Test Plan: collectives

# Category: Functional Correctness

Tests to ensure the core functionality of torch.distributed.all\_reduce operates as expected under various valid conditions, covering different operations, tensor shapes, and world sizes.

## Test Case TC\_AR\_FUNC\_001: Basic All-Reduce (SUM op)

Description: Verify all\_reduce with the default SUM operation across multiple processes using a simple tensor setup.

Implementation file: test\_all\_reduce.py

Steps:

* Initialize the distributed process group (e.g., `dist.init\_process\_group`).
* On each rank `r` (from 0 to `world\_size - 1`), create a tensor `t\_r` initialized with values based on `rank\_id` (e.g., `torch.ones(10) \* rank\_id`).
* Perform `dist.all\_reduce(t\_r, op=dist.ReduceOp.SUM)`.
* After the operation, verify that the resulting `t\_r` on each rank is equal to the sum of all initial `t\_i` tensors across all ranks (i.e., `sum(torch.ones(10) \* i for i in range(world\_size))`).

Expected Result: All tensors on all ranks should contain the element-wise sum of original values from all ranks.

Data Types: torch.float32, torch.float16, torch.float64, torch.int32, torch.int64

## Test Case TC\_AR\_FUNC\_002: All-Reduce with AVG op

Description: Verify all\_reduce with the AVG (average) operation for floating-point tensors.

Implementation file: test\_all\_reduce.py

Steps:

* Initialize the distributed process group.
* On each rank `r`, create a tensor `t\_r`.
* Perform `dist.all\_reduce(t\_r, op=dist.ReduceOp.AVG)`.
* Verify that the resulting `t\_r` on each rank equals the element-wise average of initial `t\_i` tensors from all ranks (i.e., `sum(initial\_t\_i) / world\_size`).

Expected Result: All tensors on all ranks should contain the element-wise average of original values from all ranks.

Data Types: torch.float32, torch.float16, torch.float64

## Test Case TC\_AR\_FUNC\_003: All-Reduce with MIN/MAX/PRODUCT ops

Description: Verify all\_reduce with MIN, MAX, and PRODUCT operations.

Implementation file: test\_all\_reduce.py

Steps:

* Initialize the distributed process group.
* On each rank, create tensors with varied values to ensure distinct min/max/product outcomes across ranks.
* Perform `dist.all\_reduce(tensor\_min, op=dist.ReduceOp.MIN)` and verify the minimum value.
* Perform `dist.all\_reduce(tensor\_max, op=dist.ReduceOp.MAX)` and verify the maximum value.
* Perform `dist.all\_reduce(tensor\_prod, op=dist.ReduceOp.PRODUCT)` and verify the product.
* For verification, compute the expected result locally by gathering all initial tensors (conceptually, not actually gathering for the test) and applying the operation.

Expected Result: Tensors should correctly contain the element-wise minimum, maximum, or product of original values from all ranks.

Data Types: torch.float32, torch.float16, torch.float64, torch.int32, torch.int64

## Test Case TC\_AR\_FUNC\_004: All-Reduce with various tensor shapes

Description: Test all\_reduce with 1D, 2D, and 3D tensors of different sizes to ensure shape handling correctness.

Implementation file: test\_all\_reduce.py

Steps:

* Initialize the distributed process group.
* On each rank, create tensors of varying shapes: e.g., `(10,)`, `(5, 5)`, `(2, 3, 4)`.
* For each shape, initialize with unique values per rank and perform `dist.all\_reduce` with `SUM` op.
* Verify the correctness by comparing the result with a locally computed sum for each shape.
* Ensure the output tensor maintains its original shape.

Expected Result: All\_reduce should correctly perform the reduction on tensors of different dimensions and maintain their original shapes.

Data Types: torch.float32, torch.int64

## Test Case TC\_AR\_FUNC\_005: All-Reduce with different world sizes

Description: Test all\_reduce's functionality and consistency when executed with varying numbers of participating processes (world sizes).

Implementation file: test\_all\_reduce.py

Steps:

* Execute the test script with `WORLD\_SIZE` environment variable set to different values: 2, 3, 4, and a larger value (e.g., 8 or 16).
* On each rank for each world size, create a tensor and perform `dist.all\_reduce(op=dist.ReduceOp.SUM)`.
* Verify the result for each world size, ensuring the sum correctly reflects the `WORLD\_SIZE` value used for that run.
* Ensure all ranks participate and synchronize correctly.

Expected Result: All\_reduce should function correctly and produce accurate results regardless of the number of participating processes.

Data Types: torch.float32

# Category: Data Type Compatibility

Tests to ensure all\_reduce works correctly across various numerical data types supported by PyTorch, including precision and overflow/underflow handling.

## Test Case TC\_AR\_DT\_001: All-Reduce with Float16 (Half precision)

Description: Verify all\_reduce functionality and precision with `torch.float16` tensors, especially focusing on potential precision loss or range issues.

Implementation file: test\_all\_reduce.py

Steps:

* Initialize the process group.
* Create `torch.float16` tensors on each rank with values that allow for precision checks (e.g., values that might sum to a denormal number, or large numbers summing to infinity).
* Perform `dist.all\_reduce(tensor, op=dist.ReduceOp.SUM)`.
* Verify the result against expected values, carefully considering the inherent precision limits of `float16`.
* Repeat for MIN/MAX/PRODUCT operations, and cautiously for AVG (as `float16` can have precision challenges with division).

Expected Result: All\_reduce should compute correctly for `float16` tensors within expected precision limits, handling underflow/overflow as per IEEE 754 standard.

Data Types: torch.float16

## Test Case TC\_AR\_DT\_002: All-Reduce with Integer types (Int32, Int64)

Description: Verify all\_reduce functionality with `torch.int32` and `torch.int64` tensors for supported operations (SUM, MIN, MAX, PRODUCT).

Implementation file: test\_all\_reduce.py

Steps:

* Initialize the process group.
* Create `torch.int32` and `torch.int64` tensors with varied integer values on each rank.
* Perform `dist.all\_reduce` with `SUM`, `MIN`, `MAX`, `PRODUCT` operations.
* Verify results match exact integer calculations.
* Attempt `dist.all\_reduce` with `AVG` op on an integer tensor and assert that an error (e.g., `RuntimeError`) is raised, as AVG is typically not supported for integer types.

Expected Result: All\_reduce should compute correctly for integer tensors for supported operations. An error should be raised for unsupported operations like AVG.

Data Types: torch.int32, torch.int64

# Category: Device Compatibility

Tests to ensure all\_reduce functions correctly on both CPU and GPU (CUDA) devices, adapting to the available hardware and chosen backend.

## Test Case TC\_AR\_DEV\_001: All-Reduce on CPU Tensors (Gloo Backend)

Description: Verify all\_reduce using CPU tensors, typically with the Gloo backend, ensuring correct cross-process communication.

Implementation file: test\_all\_reduce.py

Steps:

* Initialize the process group with `backend='gloo'`.
* On each rank, create tensors on CPU (`torch.ones(10, device='cpu')`).
* Perform `dist.all\_reduce` with various operations (SUM, AVG, MIN, MAX, PRODUCT).
* Verify results match expected CPU calculations.

Expected Result: All\_reduce should function correctly for CPU tensors, producing accurate results.

Data Types: torch.float32, torch.int64

## Test Case TC\_AR\_DEV\_002: All-Reduce on GPU Tensors (NCCL Backend)

Description: Verify all\_reduce using GPU tensors, typically with the NCCL backend, ensuring efficient and correct GPU-to-GPU communication. Requires CUDA-enabled environment.

Implementation file: test\_all\_reduce.py

Steps:

* Check for CUDA availability (`torch.cuda.is\_available()`). Skip test if not available.
* Initialize the process group with `backend='nccl'`.
* On each rank, create tensors on the respective GPU device (`torch.ones(10, device='cuda:rank\_id')`).
* Perform `dist.all\_reduce` with various operations (SUM, AVG, MIN, MAX, PRODUCT).
* Verify results match expected calculations, ensuring data integrity across GPUs.

Expected Result: All\_reduce should function correctly and efficiently for GPU tensors, producing accurate results.

Data Types: torch.float32, torch.float16

# Category: Edge Case Handling

Tests to verify the behavior of all\_reduce under unusual or boundary input conditions, ensuring robustness and predictable behavior.

## Test Case TC\_AR\_EDGE\_001: All-Reduce with single-element tensors

Description: Test all\_reduce with tensors containing only one element to ensure scalar operations are handled correctly.

Implementation file: test\_all\_reduce.py

Steps:

* Initialize the process group.
* On each rank, create a `torch.tensor([value])`.
* Perform `dist.all\_reduce` with SUM, MIN, MAX, PRODUCT, AVG operations.
* Verify results against simple scalar calculations of the same operation.

Expected Result: All\_reduce should correctly compute results for single-element tensors for all supported operations.

Data Types: torch.float32, torch.int64

## Test Case TC\_AR\_EDGE\_002: All-Reduce with empty tensors

Description: Test all\_reduce with tensors that have a size of 0 (e.g., `torch.empty(0)`).

Implementation file: test\_all\_reduce.py

Steps:

* Initialize the process group.
* On each rank, create an empty tensor (`torch.empty(0)` or `torch.zeros(0)`).
* Perform `dist.all\_reduce(empty\_tensor)`.
* Verify that the operation completes without error, the tensor remains empty, and it is effectively a no-op.
* Confirm no synchronization issues arise from empty tensor calls.

Expected Result: Operation should complete successfully without modifying the empty tensor or raising an error. It should behave as a no-op.

Data Types: torch.float32

## Test Case TC\_AR\_EDGE\_003: All-Reduce with NaN/Inf values

Description: Test how all\_reduce handles tensors containing Not-a-Number (NaN) or Infinity (Inf) values according to IEEE 754 floating-point rules.

Implementation file: test\_all\_reduce.py

Steps:

* Initialize the process group.
* On various ranks, create tensors containing combinations of `float('inf')`, `float('-inf')`, and `float('nan')`.
* Perform `dist.all\_reduce` with SUM, MIN, MAX, PRODUCT operations.
* Verify results rigorously against IEEE 754 standards (e.g., NaN propagates for SUM/PRODUCT, specific rules for min/max with +/- Inf and NaN).

Expected Result: Results should precisely follow IEEE 754 standards for NaN/Inf propagation and comparisons.

Data Types: torch.float32, torch.float64

## Test Case TC\_AR\_EDGE\_004: All-Reduce with very large/small values

Description: Test all\_reduce with tensors containing values close to the limits of their data type to check for correct overflow/underflow handling or precision issues.

Implementation file: test\_all\_reduce.py

Steps:

* Initialize the process group.
* On each rank, create tensors with values like `torch.finfo(dtype).max`, `torch.finfo(dtype).min`, or very small denormal numbers.
* Perform `dist.all\_reduce` (especially SUM and PRODUCT) and verify results.
* Specifically check for precision loss when summing very small numbers with very large numbers, or when products lead to overflow/underflow.

Expected Result: Results should be accurate within the limits of the data type; overflow/underflow should be handled gracefully (e.g., becoming Inf or 0, or saturating) and precision maintained where possible.

Data Types: torch.float32, torch.float16, torch.int64

# Category: Concurrency and Synchronization

Tests to ensure proper synchronization of processes during collective operations, verifying that calls block or complete correctly without deadlocks.

## Test Case TC\_AR\_SYNC\_001: Sequential All-Reduce operations

Description: Verify that multiple `all\_reduce` calls can be performed sequentially on different tensors without deadlocks or incorrect results.

Implementation file: test\_all\_reduce.py

Steps:

* Initialize the process group.
* On each rank, create tensor A and tensor B.
* Perform `dist.all\_reduce(tensor\_A)`.
* Perform `dist.all\_reduce(tensor\_B)` immediately after the first call completes.
* Verify both operations yield correct results on their respective tensors.
* Ensure no operations block indefinitely or produce stale results.

Expected Result: All sequential collective operations should complete successfully and correctly, demonstrating proper internal synchronization and state management.

Data Types: torch.float32

# Category: Error Handling

Tests to ensure that `all\_reduce` handles invalid inputs or unsupported configurations gracefully by raising appropriate, informative errors.

## Test Case TC\_AR\_ERR\_001: Invalid reduction operation

Description: Test calling `all\_reduce` with an invalid or unsupported reduction operation enum value or type.

Implementation file: test\_all\_reduce.py

Steps:

* Initialize the process group.
* Create a tensor.
* Attempt to call `dist.all\_reduce(tensor, op='INVALID\_OP\_STRING')` or `dist.all\_reduce(tensor, op=999)` (an invalid integer value).
* Verify that a `ValueError` or `TypeError` is raised indicating an invalid or unsupported operation.
* Test also with valid but non-`ReduceOp` object (e.g., `op=1` instead of `ReduceOp.SUM`).

Expected Result: An error (e.g., `ValueError`, `TypeError`, or `RuntimeError` depending on PyTorch's specific error handling for this) should be raised indicating an invalid reduction operation.

Data Types: torch.float32

## Test Case TC\_AR\_ERR\_002: Unsupported data type for operation

Description: Test calling `all\_reduce` with a data type that is semantically not supported for a specific operation (e.g., AVG on integer types).

Implementation file: test\_all\_reduce.py

Steps:

* Initialize the process group.
* Create an `int` tensor (e.g., `torch.zeros(5, dtype=torch.int32)`).
* Attempt to call `dist.all\_reduce(int\_tensor, op=dist.ReduceOp.AVG)`.
* Verify that an error is raised (e.g., `RuntimeError` or `ValueError`) indicating that AVG is not supported for integer types.

Expected Result: An error should be raised indicating the unsupported data type for the given operation (e.g., `AVG` on integer tensors).

Data Types: torch.int32, torch.int64

## Test Case TC\_AR\_ERR\_003: Uninitialized process group

Description: Test calling `all\_reduce` without first initializing the distributed process group, which should lead to a clear error.

Implementation file: test\_all\_reduce.py

Steps:

* Ensure `dist.init\_process\_group` has NOT been called.
* Attempt to call `dist.all\_reduce(tensor)` with any valid tensor.
* Verify that a `RuntimeError` or similar error is raised, clearly indicating that the distributed environment is not initialized or available.

Expected Result: A `RuntimeError` indicating an uninitialized process group or distributed environment should be raised.

Data Types: torch.float32

## Test Case TC\_AR\_ERR\_004: Mismatched tensor shapes/sizes across ranks

Description: Test `all\_reduce` with tensors that have different shapes or sizes on different ranks, which should typically result in an error.

Implementation file: test\_all\_reduce.py

Steps:

* Initialize the process group.
* On rank 0, create `torch.ones(10)`.
* On rank 1, create `torch.ones(5)` (different size).
* On rank 2, create `torch.ones(2, 5)` (different shape).
* Attempt to call `dist.all\_reduce` on these mismatched tensors.
* Verify that a `RuntimeError` or similar error indicating tensor size/shape mismatch is raised across all participating ranks.

Expected Result: A `RuntimeError` or assertion failure should occur, indicating that input tensors have mismatched shapes or sizes across ranks.

Data Types: torch.float32

# Category: Performance Considerations

Guidelines and high-level test cases for performance benchmarking and scalability analysis of `all\_reduce` across different tensor sizes, world sizes, and hardware configurations.

## Test Case TC\_AR\_PERF\_001: Latency Measurement Across Tensor Sizes

Description: Measure the average time taken for `all\_reduce` for a range of tensor sizes to characterize latency behavior.

Implementation file: test\_all\_reduce.py

Steps:

* Initialize the process group (using both Gloo and NCCL if applicable).
* For a spectrum of tensor sizes (e.g., 1KB, 10KB, 100KB, 1MB, 10MB, 100MB, 1GB, up to system limits):
* On each rank, create a tensor of the specific size.
* Execute `dist.barrier()` to ensure all ranks are synchronized.
* Start a high-resolution timer.
* Perform `dist.all\_reduce(tensor, op=dist.ReduceOp.SUM)`.
* Execute `dist.barrier()` to wait for completion.
* Stop the timer.
* Record the elapsed time for each rank. Report average, min, and max latency across ranks.

Expected Result: Detailed performance metrics (latency in ms/us) recorded for various tensor sizes. Latency is expected to increase with tensor size, but should show optimized communication patterns (e.g., logarithmic for NCCL on GPUs for smaller sizes, then bandwidth-limited).

Data Types: torch.float32

## Test Case TC\_AR\_PERF\_002: Throughput Measurement

Description: Estimate the effective data transfer throughput of `all\_reduce` based on tensor size and measured time.

Implementation file: test\_all\_reduce.py

Steps:

* Leverage data from TC\_AR\_PERF\_001.
* Calculate throughput as (Tensor Size \* World Size) / Latency (for SUM, MIN, MAX, AVG which process all data) or (Tensor Size \* 2) / Latency (for efficient algorithms like ring all-reduce where each element traverses the ring twice). Clarify the definition used.
* Plot throughput vs. tensor size.
* Compare observed throughput against theoretical network bandwidth limits.
* Repeat measurements for different network configurations (e.g., local host, high-speed interconnect).

Expected Result: Throughput metrics (e.g., GB/s) calculated and analyzed. Throughput should ideally approach network bandwidth limits for larger tensor sizes.

Data Types: torch.float32

## Test Case TC\_AR\_PERF\_003: Scalability Across World Sizes

Description: Observe how `all\_reduce` performance scales with an increasing number of processes for a fixed tensor size, revealing potential bottlenecks.

Implementation file: test\_all\_reduce.py

Steps:

* Run latency/throughput tests (as defined in TC\_AR\_PERF\_001/002) for a fixed, representative tensor size (e.g., 100MB).
* Vary the `WORLD\_SIZE` for these runs (e.g., 2, 4, 8, 16, 32, 64, etc., up to available resources).
* Plot performance (latency or inverse of throughput) vs. `WORLD\_SIZE`.
* Analyze the scaling curve to identify linearity (ideal scaling), saturation points, or performance degradation with increased process count.

Expected Result: Performance should scale efficiently with increasing world size (e.g., sub-linear or logarithmic increase in latency, or increasing throughput for optimal algorithms). Any non-linearities or bottlenecks should be identified and investigated.

Data Types: torch.float32