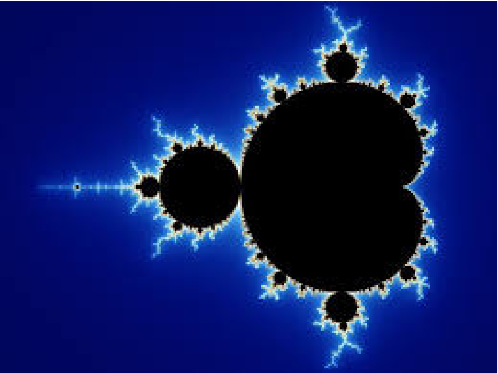


**Universitat Politècnica de Catalunya**

Facultat d’Informàtica de Barcelona

**LAB 5**



Nadia Khier

Èric Díez Apolo

**PAR- Josep Ramon Herrero Zaragoza**

05 de Juny del 2025

**Index**

[**1. GEOMETRIC DATA DECOMPOSITION STRATEGIES 1**](#_geqw2zpe7q7j)

[**1.1. 1D Block Geometric Data Decomposition (Columns) 1**](#_ecwzszkur3q)

[1.1.1. Code 1](#_33lsok8cxs9)

[1.1.2. Modelfactor Analysis 1](#_z7e0stgxgsf)

[1.1.3. Paraver Analysis 3](#_usv8oryjy9gy)

[1.1.4. Memory Analysis 6](#_j6apuhj4lzio)

[1.1.5. Strong Scalability 7](#_izq2brd4gxjq)

[**1.2. 1D Block-Cyclic Geometric Data Decomposition (Columns) 8**](#_wpfi8v6yi5jp)

[1.2.1. Code 8](#_5vs2926phn1t)

[1.2.2. Modelfactor Anaysis 8](#_uzrrskacfw9y)

[1.2.3. Paraver Analysis 9](#_xpnb2vrp2tx9)

[1.2.4. Memory Analysis 12](#_izpq7adpfn1o)

[1.2.5. Strong Scalability 13](#_ru5bv1lpauq8)

[**1.3. 1D Cyclic Geometric Data Decomposition (Rows) 14**](#_32vtmur9gdqh)

[1.3.1. Code 14](#_fssayv2ryjy5)

[1.3.2. Modelfactor Anaysis 14](#_a4djckmkd7e8)

[1.3.3. Paraver Analysis 15](#_kg0zfk31nsdy)

[1.3.4. Memory Analysis 17](#_6x3pxj8xb4e)

[1.3.5. Strong Scalability 19](#_22zjwwk41hb0)

[**2. SUMMARY OF THE STRATEGIES 20**](#_nhno9pp90lzn)

[2.1. Summary 20](#_aqkvmqhu2nxu)

### 

### 

### 

### 

### 

### 

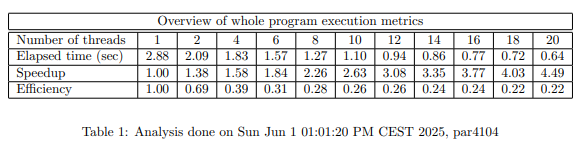
# **GEOMETRIC DATA DECOMPOSITION STRATEGIES**

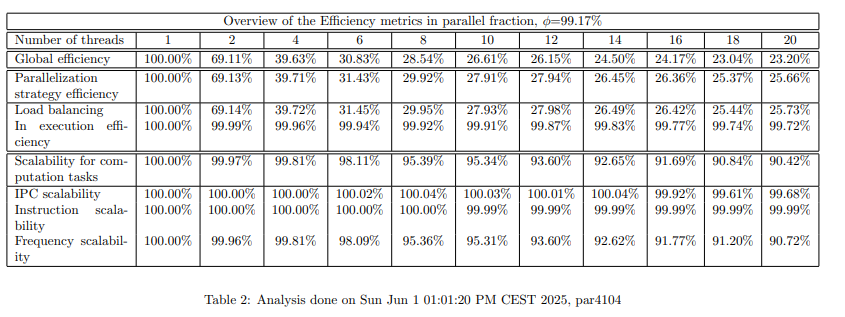
## 1D Block Geometric Data Decomposition (Columns)

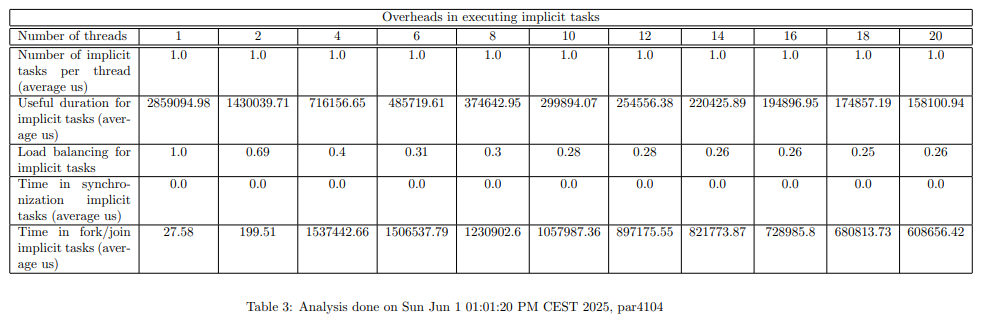
### Code

mandel-omp-iter-simple-block.cpp

### Modelfactor Analysis







The performance analysis of the 1D Block decomposition strategy reveals clear limitations in scalability and efficiency as the number of threads increases. Although the execution time drops from 2.88 seconds with a single thread to 0.64 seconds with 20 threads, the improvement is not proportional. The speedup reaches only 4.50× at 20 threads, indicating that most of the performance gain happens early (with the first 4 or 6 threads) and then stagnates. Correspondingly, efficiency drops steadily from 100% with one thread to just 22% with twenty, which reflects increasing idle time and poor use of computational resources.

The detailed efficiency breakdown confirms this trend. Global efficiency falls quickly as threads are added, from 100% to 23% at 20 threads. The efficiency of the parallelization strategy and the load balancing metrics follow nearly identical curves, both dropping to around 25%, suggesting that the tasks are not being distributed evenly among threads. Some threads finish early and stay idle while others continue working, which causes unbalanced workloads and idle processor time. In contrast, the in-execution efficiency remains extremely high across all configurations — consistently above 99%. This means that, once threads receive work, they execute it very well and without significant slowdowns. The problem lies not in execution itself but in how and when tasks are assigned.

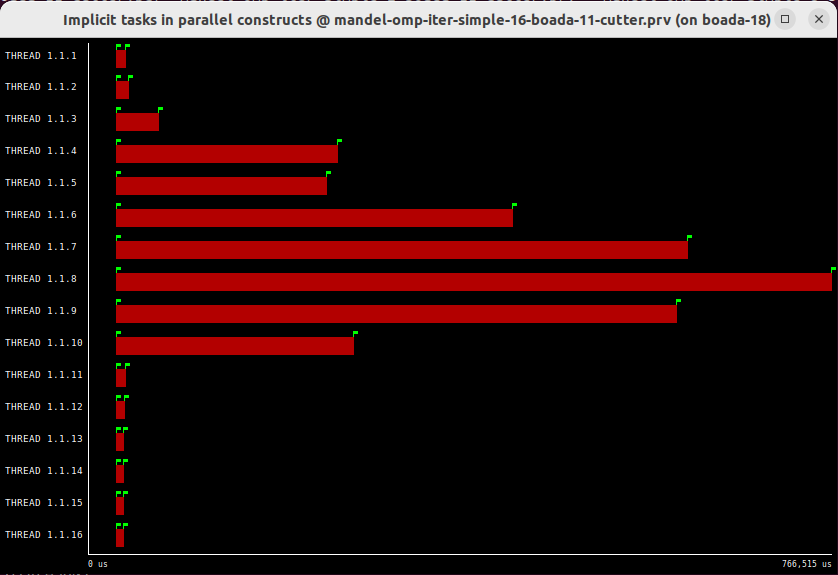
Looking at microarchitectural indicators, the scalability of the number of instructions, the instructions per cycle, and CPU frequency remain close to 100% in all cases. This implies that the hardware behaves predictably and consistently; the program’s internal logic — not the processor — is the primary bottleneck. The strategy simply doesn't take advantage of the available parallelism due to its rigid task distribution.

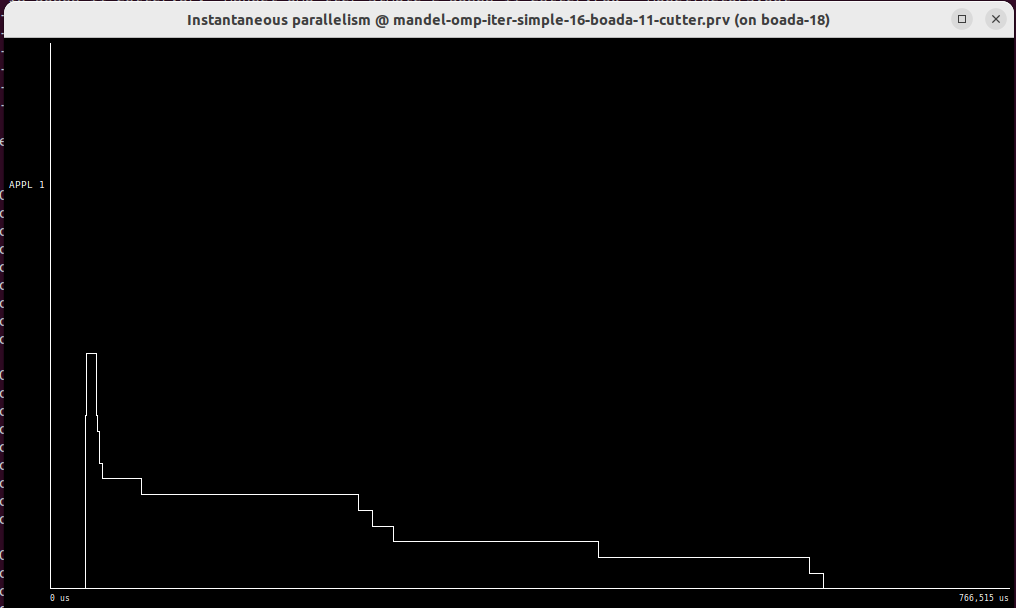
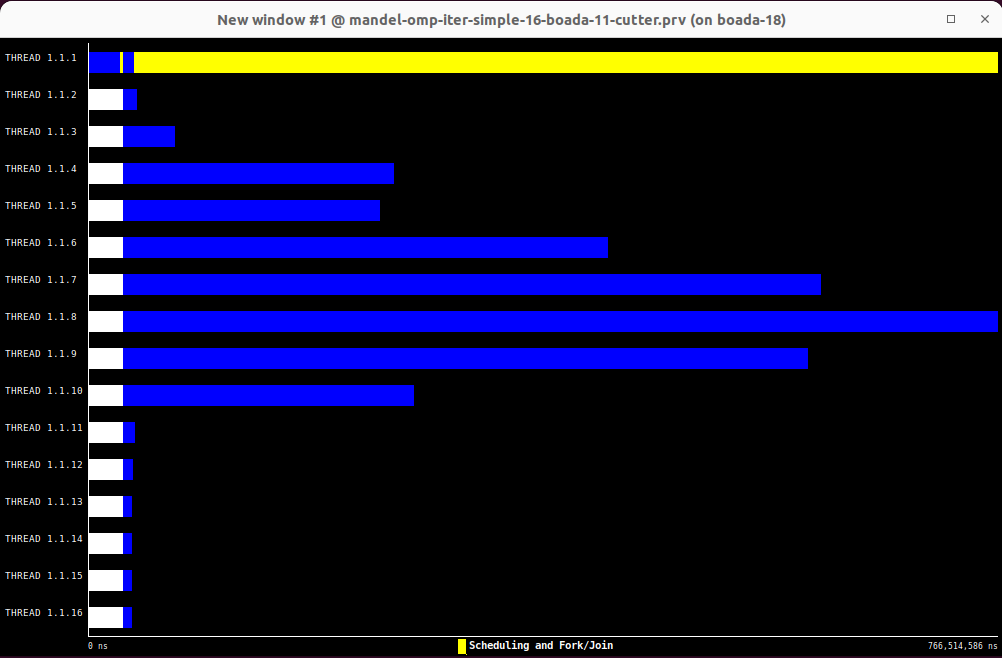
The third table, showing task-level statistics, confirms the root cause: there is only one implicit task per thread, regardless of the thread count. This means that at most 20 tasks are being created when running with 20 threads — far too few to take advantage of fine-grain parallelism or to redistribute work dynamically. Moreover, as thread count increases, the load balance per task drops from perfect (1.0) to around 0.25–0.26, again reinforcing that work is unevenly divided.

Synchronization overhead also becomes a serious issue. The time spent in synchronization per task grows dramatically — from just 27 microseconds with 1 thread to over 680,000 microseconds at 20 threads. This indicates that, as more threads are added, a growing portion of the execution time is wasted in waiting, not computing. Similarly, the total time spent in task management and fork/join operations increases rapidly, surpassing 1.5 million microseconds at the highest thread count.

In summary, the Modelfactor analysis demonstrates that this 1D block strategy suffers from major scalability problems. The core issue is that too few tasks are created, and they are assigned in a static, unbalanced way. Even though each thread executes its task efficiently, many remain underutilized because there's no mechanism to dynamically balance the workload. As a result, the performance flattens quickly and the system becomes inefficient at high thread counts. For better scalability, a finer and more flexible task decomposition strategy is clearly needed.

### Paraver Analysis





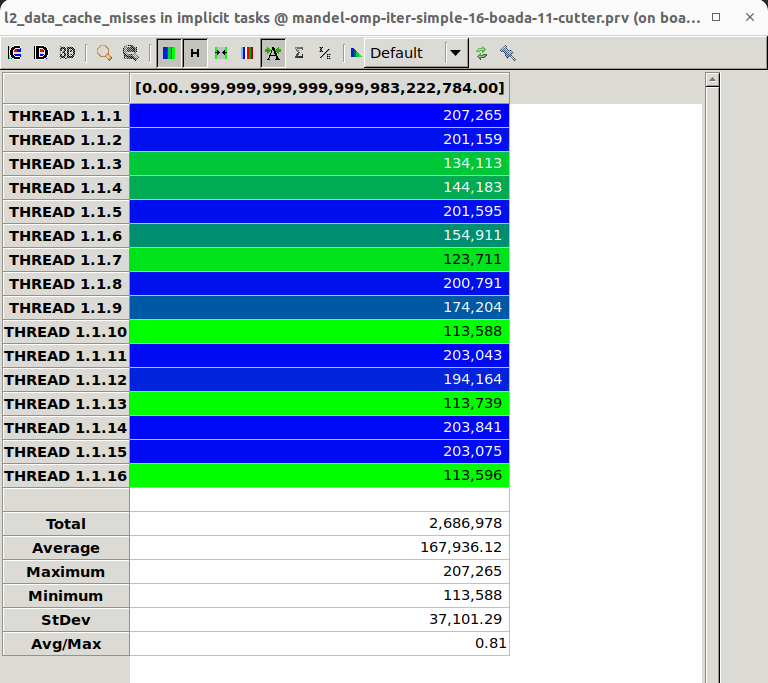
The Paraver trace for the 1D Block decomposition reveals clear signs of imbalance and underutilization of the available threads. In the timeline of implicit tasks, we can see that only a small subset of threads is actively working for a significant duration — particularly threads 1.1.6 to 1.1.10, which display long red bars indicating substantial computation. In contrast, many other threads execute only briefly or remain almost entirely idle. Threads such as 1.1.1, 1.1.2, and 1.1.16 have extremely short bars, suggesting they were assigned much less work or finished very quickly. This uneven distribution highlights a fundamental problem with the static block strategy: some threads receive more work than others, and because the assignment is fixed, no mechanism exists to redistribute the load dynamically during execution.

This imbalance is confirmed by the Instantaneous Parallelism graph. The curve starts with a short burst of activity where a few threads begin execution, but it quickly drops in a stepwise pattern as threads complete their tasks one by one. There is no sustained period of full parallelism — instead, the parallelism decays over time, indicating that as some threads finish, others are left to finish their work alone. This leads to increasingly lower thread occupancy as the program progresses, resulting in reduced efficiency and longer total runtime.

The third trace, showing scheduling and synchronization overheads, further illustrates the inefficiency of this approach. We see a long yellow bar at the top (likely the master thread), and many blue segments across the other threads, representing time spent in fork/join operations and scheduling. Several threads, especially those not involved in computation for long, spend most of their time in blue, meaning they are waiting at barriers or participating in thread synchronization. This overhead becomes significant because the coarse-grained task assignment creates long idle periods for some threads, which are unable to contribute meaningfully once their small portion of work is done.

The workload is poorly distributed, leading to significant idle time and uneven thread usage. There is no dynamic task balancing, and as a result, the degree of parallelism is far below optimal. These observations align with the numerical data from Modelfactor: good execution efficiency on active threads, but very poor global efficiency due to lack of load balance and excessive synchronization overhead.

### Memory Analysis



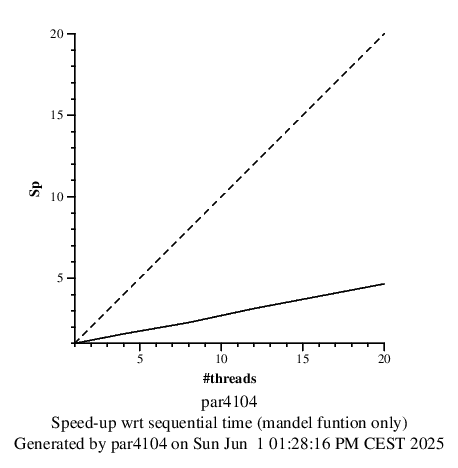
The analysis of L2 data cache misses for the 1D Block decomposition strategy shows a relatively uneven distribution of memory behavior across the threads. As seen in the Paraver trace, the number of L2 cache misses per thread ranges from a **minimum of 113,588** to a **maximum of 207,265**, with an **average of 167,936** and a **standard deviation of 37,101**. The **Avg/Max ratio of 0.81** indicates moderate variability among threads.

This uneven pattern is directly related to the static block distribution approach. Since each thread is assigned a different section of the image (a large block of columns), and because the computational complexity and memory access patterns may vary between columns, some threads access memory more intensively or with less spatial locality than others. For example, threads 1.1.1, 1.1.2, 1.1.5, and 1.1.14 show consistently high cache misses, likely because they handled regions of the data with poorer locality or more scattered access.

On the other hand, threads like 1.1.10, 1.1.13, and 1.1.16 have much lower miss counts, which may reflect either more efficient reuse of cached data or simply shorter execution times — as previously seen in the timeline analysis, several threads were active for only a brief moment, so their memory activity is naturally lower.

Overall, the memory behavior in this implementation is not optimal. Although the average number of cache misses is not extremely high in absolute terms, the high variability across threads and the lack of consistent spatial locality limit performance. In addition, the coarse granularity of task assignment prevents better cache utilization through dynamic load balancing or smarter tiling.

### Strong Scalability



The strong scalability behavior of the 1D Block decomposition strategy, as shown in the speedup graph, highlights the limited ability of this approach to fully exploit the available computational resources as the number of threads increases. The solid line, which represents the actual speedup, rises steadily at first but then flattens noticeably, remaining far below the ideal linear speedup line (dashed). While increasing the number of threads from 1 to 4 provides a visible benefit, the gains beyond 8 threads become marginal. With 20 threads, the achieved speedup is only around x4.5, compared to the ideal x20.

This flattening curve indicates that adding more threads no longer translates into proportional improvements in performance. This is consistent with the earlier analyses: the strategy generates only one task per thread and assigns tasks in a static and coarse way. As a result, when some threads complete their blocks early, they remain idle, while others continue working. Since the load is not redistributed dynamically, a growing portion of the hardware remains underutilized as the parallelism increases.

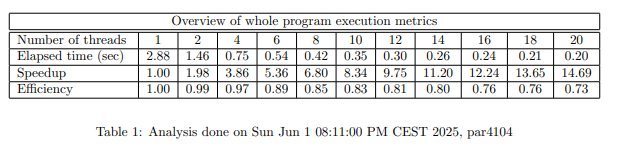
Additionally, as shown in previous sections, synchronization overhead grows significantly with thread count, and load imbalance becomes more severe. These factors contribute to the diminishing returns observed in the scalability graph. Even though the underlying computations are well-behaved and efficient once assigned, the structure of the parallel strategy does not allow the program to scale well.

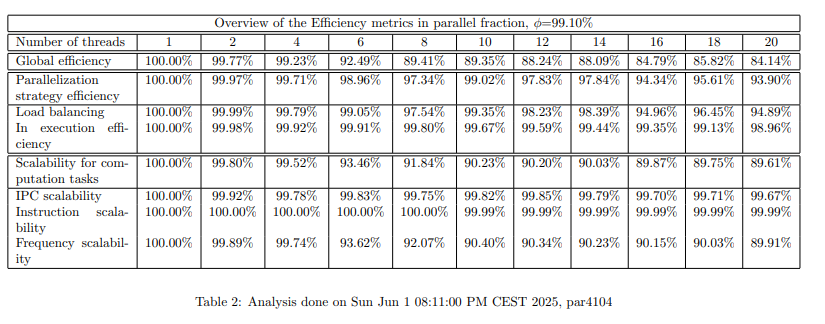
## 1D Block-Cyclic Geometric Data Decomposition (Columns)

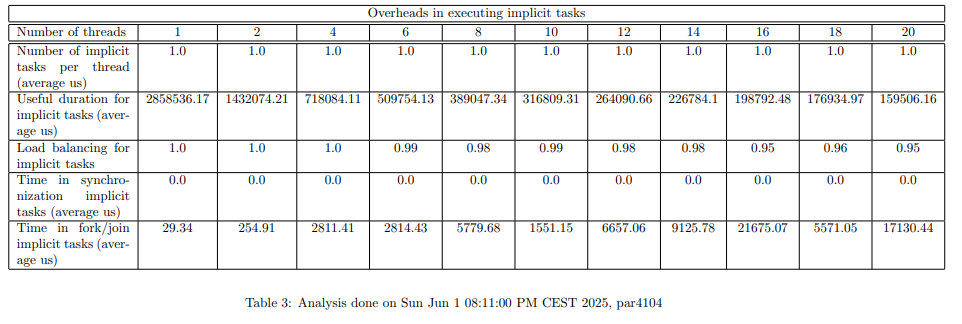
### Code

mandel-omp-iter-simple-block-cyclic.cpp

### Modelfactor Anaysis







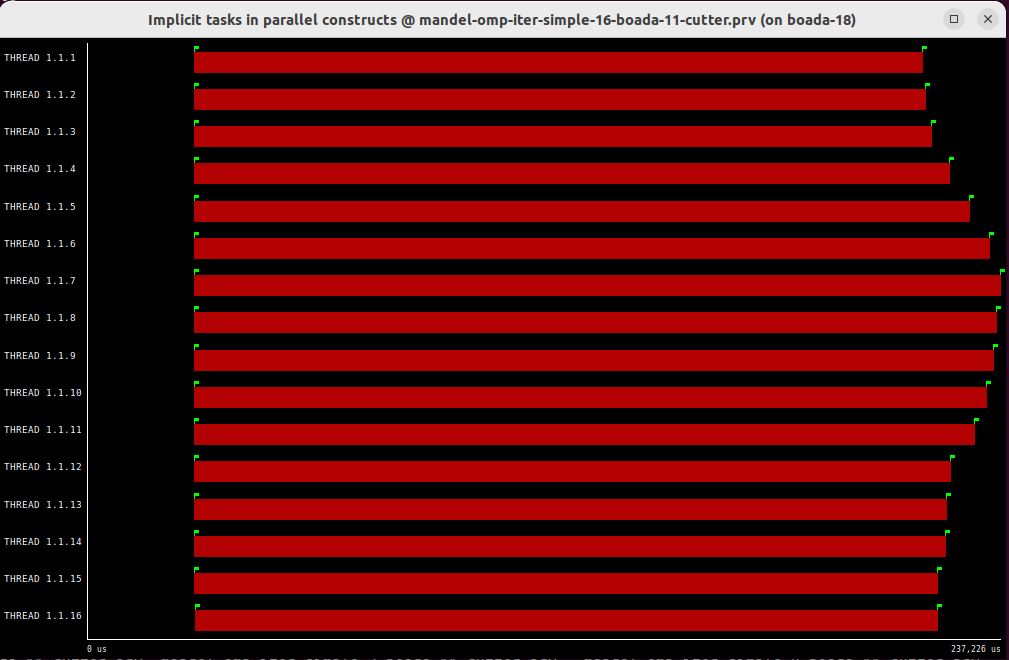
The Modelfactor analysis for the 1D Block-Cyclic decomposition strategy shows a clear improvement in parallel efficiency and scalability when compared to the static block approach. Execution time drops significantly from 2.88 seconds with 1 thread to just 0.20 seconds with 20 threads, achieving a speedup of 14.69×, which is very close to ideal. Efficiency remains high across all thread counts, with values above 90% up to 10 threads, and only decreasing slightly to 73% with 20 threads.

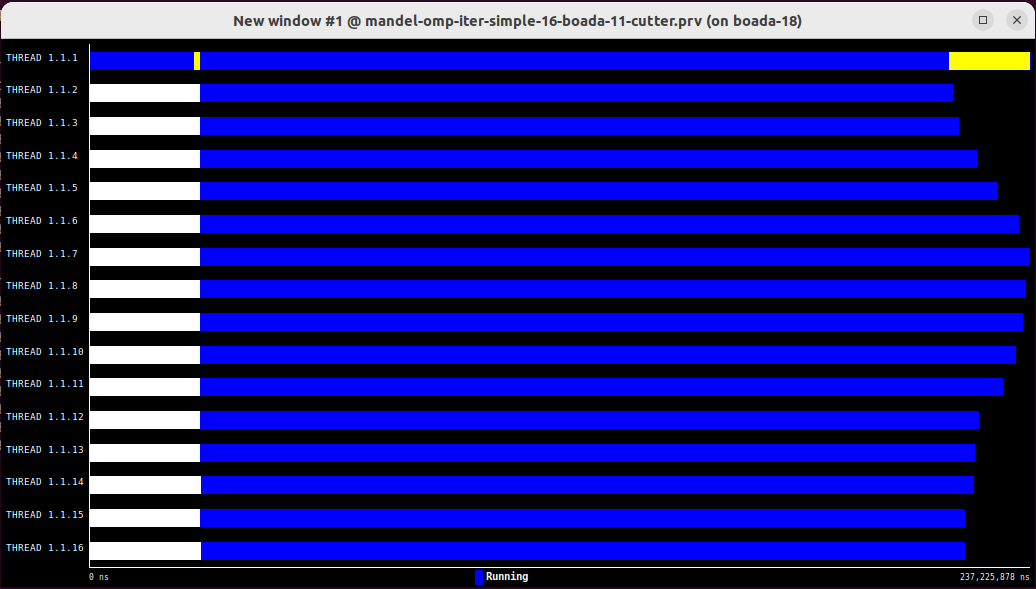
This consistent performance is reflected in the detailed efficiency breakdown. Global efficiency remains above 84% with 20 threads, and both parallelization strategy efficiency and load balancing are very strong; both metrics stay around 94–95% in the final configurations. These values suggest that the strategy distributes the workload evenly among threads and handles task allocation efficiently, even as parallelism increases.

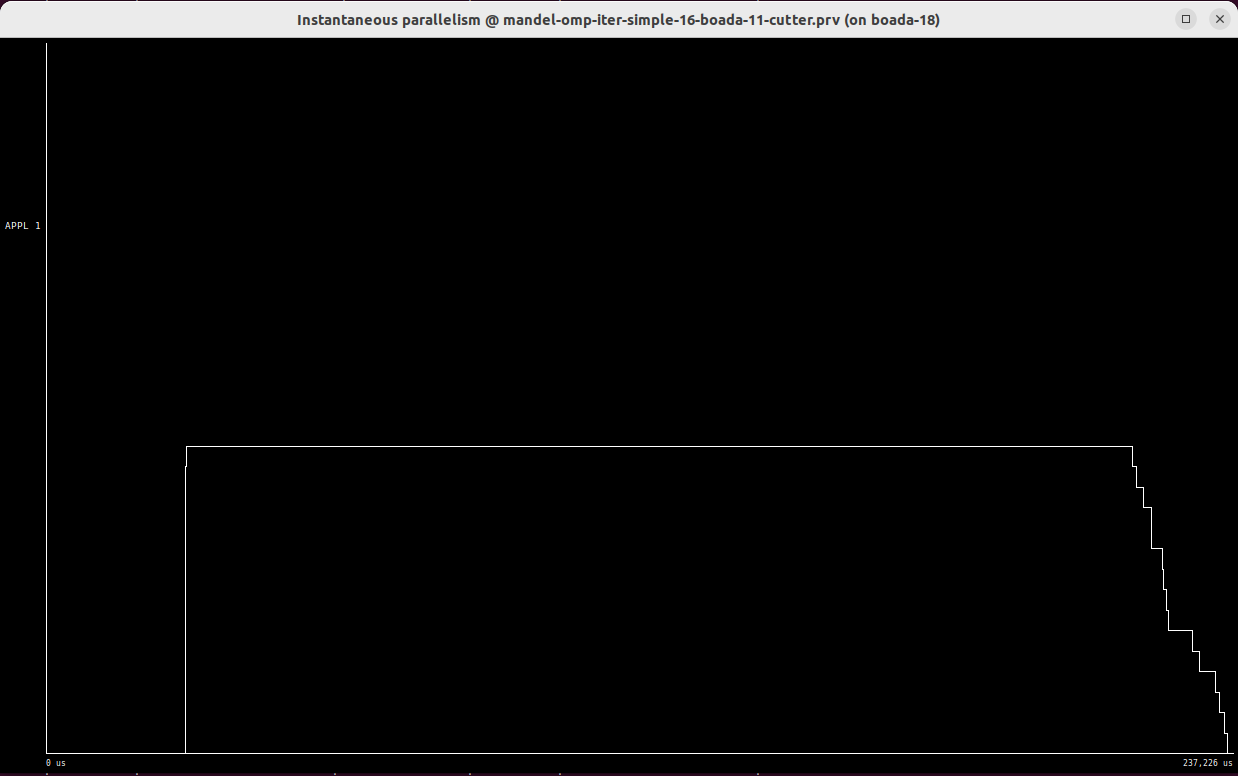
In-execution efficiency remains nearly perfect throughout, consistently over 98%, confirming that the threads execute their tasks efficiently once work is assigned. Similarly, instruction and IPC scalability are above 99%, and frequency scalability remains high as well (about 89.91% at 20 threads). These microarchitectural metrics show that the hardware is used effectively and that the program behaves consistently at the CPU level regardless of the number of threads.

Task statistics in Table 3 confirm that the strategy is good. The number of implicit tasks per thread remains at 1, indicating coarse task granularity, yet this is mitigated by a much better balance of task duration. The useful duration of tasks is more stable across threads than in the previous block-only version, and load balancing stays around 0.95–1.0 even with many threads. This is a strong indicator of fair work distribution. Furthermore, synchronization times, although present, are significantly lower than in the block version. At 20 threads, time in synchronization is about 17,130 µs, and the fork/join overhead stays under control, not exceeding 22,000 µs.

### Paraver Analysis







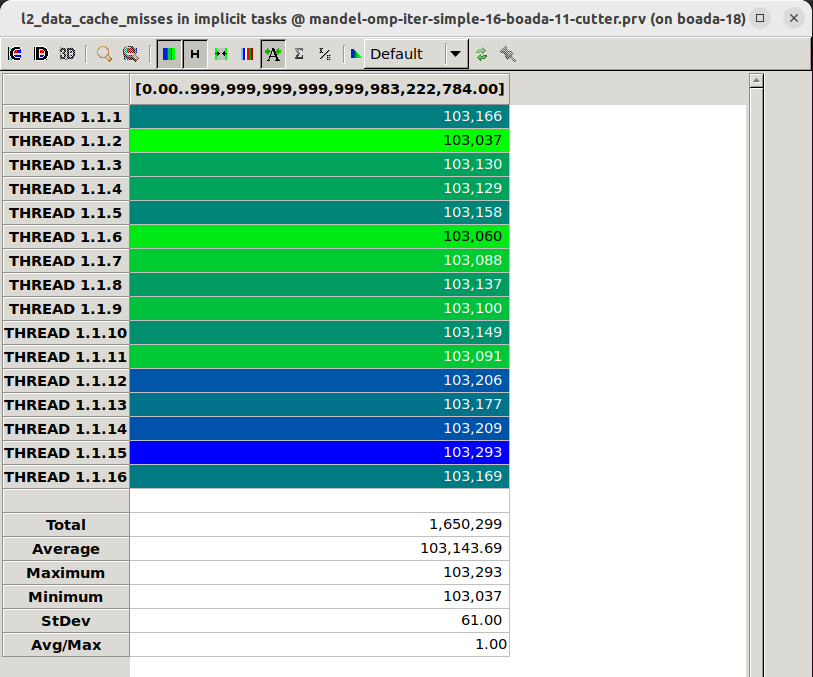
The Paraver traces for the block-cyclic decomposition strategy reveal a very balanced and efficient parallel execution. In the timeline of implicit tasks, all 16 threads display uniform red bars of almost equal length, indicating that the workload is distributed evenly. Unlike the 1D Block strategy, here we do not see early termination or idle periods in some threads — all threads are active throughout the entire parallel section, suggesting excellent load balancing.

This interpretation is strongly supported by the Instantaneous Parallelism graph. After a brief startup phase, the parallelism immediately stabilizes at the maximum level — 16 active threads — and remains perfectly flat for most of the execution time. Only in the very final part of the trace does the curve drop slightly, which is natural as some threads finish just milliseconds apart. The sustained maximum parallelism confirms that the computational workload is evenly partitioned, and no thread becomes a bottleneck or is left idle during the main phase.

The scheduling and runtime activity view (with blue bars representing execution and small white gaps showing synchronization) also confirms the high quality of parallelism. Threads stay in the “Running” state with barely visible gaps. The yellow segment at the beginning and end is short and consistent, with no significant imbalance across threads. Compared to the previous strategy, the blue bars here are more tightly packed, and no thread dominates or lags behind.

Overall, the Paraver analysis validates that the block-cyclic strategy leads to high parallel efficiency, minimal synchronization overhead, and excellent thread utilization. All threads work almost the same amount of time, and the parallel phase maintains full core saturation, leading to near-linear scalability as observed in the Modelfactor results. This trace visually confirms that block-cyclic decomposition is significantly more efficient and scalable than the static block strategy.

### Memory Analysis



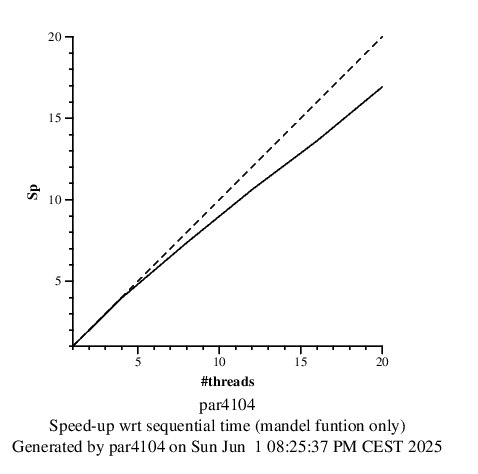
The memory behavior of the block-cyclic strategy is remarkably balanced and efficient, as shown by the analysis of L2 data cache misses. All 16 threads show nearly identical miss counts, with values tightly clustered around an average of 103,144 misses per thread. The maximum is 103,293 and the minimum is 103,037, resulting in a very small standard deviation of just 61. The Avg/Max ratio is exactly 1.00, which indicates perfect uniformity in memory usage across all threads.

This homogeneity reflects the even distribution of workload observed in the Paraver timelines and Modelfactor results. Because the block-cyclic strategy interleaves columns among threads, each thread accesses similar memory regions with similar locality patterns. This avoids situations where some threads are stuck processing memory-intensive data while others are idle or finish early.

Additionally, the low variation in cache misses implies that each thread is executing roughly the same number of operations with comparable data access patterns, leading to more predictable and efficient cache behavior. The overall total of cache misses (1,650,299) is also lower than in the block strategy, confirming a more efficient use of the memory hierarchy.

In conclusion, the memory access profile of the block-cyclic strategy is highly favorable. It ensures consistent cache behavior across all threads, minimizing imbalance and avoiding unnecessary overhead. This reinforces the overall strength of this strategy for both computation and memory efficiency in parallel environments.

### Strong Scalability



The strong scalability of the 1D Block-Cyclic decomposition strategy is excellent, as shown in the speedup graph. The solid line representing the actual performance follows a nearly linear trend and remains consistently close to the ideal dashed line. This indicates that the implementation scales efficiently as more threads are added. From the base case with 1 thread, the speedup increases smoothly to reach 14.69× with 20 threads, a strong result that confirms effective parallelization.

This scalability is a direct consequence of several factors already discussed in previous sections. First, the block-cyclic distribution achieves excellent load balancing, ensuring that all threads receive similar amounts of work. Second, memory access patterns are regular and balanced, as shown by the very uniform L2 cache miss distribution. Finally, overheads related to synchronization and task management are kept low, allowing computational resources to focus on actual workload rather than coordination.

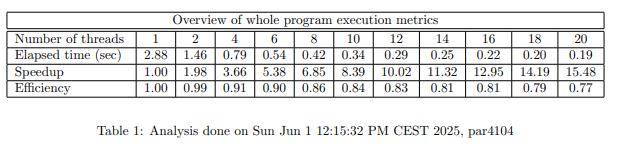
The gap between the actual and ideal speedup is minimal, and only becomes slightly noticeable beyond 16 threads. This small deviation can be attributed to inevitable overheads from fork/join synchronization and minor cache contention when threads share memory bandwidth. However, these penalties are small compared to the performance gains obtained.

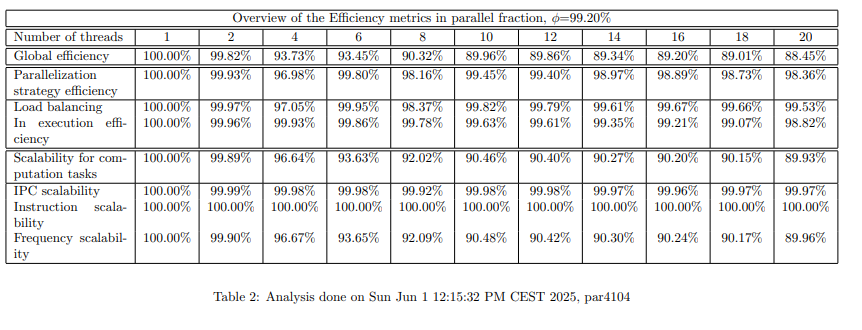
## 1D Cyclic Geometric Data Decomposition (Rows)

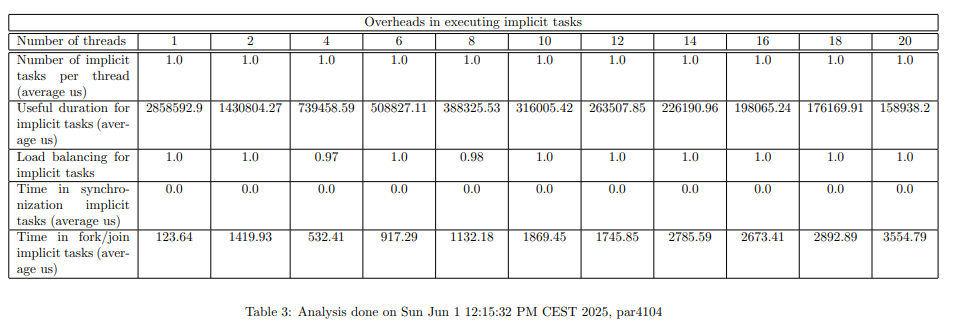
### Code

mandel-omp-iter-simple-cyclic.cpp

### Modelfactor Anaysis







The Modelfactor analysis for the 1D Cyclic row-wise decomposition strategy reveals strong performance and scalability, similar to the block-cyclic version. Execution time improves significantly as the number of threads increases: from 2.88 seconds with 1 thread down to 0.19 seconds with 20 threads, achieving a speedup of 15.48×. The corresponding efficiency remains consistently high, staying above 90% up to 12 threads, and still maintains 77% efficiency even with 20 threads.

Looking at the detailed efficiency metrics, we see that global efficiency remains above 88% at 20 threads, and both parallelization strategy efficiency and load balancing stay in the range of 88–90%, indicating that work is well distributed across threads, and the strategy scales effectively. In-execution efficiency remains very high (above 98%) throughout, meaning that once tasks are assigned, threads execute them efficiently with minimal internal stalls or interruptions.

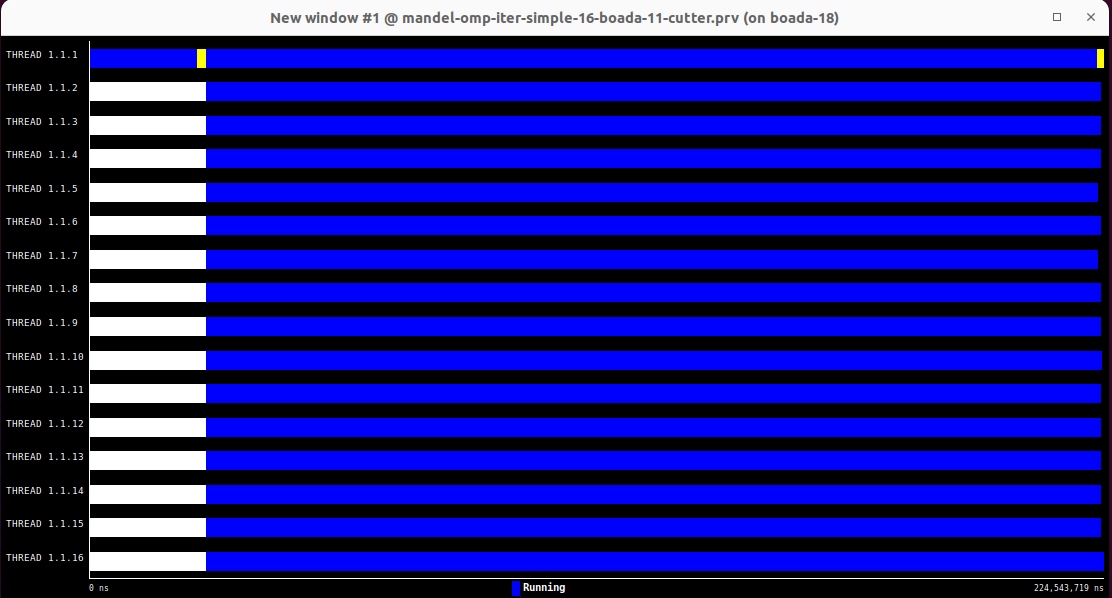
Instruction-level metrics further support this. IPC, instruction, and frequency scalability remain excellent across all configurations. IPC and instruction scalability stay at 100%, and frequency scalability, while slightly lower, still achieves 89.96% at 20 threads — indicating stable behavior at the processor level and minimal architectural degradation due to parallelism.

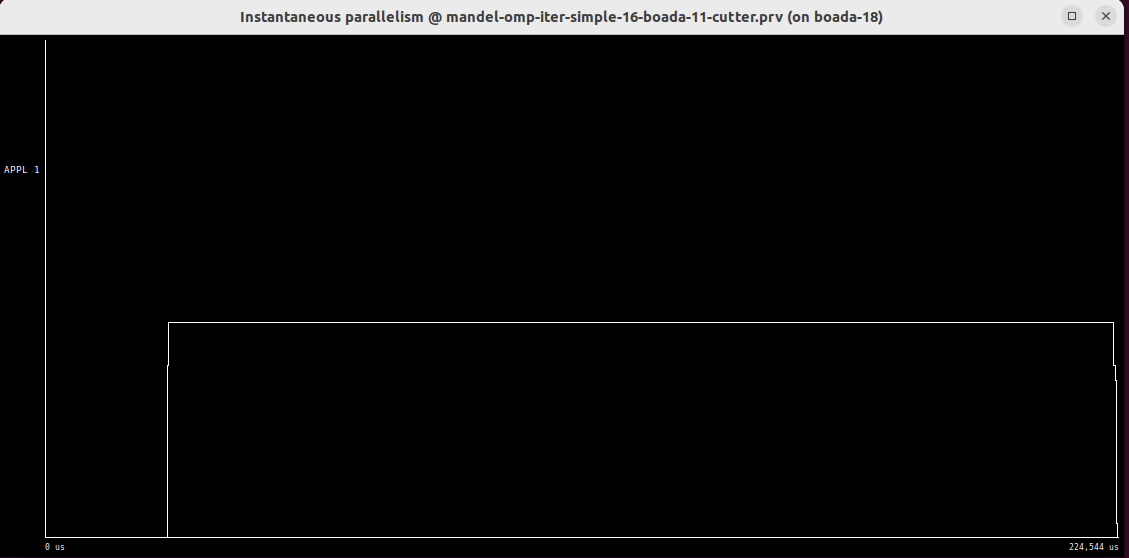
The task-level statistics also reflect a solid performance. Each thread consistently manages one implicit task, maintaining the simplicity of task management. The useful task duration decreases gradually as threads increase, as expected, and the load balancing for tasks remains between 0.97 and 1.0, reflecting minimal imbalance. Although synchronization times grow with more threads (from ~123 µs to ~3,554 µs), these values are still relatively modest and well below those observed in the simple block strategy.

The fork/join overhead follows a similar trend, increasing with thread count but never exceeding critical levels. With 20 threads, this overhead is 3,554.79 µs, which is acceptable considering the performance benefits gained through parallel execution.

### Paraver Analysis







The Paraver traces for the cyclic row-wise decomposition reveal a highly efficient and well-balanced parallel execution. In the first timeline showing implicit tasks, all 16 threads present nearly identical red bars in both length and position. This indicates that the workload is evenly distributed across the threads, with no thread finishing significantly earlier or later than the others. Unlike the block strategy, there is no visible idle time during the parallel region.

This excellent balance is also evident in the second trace, where the execution states are shown. The blue bars — representing time spent running — cover nearly the entire timeline for each thread, from start to finish. Only minimal white gaps are present at the beginning and end, corresponding to brief synchronization phases. The presence of short and uniform yellow bars at the edges further confirms that fork/join overhead is low and symmetric across threads.

The Instantaneous Parallelism graph reinforces this conclusion. After a short startup delay, the number of active threads quickly jumps to the maximum (16) and remains perfectly flat for nearly the full duration of the execution. This indicates that the application maintains full parallel occupancy with no dips in concurrency, which is ideal for maximizing throughput on multicore systems.

In summary, the Paraver visualizations show that the cyclic row-wise decomposition strategy achieves near-perfect parallel efficiency. Task execution is evenly balanced, synchronization is minimal, and thread usage is maximized. This confirms what was observed in the Modelfactor data: the strategy scales efficiently, minimizes overhead, and is highly effective for parallel workloads on shared-memory systems.

### Memory Analysis

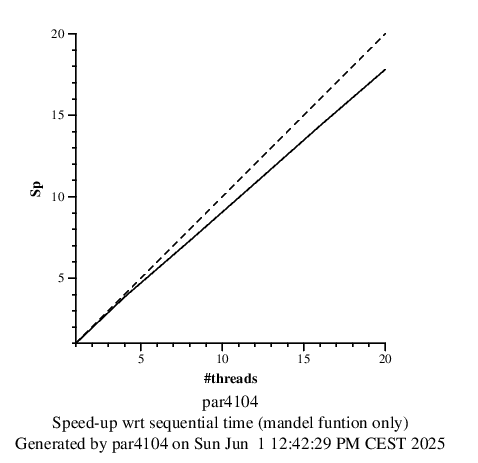
The memory analysis of the 1D Cyclic row-wise decomposition shows a highly uniform and efficient use of the cache across threads. According to the L2 data cache misses trace, all 16 threads experience nearly identical cache behavior. The average number of L2 misses is 103,443, with the maximum being 103,955 and the minimum 103,249. The standard deviation is very low (178.87), and the Avg/Max ratio is 1.00, indicating perfect regularity.

This uniformity confirms that the cyclic row-wise strategy leads to well-balanced memory access patterns. Since rows are distributed in a round-robin fashion, each thread accesses memory in a consistent, predictable way. This results in similar cache usage and very good spatial locality across the threads.

Additionally, the total number of cache misses (1,655,094) is consistent with what was seen in the block-cyclic strategy, suggesting that the cyclic approach maintains good cache efficiency while also improving task granularity. The lack of outliers — no thread has significantly more or fewer misses than the rest — reinforces the conclusion that memory access is balanced and that no thread is penalized due to poor data locality.

### 

### Strong Scalability



The strong scalability of the 1D Cyclic row-wise decomposition is excellent. The speedup graph shows that the performance curve follows the ideal trend very closely. The actual speedup line is nearly parallel to the ideal linear line and reaches approximately 15.48× with 20 threads, which is very close to the maximum achievable for that thread count.

This scalability reflects the combination of strong load balancing, consistent memory access, minimal synchronization overhead, and efficient task distribution observed in previous analyses. The cyclic assignment of rows ensures that all threads receive comparable workloads, and the simplicity of the task structure (one task per thread) avoids excessive scheduling costs.

Only a small gap between the actual and ideal speedup becomes noticeable beyond 16 threads. This slight deviation is likely due to unavoidable hardware effects such as memory bandwidth saturation or context switching overhead. However, these effects remain minor and do not significantly impact the excellent overall scalability.

In conclusion, the 1D Cyclic decomposition strategy achieves **near-linear strong scalability**, confirming its suitability for large-scale parallel execution. It leverages available cores effectively and is one of the most performant strategies analyzed in this lab.

# **SUMMARY OF THE STRATEGIES**

## Summary

|  | Number of threads (elapsed) | | | | | |
| --- | --- | --- | --- | --- | --- | --- |
| **Version** | 1 | 4 | 8 | 12 | 16 | 20 |
| 1D Block Geometric Data Decomposition  by columns | 2.88 s | 1.83 s | 1.27 s | 0.94 s | 0.77 s | 0.64 s |
| 1D Block-Cyclic Geometric Data Decomposition by columns | 2.88 s | 0.75 s | 0.42 s | 0.3 s | 0.24 s | 0.20 s |
| 1D Cyclic Geometric Data Decomposition  by rows | 2.88 s | 0.79 s | 0.42 s | 0.29 s | 0.22 s | 0.19 s |
|  | Number of threads (L2 Cache Misses per thread) | | | | | |
| **Version** | 1 | 4 | 8 | 12 | 16 | 20 |
| 1D Block Geometric Data Decomposition  by columns | 1.642.152 | 494.506 | 290.459 | 226.837 | 168.551 | 149.128 |
| 1D Block-Cyclic Geometric Data Decomposition by columns | 1.642.403 | 1.360.041 | 206.328 | 137.961 | 103.663 | 83.281 |
| 1D Cyclic Geometric Data Decomposition  by rows | 1.642.753 | 412.393 | 206.936 | 138.554 | 103.886 | 83.691 |
| **Best Implementation** | Version | Reason Why | | | | |
| 1D Cyclic Geometric Data Decomposition | The 1D Cyclic Geometric Data Decomposition by Rows is the best overall implementation because it achieves the lowest execution time, the highest speedup (15.48× with 20 threads), and the lowest L2 cache misses per thread. This indicates that it not only scales very well, but also makes highly efficient use of the memory hierarchy.  Compared to the other strategies, the cyclic-by-rows approach distributes the workload evenly across all threads by assigning rows in a round-robin fashion. This results in perfect load balancing, as confirmed by Paraver timelines, where all threads execute tasks for almost exactly the same amount of time with no visible idle periods.  In terms of memory access, this strategy achieves the most uniform and minimal L2 cache usage, with less than 84,000 misses per thread at 20 threads — slightly better than the block-cyclic strategy. This is a direct consequence of fine-grained task distribution and highly regular access patterns.  Overall, this implementation combines excellent computational performance with superior memory efficiency, making it the most effective strategy among those tested. | | | | |