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`

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()` functions

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

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) {
    int bestScore = -1;
    int bestPosition = -1;
```

```
    string bestMatch;

    // Slide query across reference DNA
    for (size_t i = 0; i <= REFERENCE_DNA.length() - query.length(); i++) {
        string segment = REFERENCE_DNA.substr(i, query.length());

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

        if (score > bestScore) {
            bestScore = score;
            bestPosition = i;
            bestMatch = segment;
        }
    }

    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

Extension Possibilities

The algorithm can be improved to:

- Smith-Waterman
- Needleman-Wunsch
- BLOSUM (substitution matrices)

Result

```
=== DNA Sequence Alignment with MPI ===
Reference DNA length: 547 bases
Number of query sequences: 20
Number of worker processes: 3
Distributing tasks among workers...
Assigned 7 tasks to worker 1
Assigned 7 tasks to worker 2
Assigned 6 tasks to worker 3
Worker 1 processed query 0 (score: 16/16)
Worker 1 processed query 3 (score: 10/16)
Worker 1 processed query 6 (score: 9/16)
Worker 1 processed query 9 (score: 9/16)
Worker 1 processed query 12 (score: 10/16)
Worker 1 processed query 15 (score: 9/16)
Worker 1 processed query 18 (score: 8/16)
Worker 3 processed query 2 (score: 16/16)
Worker 3 processed query 5 (score: 10/16)
Worker 3 processed query 8 (score: 11/16)
Worker 3 processed query 11 (score: 10/16)
Worker 3 processed query 14 (score: 9/16)
Worker 3 processed query 17 (score: 11/16)
Worker 2 processed query 1 (score: 16/16)
Worker 2 processed query 4 (score: 10/16)
Worker 2 processed query 7 (score: 10/16)
Worker 2 processed query 10 (score: 11/16)
Worker 2 processed query 13 (score: 9/16)
Worker 2 processed query 16 (score: 10/16)
Worker 2 processed query 19 (score: 11/16)

=== Alignment Results ===
Query  Position   Score      Query Sequence          Best Match
------------------------------------------------------------
    0         0      16/16  ATCGATCGATCGATC...  ATCGATCGATCGATC...  (100.0%)
    1        68      16/16  CACTGCAGGCCGGAG...  CACTGCAGGCCGGAG...  (100.0%)
    2       333      16/16  GGAAGGGACAGATTA...  GGAAGGGACAGATTA...  (100.0%)
    3       110      10/16  TTTCGGGCGCAAAGT...  CGTCGGCAGCAGAGC...  (62.5%)
    4       203      10/16  CTCTCGAGATCCCGA...  GTCTTGAGATTGGCA...  (62.5%)
    5       282      10/16  CTCCCCTCATGGAAT...  CTCCCCTCCCCCATG...  (62.5%)
    6       127       9/16  GGTCAAATTGCTTCC...  GCGCTGATTGGCTGC...  (56.2%)
    7       143      10/16  GTTAACGGTACTACG...  GATGACGCAACTGGG...  (62.5%)
    8       106      11/16  ACGGCATCAGAAGCA...  CCGCCGTCGGCAGCA...  (68.8%)
    9       138       9/16  AAGCTCAAGACACAG...  CTGCTGATGACGCAA...  (56.2%)
   10       259      11/16  GCAAAGTACCGCCGT...  GCGAGGTTCCGGCGG...  (68.8%)
   11       164      10/16  GGGCTGTGAACGTGG...  GCGCTGTGATAGGTG...  (62.5%)
   12       291      10/16  CTCATGACATTCAAT...  CCCATGAAAGCCAGG...  (62.5%)
   13       163       9/16  CGCGTTGGTATAACT...  TGCGCTGTGATAGGT...  (56.2%)
   14       163       9/16  GGCGTACTAATAAGG...  TGCGCTGTGATAGGT...  (56.2%)
   15       352       9/16  TCTAAAACTTAGGCC...  TCCGGAACCAATGCG...  (56.2%)
   16       101      10/16  TCTCTACTCTGTCGG...  CCTCCCCGCCGTCGG...  (62.5%)
   17       216      11/16  CTCTCGCAGGTGGCT...  CAGGGGCAGGTGGCT...  (68.8%)
   18        60       8/16  CTACGTACAACTGCA...  ATCGATCGCACTGCA...  (50.0%)
   19       116      11/16  AAGCTGATCAAGCCC...  CAGCAGAGCAAGCGC...  (68.8%)

Total execution time: 24 ms
```