MPI DNA Sequence Alignment

Primary Objective

This implementation is design to showcase:

- MPI programming using collective operation
- An architecture that can be scaled across multiple processes
- Application of parallel computing to real-world sequence alignment problems

Included features:

- MPI collective operations (Scatterv/Gatherv) implementation
- Fixed-size data structures optimized for MPI communication
- Round-robin load balancing algorithm
- Sliding window DNA sequence alignment

Excluded features:

- Advanced alignment algorithms
- Real genomic file
- Persistent storage/Database

Scope

The target scale is 2-16 processes, which is suited for desktop. The data scale from 10 to 1000 query sequences.

Algorithm detail

Input data

- **Reference DNA**: a large DNA sequence representing a chromosome segment
- **Sequence**: Short DNA segment that align with the reference

Program flow

- MPI_Scatterv and MPI_Gatherv for optimal data distribution

Main data structure

```
// Main data structures
struct QueryData {
    int queryIndex;
    int queryLength;
    char querySequence[64];
};

struct ResultData {
    int queryIndex, position, score;
    int queryLength, matchLength;
    char querySequence[64], matchedSegment[64];
};
```

MPI environment setup

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numProcesses);
```

We first initialize MPI runtime environment and determine the process rank from 0 to `numProcesses - 1`

OpenMP environment setup

```
int numThreads = omp_get_max_threads();
if (getenv("OMP_NUM_THREADS") == nullptr) {
    // default: use 4 threads or number of cores, whichever is smaller
    numThreads = min(4, (int)omp_get_num_procs());
    omp_set_num_threads(numThreads);
}
```

Adaptive threading since more number of thread than the number of core can bottleneck the system, making multithreading redundant.

Data generation

We will generate random query sentences

Data Structure conversion

```
// Convert strings to fixed-size MPI structures
vector<QueryData> allQueryData(numQueries);
for (int i = 0; i < numQueries; i++) {
```

```
    allQueryData[i] = QueryData(i, queries[i]);
}
```

We transform variable-length strings to fixed-size structures. This is done to optimize MPI collective communication.

Distribution Calculation

```
// Calculate send counts for round-robin distribution
vector<int> sendCounts(numProcesses, 0);
for (int i = 0; i < numQueries; i++) {
    sendCounts[i % numProcesses]++;
}

// Calculate byte-level displacements
vector<int> sendCountsBytes(numProcesses);
vector<int> displacementsBytes(numProcesses);
for (int i = 0; i < numProcesses; i++) {
    sendCountsBytes[i] = sendCounts[i] * sizeof(QueryData);
    displacementsBytes[i] = (i == 0) ? 0 :
        displacementsBytes[i-1] + sendCountsBytes[i-1];
}
```

We use Round-Robin for load balancing to ensures even distribution across processes.

Query Distribution (MPI_Scatterv)

```
// All processes participate in scatter operation
MPI_Scatterv(allQueryData.data(), // Send buffer (master only)
             sendCountsBytes.data(), // Send counts per process
             displacementsBytes.data(), // Send displacements
             MPI_BYTE, // Send type
             myQueries.data(), // Receive buffer
             sendCountsBytes[rank], // Receive count
             MPI_BYTE, // Receive type
             0, // Root process
             MPI_COMM_WORLD); // Communicator
```

Sequence matching

Each process will perform the `alignSequence()` or `smithWatermanAlignment()` functions. OpenMP is used for multithreaded functionality.

Result Gathering (MPI_Gatherv)

```
// Collective gather operation
MPI_Gatherv(myResults.data(), // Send buffer
            myResults.size() * sizeof(ResultData), // Send count
            MPI_BYTE, // Send type
            allResults.data(), // Receive buffer (master only)
            recvCountsBytes.data(), // Receive counts
            recvDisplacementsBytes.data(), // Receive displacements
            MPI_BYTE, // Receive type
            0, // Root process
            MPI_COMM_WORLD); // Communicator
```

We will determine the receive buffer layout and collect all results. The results are organized by displacement offset. This tell MPI where each process's data begin within the send/receive buffer

Result Processing at Master process

```
// Convert ResultData back to AlignmentResult
vector<AlignmentResult> finalResults;
for (const auto& resultData : allResults) {
    if (resultData.queryIndex >= 0) {  // Valid result
        finalResults.push_back(resultData.toAlignmentResult());
    }
}

// Sort by query index for ordered display
sort(finalResults.begin(), finalResults.end(),
     [](const AlignmentResult& a, const AlignmentResult& b) {
         return a.queryIndex < b.queryIndex;
     });
```

Clean up

After that we finalize MPI, release MPI resources, close communication channels, and terminate the program

Sequence matching algorithm

Simple SIding Window algorithm

The `alignSequence()` functions performs the core alignment logic:

- **Sliding Window**
  - The query sequence slides across the entire reference DNA

- At each position, we extract the segment of the reference DNA with the same number of base to the query length
- This creates a window that moves one base at a time across the reference sequence

- **Scoring System**
  - For each position, we get the alignment score.
  - The score is calculated by the number of exact base matches between the query and the reference sequence. The maximum score is the length of the query sequence, as all base in the query sequence match with the reference segment.

```
AlignmentResult alignSequence(const string& query, int queryIndex) {
    // Shared variables for best result tracking
    int bestScore = -1;
    int bestPosition = -1;
    string bestMatch;

    size_t numPositions = REFERENCE_DNA.length() - query.length() + 1;

    // Create parallel region
    #pragma omp parallel
    {
        // Thread-private variables for local optimization
        int localBestScore = -1;
        int localBestPosition = -1;
        string localBestMatch;

        // Parallel work distribution
        #pragma omp for schedule(dynamic)
        for (size_t i = 0; i < numPositions; i++) {
            // Each thread processes different positions
            string segment = REFERENCE_DNA.substr(i, query.length());

            // Calculate alignment score (number of matching bases)
            int score = 0;
            for (size_t j = 0; j < query.length(); j++) {
                if (query[j] == segment[j]) {
                    score++;
                }
            }

            // Update thread-local best result
            if (score > localBestScore) {
                localBestScore = score;
                localBestPosition = i;
                localBestMatch = segment;
```

```
            }
        }

        // Critical section for combining thread results
        #pragma omp critical(best_update)
        {
            if (localBestScore > bestScore) {
                bestScore = localBestScore;
                bestPosition = localBestPosition;
                bestMatch = localBestMatch;
            }
        }
    }

    return AlignmentResult(queryIndex, bestPosition, bestScore, query,
bestMatch);
}
```

This is quite a simple and fast algorithm to implement, making it easy to parallelize across multiple queries

OpenMP constructs explained:

- `#pragma omp for schedule(dynamic):` Distributes loop iterations
- `#pragma omp critical(best_update):` Serializes access to shared data
- Each thread will maintain its local state

Smith-Waterman algorithm

The Smith-Waterman algorithm finds the optimal alignment between two sequences using dynamic programming using a scoring system that follows:

- **MATCH_SCORE = 2**: Reward for matching bases
- **MISMATCH_SCORE = -1**: Penalty for mismatched bases
- **GAP_PENALTY = -1**: Penalty for insertions/deletions

Matrix building phase using Anti-Diagonal Parallelization

```
// Process matrix along anti-diagonals for parallelization
for (int diag = 1; diag <= queryLen + refLen; diag++) {
    int startI = max(1, diag - refLen);
    int endI = min(queryLen, diag - 1);

    if (startI <= endI) {
        #pragma omp parallel for schedule(static) reduction(max:maxScore)
```

```
        for (int i = startI; i <= endI; i++) {
            int j = diag - i;
            if (j >= 1 && j <= refLen) {
                // Calculate scores for three possible moves
                int matchScore = scoreMatrix[i-1][j-1] +
                                 ((query[i-1] == REFERENCE_DNA[j-1]) ?
MATCH_SCORE : MISMATCH_SCORE);
                int deleteScore = scoreMatrix[i-1][j] + GAP_PENALTY;
                int insertScore = scoreMatrix[i][j-1] + GAP_PENALTY;

                // Take maximum score (local alignment allows 0)
                scoreMatrix[i][j] = max({0, matchScore, deleteScore,
insertScore});
```

Anti-Diagonal because elements on the same anti-diagonal do not have any dependency on each other. Hence we can calculate different positions at the same time and still got the correct matrix.

Traceback Phase (Sequential within each alignment)

```
// Traceback to reconstruct optimal alignment
while (i > 0 && j > 0 && scoreMatrix[i][j] > 0) {
    int currentScore = scoreMatrix[i][j];
    // Calculate how we arrived at this cell

    if (currentScore == matchScore) {
        // Match/mismatch: move diagonally
        alignedQuery = query[i-1] + alignedQuery;
        alignedRef = REFERENCE_DNA[j-1] + alignedRef;
        i--; j--;
    } else if (currentScore == deleteScore) {
        // Gap in query: move up
        alignedQuery = "-" + alignedQuery;
        alignedRef = REFERENCE_DNA[j-1] + alignedRef;
        i--;
    } else if (currentScore == insertScore) {
        // Gap in reference: move left
        alignedQuery = query[i-1] + alignedQuery;
        alignedRef = "-" + alignedRef;
        j--;
    }
}
```

Result

FASTA file information

```
Reference file: coronavirus.fasta
Query file: coronavirus.fasta
Algorithm: sw
Reference DNA loaded: 11925 bases
Query sequences loaded: 9 sequences
Sequence statistics:
  Total bases: 26940
  Average length: 2993.33 bases
  Length range: 237 - 11925 bases
```

Smith Waterson Algorithm

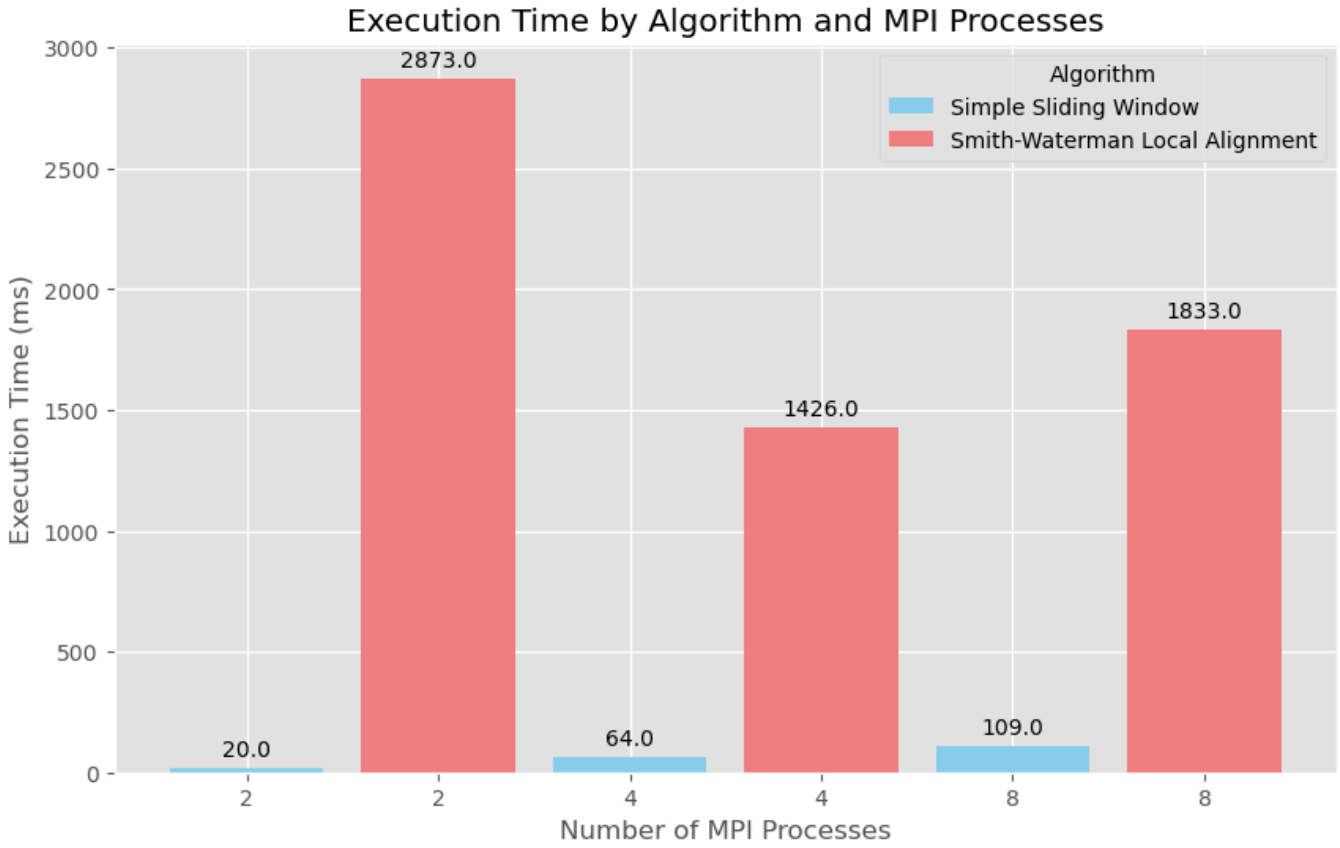
```
mpiuser@SIT315-Head:~/Cloud$ make run-sw
Running Smith-Waterman algorithm with coronavirus.fasta
Using coronavirus.fasta for reference and queries
OMP_NUM_THREADS=2 mpirun -np 4 -hostfile ./cluster ./mpi_dna_alignment coronavirus.fasta coronavirus.fasta sw
=== Hybrid MPI+OpenMP DNA Sequence Alignment ===
MPI Processes: 4
OpenMP Threads per process: 2
Total parallel workers: 8
MPI Threading support: Available


=== Loading FASTA Files ===
Reference file: coronavirus.fasta
Query file: coronavirus.fasta
Algorithm: sw
Reference DNA loaded: 11925 bases
Query sequences loaded: 9 sequences
Sequence statistics:
  Total bases: 26940
  Average length: 2993.33 bases
  Length range: 237 - 11925 bases

=== Hybrid MPI+OpenMP DNA Sequence Alignment ===
Algorithm: Smith-Waterman Local Alignment
Parallelization: MPI (distributed) + OpenMP (shared memory)
Reference DNA length: 11925 bases
Number of query sequences: 9
MPI processes: 4
OpenMP threads per process: 2
Total parallel workers: 8
Distributing tasks using MPI_Scatterv...
Master (rank 0) will process 3 queries
Process 1 will process 2 queries
Process 2 will process 2 queries
Process 3 will process 2 queries
Process 1 Thread 1 processed query 4 (lcl|NC_034972.1_cds_...)  (score: 234/237)
Process 3 Thread 1 processed query 8 (lcl|NC_034972.1_cds_...)  (score: 298/318)
Process 1 Thread 0 processed query 3 (lcl|NC_034972.1_cds_...)  (score: 588/645)
Process 3 Thread 0 processed query 7 (lcl|NC_034972.1_cds_...)  (score: 850/1170)
Process 2 Thread 1 processed query 6 (lcl|NC_034972.1_cds_...)  (score: 461/501)
Process 2 Thread 0 processed query 5 (lcl|NC_034972.1_cds_...)  (score: 665/747)
Master Thread 1 processed query 1 (lcl|NC_034972.1_cds_...) (score: 1543/8016)
Master Thread 1 processed query 2 (lcl|NC_034972.1_cds_...) (score: 901/3381)
Master Thread 0 processed query 0 (lcl|NC_034972.1_cds_...) (score: 3284/11925)

=== Alignment Results ===
Query                  Header  Position   Score      Query Sequence          Best Match
--------------------------------------------------------------------------------
    0lcl|NC_034972.1_cds_YP...      8091    3284/11925  ATGGCTAACCAATAT...  T-C-C-TGTGTGTAA...  (27.5%)
    1lcl|NC_034972.1_cds_YP...     10739    1543/8016  GAGCCCTGTAGTGAG...  TC--AT-GCCTG-T-...  (19.2%)
    2lcl|NC_034972.1_cds_YP...     11925     901/3381  ATGGCTCTCATTTTT...               Empty  (26.6%)
    3lcl|NC_034972.1_cds_YP...     11171     588/645  ATGATAGGTGGTCTT...  TG-CCAG-GTTGTTT...  (91.2%)
    4lcl|NC_034972.1_cds_YP...     11621     234/237  ATGTTACCCTCGTTT...  TATTAC-CCACGT-T...  (98.7%)
    5lcl|NC_034972.1_cds_YP...     11071     665/747  ATGGTACTCTTTTGT...  AAGGTTC--T-C-GG...  (89.0%)
    6lcl|NC_034972.1_cds_YP...     11423     461/501  ATGATTTTGGTTTTC...  TGGTTTT-GGTTTTT...  (92.0%)
    7lcl|NC_034972.1_cds_YP...     10698     850/1170  ATGAGTTCCAACGTC...  A-TGAGTTTTGAAAT...  (72.6%)
    8lcl|NC_034972.1_cds_YP...     11442     298/318  ATGTATTGTTTGTTG...  G-TACTGTTAG-TTG...  (93.7%)

Total execution time: 26565 ms
Algorithm used: Smith-Waterman
Threading model: 4 MPI processes × 2 OpenMP threads = 8 total workers
```

Sliding Window Algorithm

```
mpiuser@SIT315-Head:~/Cloud$ make run
Running with coronavirus.fasta as both reference and query source
Using coronavirus.fasta for reference and queries
OMP_NUM_THREADS=2 mpirun -np 4 -hostfile ./cluster ./mpi_dna_alignment coronavirus.fasta coronavirus.fasta simple
=== Hybrid MPI+OpenMP DNA Sequence Alignment ===
MPI Processes: 4
OpenMP Threads per process: 2
Total parallel workers: 8
MPI Threading support: Available


=== Loading FASTA Files ===
Reference file: coronavirus.fasta
Query file: coronavirus.fasta
Algorithm: simple
Reference DNA loaded: 11925 bases
Query sequences loaded: 9 sequences
Sequence statistics:
  Total bases: 26940
  Average length: 2993.33 bases
  Length range: 237 - 11925 bases

=== Hybrid MPI+OpenMP DNA Sequence Alignment ===
Algorithm: Simple Sliding Window
Parallelization: MPI (distributed) + OpenMP (shared memory)
Reference DNA length: 11925 bases
Number of query sequences: 9
MPI processes: 4
OpenMP threads per process: 2
Total parallel workers: 8
Distributing tasks using MPI_Scatterv...
Master (rank 0) will process 3 queries
Process 1 will process 2 queries
Process 2 will process 2 queries
Process 3 will process 2 queries
Master Thread 0 processed query 0 (lcl|NC_034972.1_cds_...) (score: 2368/11925)
Process 2 Thread 1 processed query 5 (lcl|NC_034972.1_cds_...)  (score: 240/747)
Process 2 Thread 0 processed query 6 (lcl|NC_034972.1_cds_...)  (score: 185/501)
Process Process 3 Thread 1 processed query 8 (lcl|NC_034972.1_cds_...)  (score: 119/1318)
Process 3 Thread 0 processed query 7 (lcl|NC_034972.1_cds_...)  (score: 333/1170)
 Thread 1 processed query 4 (lcl|NC_034972.1_cds_...)  (score: 92/237)
Process 1 Thread 0 processed query 3 (Master Thread 0 processed query lcl|NC_034972.1_cds_...)  (score: 218/645)
2 (lcl|NC_034972.1_cds_...) (score: 344/3381)
Master Thread 1 processed query 1 (lcl|NC_034972.1_cds_...) (score: 878/8016)

=== Alignment Results ===
Query                  Header  Position   Score      Query Sequence          Best Match
--------------------------------------------------------------------------------------
    0lcl|NC_034972.1_cds_YP...         0    2368/11925  ATGGCTAACCAATAT...  ATGGCTAACCAATAT...  (19.9%)
    1lcl|NC_034972.1_cds_YP...      3726     878/8016  GAGCCCTGTAGTGAG...  ATTGTTTATACTGGT...  (11.0%)
    2lcl|NC_034972.1_cds_YP...      1068     344/3381  ATGGCTCTCATTTTT...  AATGTCTCAGCCCAG...  (10.2%)
    3lcl|NC_034972.1_cds_YP...      9629     218/645  ATGATAGGTGGTCTT...  TAAACACAAGATGGT...  (33.8%)
    4lcl|NC_034972.1_cds_YP...      5770      92/237  ATGTTACCCTCGTTT...  AACTCTCCGATTTTG...  (38.8%)
    5lcl|NC_034972.1_cds_YP...      4632     240/747  ATGGTACTCTTTTGT...  GTCGGTCATGGTGAT...  (32.1%)
    6lcl|NC_034972.1_cds_YP...      6567     185/501  ATGATTTTGGTTTTC...  GTCTTGTTGTCTAGT...  (36.9%)
    7lcl|NC_034972.1_cds_YP...      8295     333/1170  ATGAGTTCCAACGTC...  TTTGAACATGCGTCT...  (28.5%)
    8lcl|NC_034972.1_cds_YP...     11307     119/318  ATGTATTGTTTGTTG...  GATGATGTTGTTATA...  (37.4%)

Total execution time: 143 ms
Algorithm used: Simple Sliding Window
Threading model: 4 MPI processes × 2 OpenMP threads = 8 total workers
```

MPI Process Scaling Experiment

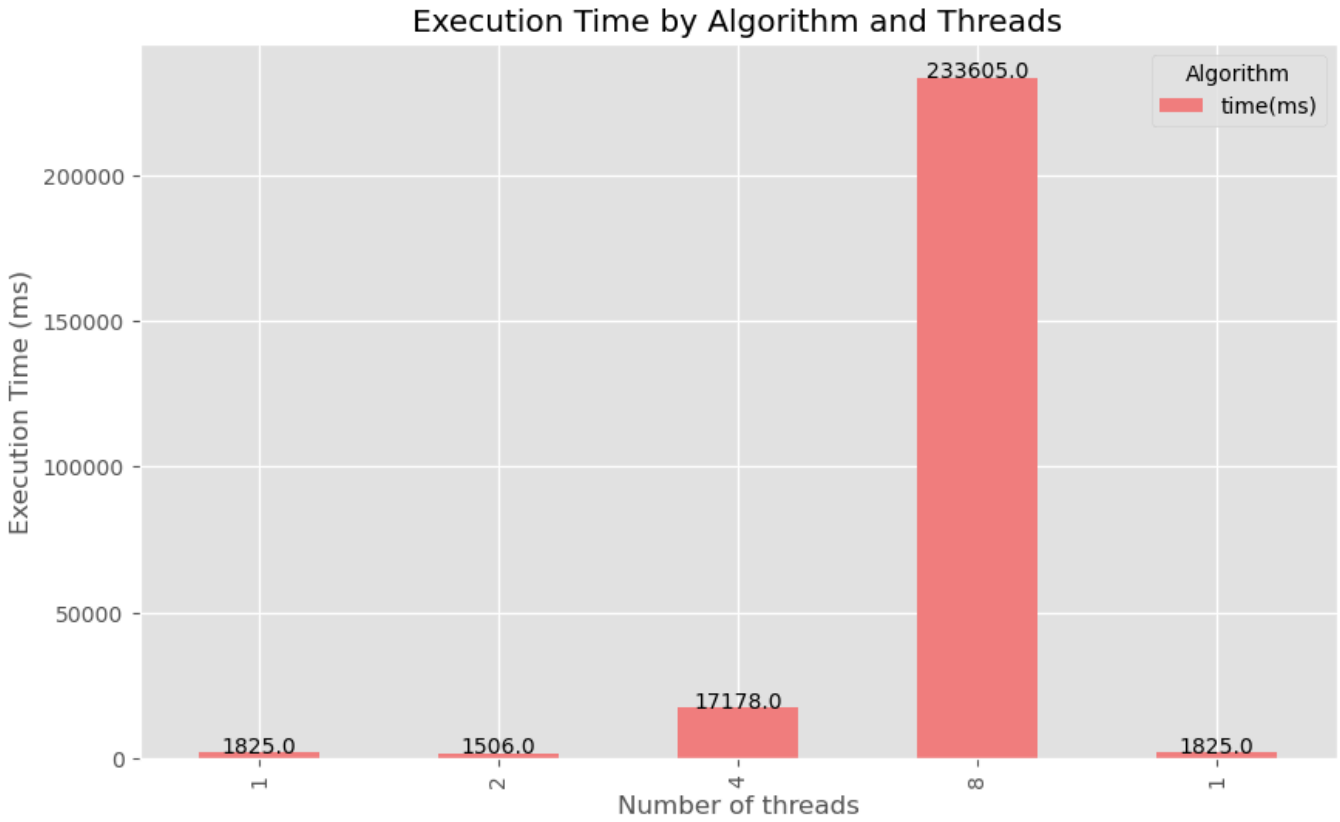
`scaling_results.txt`



- **Analysis**
  This graph show the relationship between execution time and the number of processes on two algorithms: Simple Sliding Window and Smith-Waterman Local Alignment. The result show a clear performance distinction between the two. The Simple Sliding Window algorithm execution time increase slightly when the number of processes increase to 4 and 8. This indicates that as the algorithm is simple, the processes creation overhead outweigh the computation, hence making the execution time longer. In contrast, the Smith-Waterman algorithm shows much higher execution times. At 2 processes, the time taken to finish the job is 2873ms. The execution time decrease at 4 processes and increase slightly at 8 processes. This suggests that since the algorithm is more complex, it can benefits more from the increase in processes. The overhead is less significant compared to the raw computation taken place. After 4 processes, we are starting to see diminish return, as the time taken to finish the job at 8 processes increases slightly. Overall, the graph shows that the SW algorithm is much more complicated, which result in much higher execution time. But with that, it can take better uses of the processes.

OpenMP Thread Scaling Experiment

`thread_results.txt`

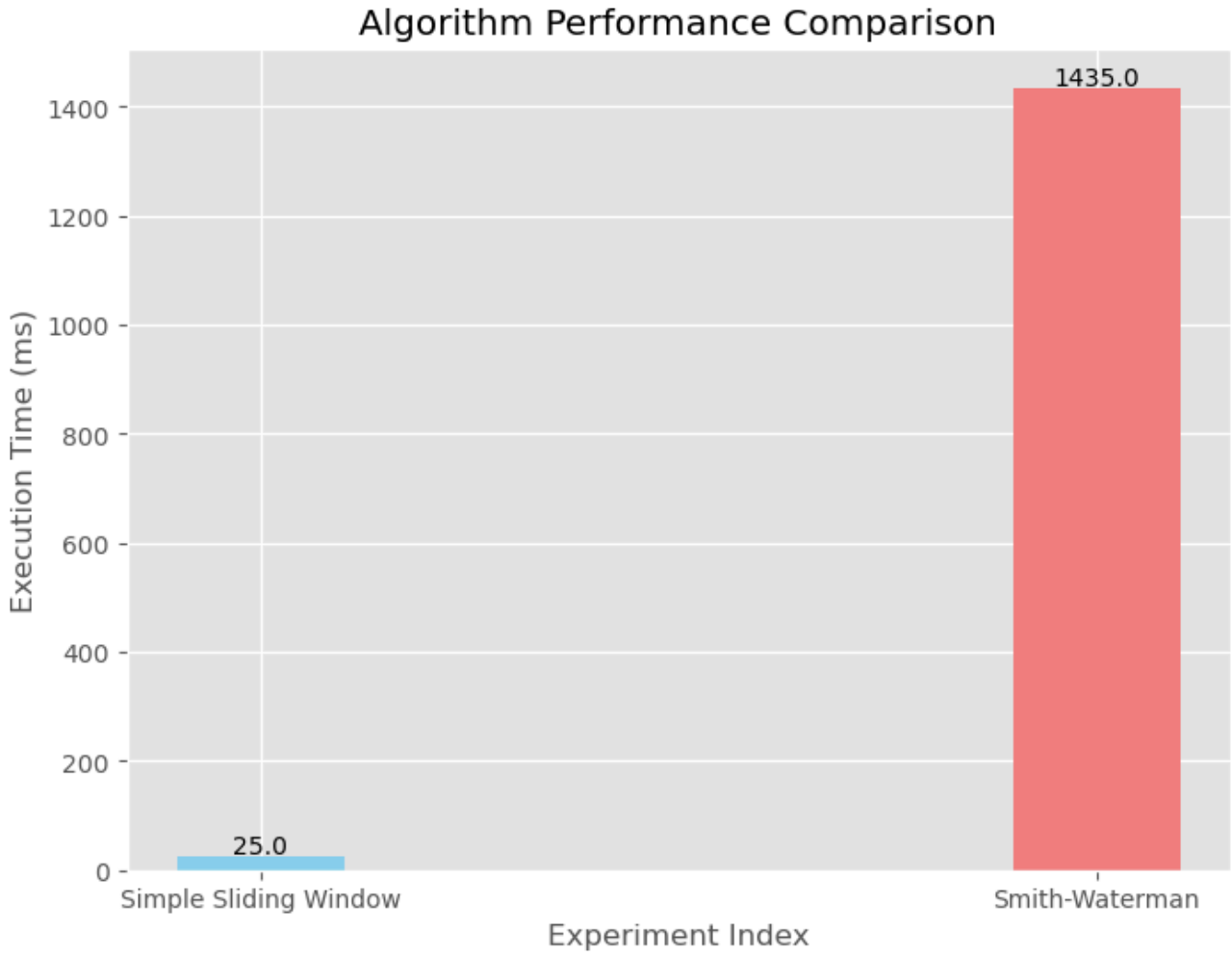


- **Analysis**
  This graph show the relationship between execution time and the number of threads used in the Simple Sliding Window algorithm. With one thread, the execution time is 1825ms. The result slightly improves to 1506ms, which indicate a modest performance gain. But with the number of threads increase, the execution time rise sharply. This suggest that the algorithm implementation has significant parallel overhead. This shows the algorithm is inefficient beyond 2 threads.

Algorithm Performance Experiment

`algorithm_results`
Both has:

- MPI processes: 4
- OpenMP threads per process: 2
  Total parallel workers: 8



- **Analysis**
  The results show the contrast in performance efficiency. The Simple Sliding Window algorithm completes execution in 25ms, while the Smith-Waterman take 1435. This significant difference highlights the difference in computation complexity of the Smith-Waterman algorithm in compare to my Simple Sliding Window algorithm.