Parallelization of Database System

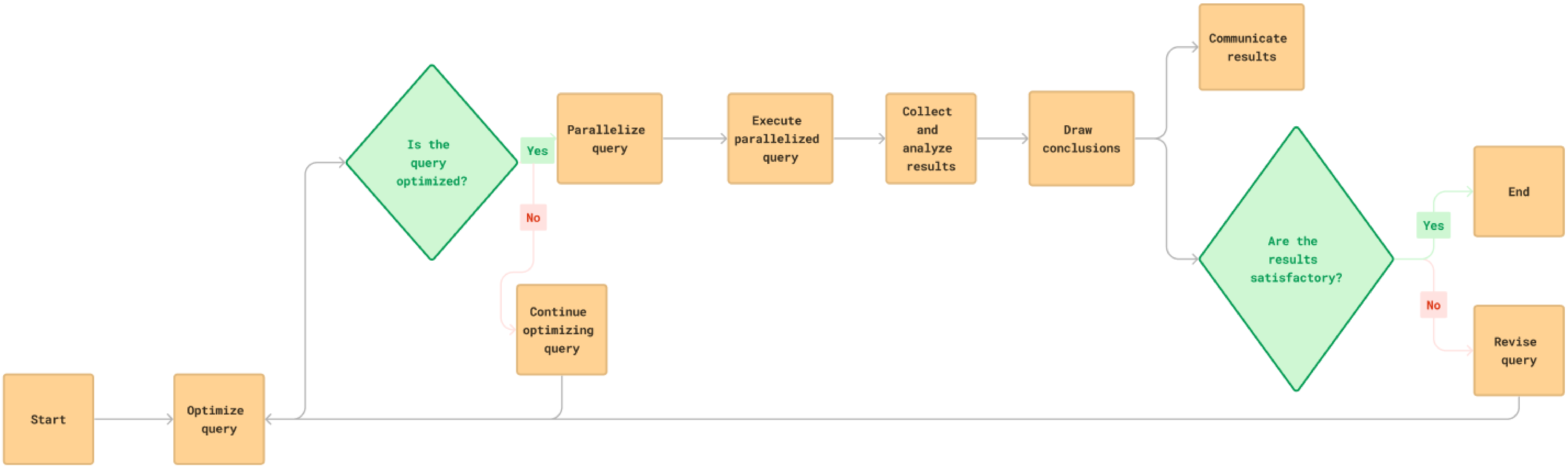
PRESENTED BY:

Pravalika Arunkumar & Shruthi Mohan

https://github.com/shruthimohan03/Parallel-Database-Architecture

Parallelization Aspects

Shared Memory Architeture	Loop Level Paralleism	Task Level Parallelism



Shared Memory Architecture:

Efficient Communication:
enables faster communication between components, reducing overhead and latency

Synchronization:
built-in mechanisms like locks ensure data consistency and prevent race conditions

Simplicity of Programming:
familiar paradigms like threads and locks simplify development

Scalability:
scale efficiently with the increasing number of CPU cores, accommodating growing demands without major architectural changes

Task Parallelism/Query Parallelism:

Task Granularity:
allows breaking down database operations into smaller, independent tasks, facilitating efficient utilization of resources and reducing contention

Parallel Task Execution:
by executing multiple tasks concurrently, task parallelism maximizes CPU utilization and throughput, enhancing overall system performance

Workload Balancing:
enables dynamic workload distribution across available resources, ensuring balanced utilization and minimizing idle time

Scalability:
scales efficiently with increasing workload demands, accommodating growing database requirements by parallelizing tasks across multiple cores or nodes

Loop-Level Parallelization/ Data Parallelization:

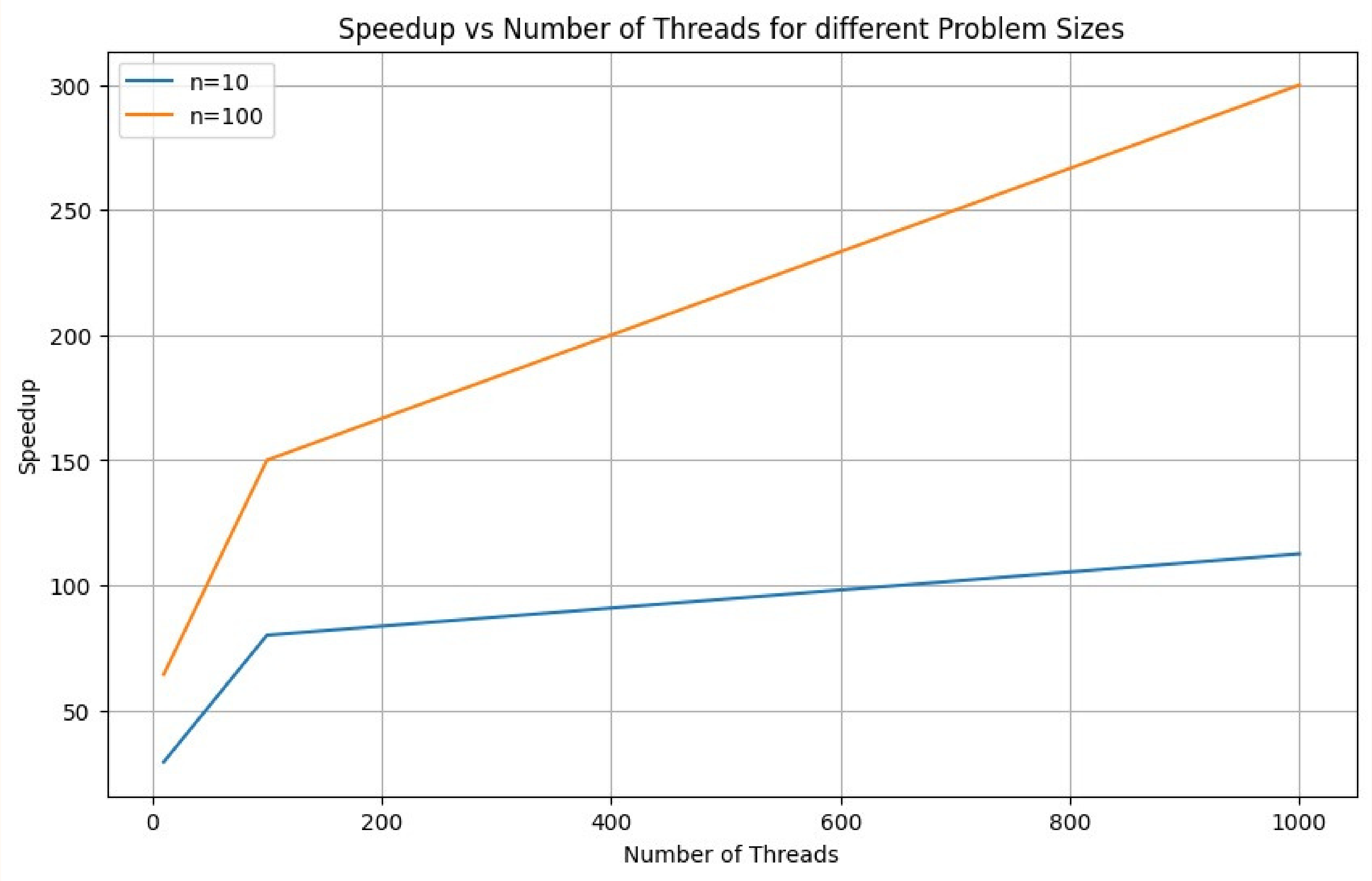
Fine-grained Optimization: optimizes specific tasks within the database, efficiently distributing workload across cores for improved performance in targeted operations

Multicore Utilization: maximizes the power of multicore processors, ensuring efficient resource usage and reduced execution time for data-intensive tasks within the database

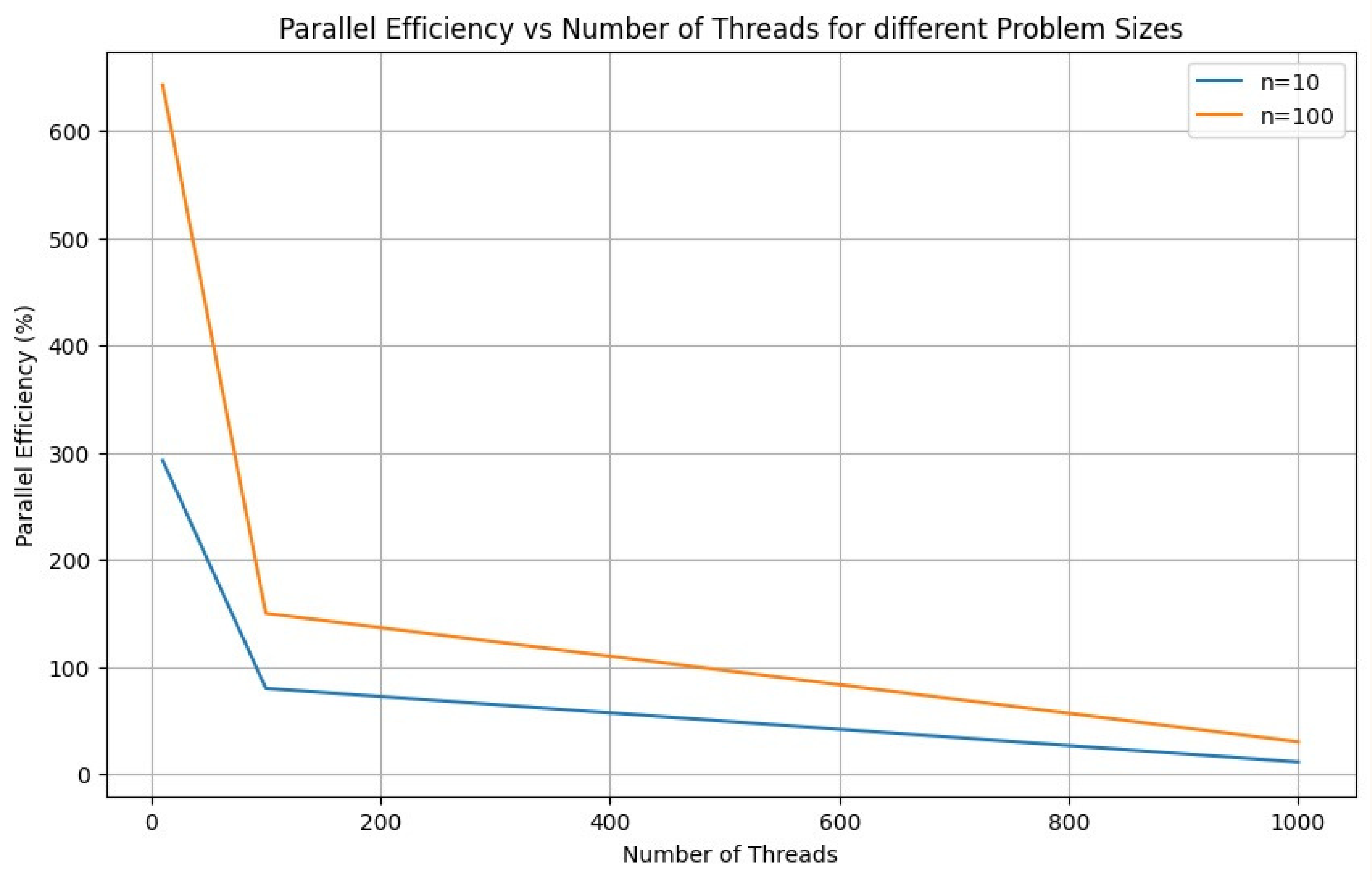
Concurrency Enhancement: boosts system concurrency by enabling concurrent execution of independent iterations, enhancing resource utilization, especially in highly parallel scenarios.

Scalability: database dynamically scales processing capacity to match workload demands, maintaining consistent performance and responsiveness as the number of cores increases or workload complexity changes

Results



Speedup vs Number of Threads for different Problem Sizes
n=10
n=100
Speedup
Number of Threads



Parallel Efficiency vs Number of Threads for different Problem Sizes
n=10
n=100
Parallel Efficiency (%)
Number of Threads

Results

```cpp
// Parallelize querying employee performance metrics
#pragma omp parallel num_threads(numThreadsToUse)
{
    // Get the thread ID
    int threadID = omp_get_thread_num();

    // Each thread performs a portion of the query
    std::vector<int> threadEmployeeIDs;
    for (int i = threadID; i < employeeIDs.size(); i += numThreadsToUse) {
        threadEmployeeIDs.push_back(employeeIDs[i]);
    }

    // Query employee performance metrics in parallel
    std::vector<PerformanceMetrics> threadResults = queryPerformanceMetrics(threadEmployeeIDs);

    // Combine the results from all threads
    #pragma omp critical
    {
        parallelResults.insert(parallelResults.end(), threadResults.begin(), threadResults.end());
    }
}
```

```
Parallel Query Results with 10 threads (Time: 0.056 seconds)
Parallel Query Results with 100 threads (Time: 0.024 seconds)
Parallel Query Results with 1000 threads (Time: 0.012 seconds)
```