OpenMP Performance Measurements

Measurement 1: Performance Comparison of OpenMP Implementations

System Setup: *1 node, 1 to 24 threads, 4096 interlines, 5 iterations, SLURM job scheduling, Hyperfine timing tool.*

1. Baseline Serial vs. Parallel Execution

Figures 1 and 2 show that the serial implementation consistently takes approximately 47.767 s for 5 iterations. This is expected as it lacks parallelization. In contrast, the OpenMP implementations (row-wise, column-wise, element-wise) exhibit significant reductions in runtime as the number of threads increases.

2. Row-wise and Column-wise Performance

Figures 1 and 2 demonstrate that both row-wise and column-wise approaches scale efficiently up to 8 threads, reducing runtime from 47.767 s to around 7.2 s. Beyond 8 threads, speedup gains decrease, reaching 3.6–3.7 s at 24 threads ($\approx 13.1\times$ speedup). This pattern reflects memory-access bottlenecks and thread synchronization overhead.

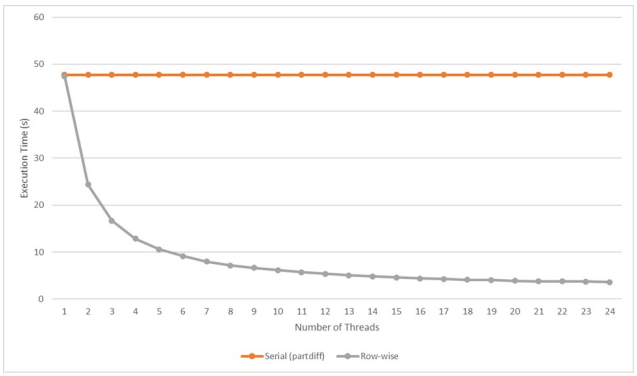

Figure 1: Row-wise vs. Serial Execution

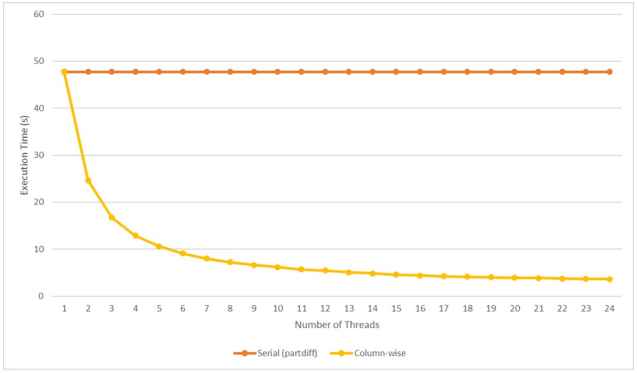

Figure 2: Column-wise vs. Serial Execution

3. Element-wise Performance

Figure 3 shows that the element-wise implementation begins with poor performance at 1 thread (92.548 s), due to fine-grained task splitting and synchronization. It improves steadily, reaching 5.772 s at 24 threads—showing better relative speedup ($\approx 16\times$) than the other two, but worse absolute runtime because of thread interference and cache contention.

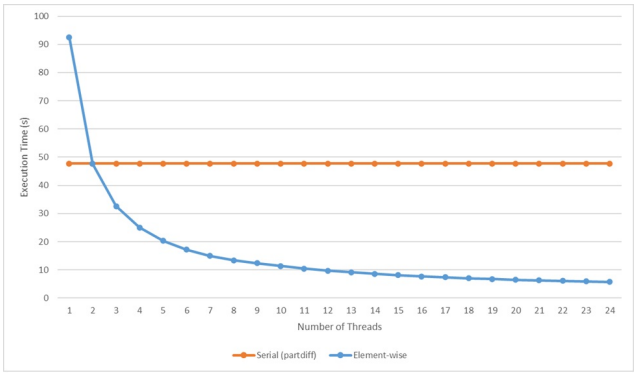

Figure 3: Element-wise vs. Serial Performance

4. Performance Table

Table 1: Execution Time (s) Across Threads

Threads	Serial	Row	Column	Element
1	47.767	47.452	47.717	92.548
2	47.767	24.396	24.641	47.691
4	47.767	12.883	12.872	25.054
8	47.767	7.186	7.216	13.428
16	47.767	4.381	4.388	7.697
24	47.767	3.636	3.649	5.772

5. Analysis and Conclusions

Scaling Efficiency: Row-wise and column-wise implementations achieve the best absolute runtime (3.6 s), while element-wise shows stronger relative scaling ($\approx 16\times$) but worse absolute performance due to fine-grained overhead.

Parallel Overhead: The element-wise approach is inefficient at low thread counts because of excessive synchronization costs.

Diminishing Returns: All three implementations exhibit plateauing benefits beyond 8–12 threads, driven by memory-access contention and synchronization overhead.

Conclusion: Among the three, row-wise or column-wise parallelization offers the best trade-off between simplicity of implementation and performance.

Measurement 2: Scaling with Interlines (Default Data Distribution)

System Setup (Measurement 2): *1 node, 24 threads, Jacobi method (method = 2), default data distribution (static block-row), 5000 iterations, Hyperfine timing tool.*

6.1. Raw Benchmark Data

Table 2: Measured Runtimes (5000 Iterations, 24 Threads, Jacobi, Default Static Block-Row)

Interlines	Runtime (s)
1	0.0465
2	0.0452
4	0.0474
8	0.0635
16	0.1012
32	0.2048
64	0.5496
128	1.913
256	7.092
512	29.408
1024	114.744
2048	455.989
4096	1813.238

6.2. log-log plot of runtime vs. Interlines

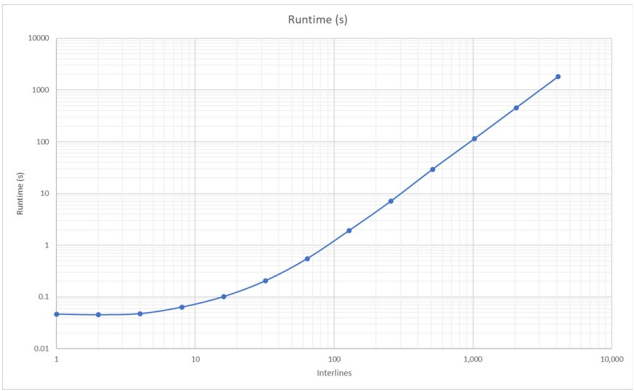

Figure 4: Runtime as a function of interlines (log-log scale).

6.3. Interpretation of results

Because the iteration count is fixed at 5000 and we only vary the number of *interlines* (each grid is size $(8 \times \text{interlines} + 9)^2$), the work per iteration grows roughly as $(\text{interlines})^2$. Hence, for large interlines, we expect:

$$T(\text{interlines}) \approx C\,(\text{interlines})^2,$$

for some hardware-dependent constant C. In a log-log plot, a pure quadratic relation appears as a straight line of slope 2—exactly what we see in Figure 4 for interlines ≥ 64).

Small-Size Overhead Region (Interlines = 1, 2, 4). At interlines $= 1, 2, 4$, runtimes are nearly constant (around 0.045 s) because the overhead of spawning and synchronizing 24 threads (and cache warm-up) dominates the small amount of arithmetic.

Transitional Region (Interlines = 8–32). From 8 to 32 interlines, runtimes increase from 0.063 s to 0.205 s. In this window, the computation cost begins to exceed the parallel overhead, but caches still partially hide memory costs.

Compute-Bound, Quadratic Regime (interlines $\geq$ 64). When interlines = 64, runtime = 0.55 s; double to 128 yields 1.91 s ($\approx 4 \times 0.55$); doubling again to 256 yields 7.09 s; then 512 $\rightarrow$ 29.41 s; 1024 $\rightarrow$ 114.74 s; 2048 $\rightarrow$ 455.99 s; 4096 $\rightarrow$ 1813.24 s. Those 4$\times$ jumps in runtime for 2$\times$ increases in interlines confirm the runtime $\propto$ $(\text{interlines})^2$ behavior once the problem is large enough to be fully compute bound.

Measurement Stability. Each data point is the mean of three runs (via `hyperfine -r 3`), with standard deviations below 0.5 s even at the largest size. This indicates a stable cluster load, so the results are reproducible.

Why "Default Data Distribution"? The binary `partdiff-row` was compiled with `-ROW`, so its solver loop is:

```
#pragma omp parallel for
for (i = 1; i < N; i++) {
  // Jacobi update on row i
}
```

Because there is no explicit `schedule(...)` clause, OpenMP uses a static schedule that distributes contiguous blocks of rows to each thread. In other words, no column-based or cyclic distribution is used, only the built-in block-row split. By keeping this 'static block row' strategy fixed, we isolate the effect of the size of the grid on performance.

Summary of Key Findings.

- **Quadratic Scaling.** For interlines $\geq$ 64, runtime grows $\propto$ $(\text{interlines})^2$.
- **Overhead-Dominated at Small Sizes.** For interlines = 1, 2, 4, parallel overhead dominates, flattening the curve.
- **Transitional Region.** Between 8 and 32 interlines, cache effects are moderate, but do not prevent the upward trend.
- **Stable Results.** Low standard deviations confirm reproducibility.
- **Fixed-Distribution Baseline.** Using OpenMP's static block-row scheduling provides a consistent baseline.