Department of Computer Science

Computer Science Master Degree

Second Assignment - Collatz

SPM Course 2024/2025

Student: Luca Rizzo

ID Number: 598992

The goal of this assignment was to develop a parallel version of the Collatz sequence computation, with a focus on evaluating performance across different scheduling strategies for distributing the input elements. The objective was to identify which version yields the best speedup, particularly in relation to the imbalance characteristics of the domain.
A theoretical analysis using the Work-Span model was also carried out to assess whether, under ideal conditions, the linear speedup limit predicted by the model is realistically achievable.

Analysis on imbalance distribution

To understand how computational load is distributed across the domain, I conducted an analysis inspired by the Mandelbrot set, where imbalance often clusters in specific contiguous chunks (making static block-based scheduling inefficient). The aim was to determine whether a similar pattern exists in the Collatz problem.
I divided the input interval into blocks of 256 elements, computed the average Collatz sequence length per block, and calculated the standard deviation of these averages. The result, 24.645, is low compared to typical averages above 150, indicating that load is fairly uniform across blocks. We can therefore conclude that, unlike the Mandelbrot case, the Collatz workload is not localized in contiguous chunk but rather evenly distributed, with high-cost elements scattered throughout the domain.

Implementation description

I chose to implement a parallelization strategy where each input range is processed independently and sequentially: the program parallelizes the computation within range1, then moves on to range2, and so on. This avoids assigning chunks from different ranges to the same thread *within the same task*, which would complicate the identification of the maximum value per range. I consider this scenario a corner case that could negatively affect performance in more general use cases, so I decided to avoid it.
The program starts by parsing command-line arguments using the parse_running_param method, and then executes one of the three available versions depending on the execution parameter: -s for the block cyclic version, -t for the dynamic version with a thread pool, -d for the dynamic version using a shared index.

Static block cyclic

In this approach, each thread receives input chunks/tasks of size chunk_size, spaced every stride elements, with each starting from an offset determined by the thread's index. For example, if num_threads = 2, chunk_size = 10, and range = [100, 1000], then:

- T_0 = [<100,110>, <120,130>, ...]
- T_1 = [<110,120>, <140,150>, ...]

Exactly num_threads threads will be spawned, and each of them is explicitly responsible for computing the maximum value over its assigned chunks, according to the shared scheduling policy (with no external master assigning work).
As a synchronization mechanism for communicating the local maximums from the workers to the master, I used a future, as it provides a one-shot notification mechanism.
The local maximums, one per thread, will then be reduced to a single global maximum by the master.

Dynamic block with shared index

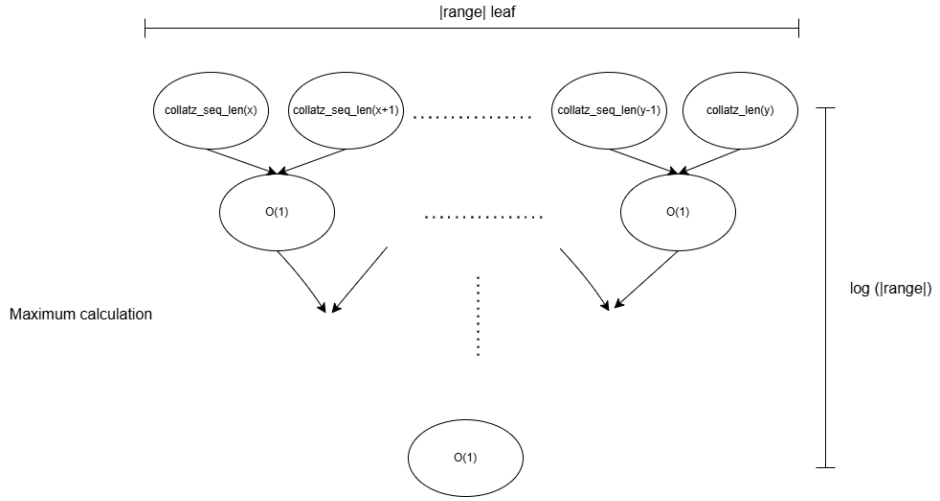
This approach employs a concurrent, shared data structure called *ChunkDispatcher*, which enables threads to atomically retrieve work chunks. The dispatcher is initialized with a given range and chunk_size, and uses a mutex to handle synchronization. An alternative implementation could leverage an atomic counter to further optimize access patterns.
Work termination is signaled by returning a range where the start is greater than the end. Each thread repeatedly retrieves chunks of size chunk_size from the dispatcher and exits when it receives such a terminating range.
Exactly num_threads are spawned, and each thread independently pulls its tasks from the dispatcher, without any involvement from a master thread (the assignment follows an internal policy managed within the shared structure).
As in the static scheduling approach, a future is used as the synchronization mechanism to report local maxima from the workers to the master, as it provides an efficient one-shot notification.
In the end, the num_threads local maximums are reduced to a single global maximum by the master.

Dynamic with threadPool

In this approach, I leveraged the TP (ThreadPool) data structure provided during the course, adopting a strategy where the master thread divides the entire range into chunks (subranges) of size chunk_size. It then progressively submits the corresponding tasks to the thread pool, which is initialized with num_threads worker threads.
In this setup, any idle thread in the pool can pick up a task, compute the maximum for the assigned subrange, and return the result via the future returned by the thread pool.
Unlike previous approaches, here it is the master thread that handles the division and (indirect) assignment of tasks by enqueueing them into the thread pool's task queue. Each thread then simply executes one chunk at a time as an individual task.
The use of future in this case is finer-grained: while in earlier strategies there were num_threads futures (one per thread), here we have one per task (that is, approximately |range| / chunk_size futures).
Finally, all the task-level maximum (|range| / chunk_size in total) are reduced by the master to compute the overall maximum.

Work span model

Each task in the model corresponds to computing the *collatz_length* for a specific value within the input ranges. From this perspective, all values across the ranges are fully independent and can be computed in parallel. So they reside at the same level in the computation graph.
Subsequently, a global maximum must be computed from these individual values. This step proceeds through pairwise reductions of progressive maximum, forming a balanced binary tree with logarithmic depth in terms of range dimension, where each level reduces the number of values by half until the final global maximum is obtained.
For simplicity, we can assume that the computational cost of each node corresponds to the length of the Collatz sequence associated with its value.
Assuming a single range [x, y], the resulting computation graph is as follows:



In this computational model, the work corresponds to the sum of the costs of computing the collatz_length for each value in the range, followed by the sequential effort required to compute the intermediate maxima during the reduction phase:

$$T_1 = \sum_{i \in range} (T(collatzLen(i))) + O(|range|)$$

Regarding the span, it is determined by the critical path of the computation. This path consists of the value within the range that requires the highest number of iterations to compute its Collatz sequence, plus the cost of performing the parallel reductions level by level — totaling log(|range|) levels in a balanced binary tree

$$T_\infty = \max_{i \in range} (T(collatzLen(i))) + O(\log |range|)$$

Let us focus on the range [1.000.000.000, 1.100.000.000], with |range| = 100.000.000. After computing the actual values, we obtain:

$$\sum_{i \in range} (T(collatzLen(i))) = 21.392.237.009 \qquad \max_{i \in range} (T(collatzLen(i))) = 984$$

so T_1= 21.492.237.009 and $T_\infty = 1011$. The parallel slack is $\frac{T_1}{T_\infty} = 21.258.395$

The parallel slack is significantly larger than any reasonable value of p, which means the corollary of Brent's theorem applies so the ***theoretical*** limit on speedup is linear in p.

$$S(p) = \frac{T_1}{T_p} \approx p \qquad if \ \frac{T_1}{T_\infty} >> p$$

Of course, we must acknowledge that this is purely a theoretical upper bound. The model does not account for overhead introduced by synchronization and task scheduling. In fact, scheduling a single number per task (and subsequent pair of maximum) leads to a considerable overhead, which is entirely ignored by this model

Performance analysis

A PHONY target named `launch_benchmark` is defined, depending on cleanall and all source files to ensure a full recompilation, especially when enabling the -march=native optimization flag. Once compiled, it runs the run_benchmark script, which executes the three parallel versions twice for each

combination of number of threads and chunk size, along with the sequential version. The results are saved in a CSV file for further analysis. To assess the system's strong scalability, the workload was fixed using a set of predefined input ranges (1–1000, 5000–100000, and 1000000–5000000). The script was launched with the following command on a cluster machine:

```
srun --nodelist=node06 make launch_benchmark
```

For each scheduling policy, the benchmark recorded performance metrics, and an analysis was performed by averaging the two runs and computing the corresponding speedup.

Policy: block cyclic

Chunk_size	8	32	512	1024	5000	10000
Num_threads						
4	3.42				3.75	
8		6.61	7.31		7.41	
16			14.48	14.65	14.77	14.76
32	17	18.18	20.29	20.39	20.53	20.51
40					17.12	17.34

Policy: dynamic with shared index

Chunk_size	8	32	512	1024	5000	10000
Num_threads						
4	3.14				3.49	
8		6.45	6.76		6.83	
16			13.5	13.57	13.64	13.65
32	7.05	19.55	20.91	20.94	20.91	20.87
40					20.88	20.84

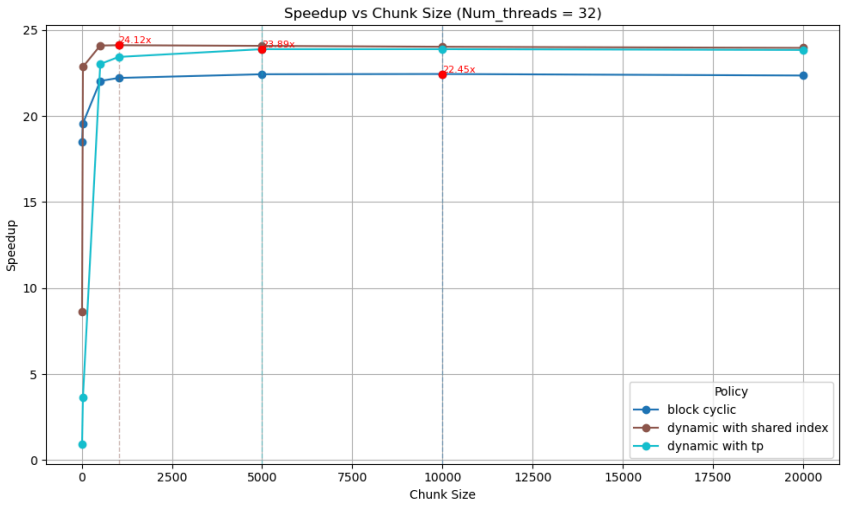
Policy: dynamic with tp

Chunk_size	8	32	512	1024	5000	10000
Num_threads						
4	1.26				3.37	
8		4.5	6.5		6.68	
16			12.83	12.73	13.11	13
32	0.8	3.15	19.84	20.12	20.51	20.51
40					20.55	20.52

All policies reached their best performance at num_threads = 32, except for the thread pool approach, which continued to scale up to 40 threads. This is consistent with the hardware setup, which features 32 cores in total, including both physical and logical ones: adding more threads beyond that point results in overprovisioning, where thread management overhead starts to outweigh any gains from additional parallelism. To simplify the analysis and avoid varying both parameters simultaneously, I chose to separate the experiments by fixing one parameter and varying the other. I think that this approach provides a clearer understanding of how each individual parameter influences the speedup.

Finding optimal chunk size

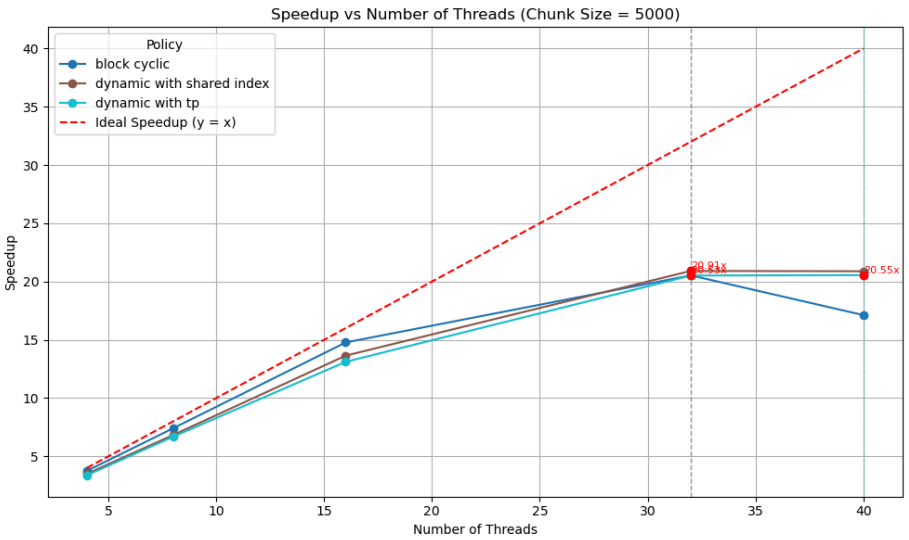
With the number of threads fixed at 32, we analyzed the impact of different chunk sizes on performance.



The block cyclic strategy consistently delivered stable results across all chunk sizes, indicating that its load distribution remains balanced regardless of granularity. This robustness comes from the uniform nature of computation of the problem domain itself: since computational load is fairly consistent across the input space, evenly dividing work among threads leads to implicit load balancing without requiring complex scheduling. On the other hand, both dynamic approaches experienced notable slowdowns when chunk sizes were too small, as execution time became dominated by scheduling and synchronization overhead, which the static policy avoids. However, all scheduling strategies showed a drop in performance beyond a certain chunk size threshold. This likely occurs because overly large chunks can lead to little load imbalance, where some threads finish earlier while others are still processing more heavy tasks, reducing overall parallel efficiency.

Scalability analysis

Finally, by fixing chunk_size = 5000, we examined how speedup changes with increasing thread counts.



All the approaches exhibited *“close to linear”* (in terms of PE) speedup up to 16 threads . This suggests that, in this range, the execution comes reasonably close to the theoretical limit predicted by Brent’s theorem, with speedup scaling efficiently with the number of threads, despite minor deviations from ideal linearity.

Beyond that point, the speedup continued to increase, but no longer as close to linear. This change in behavior may be attributed to the transition from physical to logical cores, which generally offer reduced performance due to shared resources. A peak was observed around 32 threads, after which performance began to degrade, likely as a result of increased overhead from thread management and synchronization outweighing the benefits of additional parallelism.

Similarly, the thread pool approach followed the same trend but peaked slightly later, at around 40 threads, just before the overhead became dominant.