Assignment 2

Parallel Collatz Implementations: Static vs. Dynamic Scheduling

Giuseppe Gabriele Russo 583744

April 2025

Abstract

In this report, we discuss two parallel implementations of the Collatz conjecture, focusing on static and dynamic scheduling approaches. We first provide a detailed description of each implementation, then present a theoretical estimate using the Work–Span model. We follow this with a performance analysis, highlighting the main challenges encountered during development and the solutions adopted. Finally, we draw conclusions on the efficiency and scalability of the two methods by comparing the experimental results against the theoretical bound.

1 Introduction

Although the Collatz conjecture remains an unsolved problem in mathematics, its computations are highly amenable to parallelization, given that each number's sequence can be evaluated independently. However, there is no known algorithm that can predict in advance which numbers will require more iterations: a much larger number does not necessarily need more steps than a smaller one. Consequently, load balancing among threads is inherently difficult, since we cannot anticipate the "heaviest" portions of the input ranges. In our implementation, the main challenge lies in tracking the maximum number of steps for each range while distributing the workload effectively.

We explore two parallelization strategies:

- **Static Scheduling (Block-Cyclic)**: We assign fixed-size blocks to the threads in a cyclic manner, ensuring a more uniform workload even when certain numbers require a high number of steps.
- **Dynamic Scheduling (Thread Pool)**: We rely on a provided thread pool implementation, which places chunks of work in a shared queue that all threads continuously fetch from, adapting to the load in a dynamic fashion.

In the following sections, we detail both approaches, present experimental performance results, and discuss key challenges, including load balancing, synchronization overhead, and efficiently maintaining the global maximum number of steps within each range.

2 Implementation Details

Before implementing the parallel versions of the Collatz computation, a few key optimizations were applied to the baseline sequential algorithm. These changes aimed to improve efficiency and reduce unnecessary computational overhead. Specifically:

- **Step Combination for Odd Numbers**: When the current number n is odd, the transformation $n = 3n + 1$ always produces an even number. This allows two steps of the sequence to be merged into a single operation:

```
if ((n & 1) == 0)
    n >>= 1;
else {
    n = (3 * n + 1) >> 1;
    steps++; // count the additional step
}
```

- **Bitwise Operations**: To further improve performance, modulo and division operations were replaced with faster bitwise equivalents:
  - `n % 2` was replaced by `n & 1` to test for odd/even.
  - `n / 2` was replaced by `n >> 1` for division by two.

These simple yet effective changes significantly reduced the number of instructions executed. **Naturally, these optimizations were also incorporated into all parallel versions to ensure consistent and fair performance comparisons.**

2.1 Static Scheduling Approach

In the static approach, the user specifies the number of threads and a *block size* (chunk size). The range $[start, end]$ for each input is subdivided into blocks. These blocks are then pre-assigned to the available threads in a round-robin fashion. Each thread computes the number of steps for the Collatz sequence in its assigned blocks. A high-level overview of the implementation is as follows:

1. Parse command-line arguments to retrieve the number of threads, block size, and input ranges.
2. Partition each input range into blocks of size `block_size`.
3. Distribute these blocks to the threads (static assignment). In our implementation, we create unbounded vectors (called `embedded_ranges`) that store all the subranges each thread must handle.
4. Each thread executes the Collatz procedure on every integer in its assigned blocks and tracks its local maximum step count.
5. To store and retrieve these maximum values, we use `std::promise` and `std::future` (as demonstrated in class). Specifically:
   - We create a promise for each thread and pass it (by reference) to a lambda function that runs in the thread.
   - The thread computes the maximum Collatz steps for all its assigned blocks, then sets the result into the promise.
   - In the main thread, we collect each future and combine the partial maxima into a global maximum (for each range).
6. Finally, we output the final maximum steps for every range.

This method is fairly straightforward, but it can lead to load imbalance if certain subranges contain significantly more computationally expensive numbers than others. The round-robin distribution mitigates this issue somewhat, but does not guarantee an even split of workload in all cases.

2.2 Dynamic Scheduling Approach

In the dynamic approach, we rely on a thread pool to continuously fetch tasks from a shared queue. Unlike static scheduling, where blocks are preassigned, here we break each range into smaller chunks (*tasks*), and each thread pulls these tasks one at a time until all are processed. A concise outline is as follows:

1. Parse the command-line arguments to obtain the number of threads (`num_threads`), the task size (`task_size`), and the input ranges.

2. Initialize a thread pool with `num_threads` worker threads. Internally, this pool manages a work queue in which tasks can be enqueued.
3. For each range $[start, end]$, subdivide it into chunks of size `task_size` and enqueue them into the thread pool's task queue.
4. Define a lambda function, which we refer to as `collatz_task`, that takes a single `collatz_ranges` structure (containing a subrange) as a parameter. This function computes the Collatz steps for every number in the subrange and determines the local maximum number of steps. Similar to the static approach, we use `std::promise` to store this local maximum so that, once the computation is done, the result can be retrieved via a corresponding `std::future`.
5. Each worker thread in the pool repeatedly executes the following:
   - Fetch a task (i.e., a subrange) from the shared queue.
   - Invoke `collatz_task` to compute the maximum step count for the subrange and store the result in a `promise`.
   - Retrieve the next task, if available.
6. After all tasks have been enqueued, the main thread collects the futures, which provide the local maxima computed by each subrange. These partial results are combined to obtain the final maximum steps for every input range.

In this way, the dynamic scheduling approach automates load distribution. The thread pool manages the queue of tasks, pulling them out on demand. This minimizes idle time, especially when some subranges happen to require significantly more iterations in the Collatz sequence.

3 Work-Span Model

Let us define:

- N: total number of values for which we compute the Collatz sequence.
- C: constant cost to process each single value.
- D: cost associated with dividing the workload among p threads (e.g., partitioning the ranges or creating tasks).
- A: cost for merging or combining all partial results at the end.

In the *sequential* version, there is no need to divide or merge anything, so its total running time is

$$T_1 \;=\; N \cdot C.$$

When running in parallel on p threads, we include the cost D for distributing work and A for the final aggregation. Each thread ideally processes $\frac{N}{p}$ values, contributing $\frac{N \cdot C}{p}$ to the total. Hence the time on p threads becomes

$$T_p \;=\; D \;+\; \frac{N \cdot C}{p} \;+\; A.$$

Hence, we can describe the speed-up with respect to the sequential run as

$$\text{Speed-up}(p) \;=\; \frac{T_1}{T_p} \;=\; \frac{N \cdot C}{D + \frac{N \cdot C}{p} + A}.$$

and if $N \cdot C \gg D + A$, which typically occurs when N is very large, then the division and merging costs D and A can be treated as negligible in comparison. Under that idealized view, the dominant term becomes $\frac{N \cdot C}{p}$, and hence

$$\text{Speed-up}(p) \;=\; \frac{N \cdot C}{\frac{N \cdot C}{p}} \;=\; p.$$

This p-fold speed-up is the theoretical upper bound (assuming C can be perfectly parallelized across all items), and is seldom fully realized in practice. In real systems, additional factors such as thread scheduling, synchronization, or the irreducible per-item cost still prevent the speed-up from matching p exactly.

4 Performance Analysis

We ran tests on several input ranges (**all in the internal node of the cluster machine**), focusing on how both versions scale by varying the number of threads (`1, 4, 16, 32`) and block/task size (`64, 128, 256, 512, 8192, 131072`). Specifically, we tested the Collatz computation on the following ranges:

- `1--1000`
- `50000000--100000000`
- `1000000000--1100000000`

Below is an illustrative subset of the results:

Execution Time (Static Scheduling):

Threads	64	128	256	512	8192	131072
1	54.9157	54.7525	55.1776	54.8068	54.8336	54.8517
4	15.7560	15.3423	14.6326	14.9928	14.7205	14.8324
16	4.1543	3.9874	3.8835	3.8190	3.7360	3.7546
32	2.8736	2.7659	2.6787	2.6362	2.5985	2.6317

Table 1: Execution time in seconds for static scheduling on selected ranges.

Execution Time (Dynamic Scheduling):

Threads	64	128	256	512	8192	131072
1	53.7690	52.0241	51.0346	50.5614	48.7576	48.0720
4	14.9927	14.0733	13.5162	13.1604	12.9315	12.7812
16	6.0595	3.8595	3.5518	3.5171	3.2900	3.3901
32	9.6955	4.8631	2.4424	2.3284	2.2709	2.2890

Table 2: Execution time in seconds for dynamic scheduling on selected ranges.

From these tables (1,2), we can see that both methods exhibit a notable improvement in performance as the number of threads increases, reflecting the inherent parallel nature of the Collatz computations. Dynamic scheduling tends to excel particularly when the task size is larger, whereas smaller task sizes can introduce added overhead from managing a higher volume of mini-tasks. By contrast, static scheduling maintains relatively uniform performance once the number of threads is fixed, even as the chunk size varies. On average, we observe that the dynamic approach is more efficient and often achieves faster execution times than the static alternative. Indeed, in the most favorable configurations, dynamic scheduling improves the total runtime by approximately half a second compared to the best performance achieved through static scheduling. However, we also note that increasing the chunk or task size too much—specifically to `131072`—results in slightly degraded performance, likely due to reduced parallel granularity and higher idle time. As a result, the most effective configuration in practice appears to be with a chunk size of `8192`, which consistently offers the best balance between overhead and load distribution.

4.1 Strong Scaling

To measure strong scaling, we keep the total problem size fixed and increase the number of threads, thus determining how effectively the code scales as more parallel resources are added. In practice, overheads such as task management, synchronization, and memory contention limit the real speedup to below the ideal linear speedup.

In our experiments, we present strong-scaling curves for both static and dynamic scheduling using selected parameter configurations (e.g., chunk/task size of 8192) on the ranges introduced before. The data in Figure 1 demonstrates that both approaches show an appreciable speedup as the number of threads goes from 1 to 32. Specifically, dynamic scheduling appears to follow the ideal curve more closely when the task size is sufficiently large, whereas static scheduling becomes somewhat less efficient as

more threads are introduced, likely due to load imbalance across blocks. Nonetheless, each configuration exhibits substantial performance gains compared to the plain version baseline, affirming that even partial parallelization of Collatz computations can be highly beneficial in practice.

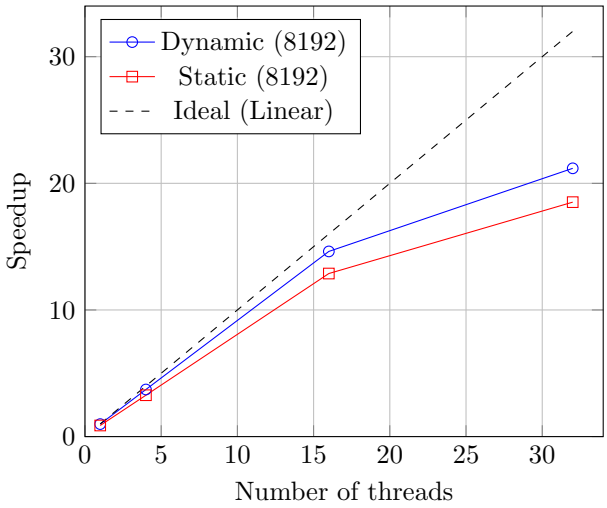

Figure 1: Strong scaling with task size/block size = 8192 for both dynamic and static scheduling, comparing measured speedups to the ideal linear trend. The tests were conducted on the input ranges $1 \times 10^0 - 1 \times 10^3$, $5 \times 10^7 - 1 \times 10^8$, and $1 \times 10^9 - 1.1 \times 10^9$.

5 Challenges and Solutions

5.1 Load Balancing

One of the main challenges in parallelizing Collatz is ensuring that each thread receives a fair share of the work. With **static scheduling**, certain subranges might have more numbers that take longer to converge to 1, causing load imbalance. We partly mitigate this by using smaller block sizes, ensuring a more uniform distribution, but at the cost of potentially increased overhead.

5.2 Synchronization Overhead

In **dynamic scheduling**, the thread pool improves load balance but it also introduces contention as multiple threads synchronize to fetch tasks. The use of the thread pool library reduces the overhead, but there may be other, more efficient solutions.

5.3 Memory and Data Sharing

When dealing with large ranges, memory usage becomes significant. We avoided unnecessary data structures and ensured that results are combined only at the end using the `futures`, minimizing data movement. Other options could be explored.

6 Conclusion

In this report, we examined two parallel implementations of the Collatz conjecture, employing static and dynamic scheduling. Our performance results indicate that while static scheduling is simpler, it may be prone to load imbalance for certain inputs. Dynamic scheduling generally yields better load distribution at the cost of extra overhead in managing tasks.

Overall, for large and potentially irregular ranges, the dynamic approach can provide more consistent results. For uniform or predictable workloads, static scheduling can be nearly as effective with proper block size selection. Future work could explore hybrid techniques that combine the low overhead of static approaches with advanced scheduling strategies informed by runtime predictions of Collatz sequence lengths, or others related to the dynamic part.