# Report Assignment2 - Parallel Collatz Implementation

Angelo Nardone

## 1 Introduction

This report concerns the second assignment of the SPM course, which focuses on the computation of the Collatz sequence, defined below.

**Definition 1** (Collatz Sequence and Collatz Problem)**.** Given an initial integer $x_0 = n$, the **Collatz sequence** is defined as:

$$x_{i+1} = \begin{cases} \frac{x_i}{2} & \text{if } x_i \equiv 0 \pmod 2 \\ 3x_i + 1 & \text{if } x_i \equiv 1 \pmod 2 \end{cases} \tag{1}$$

The sequence terminates when $x_i = 1$. The **Collatz problem** asks whether, for any given starting integer, the sequence always reaches 1, and if so, in how many steps (i.e., the smallest $i$ such that $x_i = 1$).

Although the **conjecture remains unproven**, all tested inputs have empirically resulted in termination. The objective of the assignment was to develop a sequential implementation and **two parallel implementations** of a program capable of processing one or more input ranges of integers. For each range, the program identifies the integer that generates the longest Collatz sequence and reports the corresponding number of steps.

The report begins with a theoretical analysis of the problem using the **Work-Span model**, followed by a description of the **implemented solutions**. **Experimental results** are then presented, and the report concludes with a discussion of the observed outcomes.

## 2 Work-Span Model

The theoretical analysis begins with the application of the **Work-Span model**. As a first step, a **directed acyclic graph** (DAG) is constructed to represent the dependencies among the tasks involved in solving the problem. In this graph, each **node corresponds to a task** (a unit of work) and is labeled with the **time required** for its execution, while the **edges represent dependencies**. Specifically, an edge from one node to another indicates that the second task cannot begin until the first has been completed.

To simplify the analysis without loss of generality, the input is assumed to consist of a **single range**. This assumption is justified by the fact that computations on **different ranges are independent** and follow the same structure.

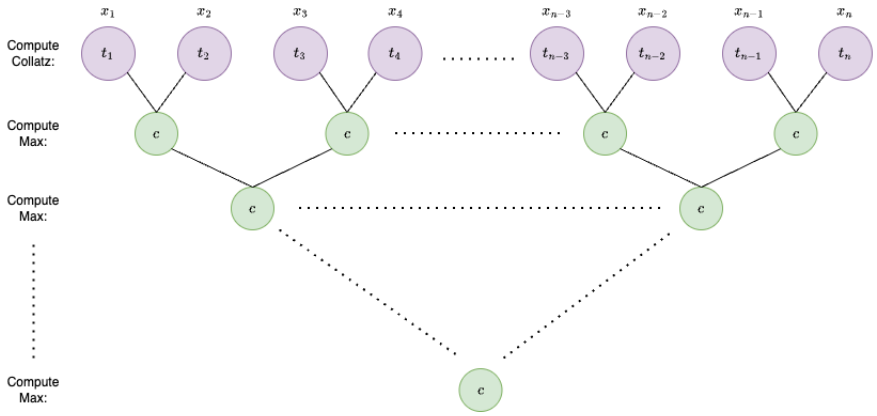


Figure 1: Dependency DAG for computing the longest Collatz sequence in a single range $[x_1, x_n]$.

Given an input range $[x_1, x_n]$, the corresponding dependency DAG is shown in Figure 1. The first level of nodes represents the **computation of the number of steps** in the Collatz sequence starting from each $x_i$. Each node

is independent from the others and is associated with a **computation time** $t_i$, which may vary significantly due to the input-dependent nature of the sequence length. In general, $t_i \neq t_j$ for $i \neq j$. The remaining levels of the DAG are responsible for **computing the maximum** over pairs of results. Each `max` operation is assumed to require a **constant time** $c$, independent of the specific values being compared.

**Observation 1.** The DAG has the structure of a **complete and balanced binary tree**. The first level contains $n$ nodes (corresponding to each Collatz computation), while the remaining $n-1$ nodes perform pairwise `max` operations. The overall depth of the tree is $\lceil \log_2(n) \rceil + 1$.

The **work** $T_1$ and the **span** $T_\infty$ can now be derived. The work represents the total time required for a **sequential execution** and is obtained by summing the durations of all tasks:

$$T_1 = \sum_{i=1}^{n} t_i + (n-1) \cdot c \tag{2}$$

The span corresponds to the execution time assuming an **infinite number of processors**, with all tasks at the same level executed in parallel. It is given by the duration of the longest task at the first level, plus the cumulative cost of the $\lceil \log_2(n) \rceil$ levels of `max` operations:

$$T_\infty = \max_{1 \leq i \leq n} t_i + \lceil \log_2 n \rceil \cdot c \tag{3}$$

**Observation 2.** In the case of **multiple ranges**, Equations (2) and (3) should be extended by introducing an **outer summation** over all ranges.

The values of $T_1$ and $T_\infty$ will be estimated numerically in the experimental section for a fixed input range. At this stage, it is possible to make the following **theoretical considerations**. For a parallel execution on $p$ processors, the following inequality holds:

$$\frac{T_1}{p} \leq T_p \boxed{\leq} \frac{T_1}{p} + T_\infty \tag{4}$$

The upper bound ($\boxed{\leq}$) follows from **Brent's Theorem**, assuming that $T_\infty \ll T_1$. Under this condition, the execution time satisfies the approximation $T_p \approx \frac{T_1}{p} + T_\infty$. This assumption is typically valid for sufficiently large input ranges, making the bound applicable in practical scenarios.

As for the **speedup** $S(p)$, the following bounds apply:

$$\min\left(p, \frac{T_1}{T_\infty}\right) \leq S(p) \leq \frac{p}{1 + p \cdot \frac{T_\infty}{T_1}} \tag{5}$$

In scenarios where $\frac{T_1}{T_\infty} \gg p$, the speedup approaches $S(p) \approx p$. This indicates that, when $T_\infty \ll T_1$, as expected in the present setting, **near-linear speedup can be achieved** for a reasonable number of processors.

# 3 Code Implementations

This section provides an overview of the **code implementations**. The function `collatz` refers to the one provided in the assignment track. Before presenting the code, a key observation is in order.

**Observation 3.** Neither the sequential nor the parallel implementations **strictly follow the DAG of the Work-Span model** illustrated in Figure 1. That graph represents an idealized version of task dependencies and does not account for **memory limitations** or **potential cache issues** arising when computing and storing Collatz sequences for a large number of elements simultaneously.

## Sequential Code

The **sequential version** adopts a straightforward structure, deliberately avoiding optimizations such as memoization via dictionaries to cache previously computed steps. This ensures that the parallel implementations **perform the same operations** as the sequential one, allowing for a **fair comparison**. Adding such optimizations would have unnecessarily complicated the implementation without contributing to the primary goal of the assignment.

The structure consists of an **outer `for`-loop** that iterates over the input ranges, an **inner `for`-loop** that processes each integer within the current range, and a helper variable that keeps track of the maximum number of steps found in that range.

### Parallel Code: Static Scheduling

The first parallel version described is based on a **static scheduling policy**, specifically a **block-cyclic policy**, as suggested in the assignment track. Among three implemented variants, only the most efficient one has been retained and is presented below.

This implementation remains relatively simple. All **threads** are created using `emplace_back` and execute the `block_cyclic_policy` function. This function receives the `thread_id` and a `CollatzData struct` containing the **range boundaries**, a shared `max_steps` **vector** (with size equal to the number of ranges), and a `mutex`. Each thread iterates over all ranges using an **outer** `for` **loop**. Within each range, an **inner loop** ensures that only the designated chunks are processed by the thread, computing the **maximum locally**. The assignment policy ensures that the first chunk of each range is always assigned to thread 1. After completing all chunks of a range, each thread uses a `lock_guard` to safely update the corresponding position in the shared `max_steps` vector.

**Observation 4.** For completeness, the two discarded versions are briefly mentioned and **provided in a separate file**. The first version relies on `promises` and `futures` to return the maximum value per range for each thread. This approach is the slowest and requires maintaining a separate vector of maxima per thread. The second version mirrors the final one but includes **additional arithmetic** to ensure that if thread $i$ computes the last chunk of range $j$, then thread $i+1$ starts the first chunk of range $j+1$. This variant is slightly slower, likely due to the overhead introduced by the additional arithmetic operations.

### Parallel Code: Dynamic Scheduling

The second parallel implementation adopts a **dynamic scheduling policy**, in which chunk assignments are made **on demand** at runtime. This strategy improves load balancing, particularly when the cost of computing the Collatz sequence varies significantly within a range.

**Observation 5.** Since the number of **steps in a Collatz sequence does not scale uniformly** with the magnitude of the input, execution times can vary greatly across ranges. As a result, a static distribution (although balanced in terms of assigned elements) may lead to **imbalanced workloads**. In contrast, the dynamic policy allows threads to pick up new tasks as they complete previous ones, reducing idle time. The trade-off is a overhead due to synchronization to **avoid redundant computations**.

The implementation relies on two key structures. The first is the shared `CollatzData` struct, containing the input **ranges**, a vector `max_steps`, a `mutex` for synchronized updates, and a vector of `DynamicTaskManager` per range. `DynamicTaskManager` is a struct that holds an `atomic` counter tracking the **current position** within the range, the chunk size, and the range end. Its core **method** `get_next_task` performs a **lock-free atomic** increment to distribute chunks among threads.

Each thread executes the `dynamic_policy` function and independently iterates over all ranges. For each range, the thread repeatedly requests new chunks via `get_next_task`. Once no more chunks are available, it proceeds to the next range. After processing all assigned chunks of a range, the thread updates the corresponding entry in `max_steps` using a `lock_guard` to ensure thread-safe access.

This design guarantees that **no chunk is processed more than once** while minimizing contention. Synchronization with mutex is limited to a single critical section per range, ensuring efficient parallel execution with dynamic load balancing.

# 4 Experiments and Results

This section presents the results obtained using the implementations previously described. Although several experiments were conducted, for simplicity, only one representative test is reported. This test includes three input ranges: a **small range** (1–1000), a **medium range** (10,000–1,000,000), and a **large range** (50,000,000–100,000,000). So, the command used is, for example:

```
./parallel_collatz -d -n 16 -c 4 1-1000 10000-1000000 50000000-100000000
```

The **sequential version** was executed 10 times, and the average execution time was computed. The same procedure was applied to the parallel versions, both **static** and **dynamic**, for each configuration. The number of threads and chunk sizes varied within the set {1, 2, 4, 8, 16, 32, 64}, leading to a total of **64 tested configurations**.

**Observation 6.** All experiments were run on **internal nodes** of the `spmcluster`. Each node has 32 physical cores, so the optimal number of threads is expected to be 32.

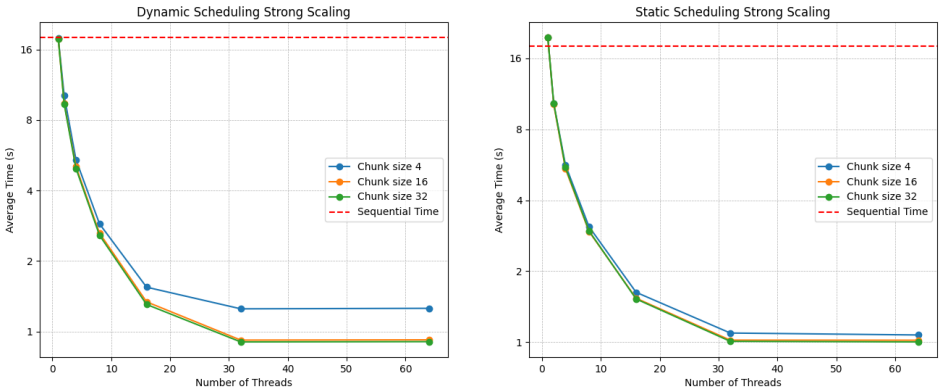


Figure 2: Strong scaling results for dynamic and static scheduling.

Figure 4 shows the **strong scaling** results. The red dashed line represents the sequential execution time, while the curves represent different fixed chunk sizes, illustrating how execution time varies with the number of threads.

From the figure, it is evident that the dynamic scheduling approach is slightly **faster** than the static one, as anticipated. The static version is **relatively unaffected** by chunk size (smaller chunk sizes cause only minimal increases in execution time). However, this is not the case for the dynamic version: when the chunk size is 4, execution time increases significantly, especially with a large number of threads. This is due to the overhead caused by more frequent chunk assignments and thread management. In the static approach, each thread is assigned a fixed portion once, keeping overhead constant. Although smaller chunk sizes increase the number of loop iterations, this results in only minor slowdowns in the static version compared to the overhead observed in the dynamic one. There is also a noticeable, though slight, increase in execution time when using 64 threads, which is consistent with expectations given the 32-core hardware limit.

In conclusion, both the static and dynamic implementations demonstrate **good strong scalability** up to 32 threads. The appendix includes a table with detailed timing results and a log-log version of Figure 4, offering a clearer view of the scalability and the small slowdown due to thread allocation near the 32-thread limit.

**Observation 7.** From Figure 4, it can be observed that while the static version with 1 thread and chunk size 1 is slower than the sequential implementation, as expected, the dynamic version with 1 thread and chunk size 1 performs slightly faster (by a few milliseconds). This unusual behavior occurs only for very large input ranges (in this case, [50,000,000–100,000,000]) and is likely due to how the compiler handles `for`-loops. In the dynamic case, the loop is split into many small chunks, while the sequential version uses a single large loop, which appears to cause **cache inefficiencies** during compilation or execution.

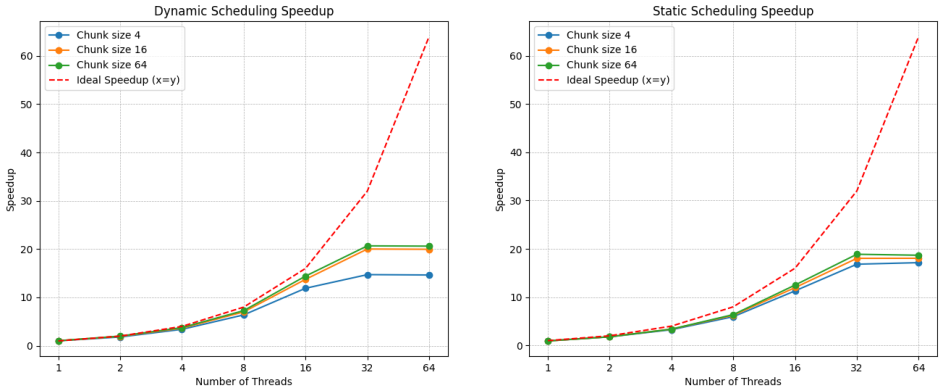


Figure 3: Speedup comparison. The red dashed line shows ideal speedup; each curve corresponds to a different chunk size.

The **speedup** is now analyzed, defined as:

$$S(p) = \frac{T_{\text{seq}}}{T_p}$$

Results are shown in Figure 5 (again, a log-log version is also available in the appendix). The red dashed line represents the **ideal speedup**, i.e., $T_p = T_{\text{seq}}/p$. Both static and dynamic versions achieve near-ideal speedup up to 32 threads.

As before, a slight divergence from the ideal curve is visible just before reaching 32 threads, again due to overhead. The same considerations from the previous figure apply here.

The code described in the previous section thus exhibits **excellent scalability** and **near-linear speedup**, as expected. Although edge cases (such as many small ranges) may slightly alter the curves, the general trend remains unchanged. The general setting shown here was chosen because it better demonstrates the algorithm's effectiveness in common scenarios.

## Experimentation on the Work-Span Model

As anticipated, a numerical evaluation of the Work-Span model described in Section 2 is now presented. It is important to note that the actual implementations do not strictly follow the structure defined by the model, due to practical considerations such as memory and cache management overhead.

Before proceeding, a few assumptions are introduced to simplify the analysis. Each `max` operation is assigned a cost of 1, and each Collatz step is similarly assumed to cost 1. While this simplification neglects the actual differences in computational cost (e.g., between a division and a multiplication-plus-addition within the Collatz step, or between various `max` evaluations) such operations are **sufficiently lightweight** to justify this abstraction. The **bottleneck remains the number of steps** executed per Collatz computation. Accordingly, the cost of each node is defined as $t_i = \#\text{steps of Collatz}(i)$.

Under these assumptions, the model is applied to the **smallest range previously analyzed**, namely $[1, 1000]$. Using Equations (2) and (3), the work and span are computed as follows:

$$T_1 = \sum_{i=1}^{n} t_i + (n-1) \cdot c = \sum_{i=1}^{1000} \#\text{Collatz steps}(i) + 999 = 59542 + 999 = 60541$$

$$T_\infty = \max_{1 \le i \le n} t_i + \lceil \log_2 n \rceil = 178 + 10 = 188$$

Even in this small range, the inequality $T_1 \gg T_\infty$ clearly holds, with $\frac{T_1}{T_\infty} \approx 322$, which significantly exceeds the maximum number of threads used in the experiments (32). For larger input ranges or multiple ranges, the gap between $T_1$ and $T_\infty$ can only increase further.

It follows that all theoretical upper and lower bounds introduced in Section 2 remain applicable, and a speedup of the form $S(p) \approx p$ can be expected. Although the actual implementation deviates from the idealized DAG structure of the Work-Span model, the **observed behavior is consistent with the theoretical predictions**. Experimental results confirm a near-linear speedup, reinforcing the validity of the model's expectations.

# 5 Conclusion

In this report, the second assignment's requirements have been addressed. A theoretical model based on the **Work-Span model** was developed, which, although not perfectly reflecting the code's behavior, provided an **estimate of the expected speedup**. In the experimental section, this estimate was confirmed, showing a **nearly linear speedup**, even for small ranges, where $T_1 \gg T_\infty$. The observed performance closely matched this expected behavior.

The **most challenging** aspect of the work was managing the parallel code, particularly the dynamic version. The static approach also presented difficulties, as it marked the first real encounter with threads, mutexes, and atomics, given that computer science was not my focus during the undergraduate studies. This is evidenced by the initial choice to implement a version using promises and futures, which appeared simpler at first but later proved unsuitable. Instead, coming from a mathematics background, the theoretical modeling aspect was **less challenging**, as it aligned with familiar concepts. Modeling the problem using the Work-Span model was both enjoyable and insightful.

It should be noted that there are **potential improvements** to the code, both for the static and dynamic parts. For the static version, replacing the mutex with an atomic operation could yield better results. Another potential optimization, which was tested but did not improve performance in this case, involved using a **vector of mutexes** instead of a single mutex. The idea was that this approach would reduce contention when multiple threads attempt to update the maximum value for different ranges simultaneously. With a single mutex, all threads are blocked, while a vector of mutexes would allow for more granular synchronization. Similar considerations apply to the dynamic version. Additionally, the code could likely be written in a more elegant or functional style (e.g., using lambda functions), an approach that was not considered during the initial implementation phase.

# A Appendix

## A.1 Log-Log Figures

Figures 4 and 5 present the same results as their counterparts in the main text, but using a **log-log scale**. This representation allows for a clearer visualization of both **strong scaling** behavior and **speedup** trends.

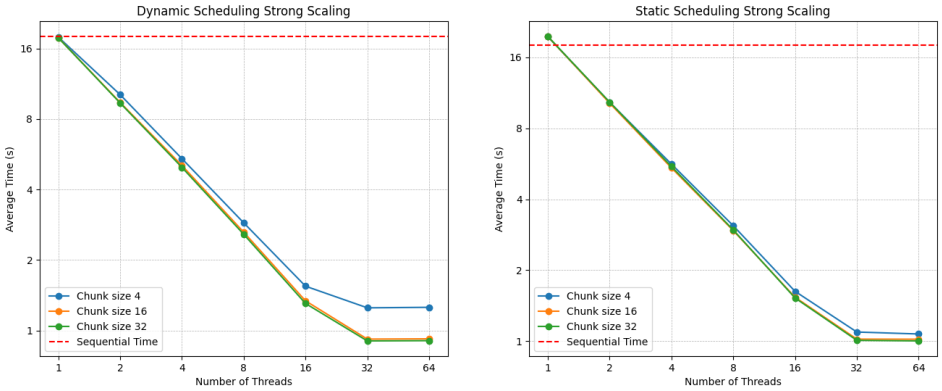


Figure 4: Strong scaling results for dynamic and static scheduling, displayed on a log-log scale.

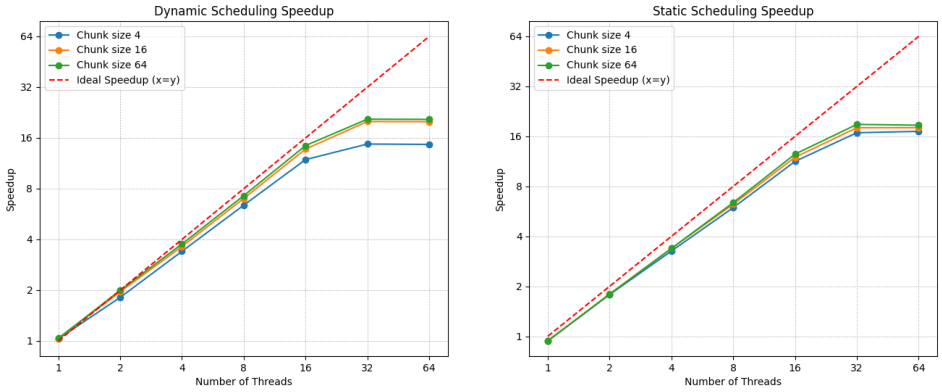


Figure 5: Speedup comparison. The red dashed line indicates ideal speedup. Each curve corresponds to a different chunk size.

## A.2 Timing Table

The Table 1 reports the execution times collected during the experiments described in Section 4.

| **Threads** | **Chunk 4** | | **Chunk 16** | | **Chunk 32** | |
|---|---|---|---|---|---|---|
| | **Dyn** | **Stat** | **Dyn** | **Stat** | **Dyn** | **Stat** |
| 1 | 17.889980 | 19.589320 | 17.774570 | 19.622010 | 17.763380 | 19.629980 |
| 2 | 10.163130 | 10.317760 | 9.411614 | 10.231580 | 9.361173 | 10.337420 |
| 4 | 5.415941 | 5.641237 | 5.065316 | 5.436891 | 4.971480 | 5.507271 |
| 8 | 2.882502 | 3.089560 | 2.629163 | 2.946542 | 2.575312 | 2.966351 |
| 16 | 1.546577 | 1.627626 | 1.339251 | 1.536466 | 1.305030 | 1.525094 |
| 32 | 1.250374 | 1.093536 | 0.919325 | 1.020098 | 0.903101 | 1.008374 |
| 64 | 1.256615 | 1.072895 | 0.921747 | 1.019272 | 0.905342 | 1.002464 |

Table 1: Execution time for different combinations of `chunk_size`, `num_thread`, and `dynamic` scheduling