Assignment 2: Parallel Collatz Problem

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. I implemented three variants to compare their characteristics:

- **Block:** Assigns large contiguous blocks per range to each thread. It features low scheduling overhead but suffers from poor load balancing, especially for non-uniform workloads[1].
- **Cyclic:** Assigns individual numbers round-robin. Offers fine-grained load balancing but can lead to suboptimal cache performance due to scattered memory accesses.
- **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. 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.

2.2 Dynamic Scheduling Strategies

For workloads like the Collatz computation where task times can vary a lot, dynamic scheduling often works better than static approaches because it lets idle threads grab new tasks as needed, improving load balance. I explored two main designs for this project: a centralized queue and decentralized work-stealing.

First, I looked at a **Centralized Task Queue**. This involves a single, shared queue, like the `TaskQueue` class I built, that all threads use. Usually, you protect it with a `std::mutex` and use a `std::condition_variable` to manage waiting threads. It's fairly straightforward to implement and naturally balances the load. The main downside, though, is that the single mutex can become a bottleneck when many threads try to access it at once, limiting how well the program scales.

The second, more advanced strategy is **Decentralized Work-Stealing**. The idea here is to give each thread its own deque (a double-ended queue). Threads mostly work on their own deque, which reduces conflicts. When a thread runs out of

[1] like Collatz where computation cost per number varies significantly.

local work, it tries to "steal" a task from another thread's deque. A way to build these deques without locks is the Chase-Lev algorithm, using only atomic operations (`std::atomic`) and careful memory ordering (`acquire/release/seq_cst`). This promises much better scalability because there are no central locks. However, implementing it right is much harder.

My initial plan was to go with the more scalable lock-free work-stealing approach, so I built a custom `ChaseLevDeque` class. I spent a good amount of time trying to get the atomic operations and memory ordering correct, especially for the tricky cases where a thread tries to pop the last element (`pop_bottom`) just as another thread tries to steal it (`steal_top`). Unfortunately, I kept running into frustrating correctness problems during testing. The program would often deadlock or hang, and whether it happened seemed to depend on subtle timing factors – sometimes adding a simple print statement for debugging would make the problem disappear or appear. This kind of behavior often points to very complex race conditions that are common in lock-free programming. Pinpointing and fixing these issues proved incredibly difficult within the time I had.

Given these challenges in making the lock-free version reliable, I made the practical decision to fall back to the simpler, centralized `TaskQueue` implementation for the dynamic scheduling benchmarks in this report. I knew this meant accepting the potential mutex bottleneck, but it ensured I had a working and correct dynamic scheduler to compare against the static ones.

3 Theoretical Performance Analysis

We applied the Work-Span model to estimate the inherent parallelism in the benchmark workloads. Work (W) was approximated 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, neglecting 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

The results in Table 1 show that theoretical Parallelism (P) varies significantly, primarily driven by the large differences in total Work (W) across workloads. 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), the $P = W/S$ ratio largely mirrors the scale of W.

Therefore, 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, not 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 suggests that achieving high speedups on smaller workloads might be challenging even theoretically, before considering practical overheads. This idealized analysis provides context for evaluating experimental speedups.

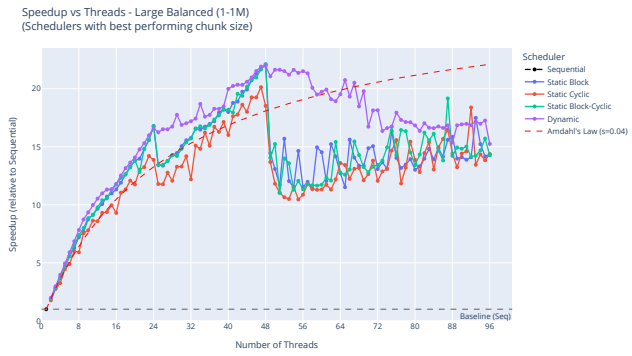
4 Experimental Performance Analysis

The parallel implementations were benchmarked on a machine equipped with two Intel(R) Xeon(R) Gold 5318Y CPUs. Each CPU has 24 physical cores supporting Hyper-Threading, providing 48 logical threads per socket, for a system total of 48 physical cores and 96 logical threads. This dual-socket configuration presents a Non-Uniform Memory Access (NUMA) architecture, where memory access latency differs depending on whether a thread accesses memory local to its own CPU socket or remote memory attached to the other socket. The system runs Ubuntu 22.04 (Linux kernel 5.15.0-119-generic) and is equipped with 1TB of RAM distributed across the sockets. All code was compiled using GCC 11.4.0 with options `-std=c++17 -O3 -pthread`. Execution times reported are the median of 10 samples, each consisting of 50

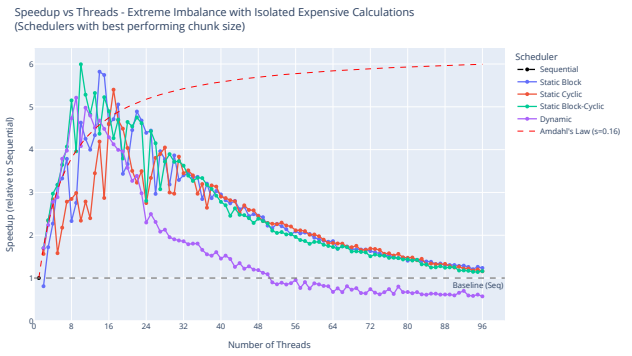
iterations, to mitigate measurement noise. Strong speedup is calculated relative to the optimized sequential baseline for each workload. Tests varied thread counts up to 96 and chunk sizes from 16 to 1024.

4.1 Static vs. Dynamic Scheduling Speedup

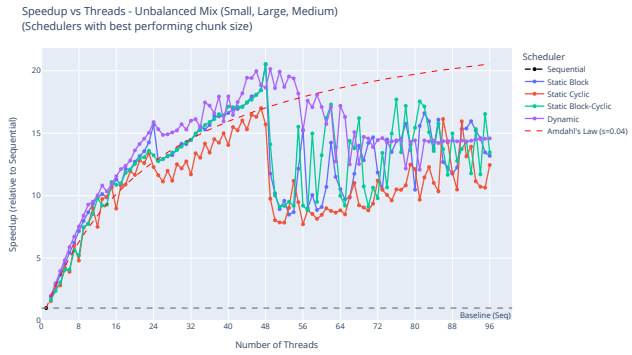
The performance comparison between static and dynamic schedulers highlights the importance of workload characteristics and the underlying hardware architecture. Figure 1 presents the speedup achieved on four representative workloads, utilizing the empirically determined optimal chunk size for chunk-dependent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



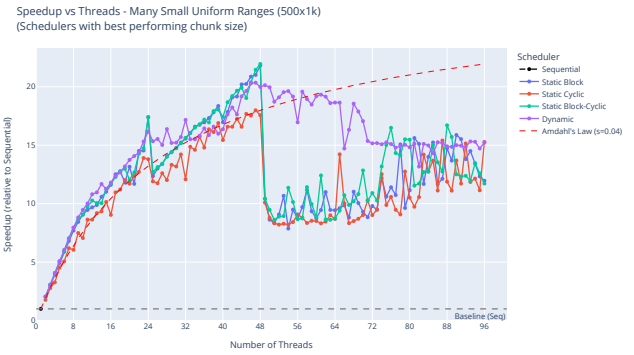
(a) Large Balanced (1-1M)



(b) Extreme Imbalance



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



(d) Many Small Uniform Ranges (500x1k)

Figure 1: Speedup vs. Threads for selected workloads (Schedulers use optimal chunk size per workload). Note varying y-axis scales and performance beyond 48 threads.

For reasonably balanced or mixed workloads (Fig 1a, 1c, 1d), the dynamic work-stealing scheduler generally achieves the highest peak speedup, often slightly exceeding Static Block-Cyclic when optimally tuned. Both scale well up to roughly the capacity of one CPU socket (48 threads).

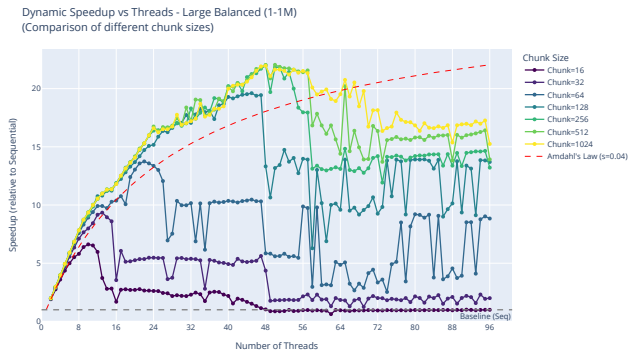
However, the "Extreme Imbalance" workload (Fig 1b) reveals a striking inversion. The dynamic work-stealing scheduler, expected to excel with imbalance, performs significantly worse than static schedulers (especially Static Block-Cyclic and Cyclic) beyond approximately 16 threads. This suggests that its overhead becomes detrimental when managing tasks with extreme duration variance (with many tasks being near-instantaneous while a few are very long). One potential cause is the high relative overhead, where the cost of lock-free deque operations, such as atomic operations and the maintenance of cache coherence, dominates the execution time for these numerous trivial tasks. Additionally, rapid idling of many threads may lead to concentrated contention or wasted effort when stealing very small tasks. In contrast, static methods, despite their load imbalance, benefit from minimal runtime overhead, which proves advantageous in this specifically highly skewed scenario.

A key observation across all workloads is performance degradation beyond 48 threads (Fig 1a). This reflects the system's

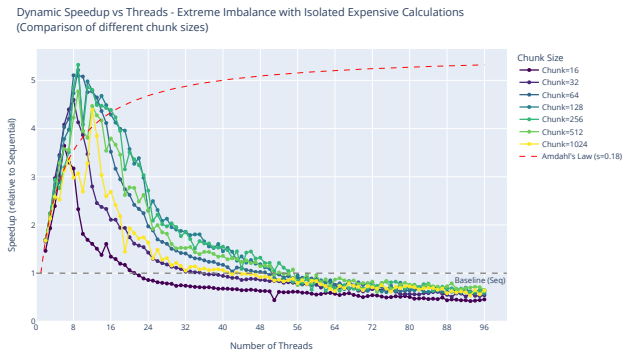
dual-socket NUMA architecture: when execution spans both sockets, cross-socket communication for shared data structures and cache coherence incurs significant latency. The implemented schedulers are NUMA-unaware, limiting their scalability across multiple sockets.

4.2 Impact of Chunk Size

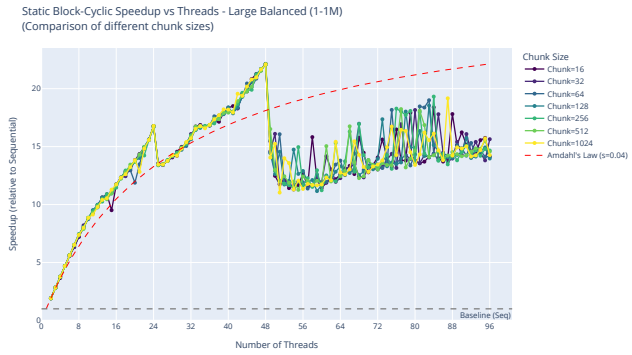
The chunk size parameter is crucial for tuning performance, mediating the trade-off between scheduling overhead (favoring larger chunks) and load balancing granularity (favoring smaller chunks). Figure 2 compares the speedup achieved by the Dynamic (Work-Stealing) and Static Block-Cyclic schedulers across various chunk sizes for both the "Large Balanced" and "Extreme Imbalance" workloads.



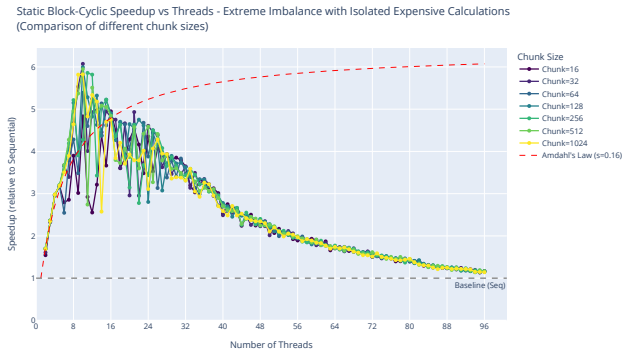
(a) Dynamic: Large Balanced (1-1M)



(b) Dynamic: Extreme Imbalance



(c) Static Block-Cyclic: Large Balanced (1-1M)



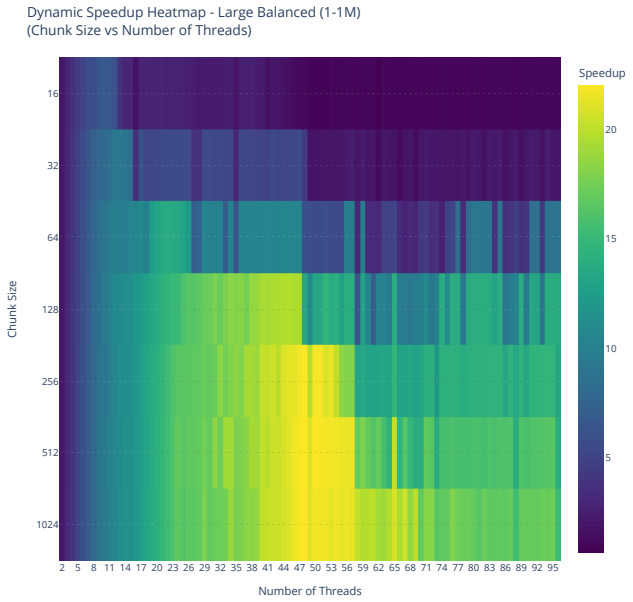
(d) Static Block-Cyclic: Extreme Imbalance

Figure 2: Impact of Chunk Size on Scheduler Speedup vs. Threads for selected workloads.

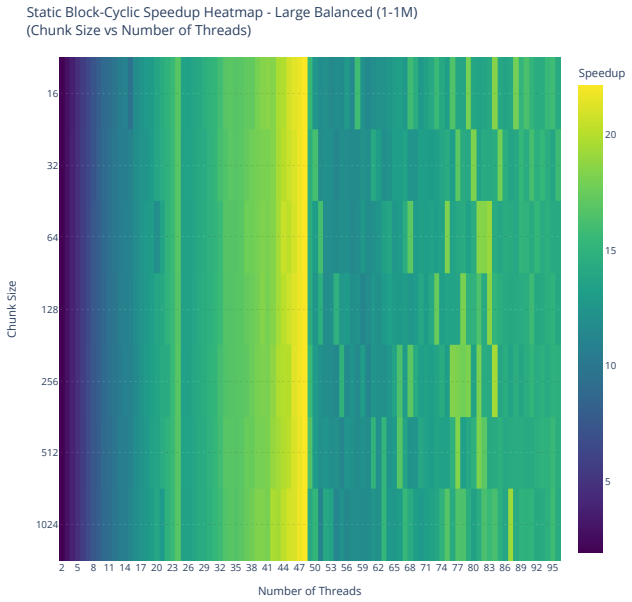
A notable observation is the difference in sensitivity to chunk size. Static Block-Cyclic (Figures 2c, 2d) appears relatively robust, showing similar performance across a wider range of medium-to-large chunk sizes once sufficient threads are active. This is likely because its overhead is primarily fixed per block assignment, regardless of the stealing activity. In contrast, the Dynamic scheduler (Figures 2a, 2b) displays stronger workload-dependent sensitivity. For the "Large Balanced" workload, larger chunks (512-1024) perform best, likely minimizing the overhead of deque operations (pushes, pops, steals) when load balancing is less critical. Conversely, for the "Extreme Imbalance" workload, these same large chunks yield poor results; smaller chunks (e.g., 64-256) become necessary, despite their higher overhead, to provide the granularity needed for work-stealing to effectively distribute the highly variable task durations. This highlights that dynamic scheduling requires more careful tuning of task granularity based on workload characteristics to maximize its benefits.

4.3 Performance Heatmaps

Heatmaps visualize performance across the thread/chunk size space, confirming the optimal regions. Figures 3 and 4 show these patterns for balanced and imbalanced workloads respectively.

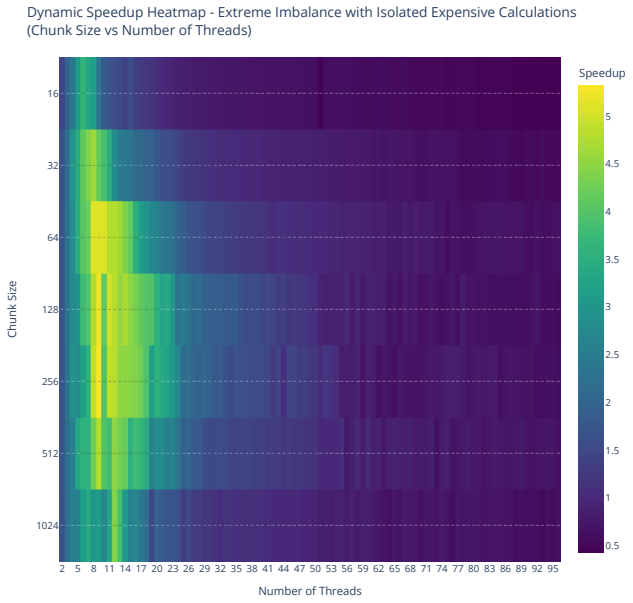


(a) Dynamic: Large Balanced (1-1M)

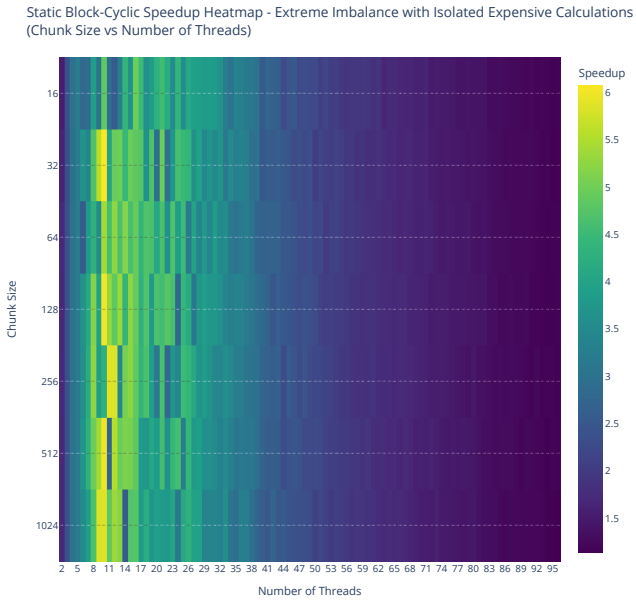


(b) Static BC: Large Balanced (1-1M)

Figure 3: Speedup Heatmaps for Balanced Workload (Chunk Size vs. Number of Threads).



(a) Dynamic: Extreme Imbalance



(b) Static BC: Extreme Imbalance

Figure 4: Speedup Heatmaps for Imbalanced Workload (Chunk Size vs. Number of Threads).

For the balanced workload (Figure 3), optimal performance (brightest regions) spans a relatively broad area of medium-to-large chunks and moderate thread counts (≈30-60) for both schedulers, confirming the observations from the line plots. For the extreme imbalance case (Figure 4), the optimal performance region is significantly narrower, concentrated at lower thread counts and smaller intermediate chunk sizes (around 64-256).