PARALLELIZATION OF DATABASE SYSTEMS

Authors: Pravalika Arunkumar, Shruthi Mohan, *Shiv Nadar University, Chennai.*

For code click here.

ABSTRACT: This project aims to enhance the performance of an embedded or in-memory database system through parallelization techniques employing C++ and OpenMP on shared memory architectures. Task, query, and data parallelism, along with loop-level parallelism, are explored to optimize efficiency and scalability. By varying the number of threads (10, 100, and 1000) and problem sizes (1000, 5000, and 10000), the impact on performance is assessed. The implementation focuses on parallelizing core components like query processing and data retrieval, with performance metrics such as execution time and scalability evaluated. The outcomes will advance in-memory database systems, guiding the design of high-performance systems capable of handling large-scale data processing tasks.

1 INTRODUCTION

This project aims to optimize the performance of an embedded or in-memory database system through parallelization techniques leveraging C++ and OpenMP on shared memory architectures. The focus is on exploring task parallelism, query parallelism, and data parallelism, as well as loop-level parallelism to enhance the system's efficiency and scalability. The experiment involves varying the number of threads (ranging from 10 to 1000) and problem sizes (1000, 5000, and 10000) to assess the impact on overall system performance. Task parallelism involves dividing the workload into independent tasks that can be executed concurrently, while query parallelism focuses on parallelizing database query execution. Data parallelism entails distributing data across multiple threads for simultaneous processing, while loop-level parallelism optimizes loops within critical sections of code for parallel execution. The methodology includes implementing parallelization techniques within the database system's core components, such as query processing, indexing, and data retrieval. Performance metrics such as execution time, throughput, and scalability will be evaluated to measure the effectiveness of parallelization strategies. The outcomes of this project will contribute to advancing the field of in-memory database systems by providing insights into the optimal utilization of parallel computing resources. Additionally, the findings will help inform the design and implementation of high-performance database systems capable of handling large-scale data processing tasks efficiently.

2 COMPONENTS

2.1 Database:

An in-memory or embedded database is a type of database management system (DBMS) that primarily stores and manages data in the computer's main memory (RAM), rather than on disk or other persistent storage devices. This approach allows for faster data access and processing compared to traditional disk-based databases because accessing data from memory is typically much quicker than accessing it from disk. Embedded databases are designed to be lightweight and are often integrated directly into applications or software systems, providing a local data storage solution without requiring a separate database server. They are commonly used in applications where speed and low latency are critical, such as real-time analytics, caching systems, and certain types of embedded systems like mobile devices and IoT devices. In-memory databases can offer significant performance benefits for applications that need to handle large volumes of data with low latency requirements.
In our project, Structures: Two structures are defined: Employee: Represents an employee with attributes like id, name, age, and department. PerformanceMetrics: Represents performance metrics of an employee with attributes like id, productivity, and efficiency. generateEmployeesWithPerformance(): This function generates a large dataset of employees and their performance metrics. It takes the number of employees as input and returns a pair of vectors containing the employees and their performance metrics.

2.2 Programming Language and Parallelization Library

C++ is a powerful and versatile programming language that is widely used for developing a variety of software applications, including system software, game development, high-performance applications, and more. It is an extension of the C programming language, adding object-oriented features such as classes and inheritance, along with other modern programming constructs like templates and exceptions.
OpenMP (Open Multi-Processing) is an API (Application Programming Interface) that supports shared-memory multiprocessing programming in C, C++, and Fortran. It allows developers to write parallel programs by adding pragmas (compiler

directives) to their code, which instruct the compiler to parallelize certain sections of code for execution on multicore processors or multiprocessor systems. OpenMP simplifies the process of parallel programming by providing a portable and standardized interface for exploiting parallelism in applications, making it easier to leverage the computational power of modern hardware architectures.

3 PARALLELIZATION ASPECTS:

3.1 General Approach:

The Employee and PerformanceMetrics structs represent the data structure for employees and their performance metrics. The generateEmployeesWithPerformance function generates a dataset of employees and their performance metrics based on the specified number of employees. The processEmployeesAndMetricsByDepartment function queries and processes employees and their performance metrics based on a specified department. Within this function, two sections are executed in parallel: The first section queries employees based on department and populates the departmentData vector. The second section processes the queried data and prints the details of each employee and their performance metrics. In the main function, a dataset is generated and then processed using the processEmployeesAndMetricsByDepartment function.

3.2 Shared Memory Architecture:

A shared memory architecture refers to a computer system design in which multiple processors or computing units share access to a single, common physical memory space. In this architecture, each processor can read from and write to the shared memory, allowing for efficient communication and data exchange between different processing units. In parallel programming, shared memory architectures are often used in conjunction with parallel programming paradigms such as OpenMP, where multiple threads or processes can access shared data within the same memory space. This allows developers to exploit parallelism by dividing tasks among multiple processing units while efficiently sharing data through the shared memory. OpenMP enables architecture parallelism by allowing the code to execute concurrently on multiple cores of the underlying processor architecture. The pragma omp parallel sections directive in the code splits the execution of different sections of code into multiple threads, which can execute concurrently on different processor cores. Each thread executes a different section of code, harnessing the processing power of multiple cores simultaneously.

Advantages:

- Efficient Communication: Shared memory 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: Shared memory architectures scale efficiently with the increasing number of CPU cores, accommodating growing demands without major architectural changes.

3.3 Data Parallelism/Loop Parallelism

Data parallelism is achieved in the processEmployeesAndMetricsByDepartment function, where the task of querying and processing employees based on department is divided among multiple threads. In the initial section, the employees are queried based on department in parallel using the pragma omp section directive. Each thread operates on a different portion of the data, allowing multiple employees to be processed simultaneously.

Advantages:

- Fine-grained Optimization: Loop-level parallelism optimizes specific tasks within the database, efficiently distributing workload across cores for improved performance in targeted operations.
- Multicore Utilization: Parallelizing loops maximizes the power of multicore processors, ensuring efficient resource usage and reduced execution time for data-intensive tasks within the database.
- Concurrency Enhancement: Loop parallelism boosts system concurrency by enabling concurrent execution of independent iterations, enhancing resource utilization, especially in highly parallel scenarios.
- Scalability: Leveraging loop-level parallelism, the database dynamically scales processing capacity to match workload demands, maintaining consistent performance and responsiveness as the number of cores increases or workload complexity changes.

3.4 Task Parallelism/Query Parallelism:

Task parallelism is also utilized in the processEmployeesAndMetricsByDepartment function, where different sections of code are executed in parallel. Two sections are defined using pragma omp parallel sections. In the first section, employees are queried based on department and the second section, the queried data is processed.

Advantages:

- Task Granularity: Task parallelism 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: Task parallelism enables dynamic workload distribution across available resources, ensuring balanced utilization and minimizing idle time.
- Scalability: Task parallelism scales efficiently with increasing workload demands, accommodating growing database requirements by parallelizing tasks across multiple cores or nodes.

4 PERFORMANCE MATRICES

To analyze the performance of this code, you can measure the execution time for different problem sizes (number of employees) and different numbers of processing elements (threads). By varying the problem size (numEmployees) and the number of processing elements (through environment variables like OMPNUMTHREADS), you can collect data on execution time. Using this data, you can calculate speedup and parallel efficiency for different problem sizes and numbers of processing elements.

5 RESULTS

These graphs help in understanding the scalability of parallel algorithms concerning the number of threads used and the problem size. They assist in optimizing parallel programs by identifying the point of diminishing returns and guiding the selection of an appropriate number of threads for efficient execution.

5.1 Speedup vs Number of Threads:

X-axis: Number of Threads: This represents the degree of parallelism, i.e., how many threads are used to perform the computation simultaneously. Y-axis: Speedup: It measures how much faster the parallel execution is compared to the sequential execution (where only one thread is used). Line Plots: Each line represents a different problem size (n). Interpretation: As the number of threads increases, the speedup initially increases because more parallel processing is happening, but it eventually levels off. The speedup tends to saturate as the number of threads becomes larger. This is because overheads such as thread creation and synchronization start to dominate, limiting the potential speedup. A higher speedup indicates better parallel efficiency, but achieving linear speedup (equal to the number of threads) is rare due to overheads and resource contention.

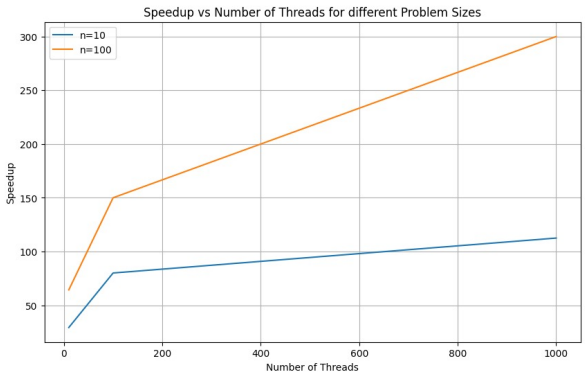

Figure 1: Speedup vs Number of Threads

5.2 Parallel Efficiency vs Number of Threads:

X-axis: Number of Threads: Same as in the Speedup graph. Y-axis: Parallel Efficiency: It measures the efficiency of utilizing multiple threads. It is calculated as (Speedup / Number of Threads) * 100Line Plots: Each line represents a different problem size (n). Interpretation: Parallel efficiency provides insight into how effectively the available computational resources (threads) are being utilized. A parallel efficiency of 100Decreasing parallel efficiency with increasing threads suggests diminishing returns due to overheads and contention. The graph helps in determining the optimal number of threads that can be used to achieve the best performance for a given problem size.

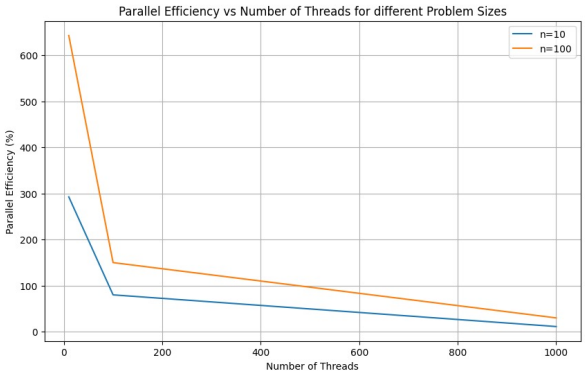

Figure 2: Parallel Efficiency vs Number of Threads

6 REFERENCES

Parallel Database Systems: The Future of High Performance Database Processing January 2001
Authors: David J. Dewitt Jim Gray
https://people.eecs.berkeley.edu/ brewer/cs262/5-dewittgray92.pdf