Assignment 2 report
SPM course
2024/25

Francesco Scarfato

April 8, 2025

1 Objective

The objective of this project is to develop a program that computes the maximum Collatz sequence length for any number within a given input range. The program includes two parallel implementations: one utilizing static scheduling and the other employing dynamic scheduling.

2 Implementation

2.1 Static scheduling

The static scheduling implementation divides the workload among threads using a block-cyclic distribution strategy. Each thread is assigned a fixed starting point within the input range and processes blocks of numbers of size `task_size`. After completing its assigned block, a thread skips ahead by a constant padding value to skip the chunks covered by other threads. This padding is computed as the product of the number of threads and the block size.
Each thread computes the maximum number of steps in its assigned numbers independently. Once the local maximum is determined, the thread synchronizes with the other threads using a mutex `max_mutex` to update the global maximum `collatz_max_steps` for the range.
This approach provides balanced load distribution when the cost of computation per number is uniform. However, since the block assignment is static, this method may suffer in cases where the computational cost of processing numbers varies significantly inside the same range.

2.2 Dynamic scheduling

The dynamic scheduling implementation assigns the computation tasks to threads at runtime using a thread pool. Unlike the static approach, where the workload is distributed in advance, this method divides each input range into smaller chunks of size `task_size`, which are dynamically assigned to the available threads.
Each thread processes its assigned block independently, computing the local maximum number of steps for the Collatz sequence within that block. The global maximum for the range can be computed either using mutex synchronization or sequentially comparing all the results from the threads.
Two versions of the dynamic scheduling are implemented:

- **Dynamic with mutex:** Threads update the global maximum for a range concurrently using a mutex to ensure consistency.

- **Dynamic without mutex:** Each thread returns its local maximum, and the final reduction is performed sequentially after all tasks complete.

This approach improves load balancing since tasks are assigned dynamically, allowing faster threads to take the next task when they finish their current task. This is particularly efficient when the computational cost is not uniform across the range, avoiding the imbalance issues of static scheduling.
However, it generally introduces higher synchronization overhead compared to static scheduling but offers better workload distribution, especially for large ranges or irregular workloads.
The following analysis is done on the static scheduling and on the dynamic scheduling without mutex.

3 Theoretical analysis using the work–span model

Assuming that each number takes $O(1)$ to be processed and each range has the same size, the total work is $T_1 = O(r \times n)$ where r is the number of the ranges and n is the total numbers in one range to process. T_1 represents the sequential execution time and is the so-called *work*.
The span T_∞ represents the longest path through the parallel computation. In the span analysis, it is assumed unlimited parallelism, so each block could potentially run in parallel. The chunk size C is known and is given to the program, so the number of chunks is $\frac{rn}{C}$. Each thread processes $\frac{rn}{Cp}$, with p number of processors.
Analyzing the implementations:

- **For the block-cyclic distribution**: One thread computes its block in $O(C)$. Finally, the mutex creates a bottleneck of $O(p)$ to compute the global maximum of a single range. The span is $T_\infty = O(r \times (C + p))$ with r number of ranges.

- **For the dynamic scheduling without mutex**: Same as static scheduling, the cost of computation of one thread is $O(C)$. In this case, the reduction is done at the end, iterating on all the ranges, so the cost is $O(r \times \frac{n}{C})$ with n size of the single range. The span is $T_\infty = O(C + \frac{rn}{C})$

From the analysis above, we can derive the upper bounds.

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

For the block-cyclic distribution:

$$S(p) \leq \frac{rn}{r \times (C + p)} = \frac{n}{C + p}$$

For the dynamic scheduling without mutex:

$$S(p) \leq \frac{rn}{C + \frac{rn}{C}}$$

From the theoretical bounds derived above, we can observe that the speedup heavily depends on the chunk size C and the number of threads p.
In the block-cyclic case, the mutex introduces an overhead of (p), which limits the speedup for smaller problem sizes or a large number of threads. Adding more processors causes a significant synchronization cost.
On the other hand, the dynamic scheduling without mutex avoids this bottleneck. Its upper bound shows that increasing the chunk size C helps reduce the reduction cost. The reduction cost grows heavily if C is too small.

4 Performance analysis

4.1 Strong scaling for small range (10K-20K)

For small range (10K–20K), using a chunk size of 1000 is the best choice, as shown in Figure 3 in the appendix.

Table 1: Execution times for small blocks with chunk size 1000

Thread Count	Sequential (s)	Static (s)	Dynamic (s)
1	0.00230269	0.00230269	0.00230269
8	–	0.000735625	0.000838429
16	–	0.000821358	0.00105963
32	–	0.00271036	0.00173517

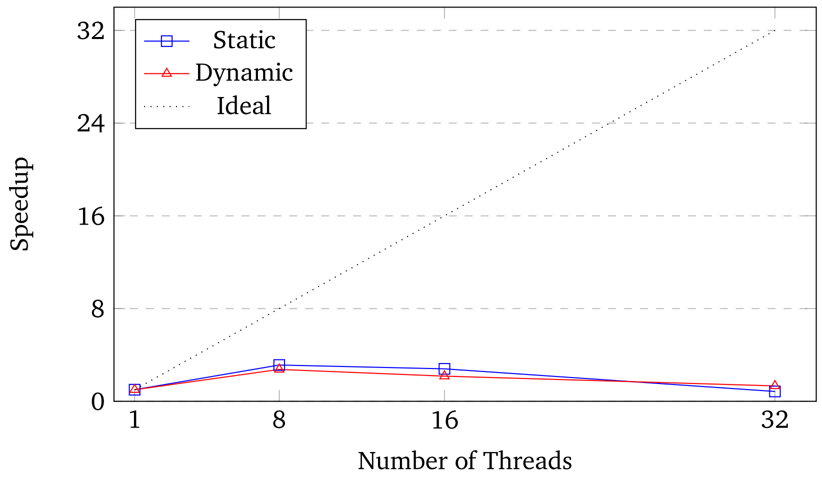


Figure 1: Strong scaling speedup for small ranges with chunk size 1000

For small workloads with a chunk size of 1000, the results show a performance peak at 8 threads, followed by a degradation as thread count increases. This occurs because the overhead associated with thread creation, synchronization, and management becomes significant relative to the actual computation time.
Static scheduling shows more degradation at 32 threads (speedup of only 0.85x) compared to dynamic scheduling (1.33x), because the work distribution in static scheduling creates load imbalance that significantly affects the running time when computational chunks become small.

4.2 Strong scaling for large ranges (50M-100M)

For large range, it is better to pick 10k chunk size, as shown in the Figure 4 in the appendix.

Table 2: Execution times for large blocks with 10K chunk size

Thread Count	Sequential (s)	Static (s)	Dynamic (s)
1	14.55664	14.69342	11.84340
8	–	1.98439	1.57764
16	–	0.99652	0.87067
32	–	0.68767	0.52211

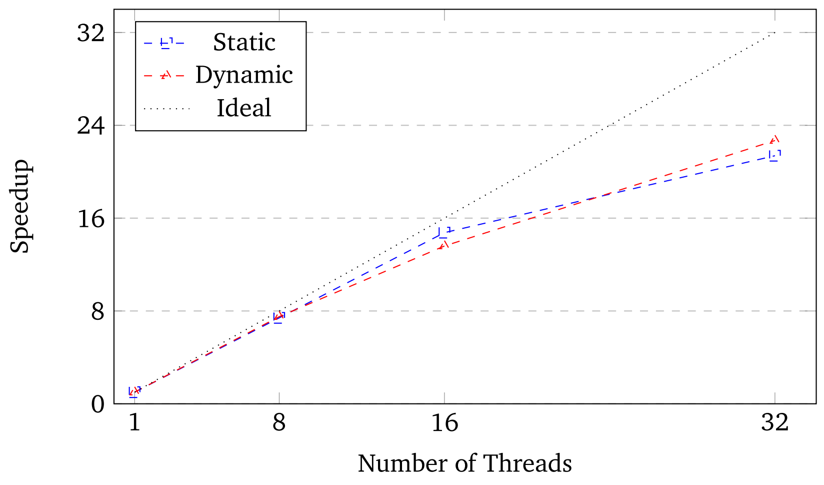


Figure 2: Strong scaling speedup for large ranges with chunk size 10K

The data for large ranges (50M-100M) with 10K chunk size shows better parallel efficiency compared to smaller workloads. With increasing thread counts, we observe that the speedup increases. This scaling occurs because the computational work overlaps the parallelization overhead, allowing effective distribution of work across threads. Dynamic scheduling has slightly better performance than static at high thread counts (22.69x versus 21.38x speedup at 32 threads).
However, while the speedup reached by 8 and 16 threads is near to the ideal speedup, with 32 threads the speedup increases but also the distance from the ideal speedup. This because the overhead due to thread spawning and synchronization became significant and affects the running time.

5 Conclusions

Both the scheduling are similar in performance. The static scheduling is efficient when the workload is uniform and large enough to overlap the synchronization cost. Dynamic scheduling provides better load balancing, especially in cases where the workload is not uniform, or the range size is small. Overall, the choice of the chunk size and the number of threads strongly impacts the performance of both scheduling.

A Chunk size analysis

A.1 Chunk size on small blocks (10K-20K)

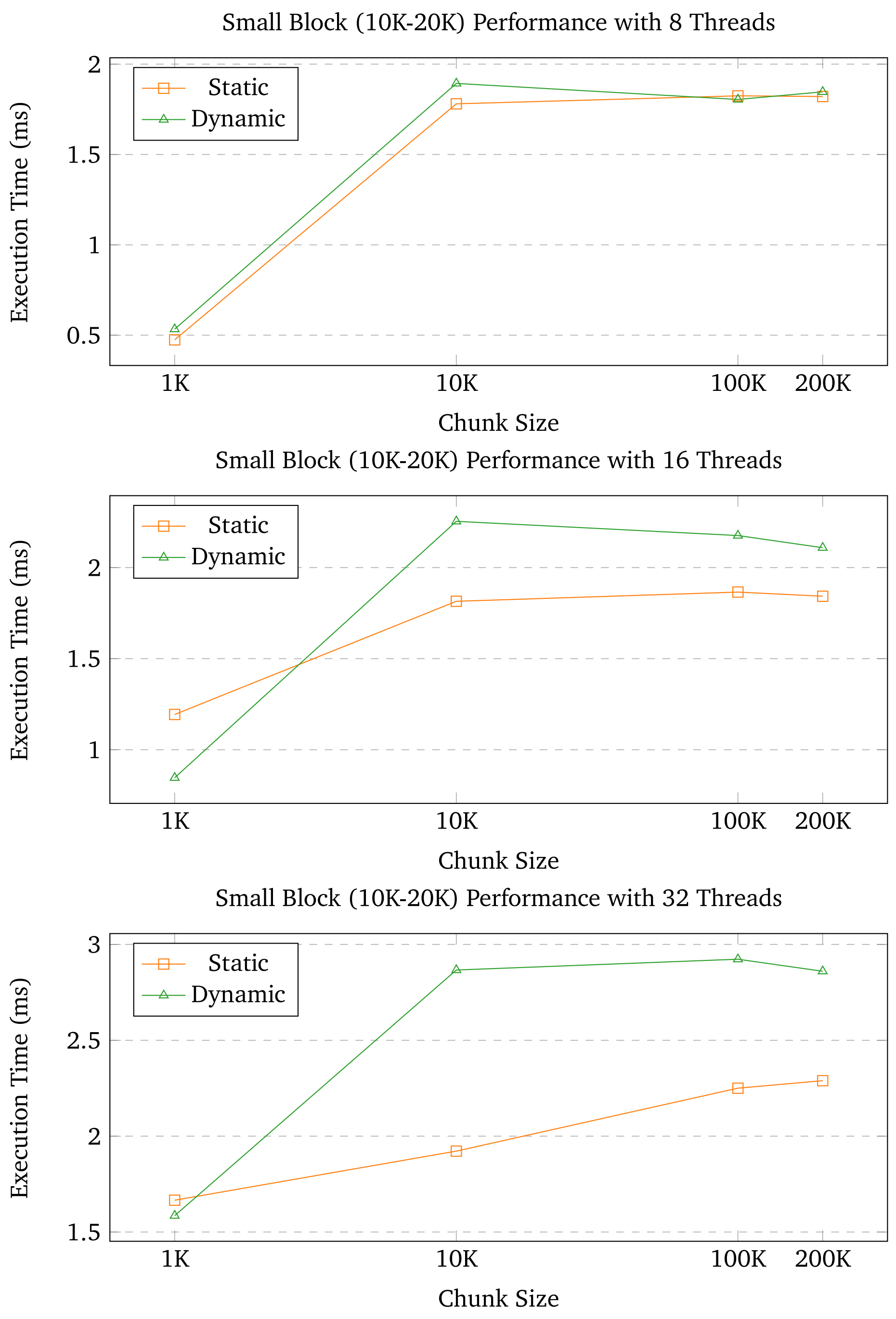
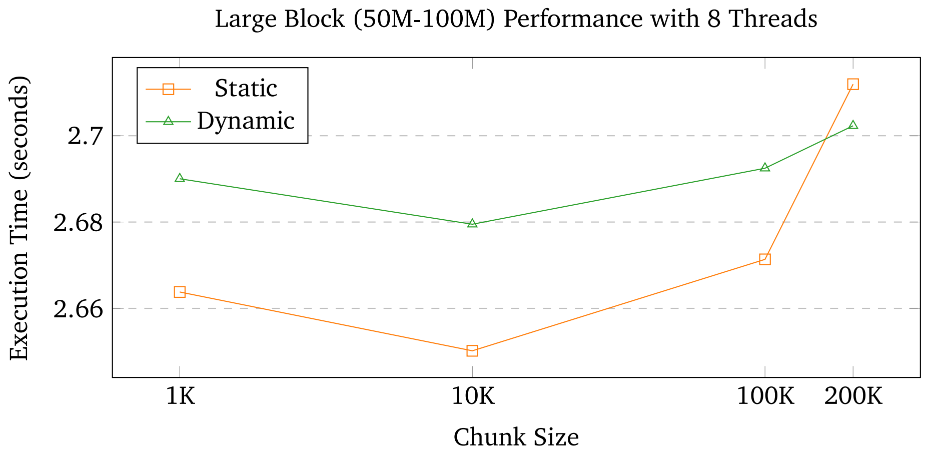
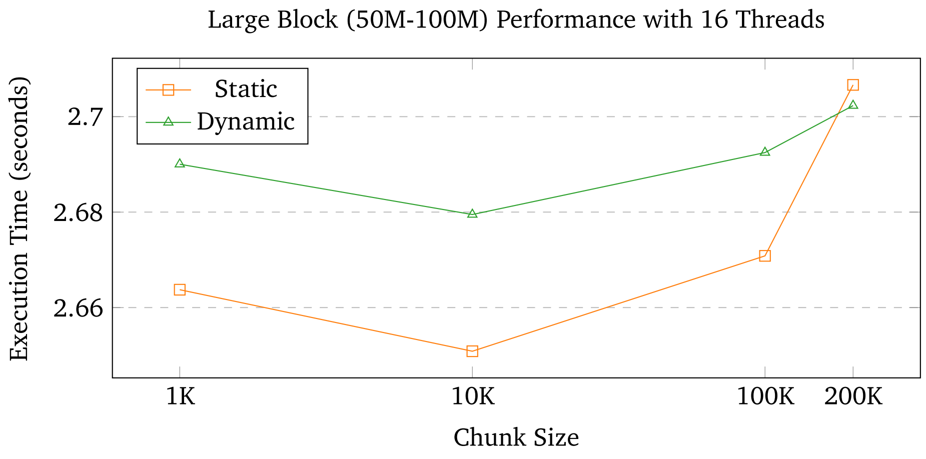


Figure 3: Impact of chunk size on small block performance across thread counts

A.2 Chunk size on large blocks (50M-100M)





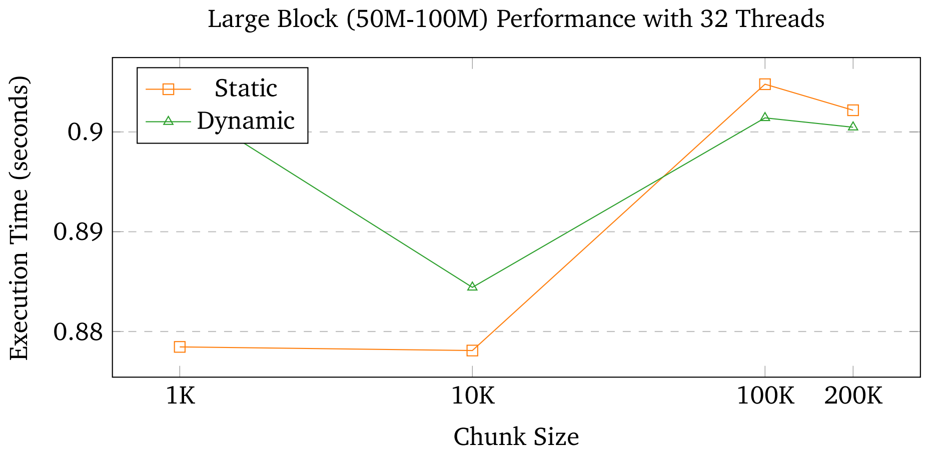


Figure 4: Impact of chunk size on large block performance across thread counts