Assignment 2: Parallel Collatz Conjecture

Luca Lombardo
SPM Course a.a. 24/25

1 Sequential Baseline and Core Function

The sequential implementation serves as our reference baseline. The `collatz_steps(ull n)` function ensures correctness using `unsigned long long` types and overflow detection. For efficiency we use bitwise operations to handle parity checks and division. A key optimization targets power-of-two inputs ($n = 2^k$): rather than iteratively dividing by two k times, the function directly computes the result k using the `__builtin_ctzll` intrinsic (count trailing zeros), significantly accelerating these cases. This implementation processes all input ranges through `find_max_steps_in_subrange`.

2 Parallel Implementation Framework

I built parallel versions using `std::thread` by partitioning input ranges into smaller tasks (chunks) that worker threads execute concurrently. For thread-safe aggregation of maximum step counts for each original range, I implemented a `RangeResult` structure containing `std::atomic<ull>` variables. Worker threads update these atomically via compare-and-swap (CAS) loops upon task completion. I chose this fine-grained approach to minimize synchronization bottlenecks during result updates. My implementations differ primarily in their task distribution (scheduling) strategy.

2.1 Static Scheduling Strategies

Static scheduling assigns tasks to threads deterministically before execution starts, with distribution patterns established upfront and fixed for the entire execution. I implemented three variants to compare their characteristics:

- **Block:** Assigns large contiguous blocks per range to each thread. For example, with numbers 1-100 and 4 threads, thread 0 processes [1-25], thread 1 processes [26-50], and so on. It features low scheduling overhead but suffers from poor load balancing, especially for non-uniform workloads
- **Cyclic:** Assigns individual numbers round-robin. With the same example, thread 0 processes [1,5,9,...], thread 1 processes [2,6,10,...], etc. This offers fine-grained load balancing but can lead to suboptimal cache performance due to scattered memory accesses, as threads frequently access non-adjacent memory locations.
- **Block-Cyclic:** Divides work into blocks of a specified `chunk_size`, which are then assigned cyclically using a global block index computed across all input ranges. With `chunk_size=4`, thread 0 processes [1-4,17-20,...], thread 1 processes [5-8,21-24,...], etc. This strategy aims to balance load better than pure Block while potentially improving cache locality compared to pure Cyclic.

Static schedulers are typically suitable for predictable, balanced workloads due to their minimal runtime overhead. The key advantage is that no synchronization is required during execution since work distribution is determined entirely in advance. For the Collatz problem specifically, the varying computation cost of different numbers (e.g., powers of 2 vs. odd numbers) makes the choice of static scheduling strategy particularly important.

2.2 Dynamic Scheduling Strategies

For workloads with variable task execution times like Collatz computation, I found dynamic scheduling strategies typically outperform static assignments. Since they let idle threads grab new tasks as they become available, they can significantly improve load balancing. I explored three dynamic scheduling approaches:

1. **Centralized Task Queue:** I implemented this straightforward approach using a single shared queue (my `TaskQueue` class) protected by a `std::mutex` and a `std::condition_variable` to efficiently manage thread signaling. This approach was simple to implement and provided natural load balancing since any free thread could take the next

task whenever it became available. However, there is a potential limitation: the single mutex protecting the queue can become a bottleneck when many threads are competing for access, limiting scalability as thread count increased.

2. **Work-Stealing with Mutex-Based Deques:** To address the scalability issues in my centralized approach, I decentralized task management by giving each thread its own deque (my `WorkStealingQueue` class). Threads primarily work with their local queue (using LIFO operations for better cache locality) but can "steal" tasks (using FIFO operations) from other threads when they become idle. This significantly reduced contention compared to my centralized queue implementation and scaled much better with higher thread counts. The LIFO access pattern for local work also improved cache locality for better performance.

3. **Lock-Free Work-Stealing:** Pushing for even better performance, I attempted using exclusively atomic operations and carefully controlled memory ordering to eliminate mutexes entirely. This approach theoretically offers the highest performance by completely avoiding lock contention, but I found it extremely difficult to implement correctly. Managing atomics and memory ordering proved challenging, especially with edge cases like concurrent operations on nearly-empty deques where multiple threads might try to take the last task simultaneously.

I spent considerable time trying to implement a lock-free work-stealing deque based on Chase-Lev principles. Despite focusing on atomic interactions and memory synchronization, my implementation was unstable - it often deadlocked or hung during testing. I noticed its behavior changed with subtle timing variations or even debugging prints, classic signs of complex race conditions. Given these challenges and the project timeframe, I made a practical decision to focus on ensuring robust implementations of the other two approaches. I fully implemented, tested and benchmarked both the Centralized Task Queue and Mutex-Based Work-Stealing approaches for this report.

3 Theoretical Performance Analysis

I applied the Work-Span model to estimate the parallelism in my benchmark workloads. For this analysis, I approximated Work (W) as the total Collatz steps across all numbers, and Span (S) as the maximum steps for any single number. The theoretical Parallelism $P = W/S$ indicates the maximum possible speedup we could achieve in an ideal scenario, neglecting all practical overheads.

Table 1: Theoretical Work-Span Analysis Results for Benchmark Workloads.

Workload Description	Work (W)	Span (S)	Parallelism (P=W/S)
Medium Balanced (1-100k)	10753840	350	30725.3
Large Balanced (1-1M)	131434424	524	250829.0
Unbalanced Mix (Small, Large, Medium)	66057430	448	147450.0
Many Small Uniform Ranges (500x1k)	62134795	448	138694.0
Ranges Around Powers of 2 (2^8 to 2^20)	1271514	363	3502.8
Extreme Imbalance with Isolated Expensive Calculations	1113876	685	1626.1

My analysis in Table 1 shows that theoretical Parallelism (P) varies significantly across workloads, primarily driven by the large differences in total Work (W). I found that workloads processing substantially more numbers (e.g., "Large Balanced", "Unbalanced Mix") naturally have orders of magnitude more total work. Since the Span (S), determined by the single longest sequence, varies relatively little (within a factor of ≈2 in this case), I observed that the $P = W/S$ ratio largely mirrors the scale of W.

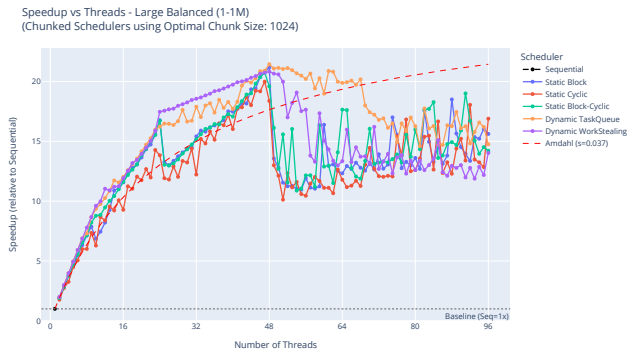
Therefore, in my analysis, while workloads like "Large Balanced" show extremely high theoretical parallelism ($P \approx 250k$), indicating vast potential for parallel execution limited mainly by processor count, workloads like "Extreme Imbalance" ($P \approx 1.6k$) are theoretically limited sooner. I found this isn't necessarily because their critical path is intrinsically problematic relative to the type of work, but because the total amount of work (W) is much smaller compared to their Span (S). This led me to conclude that achieving high speedups on smaller workloads might be challenging even theoretically, before considering any practical overheads in my implementations.

4 Experimental Performance Analysis

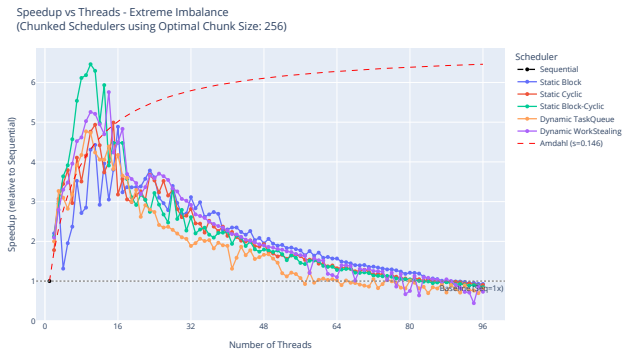
The parallel implementations were benchmarked on a dual-socket system with two Intel Xeon Gold 5318Y CPUs, offering 48 physical cores (96 logical threads) in a NUMA architecture[1]. The system runs Ubuntu 22.04 with 1TB RAM, and all code was compiled using GCC 11.4.0 with `-std=c++17 -O3 -pthread` options. Execution times reported are the median of 10 samples (each with 50 iterations) to ensure measurement reliability. Strong speedup is calculated relative to the optimized sequential baseline. Tests explored thread counts up to 96 and chunk sizes ranging from 16 to 1024.

4.1 Static vs. Dynamic Scheduling Speedup

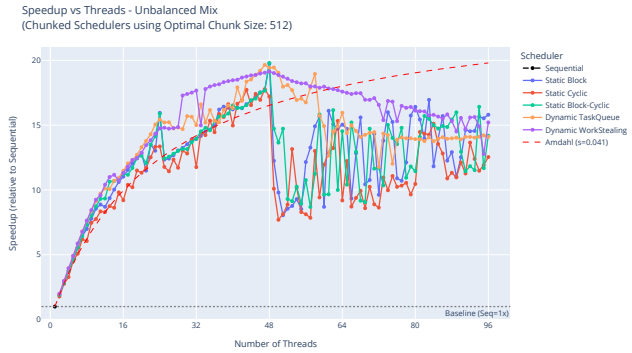
The performance comparison between static and dynamic schedulers highlights the importance of workload characteristics and the underlying hardware architecture. Figure 1 presents the speedup achieved on four representative workloads, utilizing the empirically determined optimal chunk size for chunk-dependent schedulers (Static Block-Cyclic, Dynamic TaskQueue, Dynamic WorkStealing) for each specific workload.



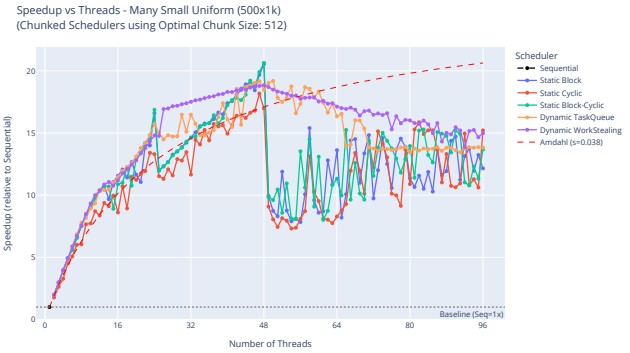
(a) Large Balanced (1-1M) - Optimal Chunk 1024



(b) Extreme Imbalance - Optimal Chunk 256



(c) Unbalanced Mix (Small, Large, Medium)



(d) Many Small Uniform Ranges (500x1k)

Figure 1: Speedup vs. Threads for selected workloads (Chunked schedulers use optimal chunk size per workload)

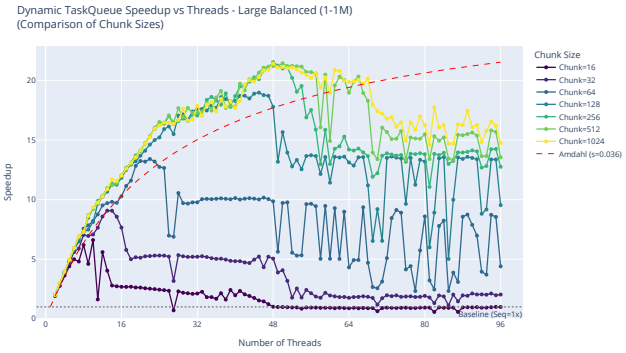
For balanced workloads with substantial computation (Fig 1a), well-tuned static and dynamic schedulers perform nearly identically, achieving excellent speedup (21-22x) at 48 threads. This indicates that when computation is uniform, scheduler overhead differences become negligible compared to the actual computation time. The "Extreme Imbalance" workload (Fig 1b) reveals a counter-intuitive result: static schedulers (particularly Block-Cyclic) outperform dynamic approaches. This suggests that for workloads with many extremely short tasks, the overhead of dynamic scheduling mechanisms (mutex contention, deque operations) becomes significant relative to task execution time. Static approaches benefit from their lower scheduling overhead despite potentially imperfect load distribution. Similar patterns appear in mixed workloads (Fig 1c, 1d).

[1] We will see that this architecture affects significantly the scalability of the implementations, that are NUMA-unaware

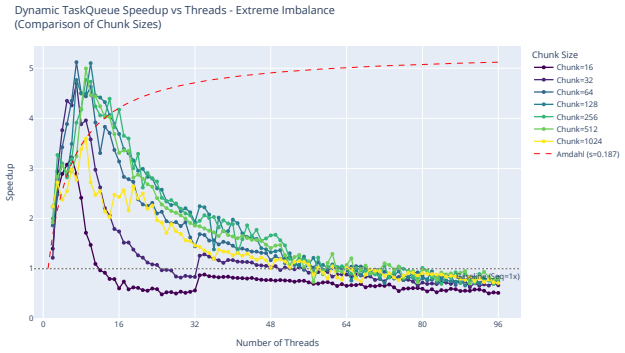
A critical observation is the performance degradation beyond 48 threads across all implementations. This clearly demonstrates the impact of the system's NUMA architecture, where cross-socket communication penalties begin to outweigh the benefits of additional threads.

4.2 Impact of Chunk Size

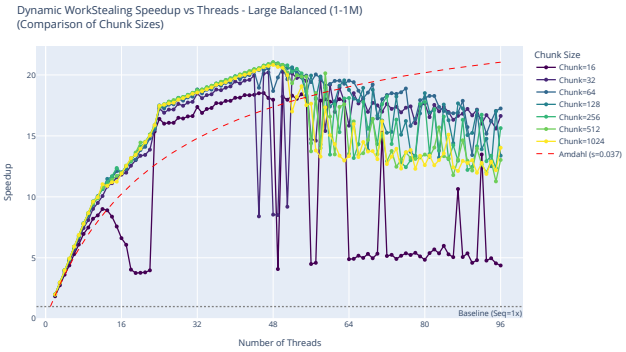
The chunk size parameter mediates the trade-off between scheduling overhead (favoring larger chunks) and load balancing granularity (favoring smaller chunks). Figure 2 illustrates this by comparing the speedup of Dynamic TaskQueue, Dynamic WorkStealing, and Static Block-Cyclic across various chunk sizes for both the "Large Balanced" and "Extreme Imbalance" workloads.



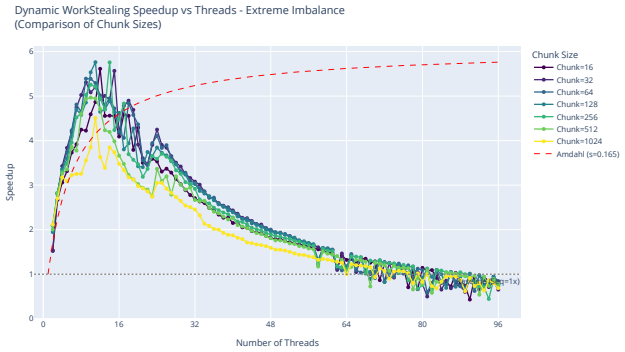
(a) Dynamic TaskQueue: Large Balanced



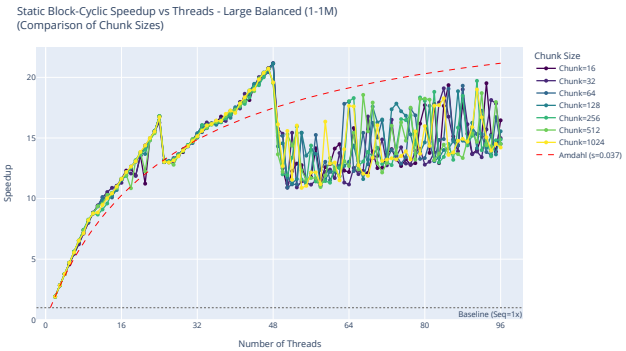
(b) Dynamic TaskQueue: Extreme Imbalance



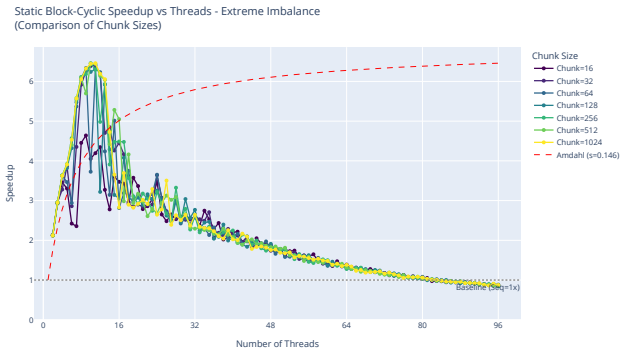
(c) Dynamic WorkStealing: Large Balanced



(d) Dynamic WorkStealing: Extreme Imbalance



(e) Static Block-Cyclic: Large Balanced



(f) Static Block-Cyclic: Extreme Imbalance

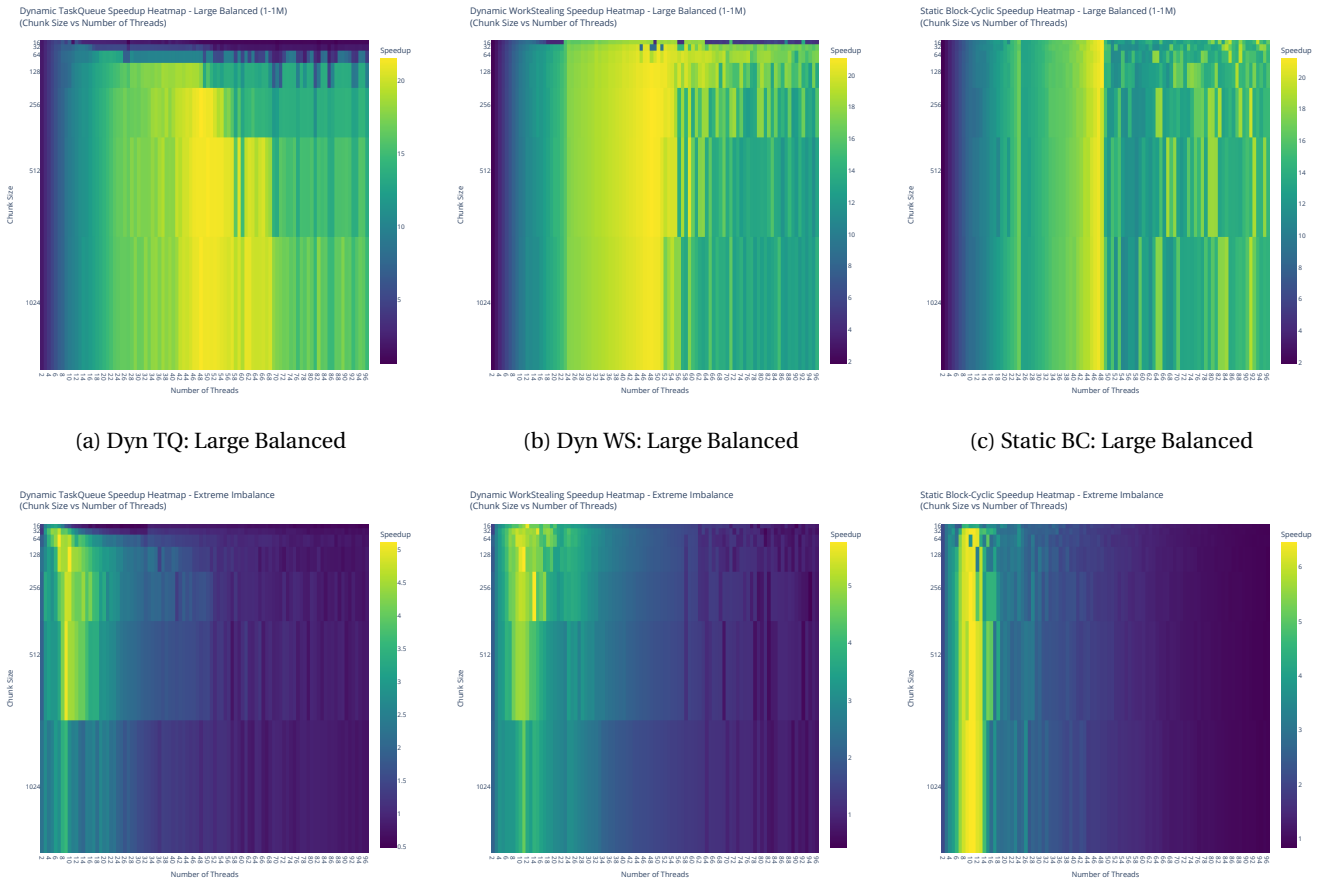
Figure 2: Impact of Chunk Size on Scheduler Speedup vs. Threads for both Static and Dynamic approaches.

Static Block-Cyclic (Figures 2e and 2f) demonstrates notable resilience to chunk size variation, particularly with balanced

workloads. Its performance overhead is primarily from initial distribution, not runtime scheduling. In contrast, dynamic schedulers (TaskQueue in Figures 2a and 2b, and WorkStealing in Figures 2c and 2d) show higher sensitivity to chunk size selection. For balanced workloads, larger chunks minimize overhead in both TaskQueue and WorkStealing approaches. However, with imbalanced workloads, medium-sized chunks provide better load distribution.

4.3 Performance Heatmaps

Heatmaps provide a comprehensive visualization of performance across the entire tested parameter space of thread count and chunk size. Figure 3 shows these heatmaps for the Dynamic TaskQueue, Dynamic WorkStealing, and Static Block-Cyclic schedulers on both balanced and imbalanced workloads.



(a) Dyn TQ: Large Balanced

(b) Dyn WS: Large Balanced

(c) Static BC: Large Balanced

(d) Dyn TQ: Extreme Imbalance

(e) Dyn WS: Extreme Imbalance

(f) Static BC: Extreme Imbalance

Figure 3: Speedup Heatmaps for Balanced (top row) and Imbalanced (bottom row) Workloads (Chunk Size vs. Number of Threads).

For the "Large Balanced" workload (Figures 3a, 3b, and 3c), all three schedulers exhibit a broad region of optimal performance (brightest yellow areas). This region typically spans medium-to-large chunk sizes (256-1024) and moderate-to-high thread counts within a single socket (≈30-48 threads). This visually confirms that multiple parameter combinations yield good results when the workload is regular.

Conversely, for the "Extreme Imbalance" workload (Figures 3d, 3e, and 3f), the optimal performance region is much narrower and located differently. High speedup is concentrated at lower thread counts (roughly 8-16 threads) and requires smaller intermediate chunk sizes (typically 64-256, sometimes up to 512). The heatmaps also visually suggest that Static Block-Cyclic (Fig 3f) achieves a slightly higher peak speedup (brighter spot) within its optimal zone compared to the dynamic schedulers (Fig 3d, 3e) for this specific challenging workload.