Traditional Parallel Computing vs Parallel Computing with Cloud

Andrew Xie, Antonio Mena, Prayag Patel

Background

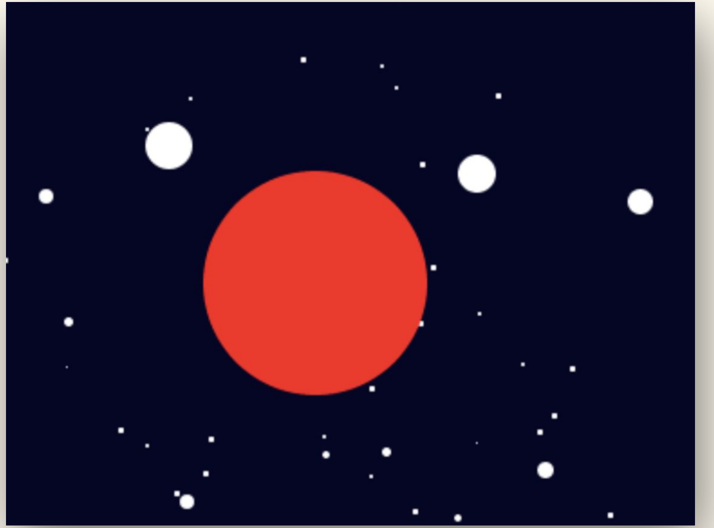
Project Goal & Implementation

Goal: Compare traditional parallel computing vs parallel computing with cloud

Implementation Task: Gravity simulator that tracks particle trajectories

Why This Problem?

- Classic scientific computing problem
- $O(n^2)$ computational complexity
- Naturally distributable workload



Technologies Used:

- Traditional: Spark (pySpark)
- Cloud: MPI on AWS EC2 nodes

Gravity Simulation Background

Particles in the simulation have:

- Position
- Velocity
- Mass

Newton's Law of Universal Gravitation

$$F = G\frac{m_1 m_2}{r^2}$$

F = force

G = gravitational constant

m_1 = mass of object 1

m_2 = mass of object 2

r = distance between centers of the masses

Applications:

- Predict cosmic collisions
- Future particle positions and their impact on other nearby cosmic objects
- Compare with real world data to verify known objects and identify unknown objects

What is Amazon EC2?

- **Amazon Elastic Compute Cloud (EC2)** provides scalable, on-demand computing resources in the AWS Cloud.
- EC2 enables you to launch virtual servers (**instances**) and scale resources based on workload requirements, reducing hardware costs and enabling faster application development and deployment.

How do EC2 Instances Work?

- **Instances:** Virtual servers that run applications and processes in the cloud. You choose the instance type based on your project's resource needs (CPU, memory, storage, and networking).
- **Instance Types:** Each type offers different combinations of resources to support different workloads, ideal for tasks like parallel computing with MPI.
- **Amazon Machine Images (AMIs):** Pre-configured templates for your instances, which include operating systems and additional software required for your project.
- **Instance Lifecycle:** You can start, stop, or terminate instances as needed. When an instance is terminated, its resources (such as temporary storage) are released.

Related Work

Existing Comparisons Between Spark and MPI

- Using the Cloud for parameter estimation problems: comparing Spark vs MPI with a case-study *(Gonzalez, Pardo, et al.)* [1]
- Performance Evaluation of Apache Spark Vs MPI: A Practical Case Study on Twitter Sentiment Analysis *(Kumar, Rahman)* [2]
- Big Data in metagenomics: Apache Spark vs MPI *(Abuín, Lopes, et al.)* [3]
- Comparing Spark vs MPI/OpenMP On Word Count MapReduce *(Junhao Li)* [4]

Existing studies similarly evaluate Spark and MPI frameworks against different use cases.

This project bridges the gap between scientific and big data workloads:

★ gravity simulation employs both **compute-heavy calculations** and **complex communication** for particle interactions

This project emphasizes parallel **cloud** computing:

★ our MPI experiment is run directly on **AWS EC2** cloud infrastructure, reflecting real-world cloud computing scenarios

Traditional Parallel Computing (Spark)

Traditional Parallel Computing (Spark)

Implementation details:

- Leveraged `pySpark` with `numpy` and `dataclasses`
- `Particle` class: stores 3D position, velocity, mass
- Force calculations using broadcasted particle data
- Algorithm steps:

    1) Force recalculation

    2) State updates

    3) Trajectory recording

Key **Challenges**:

- Potential bottleneck from data shuffle costs
- Not natively optimized for iterative workloads
- Steps in the simulation incur overhead from task scheduling and serialization (due to use of RDDs)

Parallel Cloud Computing (MPI on EC2 nodes)

Parallel Cloud Computing (MPI on EC2 Nodes)

Implementation approach:

- Custom `Particle` data structure
- Vector utility functions for 3D arithmetic
- Even distribution of work across MPI "ranks"
- Global synchronization using `MPI_Allgatherv`

MPI built-in that gathers data from all tasks and broadcasts the data to all tasks

AWS setup:

- Used `t2.micro` instances (free tier)
- 1 master node + 2 slave nodes
- Passwordless SSH for node communication
- Horizontal scaling to overcome single-processor limitation
- Leverage EC2's placement group to colocate instances and reduce communication latency

Initial Parameters

Python (pySpark)

```python
def create_solar_system() -> List[Particle]:
    return [
        Particle(
            position=np.array([0.0, 0.0, 0.0]),
            velocity=np.array([0.0, 0.0, 0.0]),
            mass=1.989e30
        ),
        Particle(
            position=np.array([1.496e11, 0.0, 0.0]),
            velocity=np.array([0.0, 29.78e3, 0.0]),
            mass=5.972e24
        ),
        Particle(
            position=np.array([2.279e11, 0.0, 0.0]),
            velocity=np.array([0.0, 24.077e3, 0.0]),
            mass=6.39e23
        )
    ]
```

C (MPI)

```c
particles[0].position = vector3_zero();
particles[0].velocity = vector3_zero();
particles[0].mass = 1.989e30;

particles[1].position.x = 1.496e11;
particles[1].position.y = 0.0;
particles[1].position.z = 0.0;
particles[1].velocity.x = 0.0;
particles[1].velocity.y = 29.78e3;
particles[1].velocity.z = 0.0;
particles[1].mass = 5.972e24;

particles[2].position.x = 2.279e11;
particles[2].position.y = 0.0;
particles[2].position.z = 0.0;
particles[2].velocity.x = 0.0;
particles[2].velocity.y = 24.077e3;
particles[2].velocity.z = 0.0;
particles[2].mass = 6.39e23;
```

Python (pySpark)

C (MPI)

Random Particles

```python
for i in range(5):
    more_particles.append(Particle(
        position=np.random.uniform(-3e11, 3e11, 3),
        velocity=np.random.uniform(-30e3, 30e3, 3),
        mass=np.random.uniform(1e23, 1e25)
    ))
```

```c
srand(time(NULL));
for (int i = 3; i < 8; i++) {
    particles[i].position.x = ((double)rand() / RAND_MAX) * 6e11 - 3e11;
    particles[i].position.y = ((double)rand() / RAND_MAX) * 6e11 - 3e11;
    particles[i].position.z = ((double)rand() / RAND_MAX) * 6e11 - 3e11;
    particles[i].velocity.x = ((double)rand() / RAND_MAX) * 6e4 - 3e4;
    particles[i].velocity.y = ((double)rand() / RAND_MAX) * 6e4 - 3e4;
    particles[i].velocity.z = ((double)rand() / RAND_MAX) * 6e4 - 3e4;
    particles[i].mass = ((double)rand() / RAND_MAX) * 9.9e24 + 1e23;
}
```

Python (pySpark)

C (MPI)

Force Calculation

```python
def calculate_force_between(p1_data, p2_data, G):
    p1_pos = np.array(p1_data['position'])
    p2_pos = np.array(p2_data['position'])
    r = p2_pos - p1_pos
    distance = np.linalg.norm(r)

    if distance < 1e-10:
        return np.zeros(3).tolist()

    force_magnitude = (G * p1_data['mass'] * p2_data['mass']) / (distance ** 2)
    return (force_magnitude * r / distance).tolist()
```

```c
Vector3 calculate_force(Particle* p1, Particle* p2) {
    Vector3 force = vector3_zero();
    Vector3 r = vector3_subtract(p2->position, p1->position);
    double distance = vector3_magnitude(r);

    if (distance < 1e-10) {
        return force;
    }

    double force_magnitude = (G * p1->mass * p2->mass) / (distance * distance);
    double scale = force_magnitude / distance;
    force = vector3_multiply(r, scale);

    return force;
}
```

Parallelization

Python (pySpark)

```python
class SparkGravitySimulator:
    G = 6.67430e-11

    def __init__(self, particles: List[Particle], dt: float = 0.01):
        self.spark = SparkSession.builder \
            .appName("GravitySimulation") \
            .config("spark.executor.memory", "2g") \
            .getOrCreate()

        self.particles_data = [p.to_dict() for p in particles]
        self.dt = dt
        self.num_particles = len(particles)

    def calculate_forces(self):
        particle_pairs = []
        for i in range(self.num_particles):
            for j in range(i + 1, self.num_particles):
                particle_pairs.append((i, j))

        sc = self.spark.sparkContext
        particles_broadcast = sc.broadcast(self.particles_data)
        G_broadcast = sc.broadcast(self.G)

        pairs_rdd = sc.parallelize(particle_pairs)

        def calculate_pair_force(pair):
            i, j = pair
            particles = particles_broadcast.value
            G = G_broadcast.value
            force = calculate_force_between(particles[i], particles[j], G)
            return (i, j, force)

        forces = pairs_rdd.map(calculate_pair_force).collect()
```

C (MPI)

```c
MPI_Get_address(&all_particles[0].position, &displacements[0]);
MPI_Get_address(&all_particles[0].velocity, &displacements[1]);
MPI_Get_address(&all_particles[0].mass, &displacements[2]);

for (int i = 2; i >= 0; i--) {
    displacements[i] = MPI_Aint_diff(displacements[i], displacements[0]);
}

MPI_Type_create_struct(3, blocklengths, displacements, types, &particle_type);
MPI_Type_commit(&particle_type);

MPI_Bcast(all_particles, num_particles, particle_type, 0, MPI_COMM_WORLD);
```

```c
    MPI_Allgatherv(
        &all_particles[start_idx], local_num_particles, particle_type,
        all_particles, recvcounts, displs, particle_type,
        MPI_COMM_WORLD
    );

    free(recvcounts);
    free(displs);

    MPI_Barrier(MPI_COMM_WORLD);
```

Output

Python (pySpark)

```
Starting gravity simulation at 20241211_020916
Configuration:
- Number of steps: 500
- Time step: 3600 seconds (1 hour)

Performance Statistics:
Total execution time: 55.43 seconds
Average time per step: 0.1109 seconds

Final positions:
Particle 0: (-40759.92361748901, 83021.92422931321,
-15339.827766454993)
Particle 1: (140072929991.45477, 52463379787.609665,
-21994.541876042145)
Particle 2: (223763210109.76465, 43076578937.28692,
-20591.140443501015)
Particle 3: (-220536191887.73007, -131004855333.84792,
-270275726748.04044)
Particle 4: (69353928904.48807, 203022216340.21124,
-168798879233.916)
Particle 5: (-203270117941.5046, -190823886331.11755,
-272352403694.52585)
Particle 6: (216587611342.19537, -42911503641.20782,
-259177568462.8093)
Particle 7: (-26352676432.34539, 13632601778.28655,
23526204261.72007)

Simulation completed successfully
```

C (MPI)

```
Starting MPI C gravity simulation at 20241211_015031
Number of processes: 3
Number of particles: 8
Steps: 500
Timestep: 3600.000000 seconds


Performance Statistics:
Total execution time: 0.28 seconds
Average time per step: 0.0006 seconds

Final positions:
Particle 0: (2.038543e+04, 7.519318e+03, 4.330912e+03)
Particle 1: (1.400729e+11, 5.246338e+10, 4.085081e+03)
Particle 2: (2.237632e+11, 4.307658e+10, 3.633190e+03)
Particle 3: (3.104993e+11, 7.523407e+10, 2.527527e+11)
Particle 4: (-7.151560e+09, 1.482256e+11, 1.320689e+11)
Particle 5: (-1.181584e+11, 1.983205e+11, 2.194526e+11)
Particle 6: (-2.203978e+11, -6.206211e+10,
-1.531738e+11)
Particle 7: (-1.774408e+11, -1.023138e+11,
3.362449e+10)

Simulation completed successfully
```

Results & Conclusion

Results

Amazon EC2

	Traditional Parallel Computing (Spark)	**Parallel Cloud Computing (MPI on EC2 nodes)**
Total execution time	55.43 seconds	0.28 seconds
Average time per step	0.1109 seconds	0.0006 seconds

~196.43 times speedup

PySpark vs MPI: Differences

- ★ MPI obviously performs better than PySpark because the gravity simulator used on AWS is written in C (statically typed compiled language vs dynamic interpreted Python)

MPI is low-level and requires better coding practices, it does not handle fault tolerance and must be written in C

PySpark is high-level, relatively easier to implement and has a higher level of abstraction

Limitations of Spark & Traditional Parallel Computing

As particle count is increased, the Spark implementation tends to **struggle more with growing communication overhead**, while MPI maintains relatively constant performance.

Spark is optimized for **data-parallel workloads** with minimal inter-task communication.

- Frequent communication between particles (tasks) in this problem is <u>critical</u>.
- At each simulation step, Spark needs to refresh data using RDDS, which implicitly incurs serialization, deserialization, and disk/memory storage overhead.

PySpark vs MPI: Increasing Efficiency?

- However, some improvements can be made to the PySpark code by optimizing memory used by the RDD, vectorizing operations, etc.
  - Better use of broadcast variables and smarter partitioning could improve performance

- The C code could also be improved by optimizing MPI, using more sophisticated algorithms, data management, etc.
  - Implementation of non-blocking communication could improve performance

Conclusion

Key insights:

- MPI & EC2 implementation significantly outperformed Spark for the gravity simulation problem
- Spark has clear limitations for tightly-coupled, iterative problems
- Cloud computing (via Amazon EC2) enabled efficient scaling

Future considerations:

- Optimization opportunities for both platforms
- Potential for hybrid approaches
- Further benchmarking and testing needed
- Exploration and comparison of approaches for different use cases and problems