```
def ar_garch_model_student_t_multi_asset_partial_pooling(
    returns,  # shape [batch_size, max_T]
    lengths,  # shape [batch_size,] giving lengths per asset
    args,  # expects args.jit, and other control flags/settings
    prior_predictive_checks: bool = False,
    device: torch.device = torch.device("cpu"),
    batch_size=None,  # support for mini-batched learning
):
    """
    Parallel AR(1)-GARCH(1,1) model with Student-T innovations
    with partial pooling/hierarchical structure for parameters
    (each asset draws its own set of parameters from global hyperpriors).

    Key modification: For (0,1) parameters (such as alpha_beta_sum, alpha_frac),
    use proper Beta partial pooling, not clamped Normals.

    Designed for Pyro JIT & masking. Can handle assets with differing available lengths.
    """

    # Move to correct device
    returns = returns.to(device)
    lengths = lengths.to(device)
    num_assets, max_T = returns.shape

    # ------------------- HIERARCHICAL PRIORS (hyperpriors for group/global parameters) ----------
----------

    # 1) GARCH omega (positive, not [0,1])
    omega_mu = pyro.sample("omega_mu", dist.Exponential(torch.tensor(1.0, device=device)))
    omega_sigma = pyro.sample("omega_sigma", dist.Exponential(torch.tensor(1.0,
device=device)))

    # 2) alpha_beta_sum ~ Beta(a, b) hierarchy
    ab_sum_a_hyper = pyro.sample(
        "ab_sum_a_hyper", dist.Exponential(torch.tensor(2.0, device=device))
    )
    ab_sum_b_hyper = pyro.sample(
        "ab_sum_b_hyper", dist.Exponential(torch.tensor(2.0, device=device))
    )
    # 3) alpha_frac ~ Beta(a, b) hierarchy
    ab_frac_a_hyper = pyro.sample(
        "ab_frac_a_hyper", dist.Exponential(torch.tensor(2.0, device=device))
    )
    ab_frac_b_hyper = pyro.sample(
        "ab_frac_b_hyper", dist.Exponential(torch.tensor(2.0, device=device))
    )

    # 4) AR(1) phi (unconstrained)
    phi_mu = pyro.sample(
        "phi_mu", dist.Normal(torch.tensor(0.0, device=device), torch.tensor(1.0, device=device))
    )
    phi_sigma = pyro.sample("phi_sigma", dist.Exponential(torch.tensor(1.0, device=device)))

    # 5) Initial GARCH sigma (positive)
```

```
sigma_init_mu = pyro.sample(
    "sigma_init_mu", dist.Exponential(torch.tensor(10.0, device=device))
)
sigma_init_sigma = pyro.sample(
    "sigma_init_sigma", dist.Exponential(torch.tensor(10.0, device=device))
)

# 6) Degrees of freedom for Student-T (constrained to >2)
df_mu = pyro.sample("df_mu", dist.Exponential(torch.tensor(1.0, device=device)))
df_sigma = pyro.sample("df_sigma", dist.Exponential(torch.tensor(1.0, device=device)))

# Global decay parameter for time-weighting - Values in (0,1): how rapidly to forget the past
lambda_decay = pyro.sample(
    "lambda_decay",
    dist.Beta(torch.tensor(2.0, device=device), torch.tensor(2.0, device=device)),
)

# ---------------------- PER-ASSET PARAMETERS  -------------------------
with ignore_jit_warnings():
    with pyro.plate("assets", num_assets, batch_size, dim=-2) as batch:
        batch_size_local = batch.shape[0] if batch is not None else num_assets

        asset_lengths = lengths[batch]  # [batch_size_local]
        asset_returns = returns[batch]  # [batch_size_local, max_T]

        # ASSET-SPECIFIC parameters from the group/hyperpriors (partial pooling!)

        # GARCH omega; positive, partial pooling via Normal
        garch_omega = pyro.sample(
            "garch_omega", dist.Normal(omega_mu, omega_sigma).expand([batch_size_local])
        )
        garch_omega = garch_omega.clamp(min=1e-4)  # safety

        # --- MODIFIED: Proper Beta partial pooling for alpha_beta_sum in (0,1) ---
        alpha_beta_sum = pyro.sample(
            "alpha_beta_sum",
            dist.Beta(ab_sum_a_hyper, ab_sum_b_hyper).expand([batch_size_local]),
        )

        # --- MODIFIED: Proper Beta partial pooling for alpha_frac in (0,1) ---
        alpha_frac = pyro.sample(
            "alpha_frac", dist.Beta(ab_frac_a_hyper, ab_frac_b_hyper).expand([batch_size_local])
        )

        # reparameterize
        garch_alpha = alpha_beta_sum * alpha_frac
        garch_beta = alpha_beta_sum * (1.0 - alpha_frac)

        # AR(1) phi (no change)
        phi = pyro.sample("phi", dist.Normal(phi_mu, phi_sigma).expand([batch_size_local]))

        # Initial GARCH sigma (positive)
        sigma_init = pyro.sample(
            "garch_sigma_init",
```

```
    dist.Normal(sigma_init_mu, sigma_init_sigma).expand([batch_size_local]),
)
sigma_init = sigma_init.clamp(min=1e-4)  # safety

# Student-T dof (must be >2)
df = pyro.sample(
    "degrees_of_freedom", dist.Normal(df_mu, df_sigma).expand([batch_size_local])
)
df = df.clamp(min=2.05)

# Per-asset likelihood weight
asset_weight = pyro.sample(
    "asset_weight",
    dist.Beta(
        torch.tensor(2.0, device=device), torch.tensor(2.0, device=device)
    ).expand([asset_returns.shape[0]]),
)

# Vectorized GARCH/AR recursion
sigma_prev = sigma_init  # [batch_size_local]
e_prev = torch.zeros(batch_size_local, device=device)
r_prev = torch.zeros(batch_size_local, device=device)

for t in pyro.markov(
    range(max_T if getattr(args, "jit", False) else asset_lengths.max().item())
):
    valid_mask = t < asset_lengths  # [batch_size_local]
    decay_exponent = max_T - t - 1

    if t == 0:
        sigma_t = sigma_prev
        mean_t = torch.zeros_like(sigma_prev)
    else:
        sigma_t = torch.sqrt(
            garch_omega + garch_alpha * (e_prev**2) + garch_beta * (sigma_prev**2)
        )
        mean_t = phi * r_prev

    obs = None
    if not prior_predictive_checks and asset_returns is not None:
        obs = asset_returns[:, t]  # [batch_size_local]

    # --- Core: apply both per-asset and temporal weighting ---
    # Each point's log likelihood is: per-asset weight × decayed by time index
    if obs is not None:
        # Calculate log likelihood "by hand" for control
        log_prob = dist.StudentT(df, mean_t, sigma_t).log_prob(
            obs
        )  # shape [batch_size_local]
        # Both per-asset and temporal scaling. Make sure shapes match!
        combined_weight = asset_weight * (lambda_decay**decay_exponent)
        # Apply mask (valid times only)
        weighted_log_prob = torch.where(
            valid_mask, combined_weight * log_prob, torch.zeros_like(log_prob)
```

```
            )
            # Use pyro.factor to modify SVI loss accordingly
            pyro.factor(f"weighted_decay_{t}", weighted_log_prob)
            # Comment: This gives the model flexibility to forget irrelevant *history* and *assets*.

        # Bookkeeping and recursion
        with poutine.mask(mask=valid_mask):
            r_t = pyro.sample(f"r_{t}", dist.StudentT(df, mean_t, sigma_t), obs=obs)

        e_prev = r_t - mean_t
        sigma_prev = sigma_t
        r_prev = r_t

#######################################################################################


def guide(
    returns,            # [num_assets, max_T]
    lengths,            # [num_assets]
    args,               # Dummy, just for API compatibility
    prior_predictive_checks: bool = False,
    device=torch.device("cpu"),
    batch_size=None
):
    returns = returns.to(device)
    lengths = lengths.to(device)
    num_assets, max_T = returns.shape

    # ---- MEAN-FIELD FOR GLOBALS ----------
    # Each uses unconstrained loc/scale for variational params
    omega_mu_loc = pyro.param("omega_mu_loc", torch.tensor(1.5, device=device))
    omega_mu_scale = pyro.param("omega_mu_scale", torch.tensor(0.5, device=device),
constraint=dist.constraints.positive)
    omega_mu = pyro.sample("omega_mu", dist.Normal(omega_mu_loc, omega_mu_scale))

    omega_sigma_loc = pyro.param("omega_sigma_loc", torch.tensor(1.5, device=device))
    omega_sigma_scale = pyro.param("omega_sigma_scale", torch.tensor(0.5, device=device),
constraint=dist.constraints.positive)
    omega_sigma = pyro.sample("omega_sigma", dist.Normal(omega_sigma_loc,
omega_sigma_scale))

    ab_sum_a_hyper_loc = pyro.param("ab_sum_a_hyper_loc", torch.tensor(2.0, device=device))
    ab_sum_a_hyper_scale = pyro.param("ab_sum_a_hyper_scale", torch.tensor(0.5, device=device),
constraint=dist.constraints.positive)
    ab_sum_a_hyper = pyro.sample("ab_sum_a_hyper", dist.Normal(ab_sum_a_hyper_loc,
ab_sum_a_hyper_scale))

    ab_sum_b_hyper_loc = pyro.param("ab_sum_b_hyper_loc", torch.tensor(2.0, device=device))
    ab_sum_b_hyper_scale = pyro.param("ab_sum_b_hyper_scale", torch.tensor(0.5,
device=device), constraint=dist.constraints.positive)
    ab_sum_b_hyper = pyro.sample("ab_sum_b_hyper", dist.Normal(ab_sum_b_hyper_loc,
ab_sum_b_hyper_scale))

    ab_frac_a_hyper_loc = pyro.param("ab_frac_a_hyper_loc", torch.tensor(2.0, device=device))
```

```
    ab_frac_a_hyper_scale = pyro.param("ab_frac_a_hyper_scale", torch.tensor(0.5, device=device), constraint=dist.constraints.positive)
    ab_frac_a_hyper = pyro.sample("ab_frac_a_hyper", dist.Normal(ab_frac_a_hyper_loc, ab_frac_a_hyper_scale))

    ab_frac_b_hyper_loc = pyro.param("ab_frac_b_hyper_loc", torch.tensor(2.0, device=device))
    ab_frac_b_hyper_scale = pyro.param("ab_frac_b_hyper_scale", torch.tensor(0.5, device=device), constraint=dist.constraints.positive)
    ab_frac_b_hyper = pyro.sample("ab_frac_b_hyper", dist.Normal(ab_frac_b_hyper_loc, ab_frac_b_hyper_scale))

    phi_mu_loc = pyro.param("phi_mu_loc", torch.tensor(0.0, device=device))
    phi_mu_scale = pyro.param("phi_mu_scale", torch.tensor(0.5, device=device), constraint=dist.constraints.positive)
    phi_mu = pyro.sample("phi_mu", dist.Normal(phi_mu_loc, phi_mu_scale))

    phi_sigma_loc = pyro.param("phi_sigma_loc", torch.tensor(1.0, device=device))
    phi_sigma_scale = pyro.param("phi_sigma_scale", torch.tensor(0.5, device=device), constraint=dist.constraints.positive)
    phi_sigma = pyro.sample("phi_sigma", dist.Normal(phi_sigma_loc, phi_sigma_scale))

    sigma_init_mu_loc = pyro.param("sigma_init_mu_loc", torch.tensor(10.0, device=device))
    sigma_init_mu_scale = pyro.param("sigma_init_mu_scale", torch.tensor(1.0, device=device), constraint=dist.constraints.positive)
    sigma_init_mu = pyro.sample("sigma_init_mu", dist.Normal(sigma_init_mu_loc, sigma_init_mu_scale))

    sigma_init_sigma_loc = pyro.param("sigma_init_sigma_loc", torch.tensor(10.0, device=device))
    sigma_init_sigma_scale = pyro.param("sigma_init_sigma_scale", torch.tensor(1.0, device=device), constraint=dist.constraints.positive)
    sigma_init_sigma = pyro.sample("sigma_init_sigma", dist.Normal(sigma_init_sigma_loc, sigma_init_sigma_scale))

    df_mu_loc = pyro.param("df_mu_loc", torch.tensor(10.0, device=device))
    df_mu_scale = pyro.param("df_mu_scale", torch.tensor(1.0, device=device), constraint=dist.constraints.positive)
    df_mu = pyro.sample("df_mu", dist.Normal(df_mu_loc, df_mu_scale))

    df_sigma_loc = pyro.param("df_sigma_loc", torch.tensor(1.0, device=device))
    df_sigma_scale = pyro.param("df_sigma_scale", torch.tensor(0.5, device=device), constraint=dist.constraints.positive)
    df_sigma = pyro.sample("df_sigma", dist.Normal(df_sigma_loc, df_sigma_scale))

    lambda_decay_alpha = pyro.param("lambda_decay_alpha", torch.tensor(2.0, device=device), constraint=dist.constraints.positive)
    lambda_decay_beta  = pyro.param("lambda_decay_beta", torch.tensor(2.0, device=device), constraint=dist.constraints.positive)
    lambda_decay = pyro.sample("lambda_decay", dist.Beta(lambda_decay_alpha, lambda_decay_beta))

    # ---- STRUCTURED MULTIVARIATE FOR LOCALS (PER ASSET) --------
    # Block: per asset, ALL local params enter a single multivariate Normal
    per_asset_param_dim = 8
```

```
# Order: garch_omega, alpha_beta_sum, alpha_frac, phi, garch_sigma_init, degrees_of_freedom,
asset_weight, (possible npad for alignment)
with pyro.plate("assets", num_assets, dim=-2):
    loc = pyro.param("local_loc", torch.zeros(num_assets, per_asset_param_dim, device=device))
    scale_tril = pyro.param(
        "local_scale_tril",
        torch.stack([torch.eye(per_asset_param_dim, device=device) for _ in range(num_assets)]),
        constraint=dist.constraints.lower_cholesky
    )
    local_latents = pyro.sample(
        "local_latents",
        dist.MultivariateNormal(loc, scale_tril=scale_tril).to_event(1)
    )

# You will then decode each latent vector into its respective parameter:
#  garch_omega_i = local_latents[:, 0]
#  alpha_beta_sum_i = local_latents[:, 1]
#  alpha_frac_i = local_latents[:, 2]
#  phi_i = local_latents[:, 3]
#  garch_sigma_init_i = local_latents[:, 4]
#  degrees_of_freedom_i = local_latents[:, 5]
#  asset_weight_i = local_latents[:, 6]
#  [Slot 7 may be left as npad/unused or you can add another parameter]

# If obs masking or batch mode is needed adjust accordingly.
```