosendaal E (2019) personal communication
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. Hercher C (2018) Über die Länge nicht-trivialer Collatz-Zyklen. Die Wurzel 6 and 7
10. Honda T, Ito Y, Nakano K (2017) GPU-accelerated exhaustive verification of the Collatz conjecture. Int J Netw Comput 7(1):69–85
11. Silva TOE (1999) Maximum excursion and stopping time record-holders for the $3x + 1$ problem: computational results. Math Comput 68(225):371–384. https://doi.org/10.1090/S0025-5718-99-01031-5
12. Hercher C (2023) There are no Collatz m-cycles with $m \leq 91$. J Integer Seq 26(3):1–22
13. Lagarias JC, Weiss A (1992) The $3x + 1$ problem: two stochastic models. Annals Appl Probab 2(1):229–261. https://doi.org/10.1214/aoap/1177005779