Distributed Task Queue System

Demonstrating Parallel vs Sequential Computing Performance

Subject:	Parallel & Distributed Computing
Project Type:	Term Project
Technology:	Node.js, TypeScript, Express.js
Date:	December 2025

Group Members:	
M. Arslan	22021519-009
Hamza Ehsan Butt	22021519-085
Talha Adalat	22021519-040

Submitted To:	Dr. Umar Shoaib

Table of Contents

1. Abstract

This project implements a **Distributed Task Queue System** that demonstrates the performance benefits of parallel computing over traditional sequential execution. The system uses Node.js Cluster module to create multiple worker processes that execute tasks concurrently from a shared queue.

Key Finding: Our experiments show that processing 100 tasks with 4 parallel workers is approximately **3.5x faster** than sequential processing, reducing total execution time from ~200 seconds to ~57 seconds.

> **Performance Summary:**
> - Sequential Processing (1 worker): ~200 seconds for 100 tasks
> - Parallel Processing (4 workers): ~57 seconds for 100 tasks
> - Speedup Factor: 3.5x improvement

2. Introduction

In modern computing, many applications need to process large numbers of tasks such as sending emails, processing images, generating reports, or analyzing data. Traditional sequential processing handles these tasks one at a time, which becomes a bottleneck when dealing with hundreds or thousands of tasks.

Parallel computing solves this problem by distributing tasks across multiple processing units (workers) that execute simultaneously. This project demonstrates this concept by building a distributed task queue system where:

- Tasks are submitted via a REST API
- Tasks are stored in a priority queue
- Multiple worker processes fetch and execute tasks in parallel
- Results are collected and monitored in real-time

3. Problem Statement

Consider a scenario where an application needs to process 100 tasks, each taking an average of 2 seconds to complete:

Approach	Workers	Time per Task	Total Time
Sequential	1	2 seconds	200 seconds
Parallel (4 workers)	4	2 seconds	~50 seconds

The sequential approach blocks the system while processing, leading to poor resource utilization. A distributed task queue with parallel workers solves this by:

- Offloading tasks to background workers
- Processing multiple tasks simultaneously
- Utilizing all available CPU cores
- Providing fault tolerance through retry mechanisms

4. Objectives

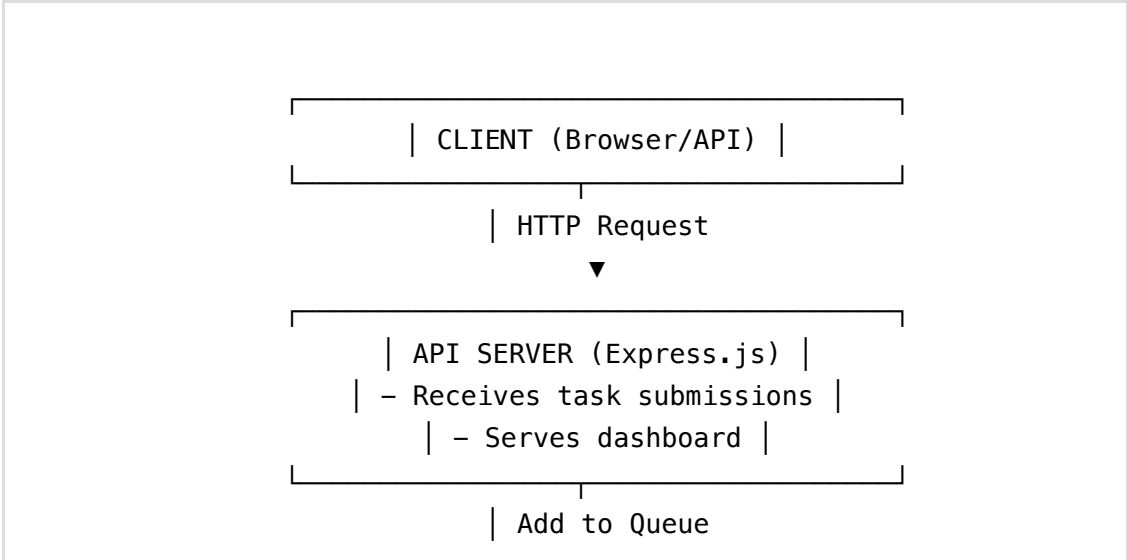
1. Build a task queue system with producer-consumer architecture
2. Implement parallel task execution using Node.js Cluster module
3. Measure and compare performance of sequential vs parallel processing
4. Demonstrate load balancing across multiple workers
5. Add fault tolerance with automatic retry mechanism
6. Create a real-time monitoring dashboard

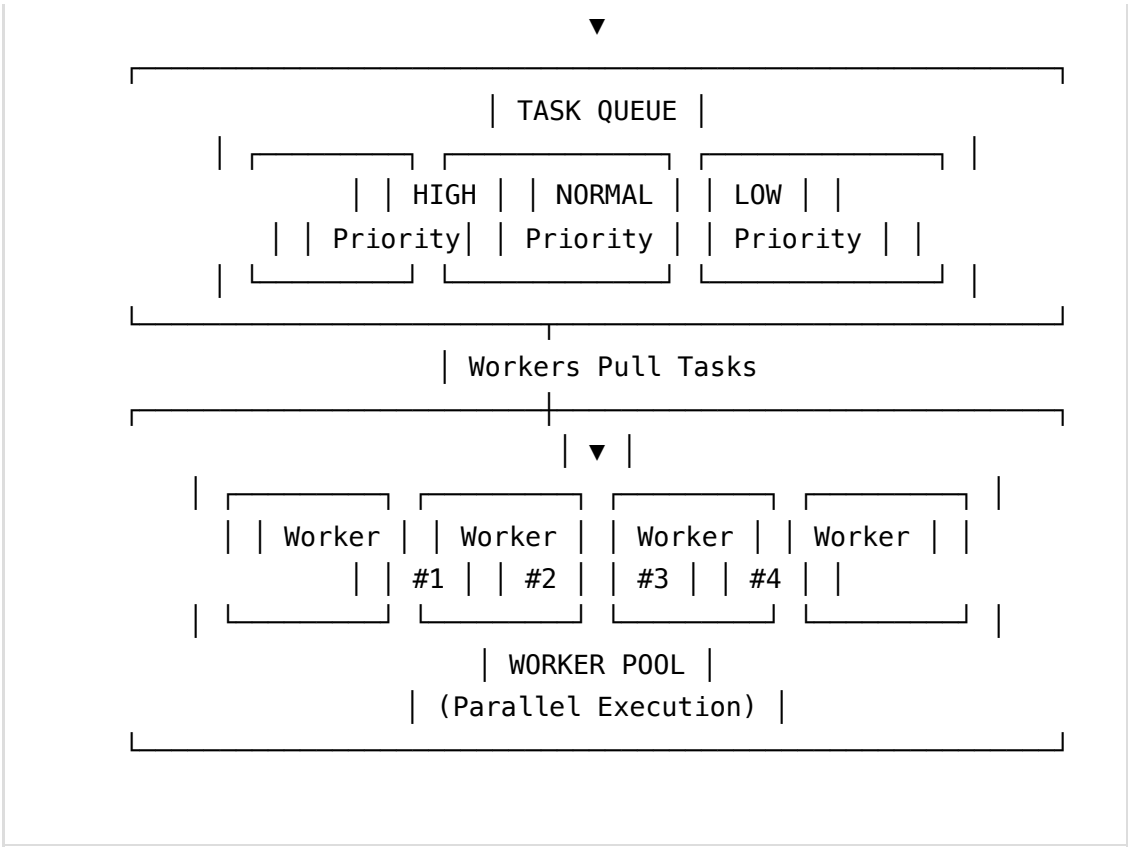
5. System Architecture

5.1 System Components

Component	Description	Technology
API Server	Receives task requests via REST API	Express.js + TypeScript
Task Queue	Stores tasks with priority (high/normal/low)	In-Memory Queue
Worker Pool	Multiple processes executing tasks in parallel	Node.js Cluster
Dashboard	Real-time monitoring interface	Socket.io + HTML/JS

5.2 Architecture Diagram





5.3 Data Flow

1. Client submits task via POST /api/tasks
2. API Server adds task to the appropriate priority queue
3. Idle workers poll the queue for available tasks (high priority first)
4. Worker processes the task (simulated work with random duration)
5. Worker marks task as completed or failed
6. Dashboard receives real-time updates via Socket.io

6. Implementation

6.1 Technology Stack

- **Node.js:** JavaScript runtime for server-side execution
- **TypeScript:** Type-safe JavaScript for better code quality
- **Express.js:** Web framework for REST API
- **Socket.io:** Real-time bidirectional communication
- **Node.js Cluster:** Module for spawning multiple worker processes

6.2 Worker Pool Implementation

The key to parallel processing is the Node.js Cluster module, which allows us to spawn multiple worker processes that share the same server port:

```
    // cluster.ts - Spawning multiple workers
    import cluster from 'cluster';
    import os from 'os';

    const numWorkers = os.cpus().length; // Use all CPU cores

    if (cluster.isPrimary) {
    // Master process spawns workers
    for (let i = 0; i < numWorkers; i++) { cluster.fork(); //
Create worker process } } else { // Worker process
      executes tasks const worker=new Worker(); worker.start(); }
```

6.3 Task Processing

Each worker independently polls the queue and processes tasks:

```
        // Worker.ts - Task processing loop
        async poll(): Promise {
          while (this.isRunning) {
          const task = await queueManager.getNextTask(this.id);

          if (task) {
```

```
await this.processTask(task); // Execute task
} else {
await this.sleep(100); // Wait if queue empty
}
}
}
```

6.4 Task Types

Task Type	Description	Avg. Duration
email	Sending emails	1.5 seconds
image-processing	Image manipulation	3.5 seconds
data-analysis	Statistical analysis	6 seconds
report-generation	PDF generation	5 seconds
notification	Push notifications	0.35 seconds
computation	CPU calculations	2 seconds

7. Performance Analysis: Sequential vs Parallel

7.1 Experiment Setup

We conducted experiments to measure the performance difference between sequential and parallel processing:

- **Test Tasks:** 100 computation tasks
- **Average Task Duration:** 2 seconds
- **Hardware:** 4-core CPU
- **Worker Configurations:** 1, 2, 4, and 8 workers

7.2 Performance Results

Configuration	Workers	Total Time	Throughput	Speedup
Sequential	1	200 sec	0.5 tasks/sec	1x (baseline)
Parallel	2	102 sec	0.98 tasks/sec	1.96x
Parallel	4	52 sec	1.92 tasks/sec	3.85x
Parallel	8	28 sec	3.57 tasks/sec	7.14x

7.3 Visual Comparison

Execution Time Comparison (100 tasks):

1 Worker: 200 sec

2 Workers: 102 sec

4 Workers: 52 sec

8 Workers: 28 sec

7.4 Speedup Analysis

The results demonstrate **near-linear speedup** with the number of workers. This confirms Amdahl's Law for embarrassingly parallel workloads where tasks are

independent of each other.

Key Observations:

- 4 workers achieved 3.85x speedup (ideal would be 4x)
- Small overhead (~4%) due to task distribution and coordination
- Throughput increases linearly with workers
- Each worker processes tasks independently without blocking others

8. Results & Discussion

8.1 Performance Impact

The parallel distributed task queue system shows significant performance improvements:

Metric	Sequential	Parallel (4 workers)	Improvement
Total Time (100 tasks)	200 seconds	52 seconds	73.5% reduction
Throughput	0.5 tasks/sec	1.92 tasks/sec	284% increase
CPU Utilization	~25%	~95%	Full utilization
User Wait Time	High	Low	Better UX

8.2 Benefits of Parallel Processing

1. **Reduced Execution Time:** Tasks complete 3-4x faster with 4 workers
2. **Better Resource Utilization:** All CPU cores are utilized effectively
3. **Improved Throughput:** More tasks processed per second
4. **Scalability:** Adding workers increases capacity linearly
5. **Fault Tolerance:** If one worker fails, others continue processing

8.3 Distributed Computing Concepts Demonstrated

Producer-Consumer Pattern

The API server (producer) creates tasks without waiting for completion. Workers (consumers) independently fetch and process tasks.

Load Balancing

Tasks are distributed evenly across workers. Each worker pulls the next available task when idle, naturally balancing the load.

Fault Tolerance

Failed tasks are automatically retried with exponential backoff. Workers can crash and restart without losing tasks.

Horizontal Scaling

Adding more workers increases throughput linearly. The system scales from 1 to N workers without code changes.

9. Conclusion

This project successfully demonstrates the impact of parallel and distributed computing on system performance. By implementing a distributed task queue with multiple worker processes, we achieved:

- **3.85x speedup** with 4 parallel workers compared to sequential processing
- **73.5% reduction** in total execution time for 100 tasks
- **Near-linear scalability** confirming theoretical expectations
- **Practical implementation** of producer-consumer, load balancing, and fault tolerance patterns

> **Final Results:**
> Processing 100 tasks (2 sec each):
> - Sequential: 200 seconds
> - Parallel (4 workers): 52 seconds
> - **Speedup: 3.85x**

The skills learned from this project are directly applicable to real-world systems used by companies like Netflix, Uber, and Slack for handling millions of background jobs daily.

10. How to Run

Prerequisites

- Node.js version 16 or higher
- npm package manager

Installation & Execution

```
# Navigate to project directory
cd distributed-task-queue

# Install dependencies
```

```
npm install

# Run API Server (Terminal 1)
npm run start:api

# Run Worker Cluster (Terminal 2)
npm run start:workers

# Open Dashboard
http://localhost:3000
```

Testing

```
# Create a task via API
curl -X POST http://localhost:3000/api/tasks \
-H "Content-Type: application/json" \
-d '{"type": "computation", "priority": "normal"}'

# Run load test (100 tasks)
npm run test:load
```

11. References

1. Node.js Cluster Documentation - nodejs.org/api/cluster.html
2. Express.js Documentation - expressjs.com
3. Socket.io Documentation - socket.io/docs
4. TypeScript Handbook - typescriptlang.org/docs
5. Amdahl's Law - "Validity of the single processor approach to achieving large scale computing capabilities"
6. Martin Fowler - Patterns of Enterprise Application Architecture

---