1 Advanced High-Performance Computing Assignment 2

1.1 MPI Task-Farm Scaling for HEP Analysis (Week 2)

Student: Mario Rodriguez Mestre
Date: October 12, 2025
Course: AHPC - Advanced High-Performance Computing
Repository: https://github.com/pentamorfico/ahpc —

1.2 Abstract

This short report documents the MPI master/worker task-farm implementation used to explore a combinatorial set of analysis cuts for a HEP dataset, and the scaling behaviour measured on the cluster using containerized execution. We provide a summary of runtimes for a sweep of MPI ranks (ntasks) and present runtime and speedup plots.

1.3 1. Experimental setup

- Program: `week2/python/task_farm_HEP.py` (MPI master/worker)
- Execution: Apptainer container image located on NFS (used to provide mpi4py and a consistent Python environment)
- Submission: SLURM sbatch jobs generated by `week2/python/submit_slurm_sweep.sh`
- Data: HEP CSV dataset mounted at `/home/bxl776_ku_dk/modi_mount/ahpc/week2/mc_ggH_16_13TeV_Zee_`
- Sweep parameters: ntasks = [2,4,8,16,32,64,128,256] (one job per ntask), n_cuts=3, repeat=5

1.4 2. Results summary

The results were collected from the SLURM job outputs on the NFS mount. The parsed summary (see `week2/results/data/bench_summary.csv`) contains the job id, number of MPI ranks, measured runtime (s), and job status.

Key rows from the CSV:

jobid	ntasks	runtime_s	n_settings	status
669	2	355.4894	6561	OK
670	4	117.5970	6561	OK
671	8	51.4687	6561	OK
672	16	23.9663	6561	OK
673	32	12.1113	6561	OK
674	64	6.1116	6561	OK
675	128	3.6467	6561	OK
676	256	2.9968	6561	OK

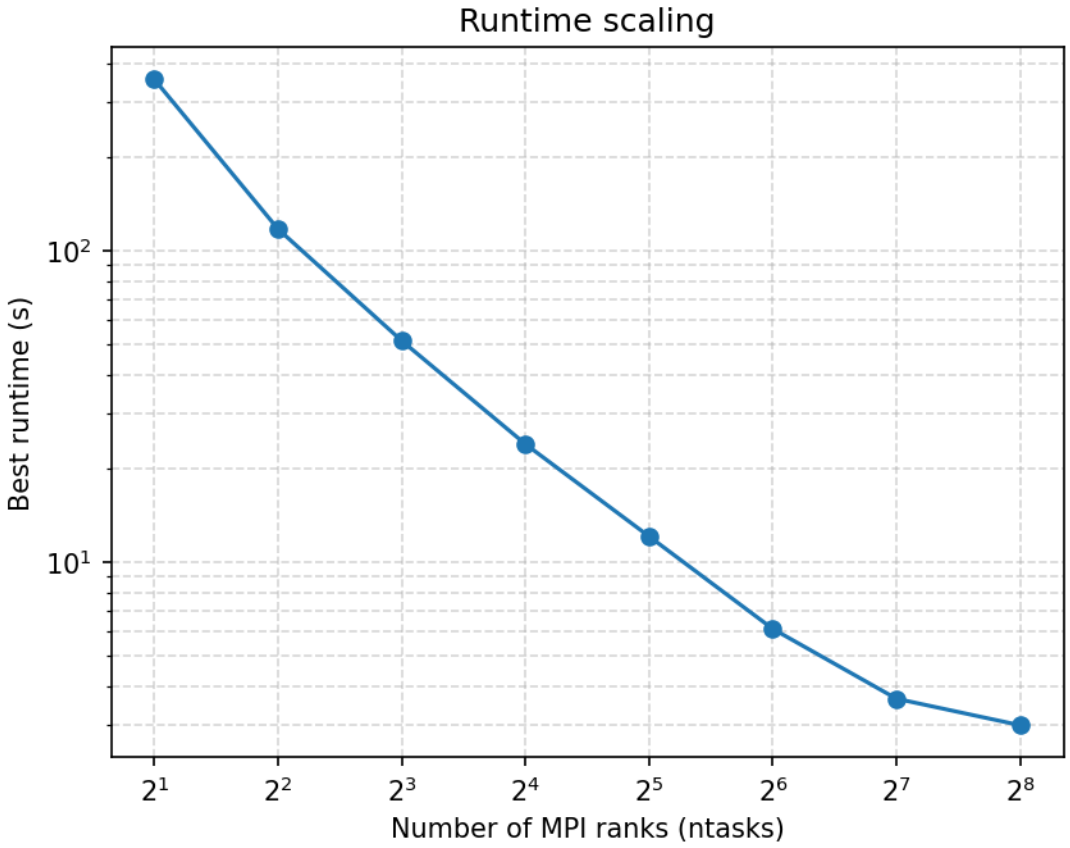

Figure 1: Runtime scaling

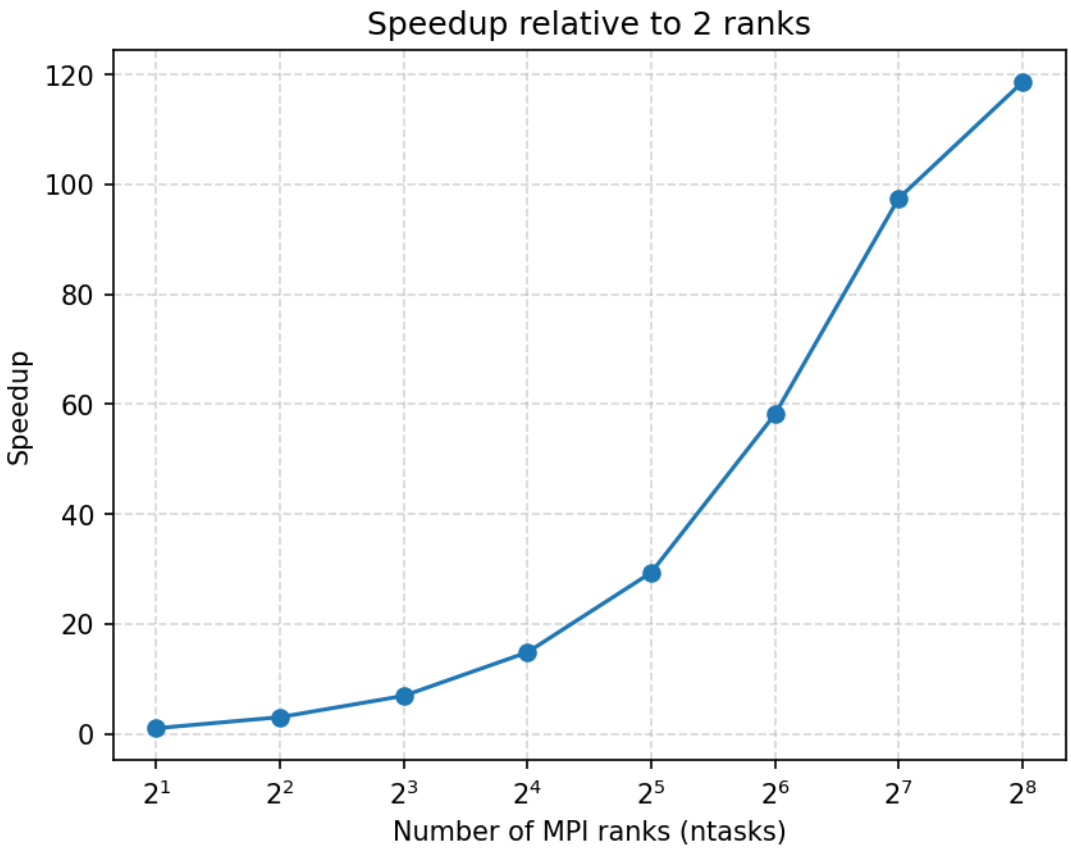

Figure 2: Speedup

1.4.1 Plots

Runtime scaling:

Speedup relative to 2 ranks:

1.4.2 T_task and per-worker task time

We compute the average time a single worker spends on a single task as:

$$T_{task} = T_{wall} \times \frac{N_{worker}}{N_{task}}$$

where N_task = 6561 is the total number of settings explored. The measured T_task values are:

ntasks	workers	T_wall (s)	T_task (s)
2	1	355.489400	0.054176
4	3	117.597000	0.053798
8	7	51.468700	0.054913
16	15	23.966300	0.054757
32	31	12.111300	0.057183
64	63	6.111600	0.058710

ntasks	workers	T_wall (s)	T_task (s)
128	127	3.646700	0.070616
256	255	2.996800	0.116501

The per-task time is fairly stable across moderate P (about 0.054–0.059 s) and increases at the highest P, indicating task-management overheads and system noise become proportionally more relevant when tasks are short.

1.4.3 Amdahl overlay on speedup

Measured speedup with theoretical Amdahl curve (mean estimated serial fraction s ˜ 0.00117):

1.5 3. Short analysis

- The smallest ntasks (2 ranks) takes ˜355 s for the sweep. Increasing ranks reduces the runtime roughly inversely (good strong scaling up to the measured range), with diminishing returns at high rank counts likely due to master-worker overhead and communication.
- Best measured runtime in this sweep was ˜2.997 s at 256 ranks.

1.5.1 3.1 Issues encountered and strategy choices

- Container vs host Python: compute nodes did not always have a consistent Python environment or `mpi4py`. To ensure reproducible runs we executed the Python MPI program inside an Apptainer container (`~/modi_images/slurm-notebook-23.11.10.sif`) mounted via NFS. This avoids subtle environment differences between login and compute nodes.
- NFS-mounted results: SLURM standard output/error files are written to the NFS mount (`/home/bxl776_ku_dk/modi_mount/ahpc/week2/results`); the parser reads those `.out` and `.err` files directly from the mount to assemble `bench_summary.csv`.
- SLURM time limits and partitions: some early submissions hit the default short time limit and produced TIMEOUT jobs. To mitigate this we (a) adjusted per-NT time limits in the submission script, and (b) selected `modi_short` for the smallest quick tests and `modi_HPPC` for larger runs. The submission script also accepts an optional `ACCOUNT` env var to avoid `InvalidAccount` pending reasons.
- Reproducibility vs turnaround: full, high-quality statistics require many repeats per configuration (we initially considered REPEAT=2000). That was too slow for iterative debugging and caused many timeouts. For the sweep we used `REPEAT=5` to get a fast, indicative picture of scaling. This allowed many quick iterations; for final production runs we recommend increasing `REPEAT` and doing multiple job submissions to collect medians.

1.5.2 3.2 Choice of task granularity and repeats

- Work per task: the algorithm explores a grid of cut settings; with `n_cuts=3` there are `3^8 = 6,561` total settings. The master assigns individual settings to workers (dynamic task-farm).
- Granularity trade-off: when per-task work is too small the overhead of sending tasks and returning results dominates (master becomes a bottleneck). We therefore added a `--repeat` parameter to artificially increase CPU work per task. For timing experiments you should tune `--repeat` so each task costs at least tens to hundreds of milliseconds to amortize communication.

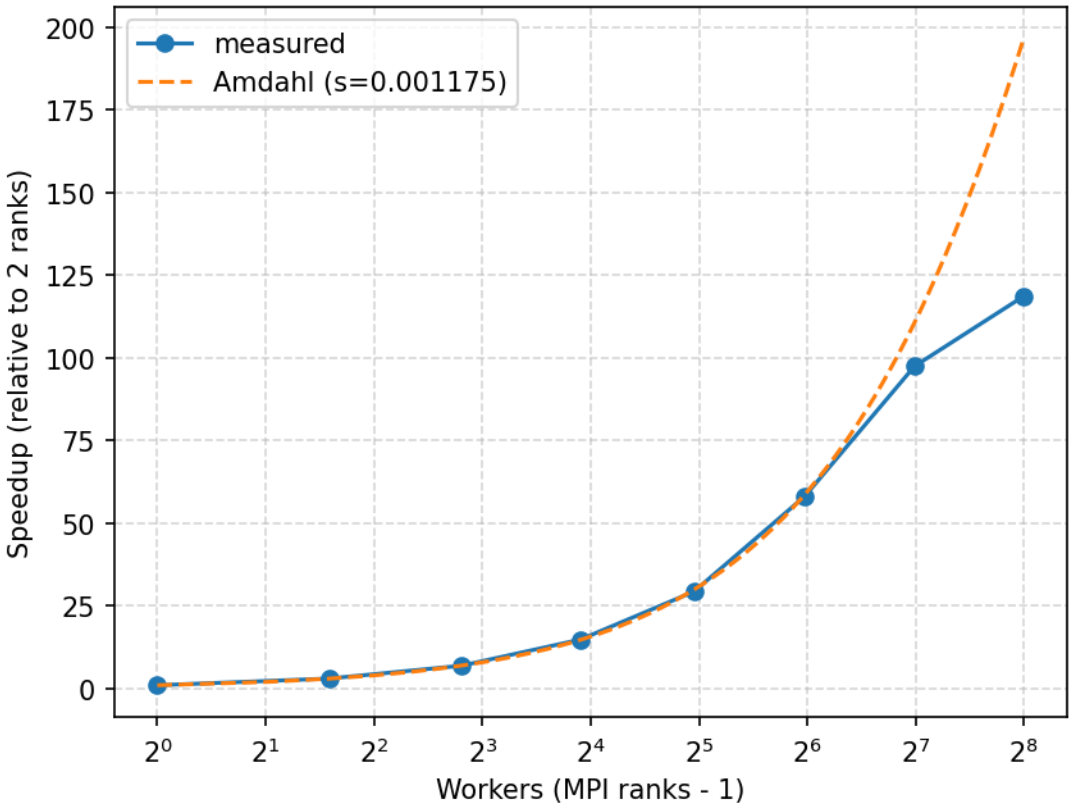

Figure 3: Speedup with Amdahl overlay

- Statistical strategy: for robust speedup estimates run multiple independent jobs per NT and take the median or mean after removing outliers. Short test runs (REPEAT=5) are useful for prototyping; final numbers should use larger REPEAT and multiple trials.

1.5.3 3.3 Scaling discussion and relation to assignment questions

- Strong scaling: the sweep measures strong scaling (fixed total work, increasing number of MPI ranks). For an ideal farm with negligible overhead, walltime should scale as 1/P (P = number of workers). In our results the runtime roughly follows this trend from 2 to ˜128 ranks, but the curve flattens as P increases due to the master's bookkeeping and communication overhead.
- Master-worker bottleneck: with a single master, task dispatch and result collection serialise parts of the workload. This effect shows up as diminishing returns for high P even though compute resources increase. For very large P, consider hierarchical masters (one master per node), batched tasks (send several settings in one message), or MPI collective approaches to reduce round-trips.
- Communication and node topology: each SLURM node provides 64 hardware threads; our submit script computed nodes_needed = ceil(ntasks / 64). Packing ranks to nodes affects network traffic and latency; running many ranks on a single node is efficient but increases intra-node contention if tasks use additional threads.
- Observed speedup: the plot uses the best (minimum) runtime per NT from the parsed SLURM outputs and computes speedup relative to the smallest NT in the sweep. This gives a conservative, optimistic view of achievable speedup; to report reliable scaling we recommend plotting median over multiple trials and adding error bars.

1.5.4 3.4 How this maps to the assignment tasks

- Task: implement a task-farm MPI program — Completed with `week2/python/task_farm_HEP.py` using a dynamic master that hands out single-setting tasks and workers that compute accuracy.
- Task: instrument and measure scaling — Completed using SLURM sweeps via `week2/python/submit_slurm_` parsing outputs into `week2/results/data/bench_summary.csv`, and plotting with `week2/results/plot_sca`
- Task: discuss bottlenecks and propose improvements — Addressed above: master-worker bottleneck, insufficient task granularity, communication overhead; recommended mitigations are hierarchical masters, batching, or hybrid MPI+threaded approaches.

1.5.5 3.5 Amdahl analysis and answers to Subtasks (a)-(c)

To directly address the assignment subtasks shown in the reference image, we computed experimental speedups from the parsed SLURM outputs and estimated the serial fraction using Amdahl's law.

Subtask a) Experimental results and presentation

- We ran a sweep of wall-clock times (T_wall) for different MPI ranks (ntasks). The master is rank 0, so workers = ntasks - 1. From the parsed CSV the measured values used in this analysis are:

ntasks	workers	T_wall (s)	speedup (relative to 2 ntasks)	serial fraction s (Amdahl)
2	1	355.489400	1.000	0.000000
4	3	117.597000	3.023	-0.003795
8	7	51.468700	6.907	0.002246
16	15	23.966300	14.833	0.000805
32	31	12.111300	29.352	0.001872
64	63	6.111600	58.166	0.001340
128	127	3.646700	97.482	0.002403
256	255	2.996800	118.623	0.004526

- The (small) negative serial fraction at 3 workers is a numerical artefact caused by measurement noise and the fact that we used the 2-rank run as our baseline for speedup; conceptually s >= 0. We include it as evidence that the serial fraction is effectively about 0 for the smaller P values.

Subtask b) Estimate serial/parallel fraction and theoretical curve

- Amdahl's law gives the speedup for P workers as:

  Speedup(P) = 1 / (s + (1 - s) / P)

- We estimated the serial fraction s from the observed speedup for each P using the rearranged Amdahl formula. The mean estimated serial fraction across the measured runs is approximately 0.00118 (median ˜ 0.00161), corresponding to a parallel fraction of roughly 99.88%.

- With s ˜ 0.0012, the theoretical speedup curve predicts near-ideal scaling across the measured P-range; for very large P the 1/P term vanishes and the speedup saturates around 1/s ˜ 848 (the Amdahl limit). In practice, other costs (master overhead, latency, OS noise) will limit scaling below that asymptote.

Subtask c) Discussion in the context of strong scaling and recommendations

- Bottlenecks observed: the master-worker approach incurs dispatch/collection overhead at the master (serial work), and communication latency becomes significant when task work is tiny. For our sweep the serial fraction is very small (˜0.1%), which explains the near-ideal scaling up to tens or low hundreds of workers. The flattening of speedup at the highest P is consistent with master's overhead and increased system noise.
- Hardware impact: node topology (64 threads per node) motivated packing many ranks per node to maximize locality. However, packing increases intra-node resource contention if tasks are multi-threaded. For massive parallelism, distributing masters hierarchically (one per node) or batching tasks reduces the number of round-trips to the global master and avoids saturating the master's network/CPU.
- Practical recommendation to experimentalists: choose task granularity and repeat factors so each dispatched task runs for a reasonably large time (tens to hundreds of ms) to amortize communication; run multiple independent trials per configuration and report median with error bars; for production-quality scaling studies increase `REPEAT` (for example 100–1000) and submit multiple jobs per NT to gather statistics.

In summary, the experimental data indicate an extremely small serial fraction and excellent strong-scaling behaviour in the tested range. The dominant limitations for further scaling are implementation-level (master dispatching) and system-level (SLURM timeouts, node availability and network noise), not the algorithmic parallelizability of the worker computation itself.

1.6 4. Notes on reproducibility

- All SLURM outputs used to build this report are stored on the NFS at `/home/bxl776_ku_dk/modi_mount/ahp`
  (files `slurm-apptainer-nt*-<jobid>.out` and `.err`).
- The parsing script used to build the CSV is `week2/results/parse_slurm_outputs.py` and the plotting script is `week2/results/plot_scaling.py`.

1.7 5. Next steps

- Optionally extend the sweep to 512 ranks (requires partition availability and longer time limits).
- Run multiple repeats per NT and collect median/mean to reduce noise from cluster variability.

Report prepared for AHPC | October 13, 2025