Interview - Codility

app.codility.com/c/interview/CL-FH3QYE-X8Z/#/

Bookmarks old Maps code all ML all system finance all personal all math tool tmp

ASKS

1 > Task 1

YSTEM DESIGN

Whiteboard

Take a tour

Accessibility mode

Task 1

Feedback

EXPLOR

WORKS

Simulated Distributed ML Training with Checkpointing

Objective:
Implement a simplified version of a distributed machine learning (ML) training job to demonstrate your understanding of parallelism, data partitioning, and fault tolerance.

Problem Statement:
You are tasked with simulating a distributed ML training job using multiple parallel trainer processes. The training is simplified to summing numeric data.

Requirements:

1. **Input Data:**
   - Use a CSV file or an in-memory array that contains one numeric value per row (e.g., `[1, 2, 3, ..., 1000]`).
2. **Parallel Trainers:**
   - Launch `N` parallel "trainer" processes.
   - Each trainer should load and process a distinct shard of the data (non-overlapping rows).
3. **Simulated Training:**
   - Each trainer computes the sum of its assigned shard.
   - Then, combine the partial results to compute the global sum.
4. **(Bonus) Checkpointing Support:**
   - Implement a mechanism to save the intermediate state (e.g., each trainer's partial sum) to disk.
   - Provide functionality to resume training from the last checkpoint in the event of a failure.
5. **Testing:**
   - Test the implementation under:

OUTLIN

TIMELI

TASKS

< 1 > Task 1

SYSTEM DESIGN

Whiteboard

Take a tour

Accessibility mode

Task 1

- Use a CSV file or an in-memory array that contains one numeric value per row (e.g., `[1, 2, 3, ..., 1000]`).

2. **Parallel Trainers:**
   - Launch `N` parallel "trainer" processes.
   - Each trainer should load and process a distinct shard of the data (non-overlapping rows).
3. **Simulated Training:**
   - Each trainer computes the sum of its assigned shard.
   - Then, combine the partial results to compute the global sum.
4. **(Bonus) Checkpointing Support:**
   - Implement a mechanism to save the intermediate state (e.g., each trainer's partial sum) to disk.
   - Provide functionality to resume training from the last checkpoint in the event of a failure.
5. **Testing:**
   - Test the implementation under:
     - A **normal case** (no failure): verify the total sum matches the expected result.
     - (Bonus) A **failure-recovery case**: simulate a process failure, resume from checkpoint, and verify that the final result is still correct.

Task 1

Simulated Distributed ML Training with Checkpointing

Objective:
Implement a simplified version of a distributed machine learning (ML) training job to demonstrate your understanding of parallelism, data partitioning, and fault tolerance.

Problem Statement:
You are tasked with simulating a distributed ML training job using multiple parallel trainer processes. The training is simplified to summing numeric data.

Requirements:

1. **Input Data:**
   - Use a CSV file or an in-memory array that contains one numeric value per row (e.g., `[1, 2, 3, ..., 1000]`).
2. **Parallel Trainers:**
   - Launch `N` parallel "trainer" processes.
   - Each trainer should load and process a distinct shard of the data (non-overlapping rows).
3. **Simulated Training:**
   - Each trainer computes the sum of its assigned shard.
   - Then, combine the partial results to compute the global sum.
4. **(Bonus) Checkpointing Support:**
   - Implement a mechanism to save the intermediate state (e.g., each trainer's partial sum) to disk.
   - Provide functionality to resume training from the last checkpoint in the event of a failure.
5. **Testing:**
   - Test the implementation under:
     - A **normal case** (no failure): verify the total sum matches the expected result.
     - (Bonus) A **failure-recovery case**: simulate a process failure, resume from checkpoint, and verify that the final result is still correct.

Feedback

EXPLORER

WORKSPACE

main.py

main.py 1

main.py > ...

```python
""" Using any AI assist is prohibited. """

import multiprocessing as mp


def worker(proc_id, shard, results):
    results[proc_id] = 0


def main():
    manager = mp.Manager()
    results = manager.list([None])
    p = mp.Process(target=worker, args=(0, [], results))
    p.start()
    p.join()
    results = list(results)
    print(results)


if __name__ == "__main__":
    main()

# Zhefu? Can you
```

Anton S.

Task 1

Simulated Distributed ML Training with Checkpointing

Objective:
Implement a simplified version of a distributed machine learning (ML) training job to demonstrate your understanding of parallelism, data partitioning, and fault tolerance.

Problem Statement:
You are tasked with simulating a distributed ML training job using multiple parallel trainer processes. The training is simplified to summing numeric data.

Requirements:

1. **Input Data:**
   - Use a CSV file or an in-memory array that contains one numeric value per row (e.g., `[1, 2, 3, ..., 1000]`).
2. **Parallel Trainers:**
   - Launch `N` parallel "trainer" processes.
   - Each trainer should load and process a distinct shard of the data (non-overlapping rows).
3. **Simulated Training:**
   - Each trainer computes the sum of its assigned shard.
   - Then, combine the partial results to compute the global sum.
4. **(Bonus) Checkpointing Support:**
   - Implement a mechanism to save the intermediate state (e.g., each trainer's partial sum) to disk.
   - Provide functionality to resume training from the last checkpoint in the event of a failure.
5. **Testing:**
   - Test the implementation under:
     - A **normal case** (no failure): verify the total sum matches the expected result.
     - (Bonus) A **failure-recovery case**: simulate a process failure, resume from checkpoint, and verify that the final result is still correct.

Feedback

workspace

EXPLORER

WORKSPACE

main.py 1

main.py > ...

```python
""" Using any AI assist is prohibited. """

import multiprocessing as mp


def worker(proc_id, shard, results):
    results[proc_id] = 0


def main():
    manager = mp.Manager()
    results = manager.list([None])
    p = mp.Process(target=worker, args=(0, [], results))
    p.start()
    p.join()
    results = list(results)
    print(results)


if __name__ == "__main__":
    main()

# Zhefu? Can you 
```

OUTLINE

y mode

//——————————————————————————————————————
myV0

Problem Statement:
You are tasked with simulating a distributed ML training job using multiple parallel trainer processes. The training is simplified to summing numeric data.

Requirements:

1. **Input Data:**
   - Use a CSV file or an in-memory array that contains one numeric value per row (e.g., `[1, 2, 3, ..., 1000]`).
2. **Parallel Trainers:**
   - Launch `N` parallel "trainer" processes.
   - Each trainer should load and process a distinct shard of the data (non-overlapping rows).
3. **Simulated Training:**
   - Each trainer computes the sum of its assigned shard.
   - Then, combine the partial results to compute the global sum.

Milie stones:

- Sum all the numbers from 1 to 1000 and print out using N processes (say 10 processes)
- Sum of the number at every training step
- Load the data data in random order
- Make the order (still random) deterministics between 2 training runs

Task 1

Problem Statement:
You are tasked with simulating a distributed ML training job using multiple parallel trainer processes. The training is simplified to summing numeric data.

Requirements:

1. **Input Data:**
   - Use a CSV file or an in-memory array that contains one numeric value per row (e.g., `[1, 2, 3, ..., 1000]`).
2. **Parallel Trainers:**
   - Launch `N` parallel "trainer" processes.
   - Each trainer should load and process a distinct shard of the data (non-overlapping rows).
3. **Simulated Training:**
   - Each trainer computes the sum of its assigned shard.
   - Then, combine the partial results to compute the global sum.

Milie stones:

- Sum all the numbers from 1 to 1000 and print out using N processes (say 10 processes)
- Sum of the number at every training step
- Load the data data in random order
- Make the order (still random) deterministics between 2 training runs

Feedback

EXPLORER

WORKSPACE

main.py 1

main.py > ...

```python
import multiprocessing as mp

# you can write to stdout for debugging purposes, e.g.
print("This is a debug message")


## 10 proccesses
##

DATA_SIZE = 1000
NUM_PROC = 4


# geberate training data for each worker
def createDataPart(data, num_proc):
    average_len = len(data) // num_proc
    data_parts = []
    for idx in range(num_proc):
        start = idx * average_len
        end = (idx+1) * average_len
        if idx == num_proc - 1:
            end = len(data)

        tup = data[start:end]
        data_parts.append(tup)

    return data_parts


# each worker logic
def worker(proc_id, data_part, results):
    local_sum = 0
    start=0

    len0 = len(data_part)
    for idx in range(len0):
        local_sum += data_part[idx]

    results[proc_id] = local_sum
    print("proc_id = ", proc_id, ", local_sum = ", local_sum)
    return

# simulate the parallel training
def simulation(data, num_proc):
    data_parts = createDataPart(data, num_proc)

    manager = mp.Manager()
```

Task 1

Problem Statement:
You are tasked with simulating a distributed ML training job using multiple parallel trainer processes. The training is simplified to summing numeric data.

Requirements:

1. **Input Data:**
   - Use a CSV file or an in-memory array that contains one numeric value per row (e.g., `[1, 2, 3, ..., 1000]`).
2. **Parallel Trainers:**
   - Launch `N` parallel "trainer" processes.
   - Each trainer should load and process a distinct shard of the data (non-overlapping rows).
3. **Simulated Training:**
   - Each trainer computes the sum of its assigned shard.
   - Then, combine the partial results to compute the global sum.

Milie stones:

- Sum all the numbers from 1 to 1000 and print out using N processes (say 10 processes)
- Sum of the number at every training step
- Load the data data in random order
- Make the order (still random) deterministics between 2 training runs

Feedback

EXPLORER

WORKSPACE

main.py 1

main.py > ...

```python
def worker(proc_id, data_part, results):

    len0 = len(data_part)
    for idx in range(len0):
        local_sum += data_part[idx]

    results[proc_id] = local_sum
    print("proc_id = ", proc_id, ", local_sum = ", local_sum)
    return

# simulate the parallel training
def simulation(data, num_proc):
    data_parts = createDataPart(data, num_proc)

    manager = mp.Manager()
    results = manager.list([None] * num_proc)
    procs = []

    for idx in range(num_proc):
        p_arg = (idx, data_parts[idx], results)

        p = mp.Process(target=worker, args=p_arg)
        procs.append(p)
        p.start()

    for p in procs:
        p.join()

    rst = 0
    for tmp in results:
        if tmp != None:
            rst += tmp

    results = list(results)
    print(results)

    print("rst = ", rst)
    return rst


def main():
    data = list(range(1, 1+DATA_SIZE))
    expected_rst = sum(data)

    final_sum = simulation(data, NUM_PROC)
    print("expected_rst = ", expected_rst, ", final_sum = ", final_sum)
```

#——————————————————————————————
simulating distributed ml training with checkpointing

objective:
implement a simplified version of a distributed machine leanring (ml) training job to demonstrate you understanding of parallelism, data partitioning, and fault tolorance

problem statement:
you are tasked with simulating a distributed ml training job using parallel trainer processes. the training is simplified to summing numeric data

requirements:
1. input data
using a csv file or an im-memory array that contains one numeric value per row(re.g., [1,2,3,...,1000])

2. parallel trainers:
launch N parallel "trainer" processes
each trainer should be load and process a distinct shard of the data (non-overlapping rows)

3. simulated training
each trainer computes the sum of its assigned shard
then combine the partial results to compute the global sum

4. sum of the number at every straining step

5. load the data in random order

6. make the order (still random) deterministic between 2 training runs

7. please use this code as an example