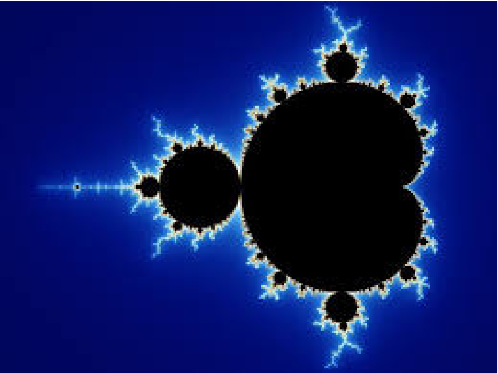


**Universitat Politècnica de Catalunya**

Facultat d’Informàtica de Barcelona

**LAB 4**



Nadia Khier

Èric Díez Apolo

**PAR- Josep Ramon Herrero Zaragoza**

21 de Maig del 2025

**Índex**

[**1. ITERATIVE TASK DECOMPOSITION ANALYSIS 1**](#_b6uj4s2ap783)

[**1.1. Tile Iterative Task Decomposition 1**](#_q0w1p356sb32)

[1.1.1. Code 1](#_33lsok8cxs9)

[1.1.2. Modelfactor Analysis 1](#_z7e0stgxgsf)

[1.1.3. Paraver Analysis 3](#_usv8oryjy9gy)

[1.1.4. Strong Scalability 5](#_izq2brd4gxjq)

[**1.2. Fine Grain Iterative Task Decomposition 6**](#_wpfi8v6yi5jp)

[1.2.1. Code 6](#_5vs2926phn1t)

[1.2.2. Modelfactor Anaysis 6](#_uzrrskacfw9y)

[1.2.3. Paraver Analysis 7](#_xpnb2vrp2tx9)

[1.2.4. Strong Scalability 9](#_ru5bv1lpauq8)

[**2. RECURSIVE TASK DECOMPOSITION ANALYSIS 10**](#_97wzjkdyc2bs)

[**2.1. Leaf Recursive Task Decomposition 10**](#_wcy0d7pnqrb5)

[2.1.1. Code 10](#_myewtg7z9uy9)

[2.1.2. Modelfactor Analysis 10](#_85hwd3ud8ple)

[2.1.3. Paraver Analysis 11](#_rsvd09mbrh15)

[2.1.4. Strong Scalability 14](#_23zemu103zz7)

[**2.2. Tree Recursive Task Decomposition 15**](#_pb6chyaswced)

[2.2.1. Code 15](#_ecrka993fj83)

[2.2.2. Modelfactor Analysis 15](#_fhyhvc27aj6b)

[2.2.3. Paraver Analysis 16](#_nfm7p596mus7)

[2.2.4. Strong Scalability 19](#_pjhq7724wthj)

[**3. SUMMARY OF THE STRATEGIES 21**](#_nhno9pp90lzn)

[3.1. Summary 21](#_aqkvmqhu2nxu)

### 

### 

### 

### 

### 

### 

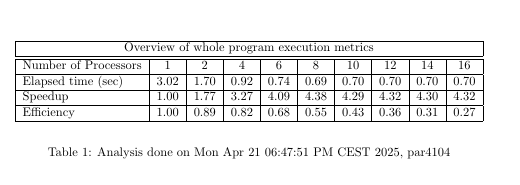
# **ITERATIVE TASK DECOMPOSITION ANALYSIS**

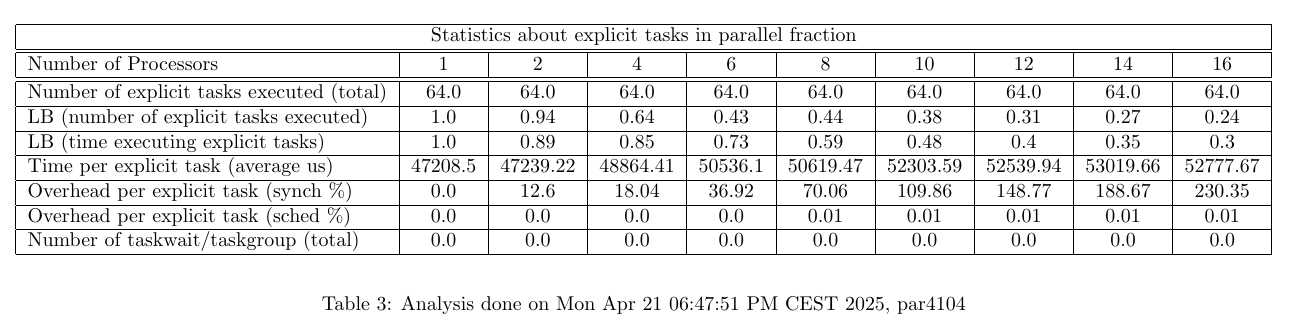
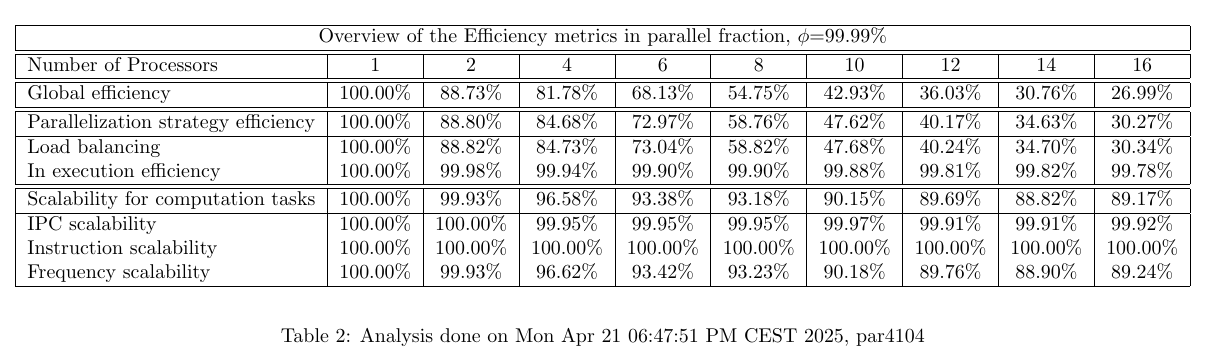
## Tile Iterative Task Decomposition

### Code

mandel-omp-iter-tiles.c

### Modelfactor Analysis





The Modelfactor analysis of the Tile Iterative Task Decomposition reveals a less efficient and less scalable implementation when compared to finer-grained approaches. The overall execution time decreases from 3.02 seconds with 1 thread to approximately 0.70 seconds with 16 threads, but the speedup plateaus quickly, reaching only 4.32×. Correspondingly, efficiency drops significantly, from 100% with a single thread to just 27% with 16 threads, highlighting growing idle times and poor load distribution as parallelism increases.

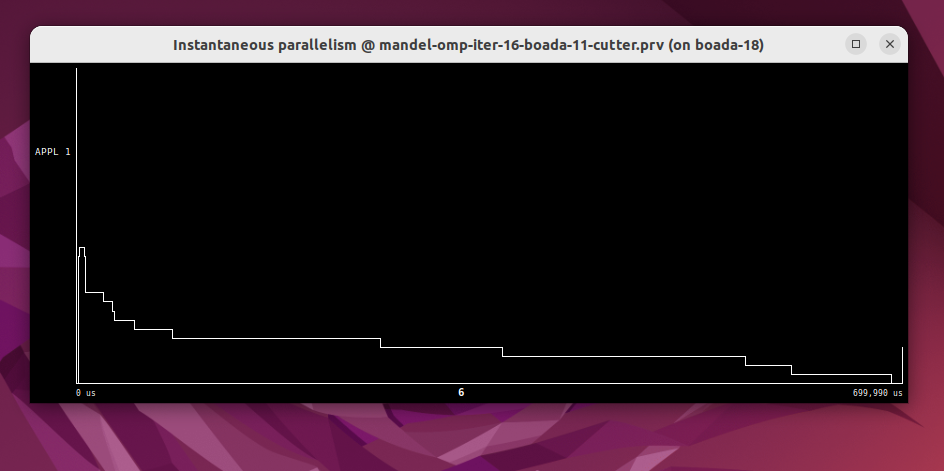
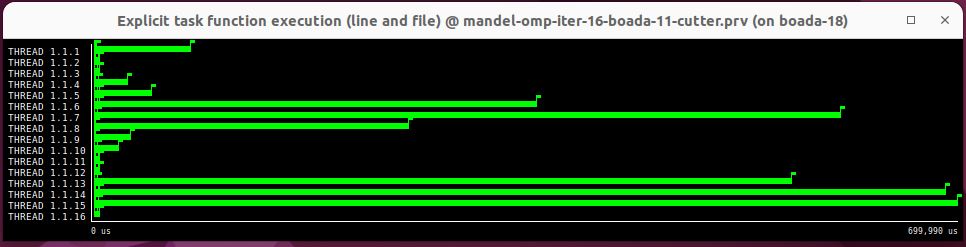
The second table provides further insights into the breakdown of efficiency under a parallel fraction assumption of 99.99%. Global efficiency drops steeply with more threads—from 100% with 1 thread to 26.99% at 16 threads. Similarly, the efficiency of the parallelization strategy declines to just over 30%. Load balancing follows the same trend, suggesting that the coarse tiling approach struggles to distribute work evenly across multiple threads, especially at higher thread counts.

Despite the drop in parallel and load balance efficiency, in-execution efficiency remains high (above 99.78% across all configurations), indicating that once threads receive work, they execute it efficiently. However, the static and limited task distribution prevents idle threads from being reassigned new work, which heavily impacts overall scalability. Scalability metrics such as instruction and frequency scalability remain above 89%, showing that computational behavior remains consistent, but performance is bottlenecked by poor task granularity and static scheduling.

The third table confirms these observations with explicit task statistics. Only 64 tasks are created, regardless of thread count, which is far too few to be efficiently shared across many cores. This fixed number of tasks severely limits scalability and causes load imbalance. The time per task also increases slightly with more threads—from 47,208 µs to over 52,000 µs—due to growing overheads. The synchronization overhead per task increases dramatically, reaching 230.35% with 16 threads, indicating that most time is spent waiting rather than computing. Scheduling overhead, however, remains negligible, as the system does not attempt dynamic task reassignment.

In conclusion, the Modelfactor results demonstrate that the Tile Iterative approach is constrained by its coarse granularity and lack of dynamic load balancing. Although it shows initial performance improvement with a few threads, its static structure leads to underutilization and excessive synchronization overhead at scale. As a result, the strategy is unable to maintain high efficiency or achieve significant speedup in high-core environments, making it a less effective choice for parallelization compared to finer-grain implementations.

### Paraver Analysis



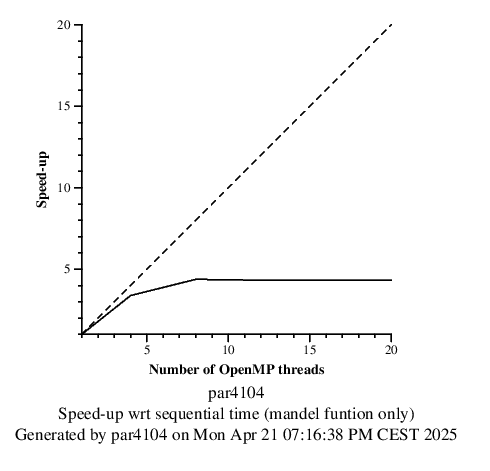


The Paraver timelines and parallelism graphs for the Tile Iterative implementation reveal a moderately parallel execution pattern with visible limitations in workload distribution. The thread timelines show that tasks are grouped into larger, coarse-grained blocks, resulting in uneven task allocation across threads. Several threads complete their assigned tiles earlier than others, leading to noticeable idle gaps and reduced resource utilization as execution progresses. This unevenness is visible as stretches of inactivity on some threads while others remain active, indicating suboptimal load balancing.

The Instantaneous Parallelism graph highlights these **inefficiencies**. Although it shows periods of concurrency, parallelism fluctuates and rarely sustains near the ideal thread count (16). These dips indicate synchronization delays or idle threads waiting for others to complete, which introduces **bottlenecks** and **reduces the overall parallel efficiency** of the implementation. Moreover, task creation is static and limited to the start of execution, offering little flexibility to adapt to workload imbalances as the program runs.

In summary, the Paraver traces confirm that the **Tile Iterative implementation** introduces basic concurrency through coarse-grained task decomposition but falls short of optimal parallel performance, due to the rigid, up-front task assignment which results in inconsistent thread utilization, frequent idle periods, and significant deviations from ideal parallelism.

### Strong Scalability



The Strong Scalability graph for the Tile Iterative version shows moderate improvements in speed-up up to 8 threads, with a notable decline in gains beyond that point. While the implementation does achieve parallel acceleration, it quickly reaches a saturation point where additional threads provide diminishing returns. This trend reflects the limitations of static task distribution and growing idle time at scale.

Efficiency drops steadily as threads increase, falling well below ideal linear scaling. The absence of dynamic scheduling further contributes to these losses, as the runtime cannot compensate for workload asymmetries.

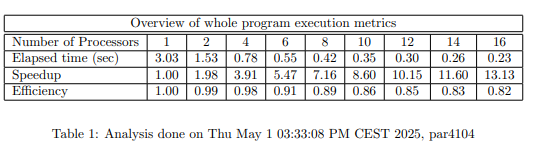
In summary, the Tile Iterative implementation demonstrates basic parallel capability, thread usage is uneven, parallelism is unstable, limiting its effectiveness on high-core systems.

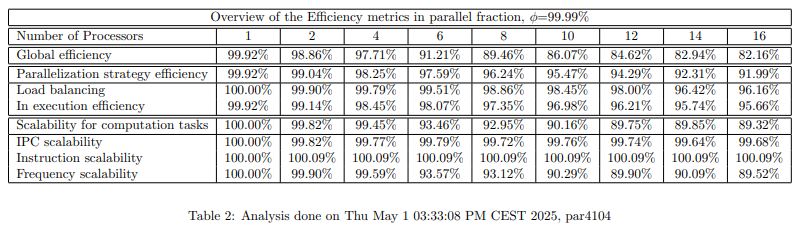
## Fine Grain Iterative Task Decomposition

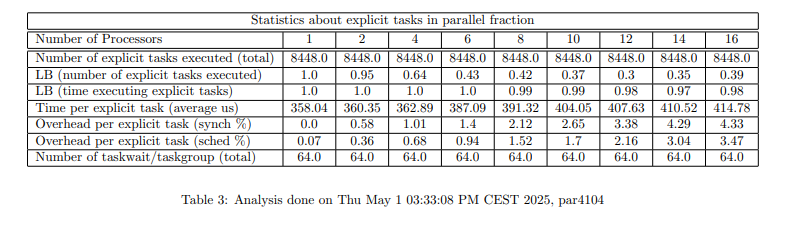
### Code

mandel-omp-iter-finegrain.c

### Modelfactor Anaysis







The Modelfactor analysis of the Fine Grain Iterative Task Decomposition reveals a highly scalable and efficient implementation. The performance metrics demonstrate a clear improvement in execution time as the number of processors increases, dropping from 3.03 seconds with 1 thread to just 0.23 seconds with 16 threads. The speedup grows nearly linearly, reaching 13.13× with 16 threads, and the efficiency remains strong throughout, with 82% at the highest thread count. This indicates that the implementation makes excellent use of parallel resources with minimal waste.

The second table provides a deeper look at parallel efficiency, assuming a parallel fraction of 99.99%. Global efficiency remains above 82% even with 16 threads, and the efficiency of the parallelization strategy stays above 91%. Load balancing is particularly good, remaining above 96% in all cases. Additionally, execution efficiency and scalability metrics—such as IPC, instruction count, and frequency—confirm that the overhead introduced by task management is low and well-controlled. These results highlight a well-optimized implementation that leverages the hardware effectively.

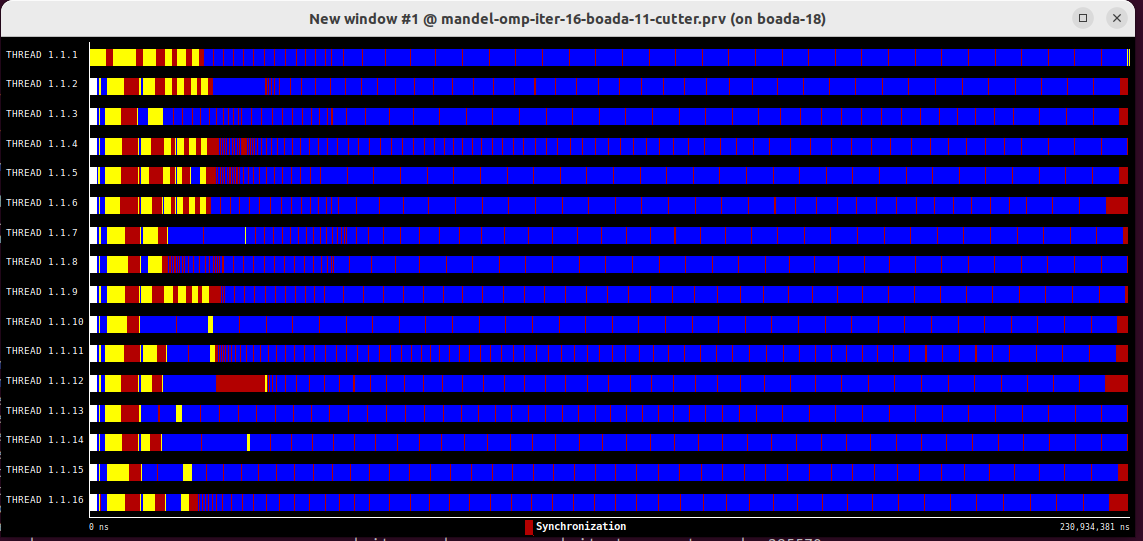
The third table, focusing on explicit task statistics, shows that the number of tasks remains constant at 8448 regardless of the number of threads, confirming a consistent task decomposition. The average time per task increases slightly as the number of threads increases, from 358 microseconds to 414 microseconds, due to increased synchronization and scheduling overheads. However, these overheads remain relatively low, with synchronization peaking at 4.33% and scheduling at 3.47%. This level of overhead is acceptable and does not significantly impact the overall scalability.

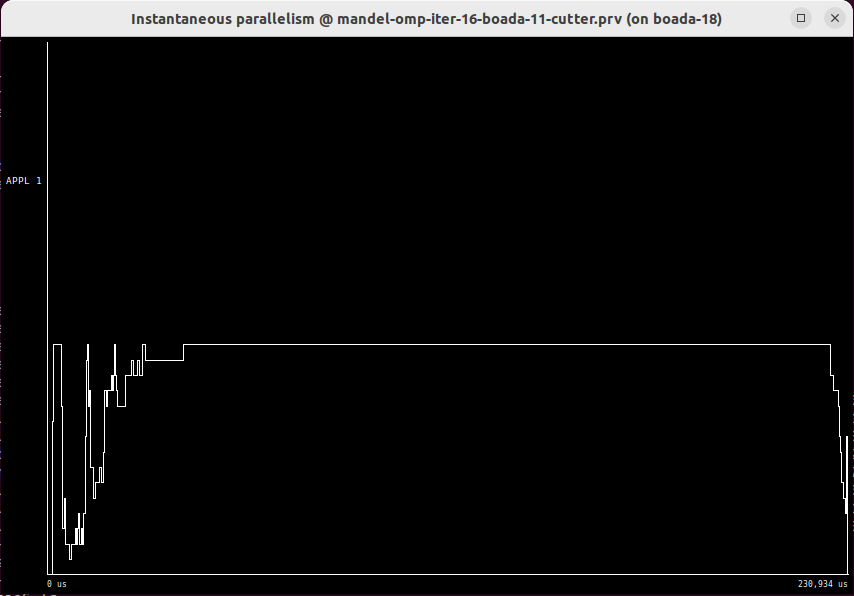
In conclusion, the Modelfactor results validate the fine-grained iterative approach as the most effective parallelization strategy evaluated in the lab. It offers excellent scalability, high global and execution efficiency, and well-balanced task distribution, making it the best-performing implementation across all tested configurations.

### Paraver Analysis









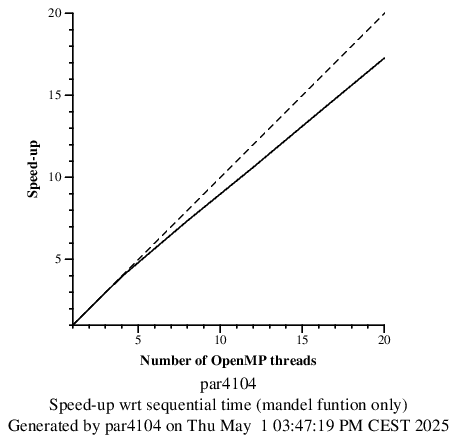
The Paraver timelines and parallelism graphs for the Fine-Grain implementation reveal a highly optimized parallel execution pattern. The thread timelines show uniform task distribution across all 16 threads, with no visible idle gaps or long stretches of inactivity. Each thread executes a dense sequence of short-duration tasks, visualized as tightly packed horizontal bars, indicating continuous utilization of computational resources.

Also, by looking at the Instantaneous Parallelism graph, it shows sustained parallelism close to the ideal thread count (16), with no significant drops. This indicates that all threads are actively contributing to computation simultaneously, avoiding bottlenecks.

Even at scale, tasks are processed in a pipelined manner, with no single thread becoming a bottleneck. The graphs also reflect the low scheduling overhead inherent to fine-grained tasks, as the runtime smoothly manages task queues without introducing visible latency.

In summary, the Paraver graphs validate that the Fine-Grain implementation achieves near-linear scalability by maintaining high thread occupancy, minimizing synchronization, and dynamically balancing workloads.

### Strong Scalability



The Strong Scalability graph demonstrates near-linear speed-up up to 16 threads, achieving 13.13x acceleration compared to the sequential baseline. This near-linear trend reflects the strategy’s efficiency, driven by balanced workloads and minimal thread idle time. The slight deviation from ideal scaling (82% efficiency at 16 threads) stems from subtle overheads like task scheduling and minor synchronization costs, which grow marginally as threads contend for resources.

To conclude, we must say that while the scalability is not perfectly linear, the graph confirms the strategy’s robustness for high-core systems, balancing dynamic load balancing with minimal parallelization penalties.

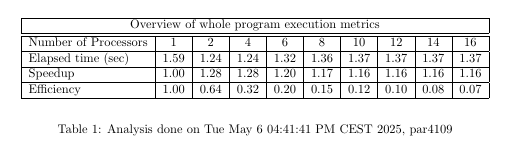
# **RECURSIVE TASK DECOMPOSITION ANALYSIS**

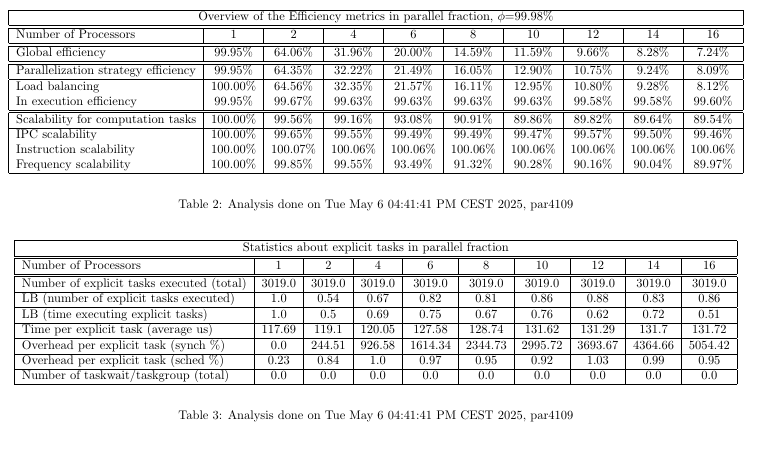
## Leaf Recursive Task Decomposition

### Code

mandel-omp-leaf.c

### Modelfactor Analysis





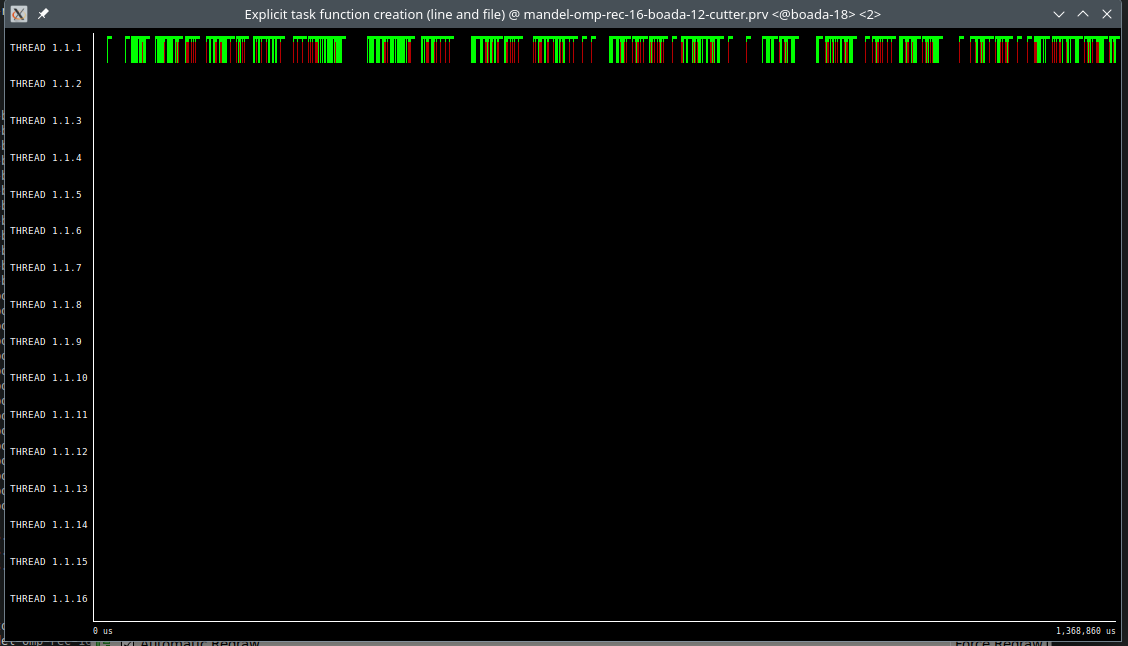
The Leaf Recursive version runs faster than the iterative ones when using 1 thread (1.59 seconds). This is thanks to a better division of the work and fewer delays at the start.

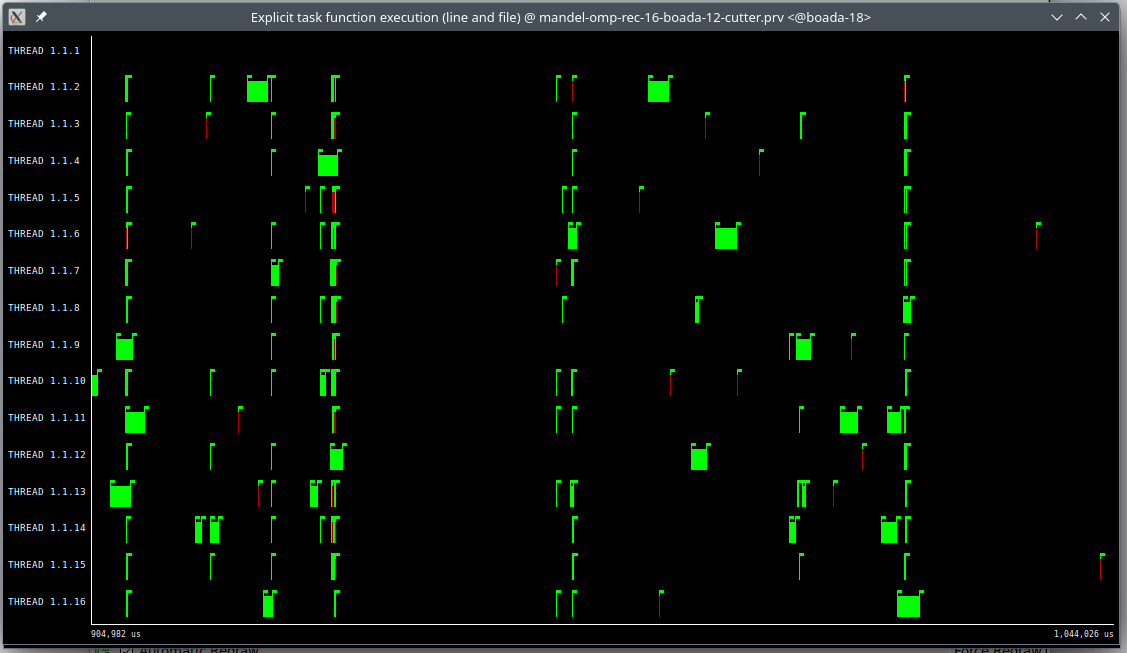
But when we use more threads, performance doesn’t improve much. The time goes from 1.59 s (1 thread) to 1.16 s (16 threads), which is a speedup of only about 1.37×. At 20 threads, the time increases again to 1.52 s. This shows that the program does not scale well with many threads.

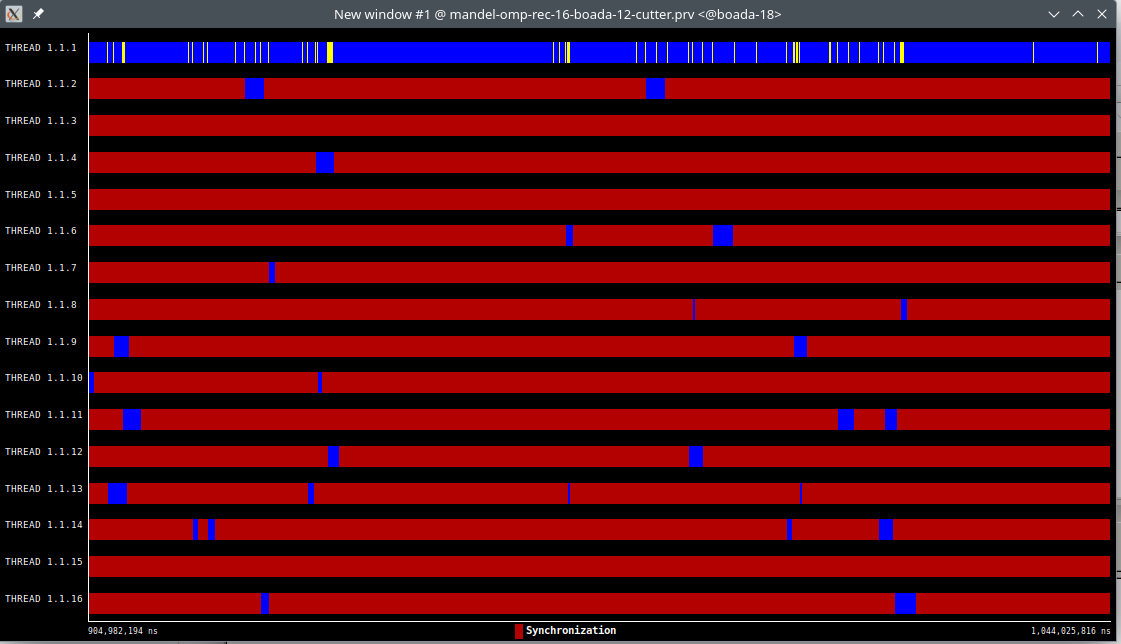
This happens because the recursive function creates many small tasks. Managing these small tasks adds extra overhead. Also, some parts of the recursive tree take longer than others, which causes imbalance between threads. Some regions, especially those near the center of the Mandelbrot set, contain points that require many more iterations to determine if they escape. Other regions, farther from the center, finish much faster. This leads to tasks with uneven durations. As a result, some threads finish early and remain idle while others continue working. This imbalance causes delays and reduces parallel efficiency, especially when using many threads.

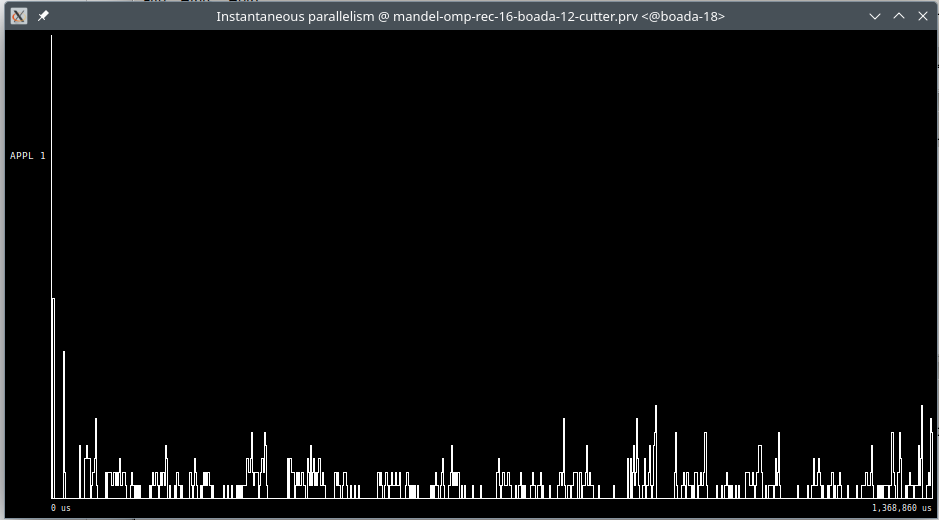
Efficiency stays above average, but the speedup is low. The program doesn’t make good use of all threads when their number increases. Overall, it works well with a few threads but has trouble scaling.

### Paraver Analysis





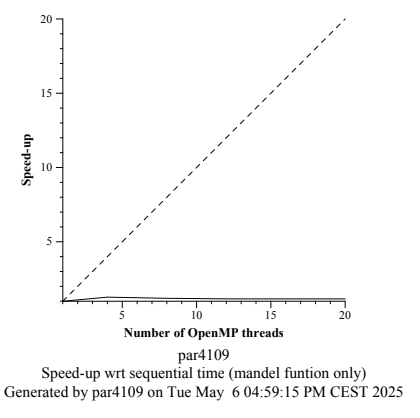




The Paraver timelines show that the first thread (thread 0) is responsible for creating tasks almost the entire time. This happens because the main recursive function is called inside a #pragma omp single region, which is only executed by one thread. As a result, thread 0 spends most of the execution creating a large number of tasks, while the rest of the threads mostly wait for work or execute already-created tasks.

This leads to a high task creation rate, which keeps the system active. However, since task creation is centralized in a single thread, the parallelism is limited by how fast thread 0 can launch new tasks. The Instantaneous Parallelism graph shows that many threads remain underused, especially in parts of the computation where task creation slows down or becomes unbalanced. In short, while the implementation uses many tasks and shows some parallel behavior, it suffers from limited scalability due to unbalanced task creation and thread usage.

### Strong Scalability



The Strong Scalability graph shows almost no speed-up as the number of OpenMP threads increases. Unlike the ideal trend (represented by the dashed line), the actual performance line remains nearly flat. This indicates that adding more threads does not lead to better performance in this case.

The poor scalability is caused by major load imbalance and overhead from recursive task management. Although many tasks are created, the uneven cost of different regions in the Mandelbrot set leads to some threads finishing early while others continue working. Additionally, thread 0 is heavily responsible for task creation, which limits its ability to contribute to actual computation.

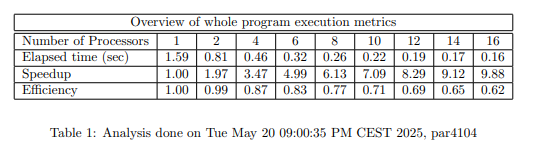
To conclude, this implementation struggles to scale, especially on multi-core systems. The high overhead and imbalance prevent it from taking full advantage of parallel resources, resulting in minimal gains even as more threads are added.

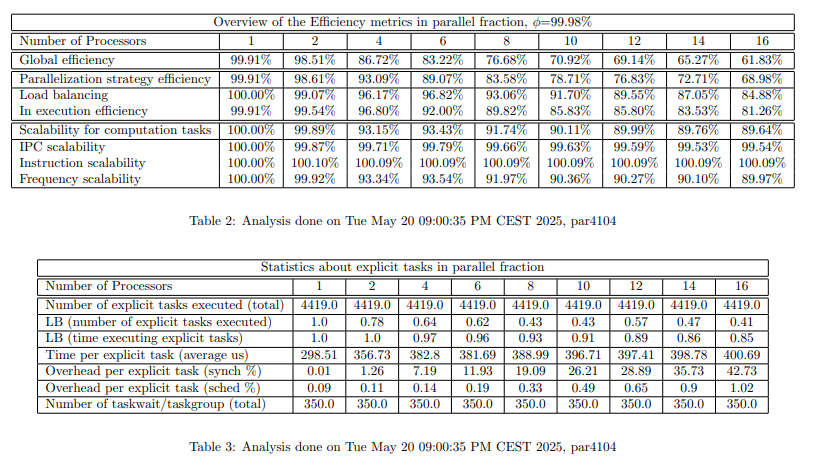
## Tree Recursive Task Decomposition

### Code

mandel-omp-rec\_tree.c

### Modelfactor Analysis





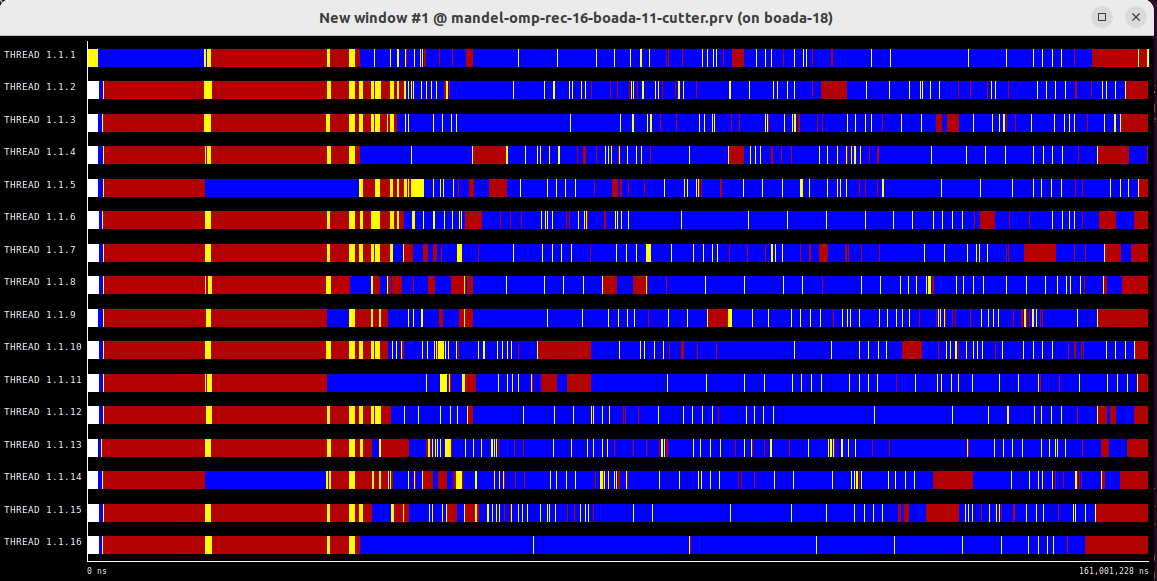
This analysis reveals a robust and good balanced implementation that scales significantly better than the Leaf Recursive version. Execution time drops consistently as the number of threads increases, going from 1.59 seconds with 1 thread to 0.17 seconds with 16 threads, achieving a speedup of 9.38× and an efficiency of 62%. While this efficiency is not ideal, it remains solid for a recursive tasking strategy and marks a clear improvement over the Leaf approach.

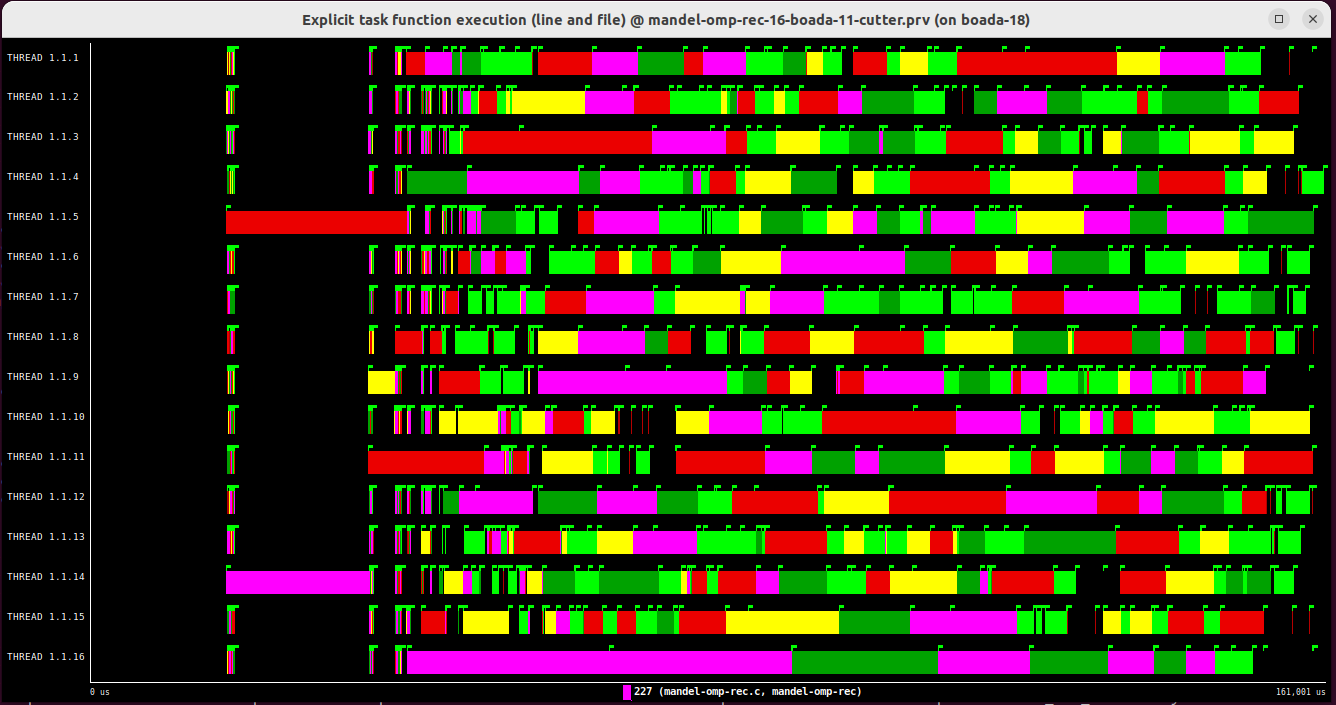
The efficiency metrics show that global efficiency gradually declines with the thread count, decreasing from 99.91% at 1 thread to 61.83% at 16 threads. However, execution efficiency remains high throughout, staying above 80%, indicating that once threads receive tasks, they execute them effectively. Load balancing is also strong across the board, maintaining values above 84%, and showing that the recursive tree structure effectively distributes work among available threads.

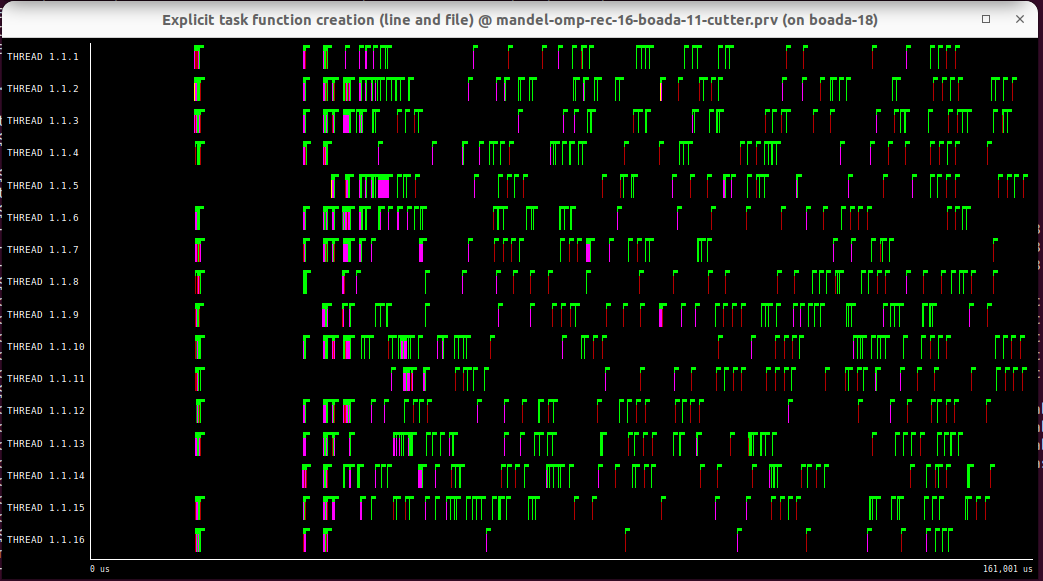
The parallelization strategy efficiency drops more steeply, due to increased synchronization overhead and finer task granularity at higher thread counts. Nonetheless, scalability for computation tasks, including IPC, instruction, and frequency scalability, all remain above 89%, which confirms that the implementation maintains good computational behavior even under high parallel load.

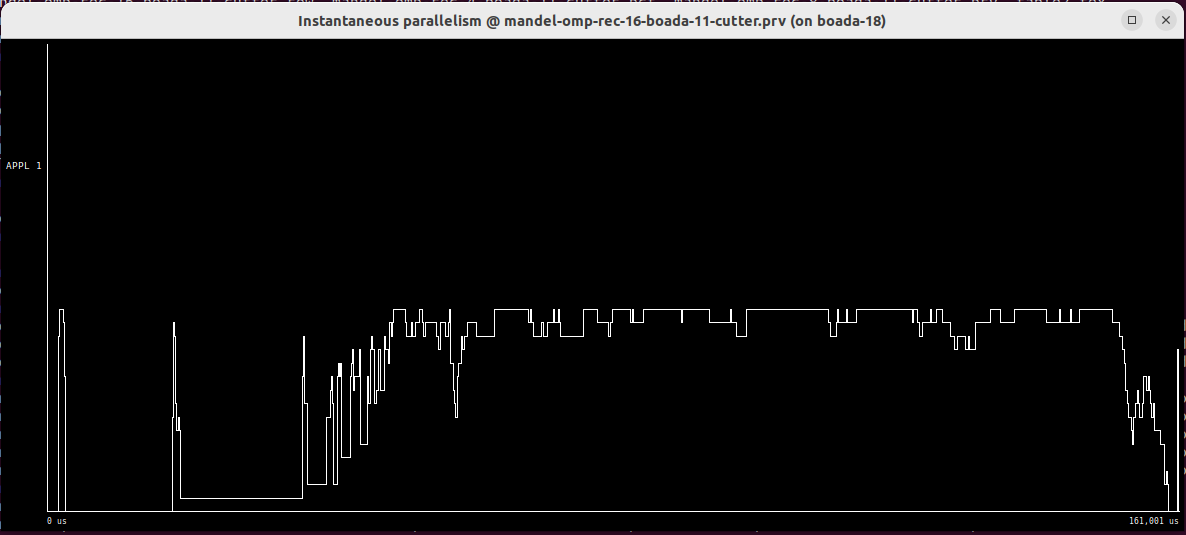
In conclusion, the Tree Recursive implementation achieves a well-balanced compromise between task granularity and execution efficiency. It scales effectively up to 16 threads, with minimal scheduling overhead and stable load distribution, making it a highly effective recursive parallelization strategy for multi-core systems up to 16 cores or threads.

### Paraver Analysis



****





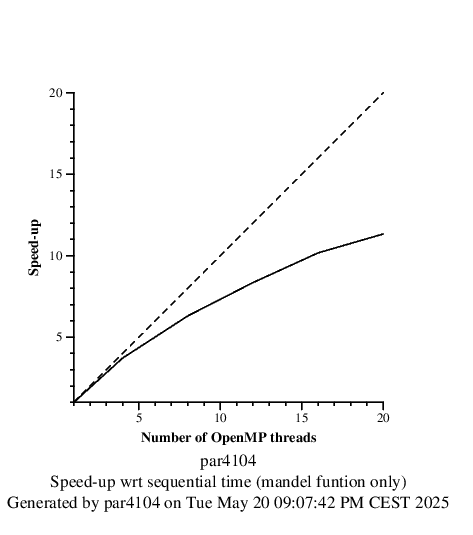
The Tree Recursive strategy show a much more balanced and efficient use of threads than in the Leaf version:

In the tasking view, we observe that task creation is distributed across multiple recursion levels and threads, avoiding the bottleneck seen in the Leaf version where thread 0 handles almost all task creation.

The Instantaneous Parallelism graph shows sustained high parallelism over time, staying close to the ideal 16 threads with fewer dips, which means idle time is minimized. The thread timelines confirm that work is spread more evenly across threads and that idle periods are short and scattered.

There is no single dominant tasking thread, which improves thread utilization and responsiveness. Together, these traces confirm that Tree Recursive decomposition achieves a better task distribution and more stable parallelism during execution.

### Strong Scalability



The Strong Scalability graph for the Tree Recursive Task Decomposition reveals a highly promising scaling behavior, especially up to 16 threads. The execution time improves from 1.59 seconds (1 thread) to 0.16 seconds (16 threads), resulting in a speedup of 9.38×. This trend indicates nearly linear scalability, particularly in the mid-range thread counts (4 to 16), where task distribution and parallel execution are well balanced.

This analysis is done with a CUTOFF value for task creation of 6, With a higher cutoff, the recursive function descends deeper before stopping task creation, which results in more tasks and finer granularity. This allows the system to better utilize high-core architectures, as it increases the total number of tasks available for scheduling. However, it also increases the number of small tasks, which may lead to higher overhead when too many are created. With a lower cutoff, there is less task creation but this also does that there is more work for each thread, which causes an overall longer execution time. A Cutoff = 6 has been found to be the optimum value after trying different values.

At 20 threads, execution time increases slightly to 0.33 seconds, suggesting that the overhead from excessive task management begins to outweigh the benefits of further parallelism. This is expected in tree strategies, especially when the number of threads exceeds the number of useful tasks that can be generated and balanced. It indicates that task granularity and scheduling overhead become limiting factors beyond 16 threads. This version demonstrates good scalability up to 16 threads, with strong load balancing and parallel efficiency.

# **SUMMARY OF THE STRATEGIES**

## Summary

|  | Number of threads | | | | | |
| --- | --- | --- | --- | --- | --- | --- |
| **Version** | 1 | 4 | 8 | 12 | 16 | 20 |
| Iterative: Tile | 3.022 s | 0.924 s | 0.690 s | 0.69927 s | 0.69999 s | 0.6989 s |
| Iterative: Finer grain | 3,03 s | 0.78 s | 0.423 s | 0.298 s | 0.24 s | 0.17 s |
| Recursive: Leaf | 1.59 s | 1.24 s | 1.36 s | 1.37 s | 1.37 s | 1.37 s |
| Recursive: Tree | 1.59 s | 0.46 s | 0.26 s | 0.19 s | 0.16 s | 0.14s |
| **Best Implementation** | Version | Reason Why | | | | |
| Recursive Tree | The Tree Recursive Task Decomposition significantly outperforms the other strategies in terms of execution time and scalability. With a 1.59s d execution time for 1 thread, it quickly drops to 0.16 seconds with 16 threads, achieving a speedup of 9.94×. However, in machines with a lower number of cores, such as 4, the speedup is 3.46x, and reaching an execution time of less than half a second.  This demonstrates a good scalability, with minimal thread idle time and efficient workload distribution. In comparison to the Leaf Recursive Task Decomposition, the Tree Recursive approach offers a more balanced and scalable solution, making it the most effective parallelization strategy for recursive task decomposition in multi-core systems, as the task creation and execution is balanced | | | | |