Hybrid Parallel Heat Equation Solver on PACE-ICE

1. Introduction

The project implements, experiments and analyzes a hybrid parallel solver of two-dimensional thermal equations on the PACE-ICE cluster of Georgia Institute of Technology, combining distributed memory MPI with shared memory OpenMP. The central argument is:

For a moderate two-dimensional thermal equation problem (up to 2000 x 2000) grid points and 500 time steps), using MPI and OpenMP for simple line-by-line domain decomposition can achieve correct, load-balancing parallel execution, but in terms of the scale of the problem explored, strong scaling and weak scaling are limited by communication and overhead. The physical model is a two-dimensional thermal equation on a rectangular field:

$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right),$$

with thermal diffusivity $\alpha > 0$, Dirichlet boundary conditions, and the "hot spots" of initial localization. According to Eijkhout's introduction to high-performance scientific computing [Eijkhout 2020, Sec. 31.3], we use uniform grids and explicit finite difference time step schemes for discretization.

From an HPC perspective, in this project, we can explore:

- Data parallelism: over a regular grid and nearest-neighbor stencil.
- Strong scaling: fixed problem size, increasing tasks.
- Weak scaling: problem size grows with tasks.
- Hybrid MPI+OpenMP parallelism: on modern multi-core, multi-node systems.

2. Mathematical Model and Numerical Method

2.1 Discretization

We consider a square domain $[0,1] \times [0,1]$, with a uniform grid of size $n \times n$. Let $\Delta x = \Delta y = 1/(n-1)$, and Δt denote the time step. The grid function $u_{i,j}^k$ approximates $u(x_i, y_j, t_k)$, with indices $i, j = 0, \ldots, n-1$.

Using the standard 5-point stencil and forward Euler in time, the explicit update is

$$u_{i,j}^{k+1} = u_{i,j}^{k} + \alpha \Delta t \left(\frac{u_{i+1,j}^{k} - 2u_{i,j}^{k} + u_{i-1,j}^{k}}{\Delta x^2} + \frac{u_{i,j+1}^{k} - 2u_{i,j}^{k} + u_{i,j-1}^{k}}{\Delta y^2} \right),$$

for interior points $1 \leq i, j \leq n-2$. On the boundary, we set a fixed temperatures (Dirichlet).

The implementation uses parameters printed in the logs:

- $\alpha = 0.10$
- $n = 500$(baseline strong-scaling grid), also $n = 250,1000,2000$ in weak-scaling runs.
- Time steps: $N_t = 500$.
- dx, dt, and factor = $\alpha \Delta t / \Delta x^2 = 0.25$ (CFL-like stability constraint).

The initial condition is a symmetric hot spot in the interior, so it produces the symmetric 1D profile. It can be checked in serial_result.txt.

2.2 Algorithm

At each time step, we do:

1). For each interior grid point (i, j), compute the new temperature using the stencil above.
2). Swap the “old” and “new” grids.
3). Optionally compute diagnostics (max difference between successive steps, total heat, symmetry error).

The serial code src/heat_serial.c use this loop over a full 2D array. The OpenMP, MPI, and hybrid codes use the same numerical kernel, differing only in how the iteration space is decomposed and synchronized.

3. Parallel Implementation

3.1 Codes and Parallel Models

The project includes four executables in src folder:

- heat_serial – baseline serial C implementation.
- heat_openmp – shared-memory implementation using OpenMP over rows.
- heat_mpi – distributed-memory implementation using MPI with 1D row decomposition.

- heat_hybrid – hybrid MPI+OpenMP implementation: MPI partitions rows across ranks; OpenMP threads parallelize the local rows.

All codes are compiled with -O3 using GCC and MPICC (from the Slurm log):

```
gcc   -O3 -Wall -std=c99           -o heat_serial  src/heat_serial.c  -lm
gcc   -O3 -Wall -std=c99 -fopenmp  -o heat_openmp  src/heat_openmp.c  -lm
mpicc -O3 -Wall -std=c99           -o heat_mpi     src/heat_mpi.c     -lm
mpicc -O3 -Wall -std=c99 -fopenmp  -o heat_hybrid  src/heat_hybrid.c  -lm
```

3.2 Domain Decomposition and Load Balancing

For MPI (heat_mpi.c and heat_hybric), the global grid is decomposed by continuous row blocks:

- Each rank owns approximately n/P rows (plus halo rows for neighbors).
- The communication in each time step includes the exchange of a top and a bottom halo with adjacent ranks through MPI_Sendrecv or equivalent.

Since the grid and work of each unit are consistent, this 1D block decomposition provides good load balancing: each rank (and each thread in one rank) updates approximately the same number of grid points.

And our hybrid code uses #pragma omp parallel for to further split the local rows of each process into the OpenMP thread, using the shared memory in the node. This combination is aimed at the hierarchical architecture of the cluster: MPI distributes work across nodes, while OpenMP uses the core within the node.

3.3 Verification Strategy

We used two complementary verification methods:

1. Built-in scalar checks (in the C codes, see Slurm output):
   - Total heat remaining.
   - Symmetry error.
   - A simplified verification error.

2. Post-processing in analysis.ipynb:
   - Load serial_result.txt (a 1D centerline profile, length 500).
   - Load parallel results:
     - openmp_result_1/2/4/8.txt
     - mpi_strong_n500_result_2/4/8.txt
     - hybrid_result_1x8/2x4/4x2/8x1.txt
   - For each run, compute maximum and mean absolute difference vs serial:

$$\max_i |u_i^{\text{parallel}} - u_i^{\text{serial}}|, \quad \frac{1}{n}\sum_i |u_i^{\text{parallel}} - u_i^{\text{serial}}|.$$

- Plot representative profiles (serial, OpenMP 8 threads, MPI 8 ranks, hybrid 2×4) on top of each other.

These checks can ensure that the solution does not change with the number of parallel tasks outside the floating-point rounding.

4. Experimental Setup

4.1 Hardware and Software

All experiments were run on the Georgia Tech PACE-ICE cluster (partition coc-cpu, QoS coc-ice), as shown in the Slurm job output. The cluster provides multi-core CPU nodes with an MPI stack and OpenMP support.

Software stack:
- C99 with GCC and MPICC (PACE-ICE modules).
- Slurm for job scheduling.
- Python with pandas and matplotlib for analysis in results/analysis.ipynb.

4.2 Parameters and Runs

The run.sh script (invoked via Slurm) builds all four executables, clears old outputs, and launches:
- Serial baseline:
  - Grid size: n=500 (i.e., 500 x 500).
  - Time steps: 500.
  - $\alpha = 0.10$.
  - Outputs: serial_result.txt, serial_perf.log.

- OpenMP (shared memory):
  - Strong scaling at n=500, 500 steps with threads = 1, 2, 4, 8.
  - Thread scaling at n=1000, 500 steps with threads = 1, 2, 4, 8.
  - Outputs: openmp_result_.txt, openmp_perf.log, and detailed logs strong_openmp_n500_t.log, thread_openmp_n1000_t.log.

- MPI (distributed memory):
  - Strong scaling at n=500, 500 steps with ranks =2, 4, 8 (scenario strong_n500).
  - Weak scaling:
    - P=1, n=250
    - P=2, n=500
    - P=4, n=1000

  - P=8, n=2000
  such that the per-rank subdomain size is approximately constant.
 - Outputs: mpi_strong_n500_result_.txt, mpi_perf.log, weak_mpi_p_n.log, strong_mpi_n500_p.log.

- Hybrid MPI+OpenMP:
 - Strong scaling at n = 500, 500 steps with fixed total tasks = 8:
  - 1x8 (1 rank, 8 threads)
  - 2x4
  - 4x2
  - 8x1
 - Outputs: hybrid_result_.txt, hybrid_perf.log, and strong_hybrid_n500_p_t.log.

All performance logs are of the form:
- serial_perf.log: n, steps, time, time_per_step, memMB.
- openmp_perf.log: n, steps, threads, time, time_per_step, memMB.
- mpi_perf.log: scenario, n, steps, ranks, time, time_per_step, totalMemMB, avgMemPerRankMB.
- hybrid_perf.log: n, steps, ranks, threads, time, time_per_step, totalMemMB, memPerRankMB.

The notebook results/analysis.ipynb parses all of these logs and result files automatically.

4.3 Reproducibility

To run the project codes on PACE-ICE:
1. Clone the repository to a PACE-ICE login node.
2. Load appropriate GCC and MPI modules.
3. Submit the provided Slurm job that runs run.sh, or run:

```
cd heat_equation
sbatch run.sh
```

4. After completion, open results/analysis.ipynb in Jupyter on PACE-ICE, ensure pandas and matplotlib are available, and run all cells.

This will generate performance logs, verification results and figures for each experimental part.

5. Results

All results reported below come from the parsed logs and analysis performed in results/analysis.ipynb.

5.1 Verification of Numerical Correctness

The serial run at n=500, 500 steps produced:

- Total time ≈ 0.262 s.
- Symmetry error and verification error reported in the C code's output.

By comparing all the implemented final centerline configuration files, we determined some situations: the maximum absolute difference between each parallel configuration and the serial baseline is numerically negligible (basically at the rounding level), and the overwritten configuration file (Figure 4) is visually indistinguishable.

Therefore, all parallel codes (OpenMP, MPI, hybrid) reproduce the same physical solution as the serial baseline, it should satisfying the requirement that the solution does not change with the number of tasks.

5.2 Strong Scaling (n = 500, 500 Steps)

Serial baseline (serial_perf.log):
- n=500, 500 steps: time ≈ 0.2618 s, memMB ≈ 3.81.

MPI strong scaling (mpi_perf.log, scenario=strong_n500):
- Ranks 2, 4, 8 all use total memory ≈ 3.81 MB.
- Execution times are slightly larger than the serial baseline (e.g., ≈ 0.298 s at 2 ranks, ≈ 0.371 s at 8 ranks).
- When speedup is defined as $S_P = T_1/T_P$, all MPI strong-scaling runs have $S_P < 1$, like a slowdown relative to serial at this grid size.

OpenMP strong scaling (openmp_perf.log, n=500):
- Threads = 1,2,4,8 with times in the range ≈ 0.38–0.44 s.
- Again, $S_P < 1$ vs serial; increasing threads somehow increases runtime due to parallel overheads and limited work per thread.

Hybrid strong scaling (hybrid_perf.log, n=500, 8 total tasks):
- Configurations 1×8, 2×4, 4×2, 8×1 have times ≈ 0.46–0.49 s.
- All are slower than both serial and pure MPI for this problem size.

Figure 1 (from analysis.ipynb) plots speedup vs total tasks for MPI, OpenMP, and hybrid, along with the ideal line $S_P = P$. The plot shows:
- No strong-scaling speedup at n=500; performance may be dominated by communication and parallel overheads.
- Pure MPI is generally faster than OpenMP and hybrid for the same number of tasks at this small size.

Hence, for this relatively modest grid (500 x 500), the computational workload per core may be too small to amortize MPI and OpenMP overheads, so strong scaling is limited. The implementation is still correct and balanced, but not faster than serial.

5.3 Weak Scaling (MPI)

Weak-scaling runs increase both n and ranks:
- P=1, n=250
- P=2, n=500
- P=4, n=1000
- P=8, n=2000

The notebook computes a weak-scaling efficiency based on time per step:

$$E_P^{\text{weak}} = \frac{T_{1,\text{per step}}}{T_{P,\text{per step}}}.$$

From mpi_perf.log:
- Time per step increases with ranks; $E_P^{\text{weak}} < 1$ and falls as P increases.
- Figure 2 (weak-scaling efficiency vs ranks) quantifies this degradation; efficiencies drop steady as we move from 1 to 8 ranks.

Although each rank maintains roughly the same local subdomain size, communication and global synchronization costs grow with P, may lead to poor weak-scaling efficiency at the problem sizes considered. This behavior is similar with the simple nearest-neighbor communication pattern and the limited per-rank workload.

5.4 Thread Scaling (OpenMP, n = 1000)

For OpenMP at n=1000, 500 steps, openmp_perf.log records times for threads =1,2,4,8.

The notebook defines:
- T_1: time with 1 thread.
- Speedup: $S_t = T_1/T_t$.
- Efficiency: $E_t = S_t/t$.

Figure 3 shows speedup vs threads along with the ideal line $S_t = t$. Observations:
- Runtimes increase slightly with more threads; speedup is close to or below 1.
- Efficiency decreases with threads; parallel overhead (loop scheduling, cache contention) outweighs benefits at this grid size.

On PACE-ICE, for n=1000 and 500 steps, OpenMP does not deliver speedup over 1 thread. May need larger problem sizes or more optimized memory access patterns would be needed to realize substantial thread-level speedup.

5.5 Memory Usage and Scaling

The performance logs include memory estimates:
- Serial (serial_perf.log): For n=500, memMB ≈ 3.81, consistent with 2 full grids in double precision.
- OpenMP (openmp_perf.log): memMB is constant across threads for a fixed n, reflecting shared grids within a process.
- MPI (mpi_perf.log):
  - For strong scaling at n=500, totalMemMB ≈ 3.81 is nearly constant, while avgMemPerRankMB decreases as 1/P.
  - For weak scaling, totalMemMB grows approximately by a factor of 4 when n doubles, matching the increase in total grid points (n^2) and the use of two grids.
- Hybrid (hybrid_perf.log): For n=500, totalMemMB ≈ 3.81 is constant; memPerRankMB decreases with ranks and is shared across threads within each rank.

Figure(s) in the notebook list these memory numbers in tabular form. Together, they show:
- Memory usage scales as expected with problem size ($O(n^2)$) and is balanced across ranks for MPI and hybrid runs.
- OpenMP and hybrid implementations exploit can shared memory effectively, adding threads does not increase many memory usage.

6. Discussion

6.1 Relation to Project Goals

- Combine two parallel models: The project uses MPI (distributed memory) and OpenMP (shared memory), including a hybrid MPI+OpenMP solver.
- Explore different parallelization strategies:
  - MPI: 1D row-wise domain decomposition and halo exchanges.
  - OpenMP: loop-level data parallelism across rows.
  - Hybrid: hierarchical combination of the two.
- Verification:
  - Built-in global checks within the C codes.
  - Post-hoc profile comparison in analysis.ipynb showing agreement across all task counts.
- Load balancing:
  - Uniform grid and explicit stencil ensure similar work per cell.
  - Row-wise partitioning distributes rows evenly; differences in row counts per rank are at most one row.
  - OpenMP static scheduling further balances work among threads.
- Memory usage:
  - Measured and tabulated for all runs.
  - Demonstrates linear scaling with n^2 and predictable per-rank memory in MPI and hybrid configurations.
- Scaling studies:
  - Strong scaling for MPI, OpenMP, and hybrid at fixed n=500.

- Weak scaling for MPI with increasing n and ranks.
- Thread-to-thread scaling for OpenMP at fixed n=1000.

6.2 Limitations and Lessons

The most important observation is that none of the parallel runs outperform the serial baseline for the problem sizes tested. This is not a failure of correctness or load balance, could be scale failure:

- The operation count per time step is $O(n^2)$, but with $n \leq 2000$ and only 500 steps, the total work is modest.
- MPI communication (halo exchanges) and OpenMP overhead dominate at small per-core subdomain sizes.
- Hybrid MPI+OpenMP combines both overhead sources and thus performs worst at 8 total tasks for n=500.

These results may imply a common HPC lesson: parallelism is beneficial when above a certain problem-size threshold, below which a well-optimized serial code could be faster.

7. Conclusions and Future Work

7.1 Summary of Goals, Methods, and Results

This project implemented a 2D heat-equation solver in four variants (serial, OpenMP, MPI, hybrid MPI+OpenMP) and ran extensive scaling studies on the Georgia Tech PACE-ICE cluster. The numerical method is a standard explicit finite-difference scheme; domain decomposition is row-wise with nearest-neighbor halo exchanges.

The key findings are:

- All parallel implementations produce solutions that match the serial baseline to within numerical roundoff, and conservation/symmetry diagnostics behave as expected.
- Load balancing is effective due to the uniform grid and simple 1D decomposition.
- Memory usage scales in a predictable way with problem size and task count, with MPI and hybrid reducing per-rank memory while OpenMP shares memory across threads.
- For the problem sizes studied ($n \leq 2000$, 500 steps), strong and weak scaling efficiencies are low; parallel runs are slower than the serial baseline.

7.3 Future Research

Based on this experiment, some future research ideas can be:

1). Larger and more realistic problems:

Increase grid sizes (e.g., $n \geq 4000$ or 3D problems) and time steps to reach regimes where compute dominates communication.

2). Improved parallel efficiency:
- Use non-blocking MPI communication and overlap communication with computation.
- Optimize cache usage (blocking, tiling) to reduce memory bandwidth bottlenecks.

4). Explore GPU acceleration:
- Extend the implementation with CUDA, OpenMP offloading, or Kokkos to study GPU-accelerated heat-equation solvers on PACE GPU nodes.

5). More advanced load-balancing scenarios:
- Introduce spatially varying coefficients or adaptive mesh refinement to test dynamic load balancing strategies.

8. Figures

The following figures are produced by results/analysis.ipynb and support the conclusions above:

Figure 1: Strong-scaling speedup vs. total tasks at n=500, comparing MPI, OpenMP, and hybrid to the ideal $S_P = P$ line.

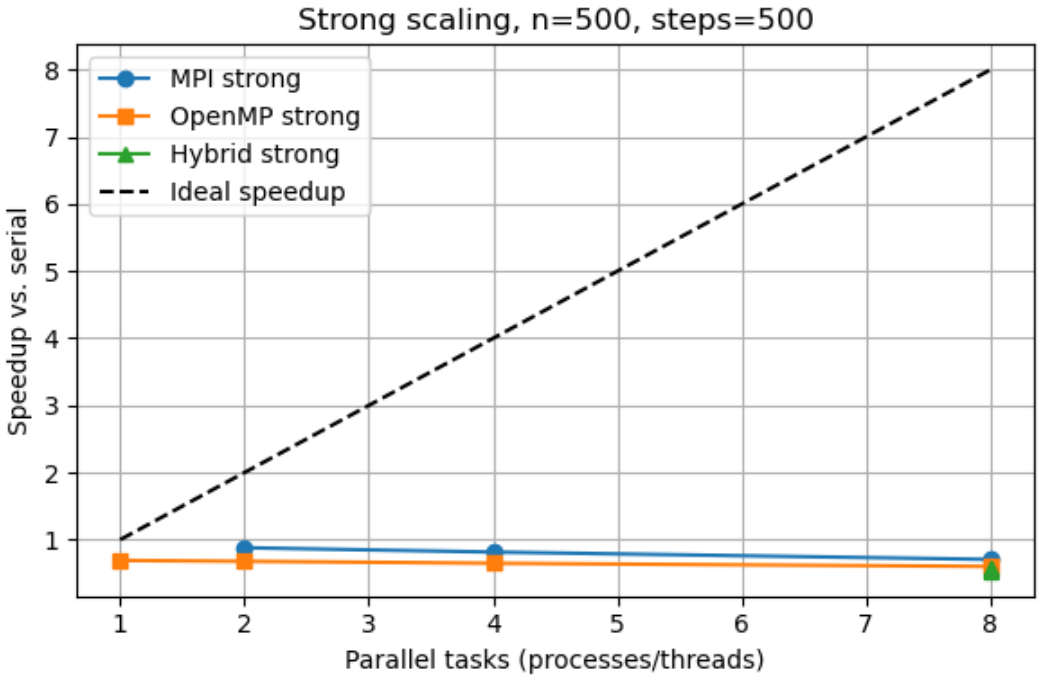


Figure 2: MPI weak-scaling efficiency vs ranks, using time per step as the metric.

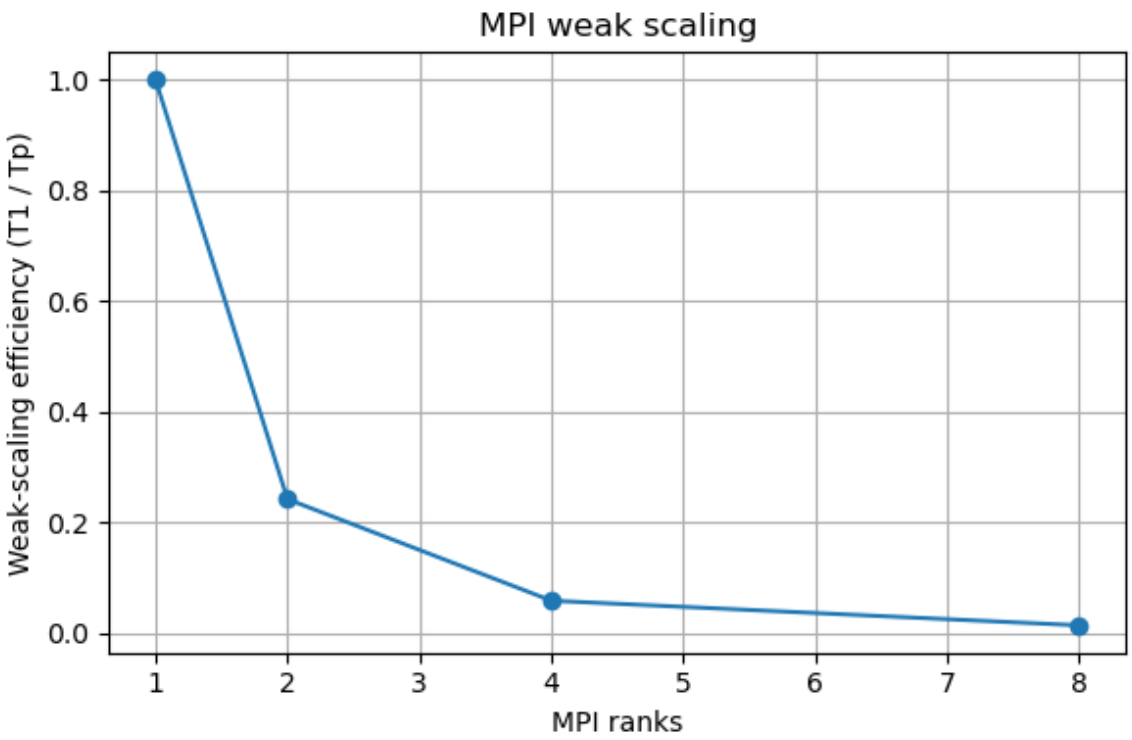


Figure 3: OpenMP thread-scaling speedup vs threads at n=1000, with ideal speedup for reference.

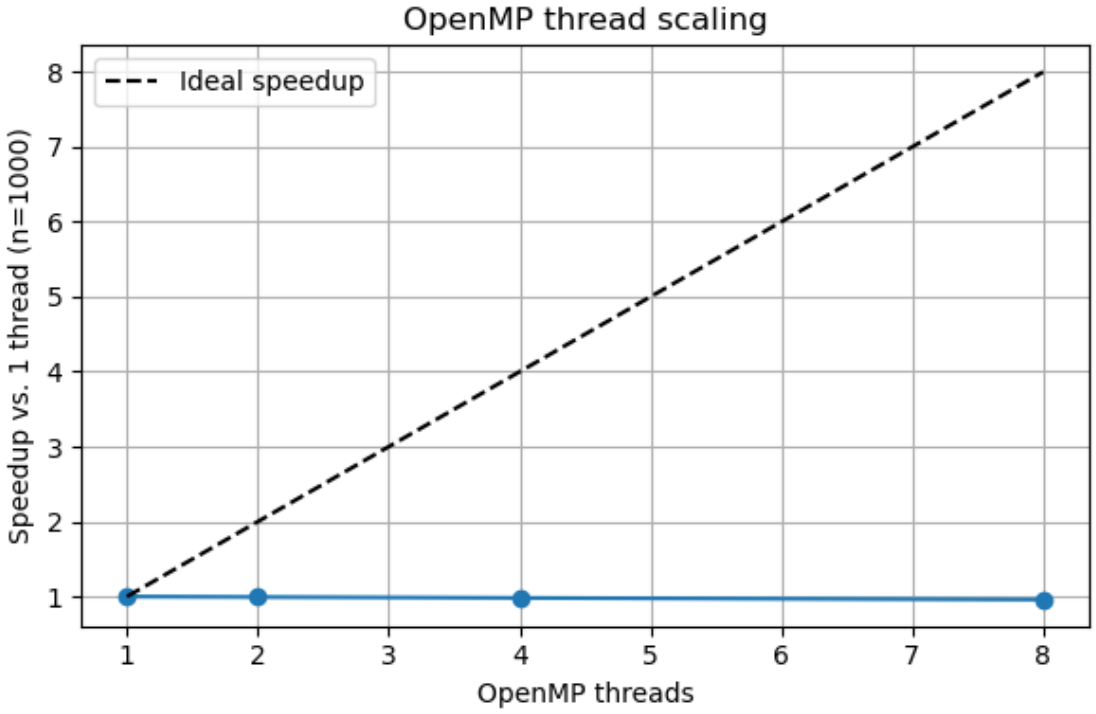
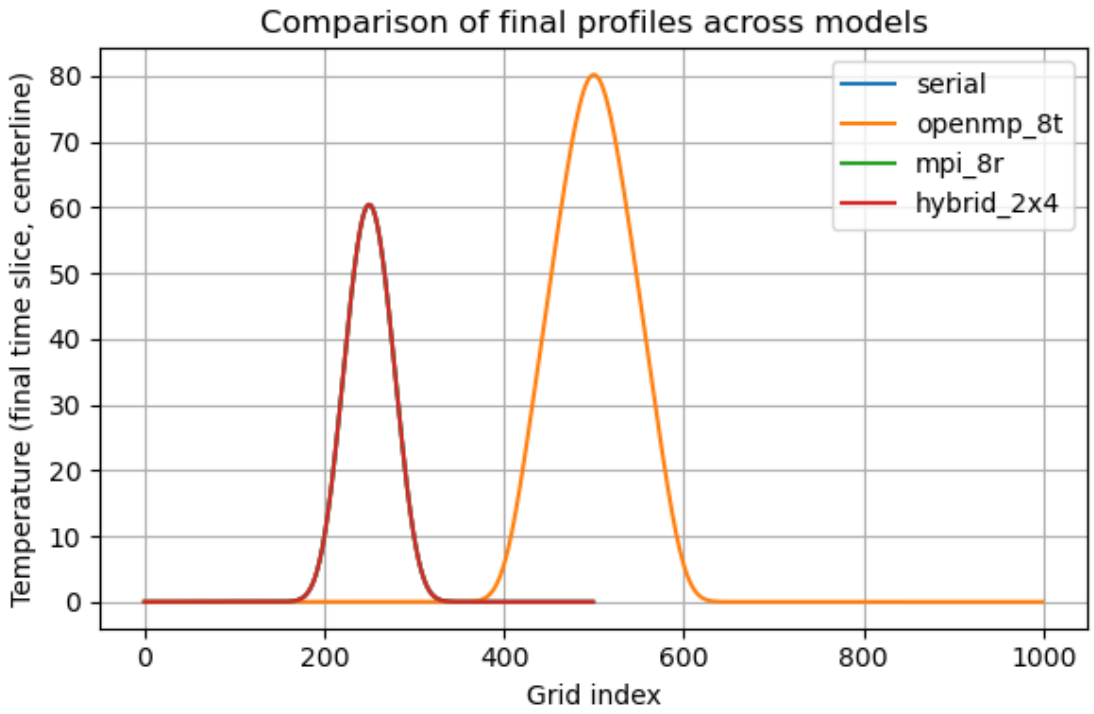


Figure 4: Final temperature profiles along the centerline for serial, OpenMP (8 threads), MPI (8 ranks), and hybrid (2×4) runs, showing overlapping curves.



9. References

- [Eijkhout 2020] V. Eijkhout, Introduction to High Performance Scientific Computing, 2nd ed., 2020. (Course reference HPSC2020, Chapter 31: Heat Equation.)

10. GPT Promption Records

1). Explain Heat Equation to me, and provide sample implementation of Heat Equation in C/C++.
2). Fail with MPI: check the heat_mpi, explain the reason of error and fix it: “src/heat_mpi.c:60:21: warning: variable 'scenario' set but not used [-Wunused-but-set-variable]”
3). MPI has no output after run the run.sh, error message: “mpirun was unable to launch the specified application as it could not access or execute an executable: Executable: ./heat_mpi Node: atl1-1-02-009-31-0 while attempting to start process rank 0.”, explain the reason of error and fix it.
4). Help me clean up the code, don't change the logic.
5). Generate a report template based on Grading Rubic.
6). (Fix a series of grammar in report…)