MPI Parallelization for Approximate Pattern Matching

To improve the efficiency of the approximate pattern matching algorithm, we implemented parallelization using **MPI (Message Passing Interface)**. The main idea is to **distribute the input text across multiple processes**, allowing each process to compute pattern matches in its assigned chunk independently. This significantly reduces the number of characters each process needs to check, improving performance, especially for large text files. Though it greatly improves performance, it doesn't significantly change the underlying logic, allowing for other optimization and parallelization techniques.

Parallelization Strategy

1. Text Partitioning

- The input text is divided among `N` processes, where `N` is the number of available MPI ranks.
- Each process reads and fixes it's own contiguous **chunk** of the text for processing.

2. Handling Overlapping Boundaries

- Since pattern matching may require examining text beyond a process's assigned chunk (to detect matches at chunk boundaries), an **overlap of `max_pattern_length - 1` characters** is included from the neighboring process.
- The first and last processes handle edge cases where no adjacent chunk exists.

3. Parallel Computation of Matches

- Each process iterates over its text segment and applies the **Levenshtein distance algorithm** to compare each pattern against substrings within its chunk.
- Matches are counted independently in each process.

4. Gathering Results

- After local computation, each process sends its results to the **master process (rank 0)** using `MPI_Reduce`.
- The master process aggregates the match counts from all ranks and produces the final result.

Performance Benefits

- **Workload distribution:** Each process handles a smaller portion of the text, reducing computational time.
- **Scalability:** The algorithm can efficiently scale across multiple nodes in a distributed system.
- **Space for other optimisations:** The task is reduced by `N` without changing the logic, allowing for further use of **OMP** and **CUDA**.

OpenMP Optimization for Approximate Pattern Matching

Following the implementation of **MPI-based parallelization**, OpenMP was introduced to further optimize the performance of the approximate pattern matching algorithm. Since MPI already distributes the workload across multiple processes, OpenMP was applied within each MPI process to **leverage multi-threading on shared-memory nodes**. The primary objective was to further reduce execution time by improving intra-node parallelism while minimizing memory overhead and ensuring thread safety.

Parallelization Strategy

The core computation consists of two nested loops: 1. **Outer loop over patterns**
- Each pattern is processed independently within an MPI process. 2. **Inner loop over text positions**
- Each position in the assigned text chunk is checked for a potential match with the current pattern using the **Levenshtein distance**.

Different OpenMP Configurations Tested

Since MPI already distributes the text across multiple processes, OpenMP was applied within each MPI rank using different configurations:

1. **Parallelizing the Outer Loop (loop over patterns)**
   - Each OpenMP thread within an MPI process was assigned a different pattern.
   - This approach provided performance improvements, as pattern computations are independent, preventing race conditions.
2. **Parallelizing the Inner Loop (loop over text positions)**
   - Each thread processed different positions in the assigned text chunk for a given pattern.
   - However, a race condition emerged due to shared memory usage in the `levenshtein()` function, where all threads accessed the same `column` array.

- A **thread-private `column` array** was introduced to eliminate conflicts, but memory allocation overhead negatively impacted performance.

3. **Parallelizing Both Loops**
   - **Nested parallelism** was attempted, where the outer loop was parallelized over patterns and the inner loop over text positions.
   - However, OpenMP's default handling of nested parallelism resulted in **thread oversubscription**, leading to excessive context-switching overhead rather than performance gains.
4. **Reducing Memory Access**
   - Since each thread required a separate `column` array, memory preallocation was tested to reduce repeated allocations inside the loop.
   - Although this reduced memory overhead, it did not significantly improve performance.

Results and Observations

Despite the optimizations, OpenMP did not provide substantial speedup beyond what was already achieved using MPI. The key limiting factors were:

1. **Levenshtein distance computation remains sequential**
   - The dynamic programming table (`column` array) must be computed iteratively, which restricts parallel efficiency.
2. **Memory overhead from per-thread allocations**
   - Allocating and freeing the `column` array for each thread introduced additional overhead, reducing expected performance gains.
3. **Diminishing returns due to MPI optimizations**
   - Since MPI had already divided the workload efficiently among processes, OpenMP's added thread-level parallelism did not yield a proportional speedup.

Conclusion

While OpenMP provided additional parallelism within MPI processes, **the primary performance improvements came from MPI-based workload distribution**. The most effective OpenMP configuration was **parallelizing only the outer loop (pattern loop)** within each MPI rank while optimizing memory allocation. However, further improvements were limited due to the inherently sequential nature of the Levenshtein distance computation.

CUDA and OpenMP Optimization for Approximate Pattern Matching

To further accelerate the **Approximate Pattern Matching** algorithm, CUDA was integrated to offload computationally expensive operations to the GPU.

Additionally, OpenMP was introduced to parallelize kernel launches at the CPU level, ensuring efficient multi-threading in a hybrid CPU-GPU environment. This approach maximizes parallelism by leveraging **GPU acceleration** for intensive computation and **OpenMP** for managing concurrent executions.

However, **OpenMP did not provide a significant performance improvement**, as the CUDA implementation was already efficiently parallelized on the GPU. The OpenMP optimization mainly helped with organizing kernel launches but did not lead to noticeable speedups in execution time.

CUDA Optimization Strategy

1. Parallelizing Text Processing on the GPU

- The **Levenshtein distance computation** for different starting positions in the text is **independent**, making it well-suited for GPU parallelization.
- A **CUDA kernel** was implemented where **each thread processes one starting position** in the text.
- **Atomic operations (`atomicAdd`)** were used to safely accumulate match counts.

2. Memory Management and Data Transfer Optimization

- Instead of copying each pattern separately, a **single large memory buffer** (`d_patterns`) was allocated for all patterns, reducing the number of memory transfers.
- A **pattern offset array (`d_pattern_offsets`)** was introduced to store the position of each pattern in `d_patterns`, ensuring correct memory access for each thread.
- **Pinned memory** could be used in future optimizations to further accelerate host-to-device data transfers.

3. CUDA Kernel Execution Model

- **Grid Configuration:**
  - Each **block** processes a subset of the text.
  - Each **thread** handles one character position in the text.
- **Workload Distribution:**
  - The **Levenshtein distance** is computed independently per thread.
  - Each thread updates the **match count** for its assigned text position.

Merging CUDA with OpenMP

1. Parallelizing Kernel Launches

- OpenMP was applied at the **CPU level** to **parallelize kernel launches**, ensuring that multiple CUDA kernels could be launched concurrently.
- OpenMP parallelization was applied to the loop that **dispatches CUDA kernels** for different patterns.

2. OpenMP's Impact on Performance

- **Despite parallelizing kernel launches, OpenMP did not lead to a significant speedup.**
- The main reason for this is that the **GPU execution was already highly parallelized**, and launching kernels concurrently at the CPU level did not significantly reduce execution time.
- **The bottleneck was in memory transfers and computation on the GPU, which OpenMP does not optimize.**

3. Avoiding Race Conditions

- **CUDA API calls (`cudaMalloc`, `cudaMemcpy`) are not thread-safe**, so all memory allocations were performed **outside the OpenMP parallel region**.
- OpenMP was **only used to parallelize kernel launches**, preventing memory corruption.
- Each pattern was **assigned a separate memory offset** to avoid memory overwrites.

Performance Improvements and Scalability

Efficient GPU Utilization

- The **Levenshtein distance computation** is highly parallelized, with **thousands of threads** running concurrently on the GPU.
- The **GPU efficiently processes large text files**, significantly reducing computation time.

Optimized Memory Access

- Using **a single memory allocation for all patterns** instead of multiple allocations reduced memory transfer overhead.
- **Minimized host-to-device and device-to-host memory copies** improved performance.

Hybrid CPU-GPU Execution (Limited OpenMP Impact)

- OpenMP parallelization at the CPU level allowed **concurrent kernel execution**, but **did not significantly improve performance** due to the dominant role of GPU execution.
- The **hybrid MPI + CUDA + OpenMP** approach ensures scalability across multiple GPUs and CPU cores.

Conclusion

The integration of **CUDA and OpenMP** successfully **accelerates approximate pattern matching**, with **CUDA providing the primary speedup**. **OpenMP's impact was minimal**, as the GPU already handled parallel execution efficiently. The main benefit of OpenMP was in **organizing kernel launches**, but it did not significantly reduce overall execution time.

Future Optimizations

- **Use shared memory in CUDA** to optimize memory access in the Levenshtein kernel.
- **Improve load balancing** by dynamically adjusting work distribution across CPU and GPU threads.
- **Explore asynchronous memory transfers** to further overlap computation and data transfer.

This optimized hybrid **CUDA + OpenMP + MPI** approach **significantly reduces execution time**, but future improvements should focus on **further optimizing GPU memory access** and **reducing data transfer overhead**, as **OpenMP does not contribute significantly to performance gains**.