# SMP Assignment 2

Workload balance

DAVIDE BORGHINI

April 2025

## Contents

# 1 Introduction

The objective of this project is to analyse how different scheduling strategies influence the performance of parallel programs. Specifically, we investigate how various methods of workload distribution impact execution time when utilizing different numbers of threads. The study focuses on a comparative analysis of static and dynamic scheduling, evaluating their respective efficiencies in a particular computational task.

The task under consideration involves computing, for multiple given ranges, the maximum number of steps required for the Collatz function to converge. The Collatz function is defined as follows:

$$n = \begin{cases} \frac{n}{2}, & \text{if } n \text{ is even} \\ 3n+1, & \text{if } n \text{ is odd} \end{cases}$$

The function is said to have converged when $n = 1$. The primary goal is to determine how different scheduling approaches affect the efficiency of this computation in a multithreaded environment.

# 2 Theoretical Analysis

Before analysing the various implementations, it is useful to first examine the theoretical dependencies of the problem. To this end, we employ the *Work-Span Model*.

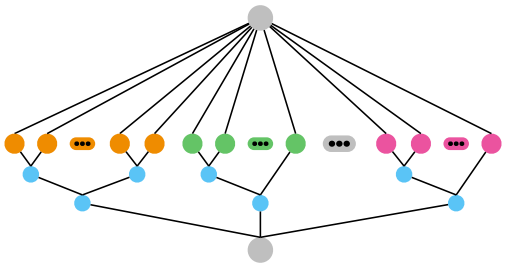

Figure 1: Work-Span Model for computing the maximum number of Collatz steps. After the initialization (gray) of the problem each range is marked with a different colour (orange, green, purple) while the max operation is denoted in blue

As illustrated in Figure 1, the computation of the number of steps in the Collatz function is independent for each number, regardless of the range. Within each range, the maximum number of steps can be determined using a `reduce` operation. The best achievable parallelism in this context involves distributing the $n-1$ comparisons across $\log_2(n)$ levels, assuming a binary max operation.

Since we have some interesting bounds and theorems that depend on the execution times $T_1$ and $T_\infty$, it's useful to express them formally.

- $T_1$ is the total work, simply the sum of all operations:

$$T_1 = \sum_{i=1}^{n} c_i + (n-1) \cdot m$$

where $c_i$ is the time required to compute the Collatz sequence for the $i^{\text{th}}$ number, $n$ is the number of elements, and $m$ is the time for a single maximum operation.

- $T_\infty$ is the span (the longest chain of dependent operations), computed as:

$$T_\infty = \max_i c_i + \log_2(n) \cdot m$$

From these, using Brent's Theorem, we can derive a lower bound on the speed-up for a given number of processors $p$:

$$S(p) \geq \frac{p}{1 + p \cdot \frac{\max_i c_i + \log_2(n) \cdot m}{\sum_{i=1}^{n} c_i + (n-1) \cdot m}}$$

We can reasonably assume that the time for computing a maximum is much smaller than computing a Collatz sequence. With this assumption, we can simplify the expression:

$$S(p) \geq \frac{p}{1 + p \cdot \frac{\max_i c_i + \log_2(n) \cdot m}{\sum_{i=1}^{n} c_i}}$$

We can also derive an upper bound to the speed-up, from the fact that

$$S(p) \leq \min(p, \frac{T_1}{T_\infty})$$

We can restrict the speed-up in the range

$$[\frac{p}{1 + p \cdot \frac{\max_i c_i + \log_2(n) \cdot m}{\sum_{i=1}^{n} c_i}}, \min(p, \frac{\sum_{i=1}^{n} c_i}{\max_i c_i + \log_2(n) \cdot m})]$$

For even further simplification, if we consider $T_\infty \ll T_1$, the range converge to a point in which we have $S(p) \approx p$. While this bound is theoretical and context-dependent, it captures a fundamental property of the Work-Span model: near-linear speedup is achievable under ideal conditions, but it also represents an upper limit that cannot be exceeded.

# 3 Implementations

*Now that we know that we should be able to reach almost linear speed-up the pressure on us is quite high...*

## 3.1 Sequential

The sequential implementation is the trivial one in which each number is computed inside a for loop updating the maximum each step. I tried to apply loop unrolling but without obtaining any performance improvement.

## 3.2 Static scheduling

In the static version, tasks are assigned to threads in a block-cyclic fashion for each range. This means that, for example, thread 0 will always receive the first block of a given range. This (slightly naïve) approach was chosen for its implementation simplicity. However, it can lead to performance issues when the ratio between chunk size and range size becomes large.

An extreme case 4 is when the block size is greater than the range itself; in this scenario, only one thread ends up computing all the Collatz sequences.

There are two possible alternatives to address this issue: if we know in advance that we'll be dealing with many small ranges, it may be reasonable to assign full ranges in a block-cyclic manner. A more generally effective approach, however, is to assign fixed-size chunks (equal to the chunk size) without restarting from the first thread each time a new chunk is assigned. This strategy tends to distribute work more evenly across threads in most situations.

### 3.3 Dynamic scheduling

The first implementation of dynamic scheduling used a single mutex to control a shared index, which was updated each time a thread picked a chunk to compute. In the same critical section, if the worker detected a change of range, it would also check whether it needed to update the maximum for that range.

Since this critical section became a bottleneck, I introduced a dedicated mutex for accessing the maximums. These are maintained separately for each thread and updated only at the end of the computation. Even with this optimization, I wasn't satisfied with the performance—preliminary results showed significantly slower execution times compared to the static version.

An alternative could have been to introduce multiple locks, one for each range. However, this approach might cause issues when dealing with a high number of ranges. For this reason—and out of curiosity—I implemented another parallel version that is completely lock-free.

### 3.4 Dynamic scheduling lock-free

In this final implementation, a vector of shared indices was used to keep track of the chunks being processed. Both the indices and the maximum values were implemented as atomic variables. With this approach, threads no longer need to compete for shared resources, which eliminates contention. As a result, this version performs better than the traditional lock-based implementation.

## 4 Experiments and Results

Since there are many variables to analyse in this problem (such as the number of threads, chunk size, etc.), I defined three setups for the experiments. The first setup requires computing many small ranges, the second involves a few large ranges, and the third is a mix of both.

### 4.1 Experimental result

From Figures 2 and 3, we can see that the dynamic scheduling implemented using the lock-free paradigm is faster than any other implementation[1] across all setups, regardless of the number of threads.

This is likely because the dynamic scheduling distribute better the work and the use of atomic almost eliminate the synchronization overhead.

[1] Strangely, when the lock-free program is run with a single thread, it still outperforms the sequential one. This could be due to how the compiler optimizes the code and how the cache is used differently between the two implementations. This was suggested by a roommate of a friend of mine, and whether it's the right answer or not, I don't think it's worth diving deeper into this issue.

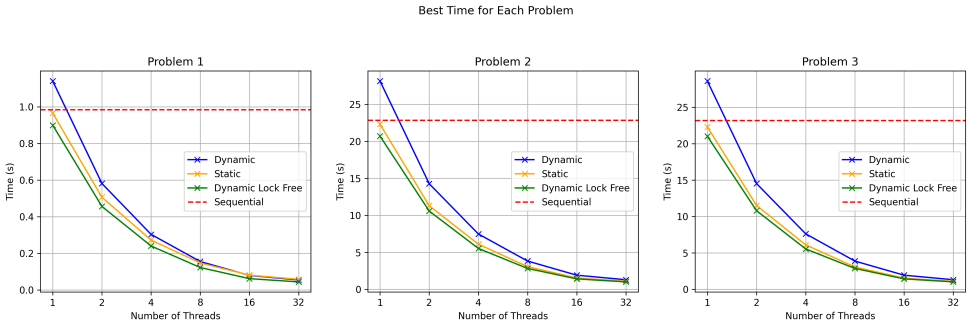


Figure 2: Graphs showing the time comparison between different implementations among three different problems, using optimal chunk size.

Another interesting aspect to analyse is the speed-up and strong scalability, as shown in Figure 3. The dynamic lock-free implementation demonstrates almost linear speed-up up to 16 threads across all three problems, and it continues to scale effectively even up to 32 threads. This suggests that the overhead introduced by atomic operations remains minimal, allowing the implementation to take full advantage of increased parallelism.

The static implementation also scales quite well, particularly in Problems 2 and 3, where it approaches the performance of the lock-free version. This might be due to better load balancing in scenarios with fewer, larger ranges. On the other hand, the classic dynamic version is always behind the other two, and the gap becomes more noticeable as the number of threads increases. This is probably because of the overhead caused by locks and contention between threads. In general, these results show that avoiding these bottlenecks—either by distributing chunks more smartly or by going fully lock-free—can really boost performance when scaling up.

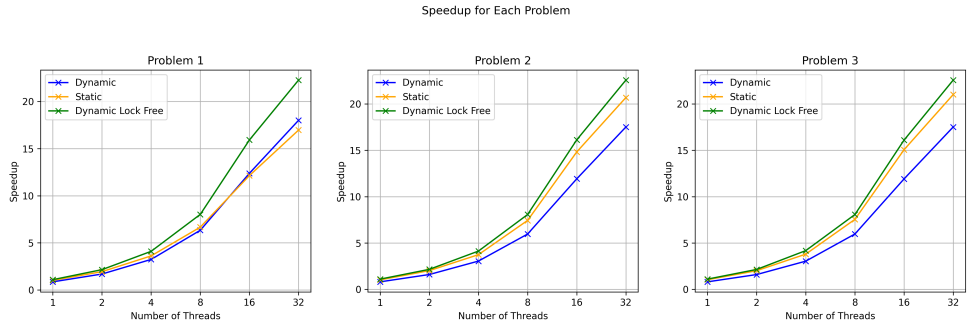


Figure 3: Graphs showing the speed-ups of the different implementations among three different problems, using the optimal chunk size.

A last analysis I want to perform is the one about chunk size (some supporting plots are included in the appendix 6).

As expected, when dealing with many small ranges, static scheduling performs better with smaller chunk sizes. As we move towards larger ranges, the impact of chunk size becomes less significant, although larger sizes seem to offer some minor improvements.

For the classic dynamic version (the one using locks), performance consistently improves with larger chunk sizes. This is because bigger chunks reduce the number of times threads need to enter a critical section to fetch new numbers to compute. The same is true for the lock-free version, although the issue here isn't entering a critical section,

but rather the overhead from the operations needed to compute the correct index each time a thread needs to fetch a new chunk. So, even in this case, using larger chunk sizes helps reduce that overhead.

### 4.2 Comparison with Work Span model

It's also interesting to compare these experimental results with the theoretical predictions from the Work-Span model. While the model gives us an idealized view of parallel performance, it's useful to check how closely the real-world behaviour aligns with those bounds.

The Work-Span model assumes $p$ identical processors and a greedy task scheduling strategy. These assumptions are respected in the dynamic scheduling version[2].

The analysis focuses on a single case, as covering multiple problem sizes would require considerable testing. In particular, we consider computing the maximum number of steps in the range 50,000,000–100,000,000 using 16 threads. We've already observed that at 32 threads the speed-up is far from linear. In this range, there are 9,322,961,605 total steps, which take about 18.57 s to compute sequentially. The maximum number of steps is 949, but its impact on runtime is negligible.

In theory, this setup should yield close to linear speed-up. However, running `srun parallel_collatz.out -d -f -n 16 -c 1 50000000-100000000` gives a runtime of 3.23 s, resulting in a speed-up of 5.75. This is far from ideal, mainly due to overhead introduced when threads fetch new chunks. The model does not account for this. If we increase the chunk size to 1000, the same computation takes only 1.21 s, leading to a speed-up of 15.35—much closer to the theoretical limit.

## 5 Conclusions and Possible Further Optimizations

After this discussion, we can say that dynamic scheduling proved to be superior to the specific implementation of static scheduling tested here. This was mainly because we were able to reduce synchronization overhead to a minimum. However, to truly claim the superiority of dynamic scheduling for this problem, a more refined static scheduling policy should be explored—possibly one that uses C++ atomics for updating the maximum and distributes work more evenly.

The theoretical model provided a useful lower bound, although an implementation that strictly follows the computation graph would not make much sense in practice, as shown in the analysis in Section 4.2. Still, with an implementation that accounts for synchronization costs (i.e., uses chunk sizes greater than 1 and computes local maxima instead of having workers perform a parallel reduction), it is possible—at least for this specific problem—to get quite close to the theoretical limit.

[2]This analysis refers only to the lock-free implementation, as it is both the most effective and the one that aligns best with the theoretical model. The time to compute the maximum is not considered here.

# 6 Appendix

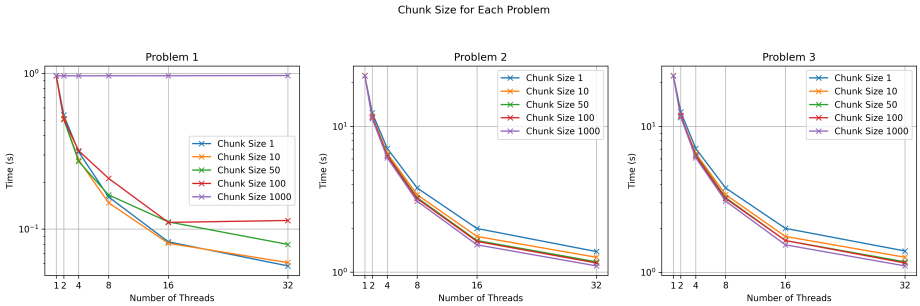


Figure 4: Graphs showing the time of static scheduling for different chunk sizes.

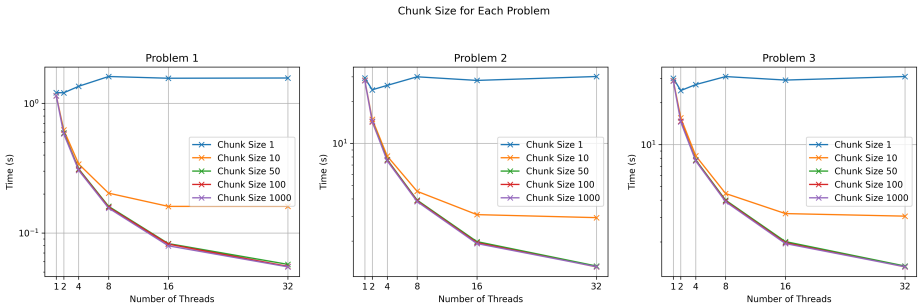


Figure 5: Graphs showing the time of dynamic scheduling for different chunk sizes.

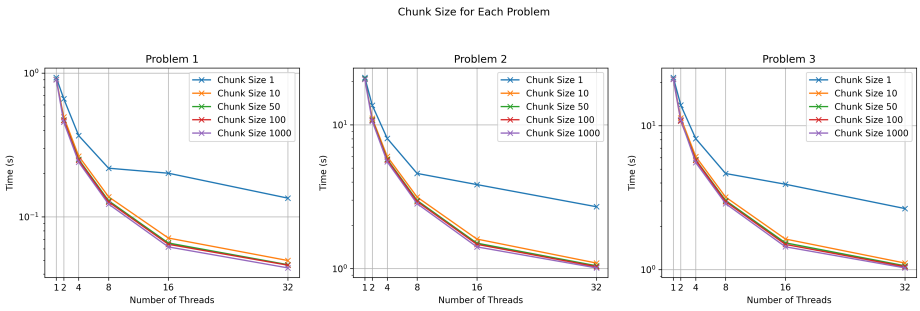


Figure 6: Graphs showing the time of dynamic scheduling (using lock free programming) for different problems and different chunk sizes.