CS542200 Parallel Programming
Homework 2: Mandelbrot Set

Kai-Yuan Jeng 鄭凱元
113062529
kaiyuanjeng@gapp.nthu.edu.tw

1 Implementation

This section details the parallel implementations of the Mandelbrot Set algorithm. We describe the task partitioning strategies and optimization techniques employed to maximize performance and ensure scalability across both shared-memory and distributed-memory architectures.

1.1 Overview: Parallelization Strategies

Two distinct parallel implementations were developed to target different computational environments:

1. **hw2a (Pthread):** A shared-memory parallelization for multi-core processors on a single node.
2. **hw2b (MPI+OpenMP):** A hybrid parallelization combining distributed-memory (MPI) for multi-node execution with shared-memory (OpenMP) for intra-node parallelism.

1.2 hw2a: Centralized Dynamic Task Queue

To address the highly non-uniform computational load of the Mandelbrot set, the Pthread implementation utilizes a **fine-grained dynamic task queue**. This strategy is crucial for achieving high parallel efficiency on a multi-core system.

Task Partitioning Strategy. Instead of statically partitioning image rows, we implement a centralized task dispatcher. A global counter, `next_row`, protected by a `pthread_mutex`, serves as the task queue.

```
#define CHUNK_SIZE 1 // for maximal load balancing

pthread_mutex_t task_mutex = PTHREAD_MUTEX_INITIALIZER;
int next_row = 0;

void *local_mandelbrot(void *argv) {
    while (1) {
        // Each thread atomically acquires its next unit of work
        pthread_mutex_lock(&task_mutex);
        const int my_start_row = next_row;
        next_row += CHUNK_SIZE;
        pthread_mutex_unlock(&task_mutex);

        if (my_start_row >= height) break; // No more rows to compute

        // ... compute the assigned rows ...
    }
}
```

By setting CHUNK_SIZE to 1, we create a "work-dealing" model where each thread processes one row at a time. This ensures that threads completing tasks in low-computation regions can immediately acquire new work, minimizing idle time and maximizing core utilization.

1.3 hw2b: Two-Level Hierarchical Load Balancing

The hybrid implementation employs a **two-level hierarchical load-balancing strategy** to manage complexity across nodes and cores.

MPI Layer: Inter-Process Static Cyclic Distribution. At the inter-process level, rows are distributed cyclically among MPI processes. This coarse-grained strategy ensures that each process receives a structurally similar subset of the computational domain, mixing both high-cost and low-cost regions, thus preventing any single process from being burdened with a contiguous, computationally expensive fractal boundary.

```
// In each process, OpenMP threads work on a cyclically assigned subset of rows
#pragma omp for schedule(dynamic, 1) nowait
for (int local_j = 0; local_j < my_count; ++local_j) {
    // Between each MPI process, Process `my_rank` computes rows: rank,
    ↪  rank+numtasks, rank+2*numtasks, ...
    const int j = my_rank + local_j * numtasks; // Cyclic (interleaved) global row
    ↪  index

    // ... compute row j using OpenMP threads ...
}
```

OpenMP Layer: Intra-Process Dynamic Scheduling. Within each MPI process, the `#pragma omp for schedule(dynamic, 1)` directive distributes the assigned rows dynamically among the available threads, providing a second, fine-grained level of load balancing to smooth out any remaining workload variations.

1.4 Core Optimization Techniques

To further enhance performance, several key optimization techniques were applied to both implementations.

SIMD Vectorization with SSE2. The core Mandelbrot iteration loop is vectorized using **SSE2 intrinsics**. By packing two 64-bit double-precision numbers into a single 128-bit SSE register, this technique achieves data-level parallelism, processing two pixels simultaneously. This can be viewed as an **intrinsic loop unroll of factor 2**, which ideally provides a 2x throughput gain per instruction. Furthermore, an early-exit condition using _mm_movemask_pd terminates iterations for a vector once all its pixels have diverged, avoiding unnecessary calculations.

Manual Loop Unrolling. In addition to vectorization, the outer loop over image width was manually unrolled. Empirical testing showed an unroll factor of 8 delivered optimal performance by reducing loop overhead and improving instruction scheduling. Higher factors led to performance degradation, likely due to increased register pressure and register spilling, and also instruction cache misses. Both of our implemented codes demonstrate this approach by processing 8 pixels (four __m128d vectors) per outer loop iteration.

Data Gathering and Reordering in hw2b. A critical challenge in the hybrid model is reassembling the final image from the cyclically distributed results. This is solved with a two-phase process on rank 0:

1. **Gather:** MPI_Gatherv collects the computed row segments from all processes into a single temporary buffer.
2. **Reorder:** The data is then reordered from the temporary buffer into the final image buffer, mapping the cyclically computed data back to its correct linear row sequence using memcpy.

This reordering step, while introducing a one-time communication and data movement overhead, is essential for correctness in the distributed-memory setting. The code for this logic is shown below.

```
if (my_rank == 0) {
    // ... setup recvcounts and displacements for MPI_Gatherv ...

    int *temp_image = (int *)malloc(width * height * sizeof(int));
    MPI_Gatherv(..., temp_image, ...); // Phase 1: Gather

    // Phase 2: Reorder from cyclic layout to linear layout
    for (int rank = 0; rank < numtasks; ++rank) {
        // ... calculate count and offset for the current rank ...
        for (int local_j = 0; local_j < count; ++local_j) {
            const int global_row = rank + local_j * numtasks;
            memcpy(&global_image[global_row * width],
                   &temp_image[offset + local_j * width],
                   width * sizeof(int));
        }
    }
    // ... free memory ...
}
```

1.5 Profiling and Measurement Infrastructure

To enable rigorous performance analysis, a profiling infrastructure, enabled via the `-DPROFILING` compiler flag which defines the `PROFILING` macro, was integrated into both programs.

- **Timing:** `gettimeofday` (in `hw2a`) and `MPI_Wtime` (in `hw2b`) are used to measure computation, synchronization, and communication timings.
- **Statistics:** `MPI_Reduce` aggregates timing data from all processes, allowing for the calculation of load imbalance and overheads across nodes.
- **Visualization:** NVIDIA's NVTX markers are used to annotate code regions, enabling detailed visual performance analysis in tools like Nsight Systems.

2 Experiment and Analysis

This section presents a systematic evaluation of our parallel implementations. The primary objective is to quantify the performance impact of key optimization choices and to measure the overall scalability of the final programs. Our analysis proceeds in three stages:

1. **Micro-optimization Tuning:** We first determine the optimal **manual loop unrolling factor**. Given the compute-intensive nature of the Mandelbrot iteration, this factor significantly influences instruction-level parallelism and cache performance.
2. **Load Balancing Strategy Evaluation:** Second, we compare various **load balancing strategies**. The computational cost per pixel varies dramatically across the Mandelbrot set, making effective load balancing critical. Performance in a parallel system is limited by the execution time of the slowest thread (the long pole), which is directly governed by workload distribution.
3. **Strong Scalability Analysis:** Finally, using the optimal configuration derived from the first two stages, we conduct **strong scalability experiments**. We measure the speedup of both **hw2a** and **hw2b** as the number of cores increases, comparing the results against the ideal linear speedup. This allows us to quantify the impact of parallel overheads, such as thread synchronization (Pthread/OpenMP) and inter-process communication (MPI).

2.1 Methodology

System Specification. All experiments were conducted on the **Apollo/Origo** cluster provided for the course. Each compute node is equipped with a dual-socket **Intel Xeon X5670** processor, providing a total of **12 physical cores** (2 sockets $\times$ 6 cores/socket) running at a base frequency of

2.93 GHz. Each core has access to a 32 KiB L1d cache, a 32 KiB L1i cache, and a 256 KiB L2 cache. The two sockets share a 24 MiB L3 cache. The system architecture features 2 NUMA nodes, with 6 cores per node. Our `hw2a` (Pthread) experiments were confined to a single node, utilizing up to all 12 cores, while `hw2b` (MPI+OpenMP) experiments spanned multiple nodes. The operating system is a 64-bit x86_64 Linux distribution.

Performance Metrics and Measurement. Performance was measured using a custom profiling infrastructure (see also Section 1.5), enabled via a `-DPROFILING` compile-time flag. The code was instrumented with high-precision timers (`gettimeofday` for Pthread and `MPI_Wtime` for MPI) to capture wall-clock time. The primary metrics used in our analysis are defined as follows:

- **Total Execution Time (T_p):** The wall-clock time from the start of the parallel computation to the completion of the final I/O operation. In a multi-process/thread environment, this is determined by the last process/thread to finish.
- **Speedup (S_p):** The ratio of the sequential execution time (T_1) (or the single-node single thread configuration in our setting) to the parallel execution time (T_p) on p cores. It is calculated as:
$$S_p = \frac{T_1}{T_p}$$
- **Load Imbalance (I):** A measure of the workload distribution unevenness among parallel workers (threads or processes). It is calculated as the normalized difference between the maximum and minimum computation times:
$$I = \frac{T_{max\_compute} - T_{min\_compute}}{T_{max\_compute}} \times 100\%$$
where $T_{max\_compute}$ and $T_{min\_compute}$ are the maximum and minimum wall-clock times spent in the core computation loop across all workers.

Test Case Description. To ensure the robustness of our findings, all experiments were conducted on a comprehensive test suite comprising all **68 public test cases** provided for the assignment. For plots showing aggregate performance, the total execution time is the sum of times across all 68 cases, and the reported load imbalance is the average imbalance value across all cases.

Parallel Configurations. Unless stated otherwise, the default parallel configurations for the experiments are as follows:

- **hw2a (Pthread):** Deployed on a single compute node, utilizing all 12 available cores. The configuration is 1 process with 12 threads, launched via:
```
srun -n1 -c12 ./hw2a <args...>
```
- **hw2b (MPI+OpenMP):** Deployed across 4 compute nodes, utilizing a total of 48 cores. The configuration is 4 processes (one per node), with each process spawning 12 threads to saturate the node's cores. This is launched via:
```
srun -n4 -c12 ./hw2b <args...>
```

2.2 Analysis of Micro-optimizations: Loop Unrolling

The first experiment evaluates the impact of manual loop unrolling on performance. Given that the Mandelbrot computation is entirely data-parallel and bound by floating-point operations, maximizing instruction-level parallelism (ILP) is critical. Our implementation combines SSE2 vectorization with loop unrolling to process multiple pixels within a single loop iteration, aiming to saturate the CPU's execution units and hide instruction latency.

In this experiment, the "unroll factor" corresponds to the number of pixels processed per outer loop iteration. A factor of 1 represents the scalar (non-vectorized) baseline. A factor of 2 represents our basic SSE2 implementation, which processes 2 pixels simultaneously using one `__m128d` vector. A factor of N therefore processes $N/2$ such vectors. The results for both `hw2a` and `hw2b`, executed with their default parallel configurations, are presented in Figure 1.

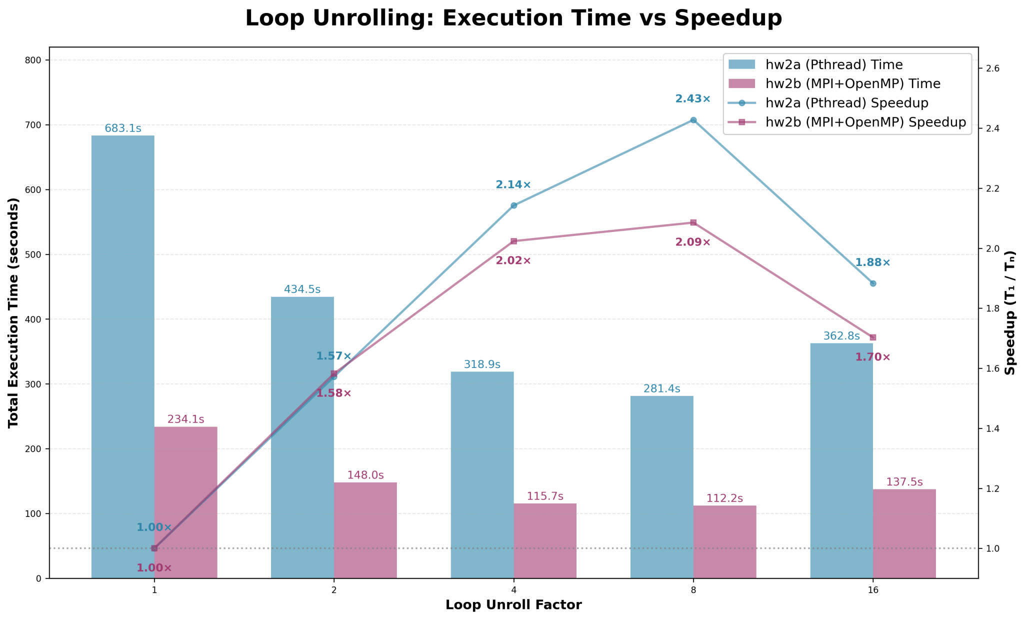

Figure 1: Performance impact of different loop unrolling factors on `hw2a` (12 threads) and `hw2b` (4 processes, 48 threads total). The baseline (factor 1) is the non-vectorized version. Speedup is calculated relative to each version's own baseline.

Results and Discussion. The results clearly demonstrate that loop unrolling yields significant performance gains, but only up to an optimal point.

- **Optimal Unroll Factor:** As shown in Figure 1, performance for both implementations improves substantially as the unroll factor increases from 1 to 8. **hw2a** achieves its peak relative speedup of **2.43x** at a factor of 8, reducing execution time from 683.1s to 281.4s. Similarly, **hw2b** also peaks at a factor of 8, reaching a speedup of **2.09x**. This confirms that unrolling helps the compiler generate more efficient machine code by reducing loop control overhead and improving instruction scheduling.
- **Performance Degradation:** Beyond the optimal point, performance degrades sharply. At an unroll factor of 16, the speedup for `hw2a` drops to 1.88x, a level worse than that of factor 4. This behavior is characteristic of excessive unrolling, where the benefits are outweighed by negative secondary effects, primarily increased **register pressure**. The large number of live variables in an unrolled loop body exceeds the available physical registers, forcing the compiler to generate "spill code" to move data back and forth to memory, which incurs significant overhead. Reduced instruction cache locality could also be a contributing factor.
- **Speedup Gap:** An interesting observation is the consistent gap in relative speedup between the two versions. `hw2a` benefits more from unrolling (2.43x peak) than `hw2b` (2.09x peak). This suggests that the inherent overheads of the hybrid model—particularly inter-process communication and synchronization in MPI—partially mask the gains from instruction-level micro-optimizations. The lightweight, single-node Pthread model is more sensitive to core computation improvements.

Conclusion. Based on this analysis, an unroll factor of 8 provides the optimal performance for both implementations on this hardware architecture. All subsequent experiments in this report will use this configuration.

2.3 Analysis of Macro-optimizations: Load Balancing Strategy

The second set of experiments evaluates the impact of different task partitioning and scheduling strategies on performance. Given the non-uniform workload of the Mandelbrot set, effective load balancing is arguably the most critical factor for achieving high parallel efficiency. Our goal is to identify the optimal strategy for both the shared-memory (`hw2a`) and hybrid (`hw2b`) implementations. The relationship between load imbalance and total execution time for each configuration is presented in Figure 2 and Figure 3.

Pthread Implementation (`hw2a`). For the single-node Pthread version, we compared 9 different strategies based on how image rows are assigned to threads:

- **Static Partitioning (2 strategies):**
  - *Continuous:* The image is divided into contiguous blocks of rows, with each thread assigned one block. (e.g., ‘hw2a_pthread_static_continuous‘)
  - *Cyclic:* Rows are assigned to threads in a round-robin fashion. (e.g., ‘hw2a_pthread_static_cyclic1‘)
- **Dynamic Partitioning (7 strategies):** Using the centralized task queue model described in Section 1.2, we varied the `CHUNK_SIZE` to study the effect of task granularity. The chunk sizes tested were 1, 2, 4, 8, 16, 32, and 64 rows. A configuration is denoted by its chunk size, e.g., ‘hw2a_pthread_dynamic_chunk8‘.

Hybrid MPI+OpenMP Implementation (`hw2b`). For the hybrid version, we evaluated a two-level strategy, combining inter-process partitioning at the MPI level with intra-process scheduling at the OpenMP level. We tested a total of 24 configurations.

- **MPI Process-Level (Inter-Process):** We tested two static partitioning schemes for distributing rows among MPI processes: *Continuous* and *Cyclic*. We conjectured that a dynamic scheme at this level would introduce excessive communication overhead and thus was not considered.
- **OpenMP Thread-Level (Intra-Process):** For each MPI-level partitioning, we explored various OpenMP scheduling policies to manage the workload within each process:
  - *Static:* Default contiguous scheduling.
  - *Static, N:* Cyclic scheduling with a chunk size of N (tested with N = 1, 2, 4, 8, 16).
  - *Dynamic, N:* Dynamic scheduling with a fixed chunk size of N (tested with N = 1, 2, 4, 8, 16).
  - *Guided:* Dynamic scheduling with exponentially decreasing chunk sizes.

Results and Discussion. The experimental results underscore the paramount importance of load balancing and reveal distinct behaviors in the shared-memory and hybrid environments.

- **Analysis of `hw2a` (Pthread):** As illustrated in Figure 2, there is a strong, near-linear correlation between thread imbalance and execution time.
  - The naive *static continuous* partitioning (point 9) is unequivocally the worst strategy, resulting in a staggering 74.3% imbalance and the longest execution time (487.5s). This is expected, as it assigns contiguous, computationally-intense fractal regions to single threads, leaving others idle.
  - In contrast, *dynamic scheduling* with a small chunk size proves highly effective. Performance improves monotonically as the chunk size decreases. The optimal strategy is **dynamic with chunk size 1** (point 1), achieving the lowest imbalance (3.5%) and the fastest execution time (281.3s). This confirms that a fine-grained, "work-dealing" approach is ideal for this problem in a shared-memory context.
  - A notable observation is the performance of *static cyclic* partitioning (point 3). It achieves a low 5.8% imbalance and an execution time (285.7s) nearly as fast as the best dynamic strategy. This suggests that the synchronization overhead of the mutex lock in our dynamic implementation, though small, is non-zero. The lock-free nature of static cyclic makes it a surprisingly competitive alternative, although dynamic scheduling remains superior for minimizing imbalance.
- **Analysis of `hw2b` (Hybrid):** The behavior of the hybrid model, shown in Figure 3, is more nuanced, highlighting the interplay between inter-process and intra-process balancing.

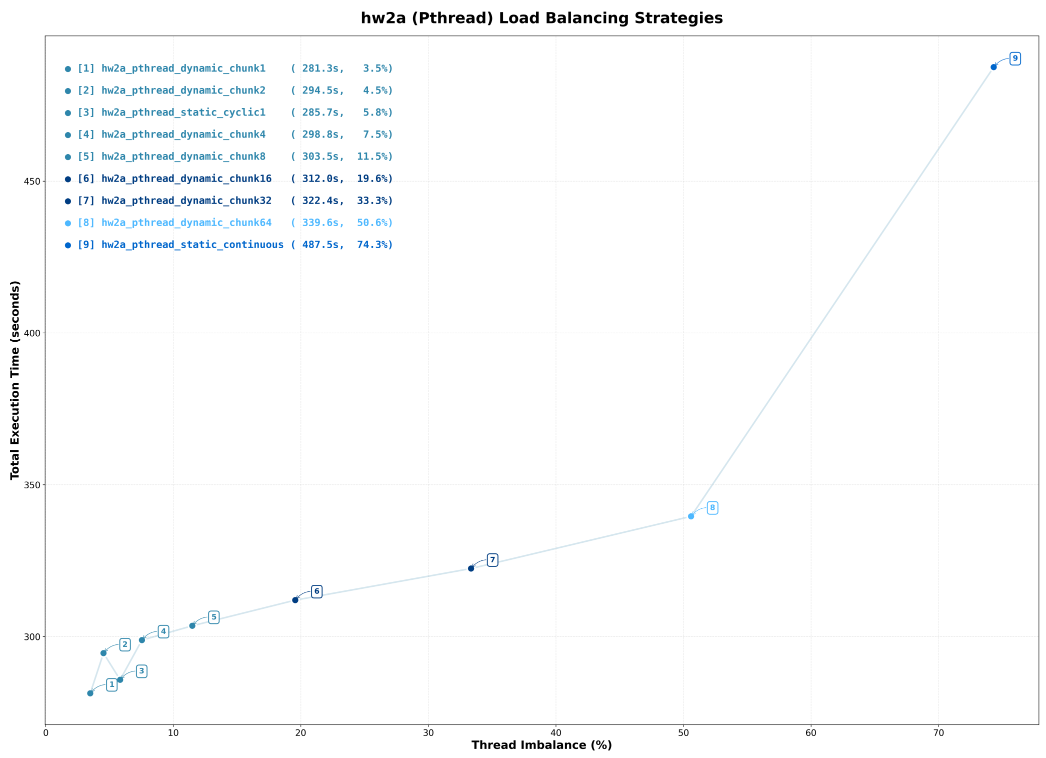

Figure 2: Execution Time vs. Thread Imbalance for different load balancing strategies in the `hw2a` (Pthread) implementation on 12 threads.

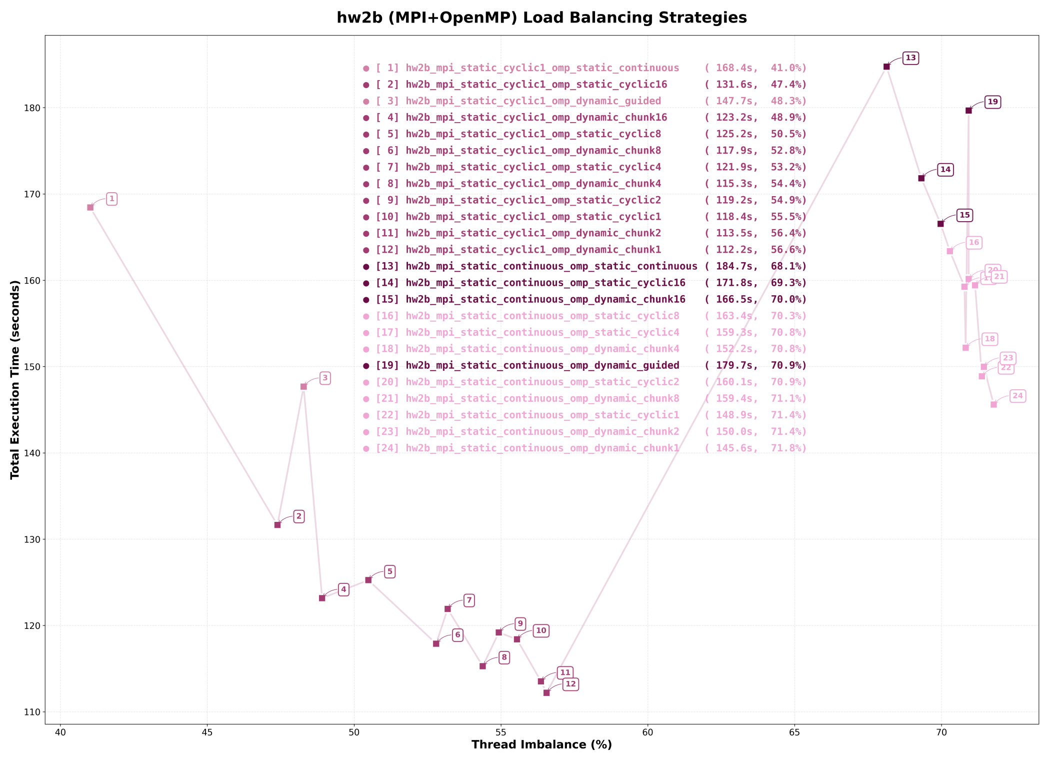

Figure 3: Execution Time vs. Thread Imbalance for different two-level load balancing strategies in the `hw2b` (MPI+OpenMP) implementation on 48 cores (4 processes, 12 threads each).

- Firstly, it is evident that MPI-level partitioning is dominant. All configurations using *MPI static cyclic* (points 1-12, the cluster on the left) vastly outperform those using *MPI static continuous* (points 13-24, the cluster on the right), regardless of the OpenMP policy. This confirms our hypothesis that providing each process with a balanced mix of global rows is the most critical first step.
- Secondly, an interesting trade-off emerges between imbalance and execution time. Consider the best-performing MPI cyclic group: point 1 (`omp static`) has the lowest imbalance (41.0%) but a slow execution time (168.4s), whereas point 12 (`omp dynamic, chunk=1`) has a higher imbalance (56.6%) yet achieves the **fastest execution time (112.2s)**.
- This apparent paradox can be explained. In the static OpenMP case (point 1), threads within a process are assigned fixed portions of that process's already-balanced workload, leading to low measured imbalance but no flexibility. If one thread gets a slightly harder chunk, all others must wait. In the dynamic OpenMP case (point 12), while the *final* measured imbalance might be higher (because some threads may have worked continuously on hard tasks), the overall system throughput is maximized. Idle threads are constantly repurposed to tackle remaining work, effectively hiding the latency of the "long pole" tasks and driving down the total wall-clock time. This demonstrates that for system throughput, minimizing idle time via dynamic scheduling is more important than achieving a perfectly balanced final workload accounting.

Conclusion. Based on this analysis, the optimal load balancing strategies were determined for each implementation. For `hw2a`, we select the **dynamic task queue with a chunk size of 1**. For `hw2b`, the best-performing combination is **static cyclic partitioning at the MPI level, combined with dynamic scheduling (chunk size 1) at the OpenMP level**. All subsequent scalability experiments will use these optimized configurations.

2.4 Strong Scalability Analysis

Having determined the optimal configurations from the previous experiments (unroll factor of 8 and dynamic load balancing), we now evaluate the strong scalability of both implementations. Strong scalability measures how effectively a parallel program utilizes additional processing resources for a fixed-size problem.

Scalability of `hw2a` (Pthread): Single-Node Thread Scaling. For this analysis, we measure the speedup relative to a single-threaded execution on one node. The Pthread implementation was benchmarked on a single compute node, scaling the number of threads from 1 to 12. The results, presented in the left panel of Figure 4, demonstrate excellent strong scalability.

- **Performance:** At 12 threads, `hw2a` achieves a speedup of **8.83x**, which corresponds to a high parallel efficiency of 73.6% (8.83/12). The total execution time is reduced from 2485 seconds down to 281 seconds.
- **Analysis:** The speedup curve closely follows the ideal linear trajectory, especially up to 8 threads, where efficiency remains above 75%. The minor deviation from ideal scalability can be attributed to inherent parallel overheads: (1) the cost of Pthread library calls and context switching, and (2) the contention on the mutex lock for the centralized task queue. Nonetheless, achieving over 8x speedup on 12 cores for a problem with such irregular computation is an outstanding result, validating our choice of a fine-grained dynamic scheduling strategy.

Scalability of `hw2b` (Hybrid): Multi-Node Scaling. For this analysis, we measure the speedup relative to a single-processed 12-threaded execution on one node. The hybrid implementation was evaluated by scaling the number of MPI processes from 1 to 4 (i.e., from 12 to 48 total cores across 4 nodes), with each process utilizing 12 OpenMP threads. The results are shown in the right panel of Figure 4.

- **Performance:** Scaling from 12 to 48 cores (a 4x increase in resources) yields a speedup of **2.47x** over the single-node 12-core baseline. While significant, this is considerably lower than the ideal 4x speedup.

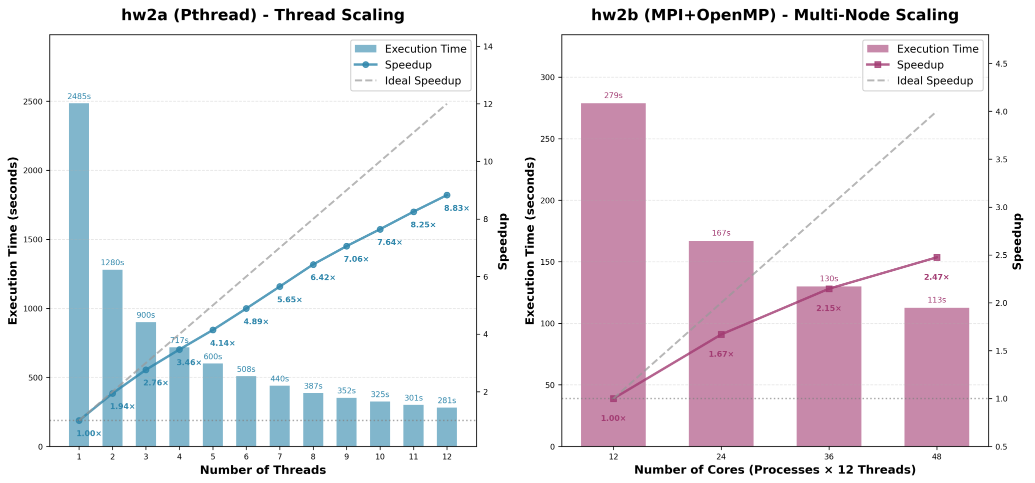

Figure 4: Strong scalability comparison. **Left:** `hw2a` (Pthread) scaling from 1 to 12 threads on a single node. **Right:** `hw2b` (MPI+OpenMP) scaling from 1 to 4 processes (12 to 48 cores) across nodes. Dashed lines represent ideal linear speedup.

- **Analysis:** This sub-linear scalability highlights the fundamental challenge of distributed-memory parallelism: **communication overhead**. The primary bottleneck is the final data gathering stage, where the `MPI_Gatherv` operation must collect and reorder image segments from all nodes. As the number of processes increases, both the latency of cross-node communication and the complexity of data aggregation become dominant factors, limiting overall acceleration.

Ablation Study: Process vs. Thread Trade-offs in the Hybrid Model. To further diagnose the performance characteristics of the hybrid model, we conducted an ablation study by fixing the total core count at 48 while varying the process-thread configuration. Six combinations were tested, from MPI-heavy (48p, 1t) to OpenMP-heavy (4p, 12t). The results are presented in Figure 5.

- **Performance:** Remarkably, all six configurations deliver nearly identical performance. Execution times are clustered within a tight 9% band (111s to 121s), with speedups hovering around 2.4x to 2.5x. The optimal configuration, (16p, 3t), is only marginally faster than the others.
- **Analysis:** This result leads to a crucial insight: for a fixed total core count, the specific balance between MPI processes and OpenMP threads has minimal impact on performance for this problem. The dominant bottleneck is not the intra-node scheduling strategy (OpenMP) but the **inter-node communication cost (MPI)**. As long as each node's cores are saturated with work, the total execution time is dictated by the cost of moving data between nodes. The slightly worse performance of the 48p-1t configuration is consistent with this, as it maximizes the number of communicating MPI processes, thereby maximizing communication overhead.

Conclusion. The scalability experiments confirm that our shared-memory implementation (`hw2a`) is highly efficient, achieving near-ideal strong scalability on a single multi-core node. In contrast, while the hybrid implementation (`hw2b`) does scale across nodes, its performance is fundamentally limited by communication overhead. The ablation study further reinforces this, showing that once you move to a distributed-memory model for this problem, the cost of data aggregation at the MPI level becomes the dominant factor, dwarfing any nuanced differences in intra-node thread scheduling.

3 Experience and Conclusion

This assignment provided extensive hands-on experience with advanced parallel programming techniques, spanning shared-memory (Pthread), distributed-memory (MPI), and hybrid (MPI+OpenMP)

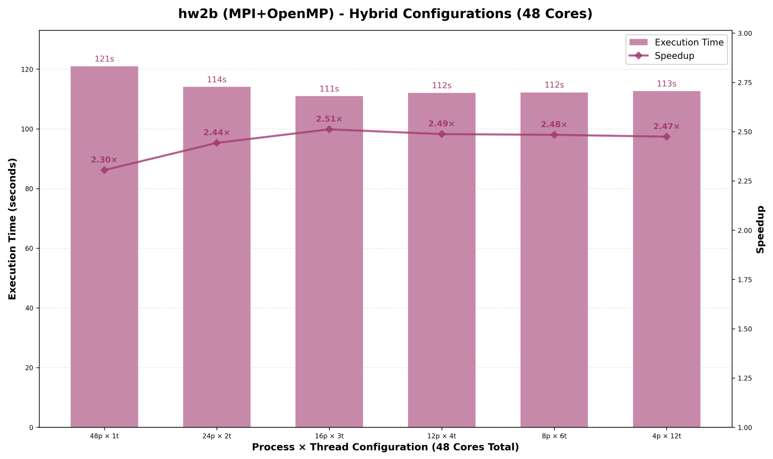

Figure 5: Ablation study of `hw2b` performance at a fixed total of 48 cores. The minimal variation in execution time across six different (process, thread) configurations indicates that inter-node MPI communication, not intra-node scheduling, is the primary performance bottleneck.

models. Through a systematic process of implementation, profiling, and optimization, we developed a highly efficient parallel solution for rendering the Mandelbrot set. The key takeaway is the hierarchical nature of performance optimization: macro-level strategies like load balancing establish the upper bound on scalability, while micro-level optimizations like vectorization determine the raw computational throughput.

Key Learnings. The most significant lesson was the dramatic performance impact of instruction-level parallelism. Learning to effectively utilize **SIMD intrinsics (SSE2)** and combine them with manual loop unrolling was challenging but rewarding. As our experiments showed, these micro-optimizations in the innermost compute kernel provided a greater than 2x performance boost, a gain that cascaded through all levels of parallelism. This highlights a critical principle in HPC: any inefficiency in the core computation is magnified when scaled across many cores.

A second key learning was the trade-off between different load balancing strategies. For the shared-memory model, a fine-grained dynamic task queue proved optimal. For the hybrid model, a two-level approach combining coarse-grained static cyclic distribution at the MPI level with fine-grained dynamic scheduling at the OpenMP level yielded the best results. This demonstrates that the ideal strategy is context-dependent, balancing algorithmic elegance with the architectural realities of communication and synchronization costs.

Difficulties Encountered. The primary difficulty was mastering the intricacies of SSE2 vectorization. Writing correct and efficient SIMD code required careful management of data alignment, register usage, and control flow (e.g., handling early exits with `_mm_movemask_pd`), demanding a deeper understanding of low-level CPU architecture than typical high-level programming.

Feedback. This assignment was an excellent exercise in real-world performance engineering. To further enhance the learning experience, it would be highly beneficial for the teaching staff to provide a set of optimized reference implementations after the submission deadline. Access to such "gold-standard" code would allow students to compare their approaches, discover alternative optimization techniques, and understand the performance gap between their own solutions and a highly tuned implementation. This would provide invaluable insights for future work.