Overview

This document outlines a specific use-case for the proposed methodology to a Genome analysis workflow (chunk size = 80,000). Due to the nature of this application it requires a different degree of parallelism for different task. At first the proposed method is shown on the `individual` tasks and later a similar analysis is done for the `frequency` tasks.

At first the baseline, which does not rely on any parallel processing that is tuned later, (see Figure 1) is run 3 times, then the average is calculated.

The tests were executed on my local machine (for now), results depend on machine used, but the trend should stay the same.

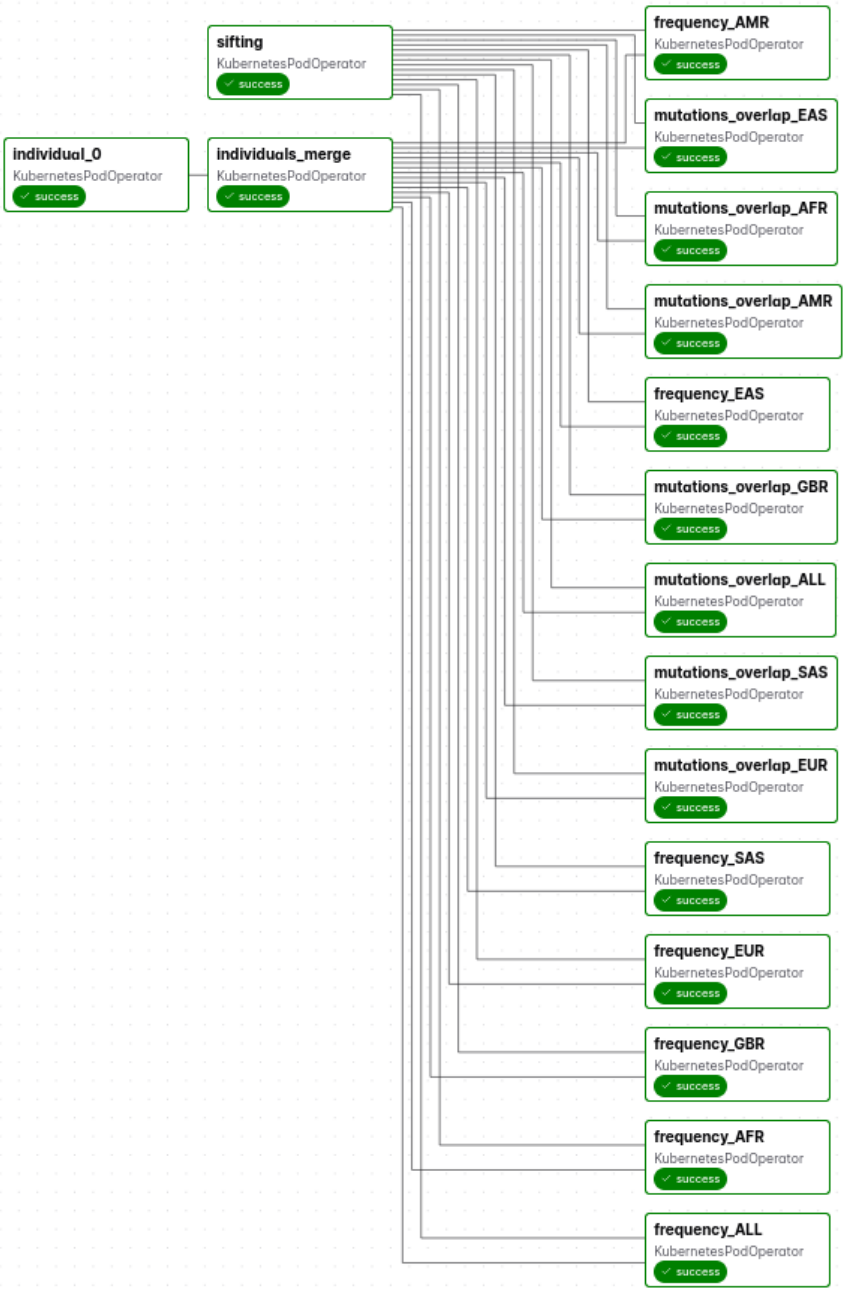


Figure 1: DAG structure of baseline

Run	Total Time (s)	Time `individuals` (s)	Time `frequency` (s)
1	451	295	128
2	448	291	128
3	447	290	130
Average	**449**	**292**	**129**

Table 1: Results of baseline

Individuals Analysis

This part of the analysis only tunes and examines the parameters concerning the `individual` tasks.

Step 1 (Perfect World)

In this step, Amdahl's Law is used to predict the execution time, as shown in Equation 1.

$$T(s) = (1-p) \cdot T + \frac{p}{s} \cdot T \tag{1}$$

where T is total execution time, p is the parallelizable portion and s defines the degree of parallelism.

In the current version it implicitly assumes $p = 1$. In the analysis later it can be seen, that this is not too bad for `individual` tasks, but not accurate for the `frequency` tasks.

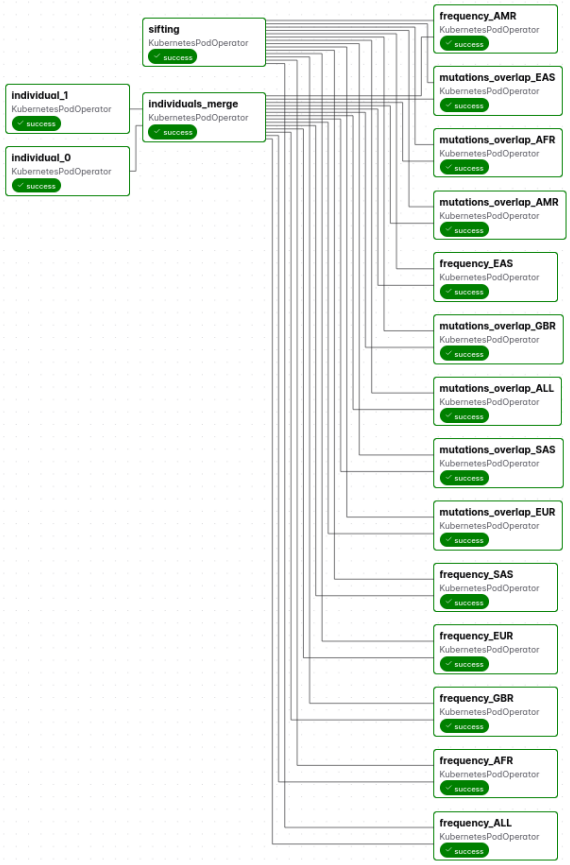
This leads to the following execution time predictions (only focuses on `individual` tasks, since other tasks are stable and not changed from baseline).

s	T(s)
2	146
4	73
6	49
8	37

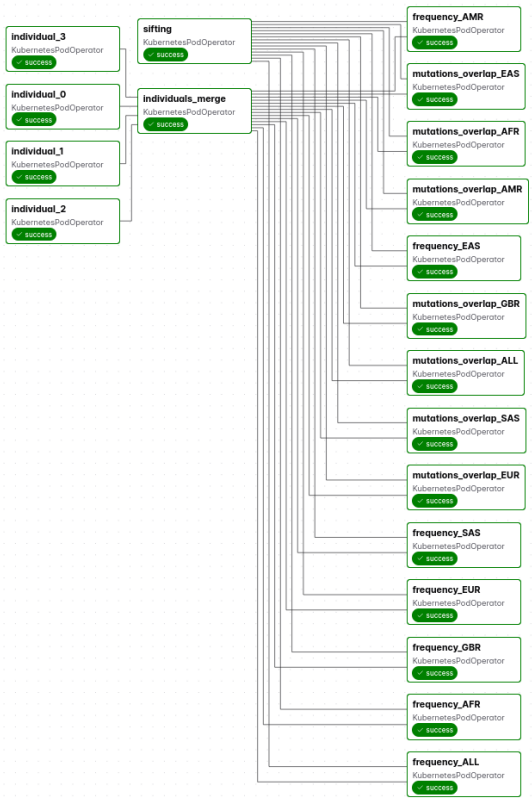
Table 2: Predictions for T(s)

Step 2 (Reality)

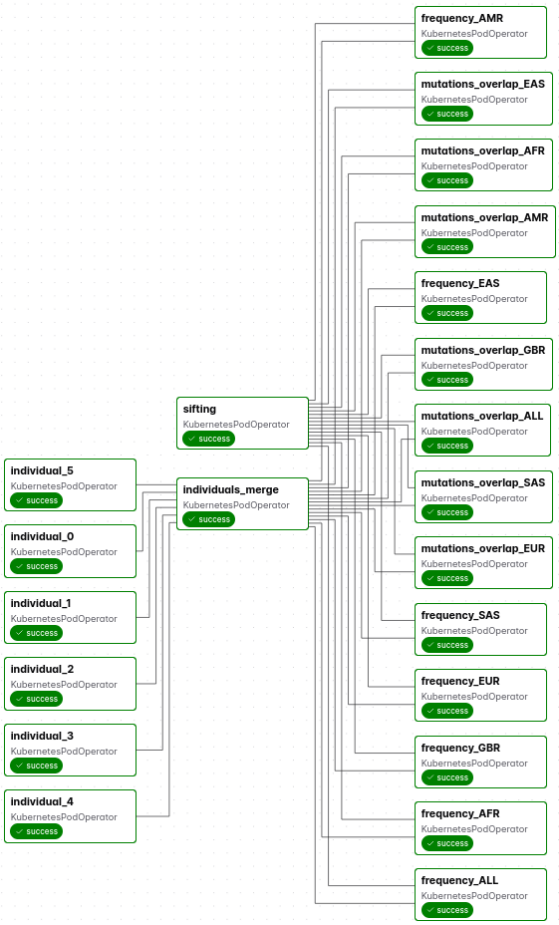
Here, the DAG was executed 3 times per configuration to get an average execution time. Since parallel tasks have varying durations, the slowest task was recorded as the stage time. Figure 2 illustrates how the DAG structure changes as the parallelism for individual tasks increases. Detailed results can be found in Table 3, which are visualized in Figure 3.



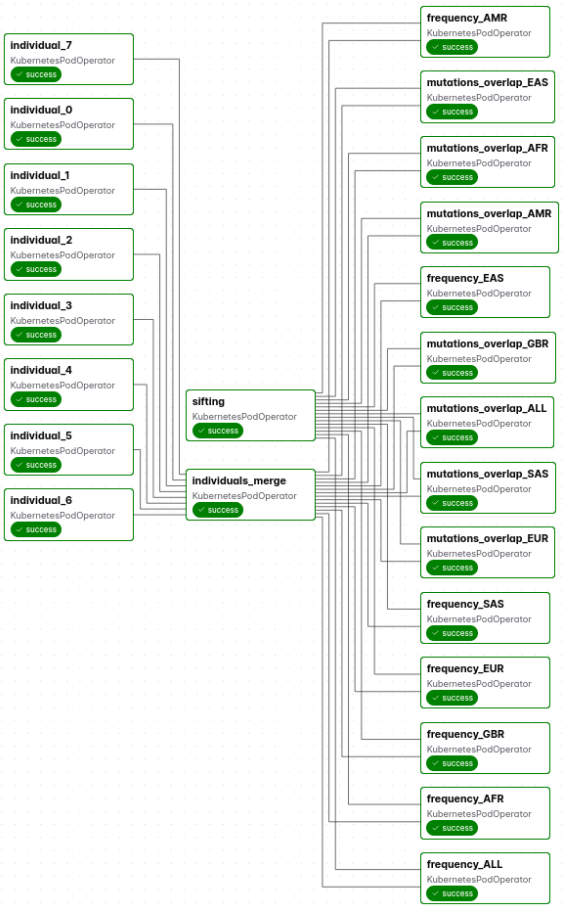
(a) Structure for $s = 2$ workers.



(b) Structure for $s = 4$ workers.

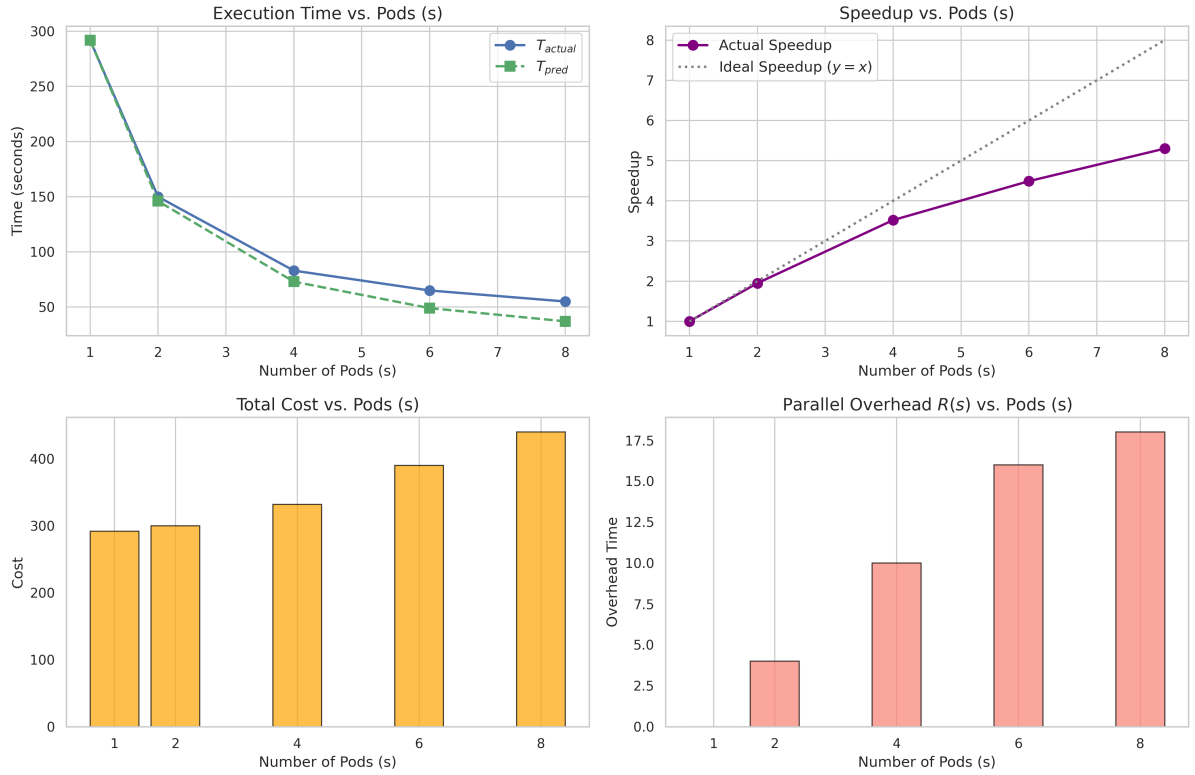


(c) Structure for $s = 6$ workers.

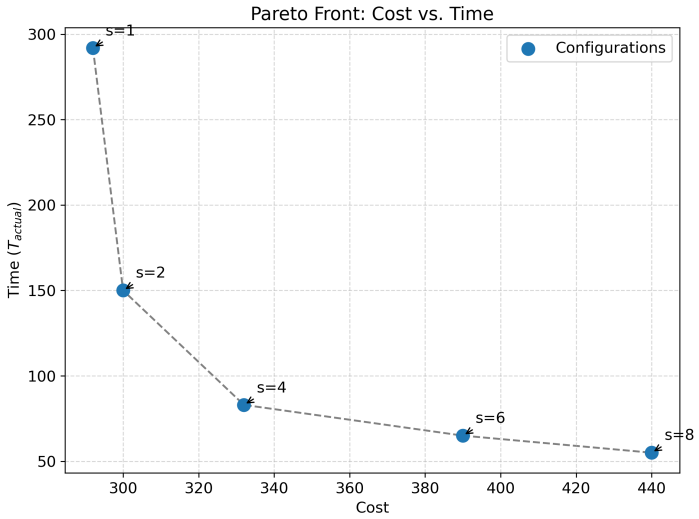


(d) Structure for $s = 8$ workers.

Figure 2: Evolution of the DAG structure when scaling the "individuals" task from 2 to 8 workers.



(a) Performance metrics: Execution Time, Speedup, Cost analysis and Overhead



(b) Pareto Front showing the trade-off between Cost and Time.

Figure 3: Visual analysis of the parallel execution results.

s	T_{actual}	$T_{pred}(s)$	R(s)	Cost	Speedup (task)
1	292	-	-	292	1
2	150	146	4	300	1.95
4	83	73	10	332	3.52
6	65	49	16	390	4.49
8	55	37	18	440	5.3

Table 3: Results of parallel version

Frequency Analysis

This section focuses on tuning and examining the parameters specifically for the `frequency` tasks.

Unlike the previous step, the `frequency` stage processes multiple distinct populations simultaneously. Experimental observation reveals a disparity in execution times: the task processing the `ALL` population is computationally more intensive than the other populations, which exhibit shorter and similar runtimes.

To optimize resource usage, we classify these tasks into two distinct scaling groups:

- **Class A (Heavy):** The `ALL` population task. Due to its long duration, this is scaled independently to reduce the critical path.
- **Class B (Light):** All other populations (e.g., *EUR, AMR, AFR, etc.*). These are grouped and scaled together as their computational footprint is uniform.

Consequently, the following analysis separates the scaling effects for the `ALL` population from the rest.

Since my machine/the cluster don't have enough resources a very simplified version of the DAG is used. It also removed the `individual`, `sifting` and `individuals_merge`, since they don't have any effect on the execution time of the `frequency` tasks, see Figure 4.

Note, this analysis assumes p of 1, due to simplicity (hard to automatically detect) and additional overhead is caught by the residual R.

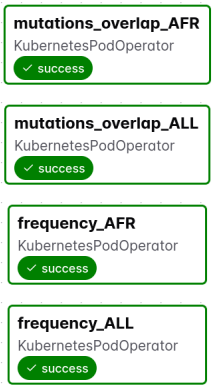


Figure 4: DAG structure of simplified version

Run	Time `frequency_ALL` (s)	Time `frequency_AFR` (s)
1	98	52
2	99	54
3	105	60
Average	**101**	**55**

Table 4: Results of simplified baseline

Step 1 (Perfect World)

Using Amdahl's Law, we get the following predictions:

s	$T_{all}(s)$	$T_{afr}(s)$
2	50	28
3	34	18
4	25	14

Table 5: Predictions for T(s)

Step 2 (Reality)

Here, the DAG was also executed 3 times per configuration to get an average execution time, Figure 5 is showing how the DAG structure changes for different configurations.

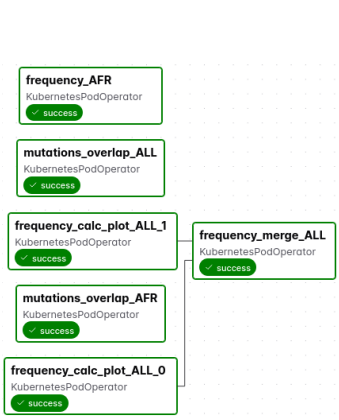
Note that `frequency_AFR` is representing all other populations, and time is sum of duration for all `frequency` tasks.

Workers		ALL Task			AFR Task			Global	
s_{all}	s_{afr}	T	T_{pred}	R	T	T_{pred}	R	Cost	Speedup
1	1	**101**	-	-	55	-	-	156	1.00
2	1	**72**	50	22	64	55	9	201	1.4
2	2	**76**	50	26	57	28	29	251	1.33
3	1	60	34	26	**66**	55	11	232	1.53
3	2	**65**	34	31	60	28	32	294	1.55
3	3	**67**	34	33	50	14	36	324	1.51
4	1	54	25	29	**69**	55	14	265	1.46
4	2	46	25	21	**50**	28	22	258	2.02
4	3	**50**	25	25	42	18	24	277	2.02
4	4	**48**	25	23	37	14	23	297	2.1

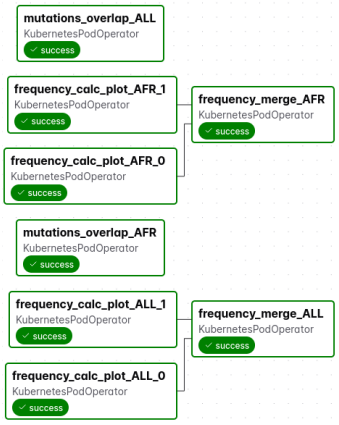
Table 6: Detailed performance metrics for Frequency tasks. *R* denotes the Residual.

Note the correlation between `frequency_ALL` and `frequency_AFR`, in the second row, only ALL is scaled, and AFR is using only one pod, so it could be expected that the time of `freuqency_AFR` stays the same, but due to resource contention (inference), it gets higher, so it takes special care when dealing with concurrent tasks that are scaled independently.

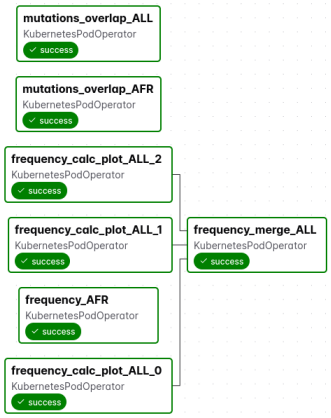
The results are also visualized in Figure 6.



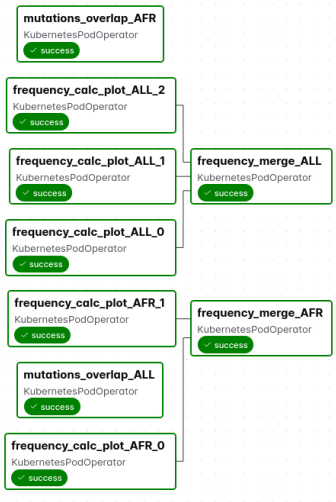
(a) $s_{all} = 2; s_{afr} = 1$



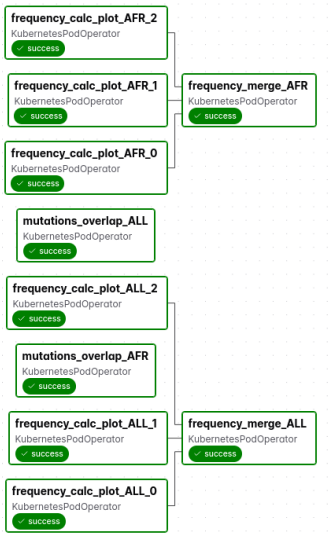
(b) $s_{all} = 2; s_{afr} = 2$



(c) $s_{all} = 3; s_{afr} = 1$



(d) $s_{all} = 3; s_{afr} = 2$

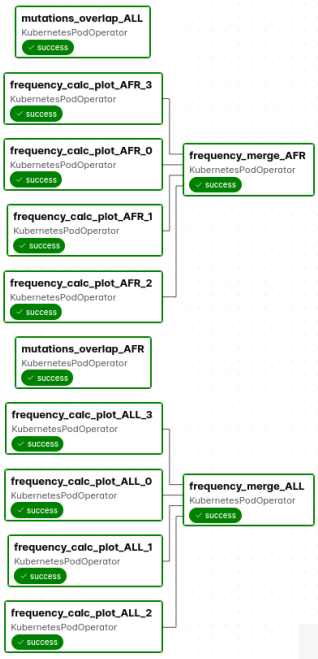


(e) $s_{all} = 3; s_{afr} = 3$

(f) $s_{all} = 4; s_{afr} = 1$

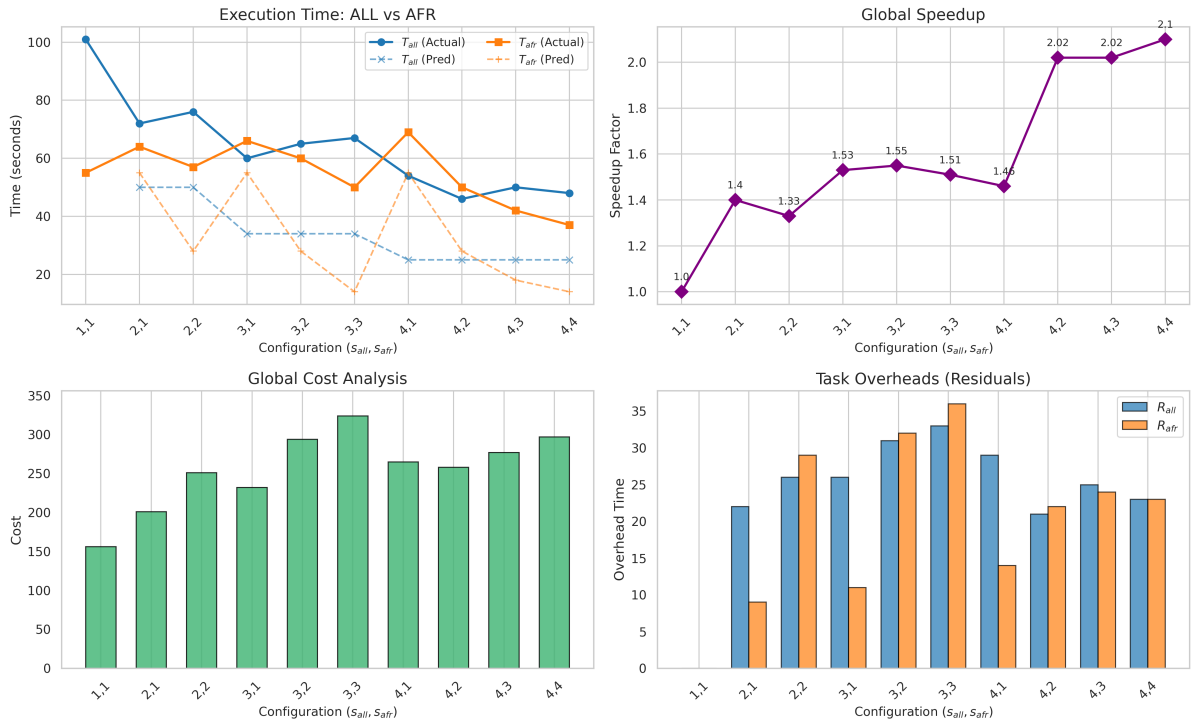
(g) $s_{all} = 4; s_{afr} = 2$

(h) $s_{all} = 4; s_{afr} = 3$

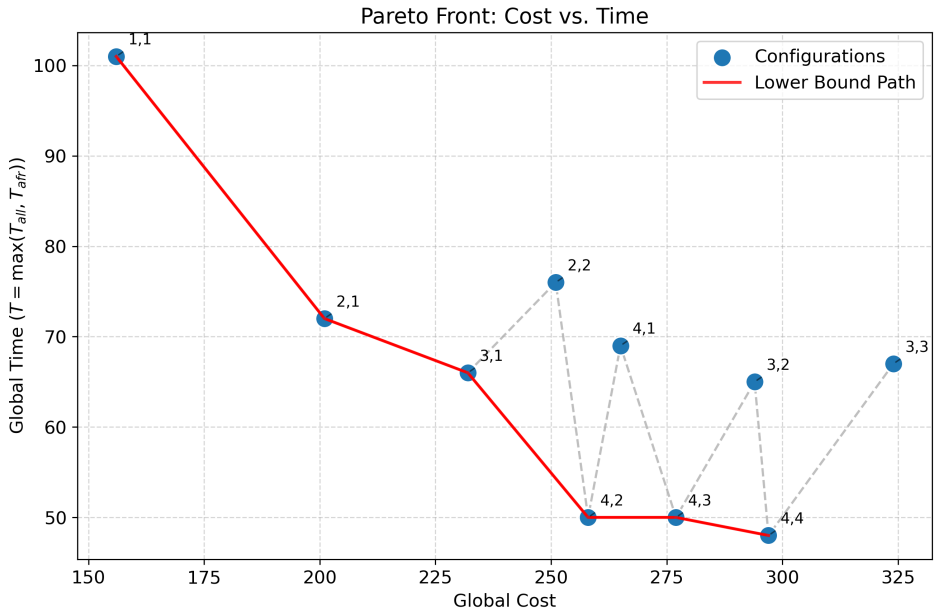


(i) $s_{all} = 4; s_{afr} = 4$

Figure 5: DAG structures of different configurations.



(a) Performance metrics: Execution Time, Speedup, Cost analysis and Overhead



(b) Pareto Front showing the trade-off between Cost and Time.

Figure 6: Visual analysis of the parallel execution results.

Optimized Approach for p

To improve the accuracy of the residual (R) the parameter p can be observed through execution times and not blindly set to $p = 1$.

$$
\begin{aligned}
T(s) &= (1-p)\cdot T(1) + \frac{p}{s}\cdot T(1) \\
\frac{T(s)}{T(1)} &= (1-p) + \frac{p}{s} \\
\frac{T(s)}{T(1)} - 1 &= -p + \frac{p}{s} \\
\frac{T(s)}{T(1)} - 1 &= \frac{p}{s} - p \\
\frac{T(s) - T(1)}{T(1)} &= p \cdot \left(\frac{1}{s} - 1\right) \\
\frac{T(s) - T(1)}{T(1)} &= p \cdot \left(\frac{1-s}{s}\right) \\
\frac{\frac{T(s)-T(1)}{T(1)}}{\frac{1-s}{s}} &= p \\
\frac{T(s) - T(1)}{T(1)} \cdot \frac{s}{1-s} &= p \\
\left(\frac{T(s)}{T(1)} - \frac{T(1)}{T(1)}\right) \cdot \frac{s}{1-s} &= p \\
\left(\frac{T(s)}{T(1)} - 1\right) \cdot \frac{s}{1-s} &= p \\
\left(1 - \frac{T(s)}{T(1)}\right) \cdot \frac{s}{s-1} &= p \\
\frac{\left(1 - \frac{T(s)}{T(1)}\right) \cdot s}{s-1} &= p
\end{aligned}
$$

resulting in

$$
p_{obs} = \frac{s \cdot \left(1 - \frac{T(s)}{T(1)}\right)}{s-1} = \frac{s}{s-1} \cdot \left(1 - \frac{T(s)}{T(1)}\right)
$$

This requires the baseline time, and a parallel execution from which p can be inferred.

To test this approach, the results of Table 3 and Table 6 are used and compared to check if it performs better or not.

Individual Comparison

It uses the same baseline and one execution with $s = 4 \rightarrow T(4) = 82$, to infer P_{obs} and then the table is filled, which results in $P_{obs} = 0.96$.

As visible in Table 8 and Figure 7 the "fitted" model predicts the residual better.

s	T(s)
2	152
4	82
6	58
8	47

Table 7: Predictions for T(s) with p_{obs}

		Naive Model ($p = 1$)		**Fitted Model** ($p \approx 0.96$)	
s	T_{actual}	T_{pred}	R	T_{pred}	R
1	292	-	-	-	-
2	150	146	4	152	-2
4	83	73	10	82	1
6	65	49	16	58	7
8	55	37	18	47	8

Table 8: Comparison of Naive ($p = 1$) vs. Fitted (p_{obs}) prediction models. Note that the Fitted model reduces the unexplained Residual (R), isolating the true cluster overhead.

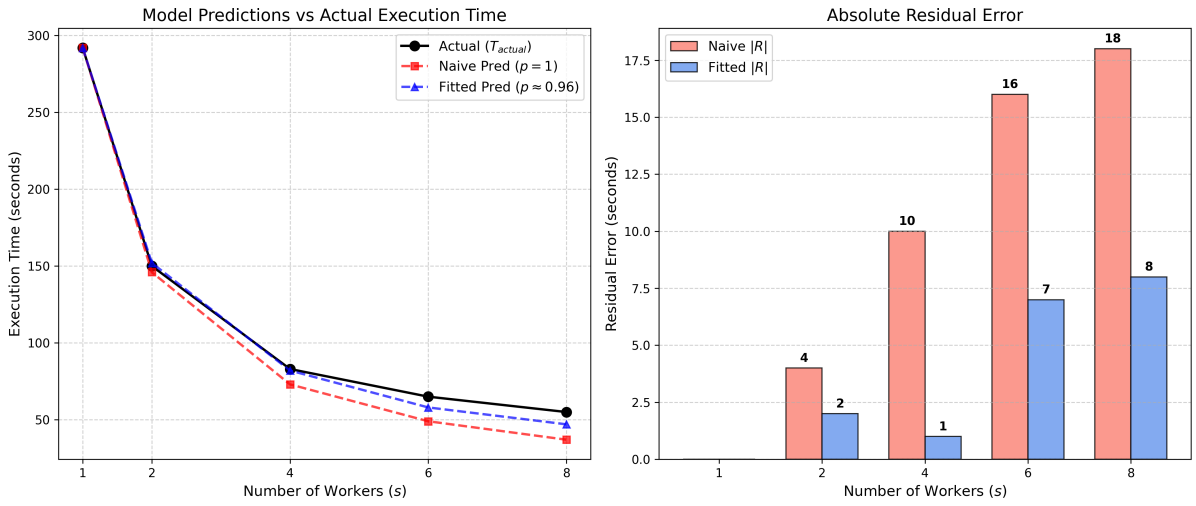


Figure 7: Differences between naive and fitted residual predictions

Frequency Comparison

It also uses the same baseline and one execution and an execution with $s = 4$, since ALL and AFR are scaled differently it uses two different p values. This results in: $p_{obs}^{all} = 0.61$ and

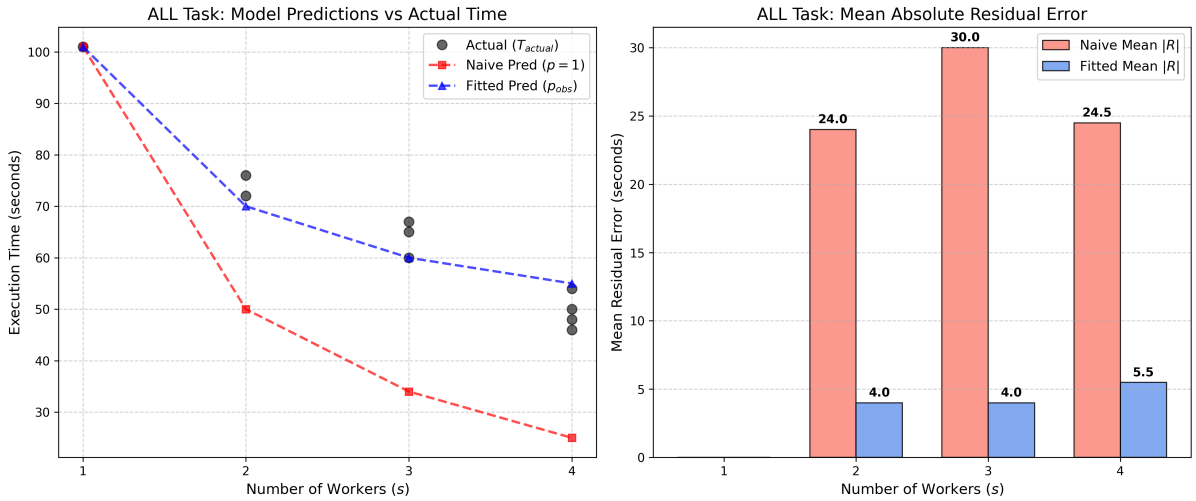
$p_{obs}^{afr} = 0.58$. These values were taken from executions where one task had s=4 and the other s=1.

s	$T_{all}(s)$	$T_{afr}(s)$
2	70	39
3	60	34
4	55	31

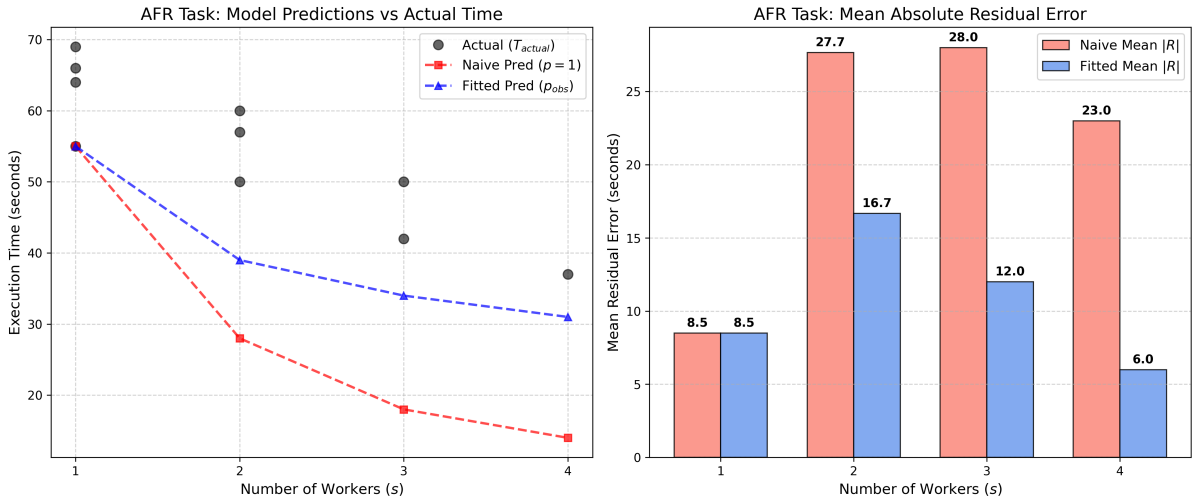
Table 9: Predictions for T(s) with p_{obs}^{all} and p_{obs}^{afr}

Workers		**ALL Task**					**AFR Task**				
			Naive		**Fitted** ($p \approx 0.61$)			**Naive**		**Fitted** ($p \approx 0.58$)	
s_{all}	s_{afr}	T_{act}	T_{pred}	R	T_{pred}	R	T_{act}	T_{pred}	R	T_{pred}	R
1	1	**101**	-	-	-	-	55	-	-	-	-
2	1	**72**	50	22	70	2	64	55	9	55	9
2	2	**76**	50	26	70	6	57	28	29	39	18
3	1	60	34	26	60	0	**66**	55	11	55	11
3	2	**65**	34	31	60	5	60	28	32	39	21
3	3	**67**	34	33	60	7	50	14	36	34	16
4	1	54	25	29	55	-1	**69**	55	14	55	14
4	2	46	25	21	55	-9	**50**	28	22	39	11
4	3	**50**	25	25	55	-5	42	18	24	34	8
4	4	**48**	25	23	55	-7	37	14	23	31	6

Table 10: Comparison of Naive ($p = 1$) vs. Fitted (p_{obs}) prediction models. The fitted model significantly reduces residuals for the primary ALL task. For the AFR task, persistent residuals in rows where $s_{all} > s_{afr}$ quantify the cross-task interference (Noisy Neighbor effect) caused by resource contention.



(a) Comparison of `frequency_ALL`



(b) Comparison of `frequency_AFR`

Figure 8: Comparison of Residual for naive and fitted model

Open Questions & Possible Improvements

Dynamic Estimation of p_{obs}

The observed parallelizable portion (p_{obs}) is currently estimated after the first parallel execution (may be biased by current cluster state). To improve robustness, p_{obs} could be treated as a dynamic variable, updated after each execution k with "Online Empirical Average with Learning Rate":

$$p_{obs}^{(k+1)} = \alpha \cdot p_{obs}^{(k)} + (1-\alpha) \cdot p_{current} \quad (\alpha \in \mathbb{R} : 0 < \alpha \leqslant 1)$$

where α is the learning rate.

Input Size

Since the execution time of workflows highly depends on the input size, the model needs to be aware of it. Using absolute units (e.g. "seconds per MB of data") is very problematic (e.g. number of genome chunks vs. number of images (IISAS)).

The following is based on the assumption that the execution time scales proportionally with the input size ($T \propto N$).

This uses a relative scaling factor γ. If baseline had input size N_{base} and current run had input size N_{curr} it has a scaling factor:

$$\gamma = \frac{N_{curr}}{N_{base}}$$

The input size could be measured from download size? or user input

The time is presumed to scale linearly with γ:

$$T_{Amdahl}(s,\gamma) = \underbrace{C_{startup}}_{\text{Fixed Overhead}} + \gamma \cdot ((1-p_{obs}) \cdot T(1)) + \gamma \cdot \left(\frac{p_{obs}}{s} \cdot T(1)\right)$$

$$P(s,\gamma) = T_{Amdahl}(s,\gamma) + R(s,\gamma)$$

This relies on knowing the constant overhead pod start-up time.

Resource Contention

A workflow is influenced by the total load of the cluster (see Table 6), the model needs to account for this interference. Optimizing a workflow in isolation might lead to performance degradation caused by other concurrent workflows.

It requires an additional variable measuring the cluster load, which aggregates the parallelism of all other active tasks:

$$L_{cluster} = \sum_{i \neq j} s^i_{task}$$

The residual function R needs to be expanded, such that GP learn correlation between high cluster pressure and increased execution time:

$$P(s, L_{cluster}) = T_{Amdahl}(s) + R(s, L_{cluster})$$

A questions still remains, when to calculate the cluster load, for it to be most accurate it needs to be right before scaling a task.

Unified Residual Model

To capture the interplay between infrastructure, input size and cluster pressure, it can be combined into a single multi-dimensional Gaussian Process, which is defined as:

$$P(s, \gamma, L_{cluster}) = T_{Amdahl}(s, \gamma) + R(\mathbf{x})$$

where:

- T_{Amdahl} is theoretical time accounting for input size:

$$T_{Amdahl}(s, \gamma) = \underbrace{C_{startup}}_{\text{Fixed Overhead}} + \gamma \cdot ((1 - p_{obs}) \cdot T(1)) + \gamma \cdot \left(\frac{p_{obs}}{s} \cdot T(1) \right)$$

- $R(\mathbf{x})$ is the learned residual, where $\mathbf{x}$ is defined as:

$$\mathbf{x} = [s, \gamma, L_{cluster}]$$

Proposed Algorithm

Based on the analysis and derivations above, the algorithm is shown (might still change). The algorithm works on a Task-Specific Basis, since specific tasks in the DAG exhibit unique computational characteristics. It consists of two Phases:

Phase 1: Initialization

Before optimizations are possible, the system needs a baseline for the workflow structure.

1. **Measure Baseline ($s = 1$):** Execute workflow sequentially, record execution time $T_{base}(1)$, the reference input size N_{base} and the cluster load during execution $L_{cluster}$
2. **Measure Overhead ($C_{startup}$):** Estimate fixed infrastructure overhead (e.g., Pod spin-up time). This is treated as a constant.
3. **Initial Parallel Execution:** Execute the workflow once with a moderate parallelism (e.g., $s = 4$). Record the observed time T_{obs} and the cluster load L_{probe}.
4. **Derive Initial p_{obs}:** Calculate the initial parallelizable fraction using the inverse Amdahl function:
$$p_{obs}^{(0)} = \frac{s}{s-1} \cdot \left(1 - \frac{T_{obs} - C_{startup}}{T_{base}(1)} \right)$$
5. **Initialize GP:** Initialize the Gaussian Process with the data point from the probe execution. The initial training pair $(\mathbf{x}, y)$ is:
$$\mathbf{x} = [s = 4, \gamma = 1, L_{probe}], \quad y = T_{obs} - T_{Amdahl}(4, 1)$$

Phase 2: Optimization Loop

For every new (k) requested execution the following steps are performed:

1. **Model Retrieval:** Retrieve the specific parameters (p_{obs}, N_{base}, ...) and the (trained) GP from a registry (e.g. database, or similar)

2. **Context Gathering:**
   - Measure current Input Size (N_{curr}) and calculate scaling factor $\gamma = N_{curr}/N_{base}$
   - snapshot current Cluster Load $L_{cluster} = \sum_{i \neq j} s^i_{task}$

   An open question which remains is, how to handle the case where the first task can already be parallelized, possible strategies:
   - Insert dummy task, that downloads the same data → not useful if input data is very big
   - rely on user input
   - force specific values (e.g. $\gamma = 1$)
3. **Prediction:** For a given candidate range $s \in [1, S_{max}]$ calculate the predicted execution time:

$$P(s, \gamma, L_{cluster}) = \underbrace{C_{startup} + \gamma \cdot ((1 - p_{obs}) \cdot T_{base}(1)) + \gamma \cdot \left(\frac{p_{obs}}{s} \cdot T_{base}(1)\right)}_{T_{Amdahl(s,\gamma)}} + \underbrace{GP(s, \gamma, L_{cluster})}_{R(\mathbf{x})}$$

4. **Decision:** Select optimal S_{opt} that minimizes the Score function (balancing Cost vs. Speed):

$$s_{opt} = \underset{s}{\operatorname{argmin}} \left(P(s, \gamma, L_{cluster})^a \cdot \text{Cost}(s)\right) \qquad \text{original formula used Cost(p), check!}$$

   where a is the urgency factor ($a = 0.5 \rightarrow$ save money; $a = 1 \rightarrow$ balance; $a = 2 \rightarrow$ save time)
5. **Execution & Feedback:** Execute task with s_{opt} and measure actual execution time T_{actual}
6. **Model Update:**
   - **Calculate $p_{current}$:** Derive the parallelizable fraction from the observed time, accounting for input scaling (γ) and fixed overhead ($C_{startup}$):

$$p_{current} = \frac{s}{s-1} \cdot \left(1 - \frac{T_{actual} - C_{startup}}{\gamma \cdot T_{base}(1)}\right)$$

   - **Update p_{obs}:**

$$p^k_{obs} = \alpha \cdot p^{k-1}_{obs} + (1 - \alpha) \cdot p_{current}$$

   - **Update GP:** Add new observation:

$$\mathbf{x} = [s_{opt}, \gamma, L_{cluster}], y = T_{actual} - T_{Amdahl}(s_{opt}, \gamma)$$

     to the Gaussian Process training

Model Maintenance: Rolling Window Strategy

To maintain low computational overhead and ensure adaptability to "Concept Drift" (e.g., permanent shifts in cluster performance, hardware upgrades, or changes in baseline load), the Gaussian Process is restricted to a fixed-size history buffer W (e.g., $W = 500$).

Instead of training on the potentially infinite set of all historical executions $\mathcal{H}_{all}$, the model uses a localized training set $\mathcal{H}_{active}$ containing only the W most recent observations:

$$\mathcal{H}_{active}^{(k)} = \{(\mathbf{x}_i, y_i) \mid i \in [\max(0, k-W), k]\} \tag{2}$$

This ensures that the inference complexity remains constant at $\mathcal{O}(W^3)$ rather than growing cubically with total usage time ($\mathcal{O}(k^3)$), while automatically discarding obsolete infrastructure data that may no longer reflect the current system state.

Refined Algorithm

The optimization loop works on a Task-Specific Basis. For every new execution request k:

1. **Context Gathering:** Measure the current input size N_{curr} to derive the scaling factor $\gamma = N_{curr}/N_{base}$, and snapshot the current cluster load $L_{cluster}$.
2. **Probabilistic Prediction:** Using the GP trained on the active history $\mathcal{H}_{active}^{(k-1)}$, predict the residual statistics for a candidate parallelism s:

   $$\mu_{res}, \sigma_{res}^2 = GP(s, \gamma, L_{cluster} \mid \mathcal{H}_{active}^{(k-1)})$$

   The total predicted execution time is modeled as a distribution:

   $$P(s) \sim \mathcal{N}\left(T_{Amdahl}(s, \gamma) + \mu_{res}, \sigma_{res}^2\right)$$

3. **Acquisition & Decision:** Select the optimal parallelism s_{opt} that minimizes the objective function. To account for uncertainty, we minimize the *Expected Cost* rather than a point estimate:

   $$s_{opt} = \underset{s}{\operatorname{argmin}} \left(\mathbb{E}[P(s)]^a \cdot \mathrm{Cost}(s)\right)$$

   (Note: More advanced Acquisition Functions like Expected Improvement (EI) can be substituted here if exploration is desired.)
4. **Execution:** Execute the task with s_{opt} and measure the actual execution time T_{actual}.
5. **Rolling Update:**
   - **Compute Residual:** Calculate the discrepancy between theory and reality:

     $$y_{new} = T_{actual} - T_{Amdahl}(s_{opt}, \gamma)$$

   - **Construct Data Point:** Form the feature vector $\mathbf{x}_{new} = [s_{opt}, \gamma, L_{cluster}]$.
   - **Update History:** Append the new observation to the buffer:

     $$\mathcal{H}_{active}^{(k)} \leftarrow \mathcal{H}_{active}^{(k-1)} \cup \{(\mathbf{x}_{new}, y_{new})\}$$

- **Pruning:** If the buffer size $|\mathscr{H}_{active}^{(k)}| > W$, remove the oldest data point $(\mathbf{x}_{k-W}, y_{k-W})$ to maintain the window size.
- **Retrain:** Re-optimize the Gaussian Process hyperparameters (length-scales and noise variance) using the updated $\mathscr{H}_{active}^{(k)}$.