Distributed Chess Engine Using MPI: Project Report

1 Project Overview

The Distributed Chess Engine is a parallelized chess-playing system built using the Message Passing Interface (MPI) and Python. It leverages distributed computing to enhance the performance of the chess engine by distributing the minimax search across multiple processes. The system includes a graphical user interface (GUI) for user interaction, a chess engine with an advanced evaluation function, and a master-worker architecture for task distribution. The project integrates the `python-chess` library for chess logic, `mpi4py` for parallelization, and `pygame` for the GUI.

1.1 Key Features

- **Distributed Minimax Search**: Utilizes iterative deepening with alpha-beta pruning, distributed across multiple worker processes.
- **GUI**: A user-friendly interface for playing against the AI, with visual highlights for moves and game states.
- **Difficulty Levels**: Configurable AI difficulty (Easy, Medium, Hard) with varying depths and time limits.
- **Evaluation Function**: Considers material, piece-square tables, mobility, pawn structure, king safety, and game phase.
- **Error Handling**: Robust handling of timeouts, invalid moves, and worker failures.
- **Move Ordering**: Optimizes search by prioritizing previous best moves.

1.2 Files

- **utils.py**: Logging and serialization utilities.
- **chess_engine.py**: Core chess logic, including minimax and evaluation functions.
- **master.py**: Manages task distribution and collects results.
- **worker.py**: Executes minimax tasks assigned by the master.
- **chess_gui.py**: Implements the GUI and integrates the engine.

2 System Architecture

The system follows a master-worker architecture, where one process (rank 0) acts as the master, handling the GUI and task coordination, while other processes (rank 1 to N) act as workers, performing minimax computations.

2.1 Architecture Components

1. **Master Process (Rank 0)**:
   - Runs the GUI (`chess_gui.py`) using `pygame`.
   - Accepts user moves and initiates AI move computation.
   - Distributes tasks to workers using `distribute_and_collect` (`master.py`).
   - Collects and aggregates results to select the best move.
2. **Worker Processes (Rank 1 to N)**:
   - Execute `worker_process` (`worker.py`).
   - Receive board positions, depths, and time limits from the master.
   - Perform minimax search with alpha-beta pruning (`chess_engine.py`).
   - Return scores and moves to the master.
3. **Chess Engine**:
   - Implements minimax with alpha-beta pruning and a sophisticated evaluation function (`chess_engine.py`).
   - Evaluates board positions based on material, mobility, pawn structure, king safety, and game phase.
4. **Utilities**:
   - Logging to track execution (`utils.py`).
   - Serialization/deserialization for MPI communication (`utils.py`).

2.2 Architecture Diagram

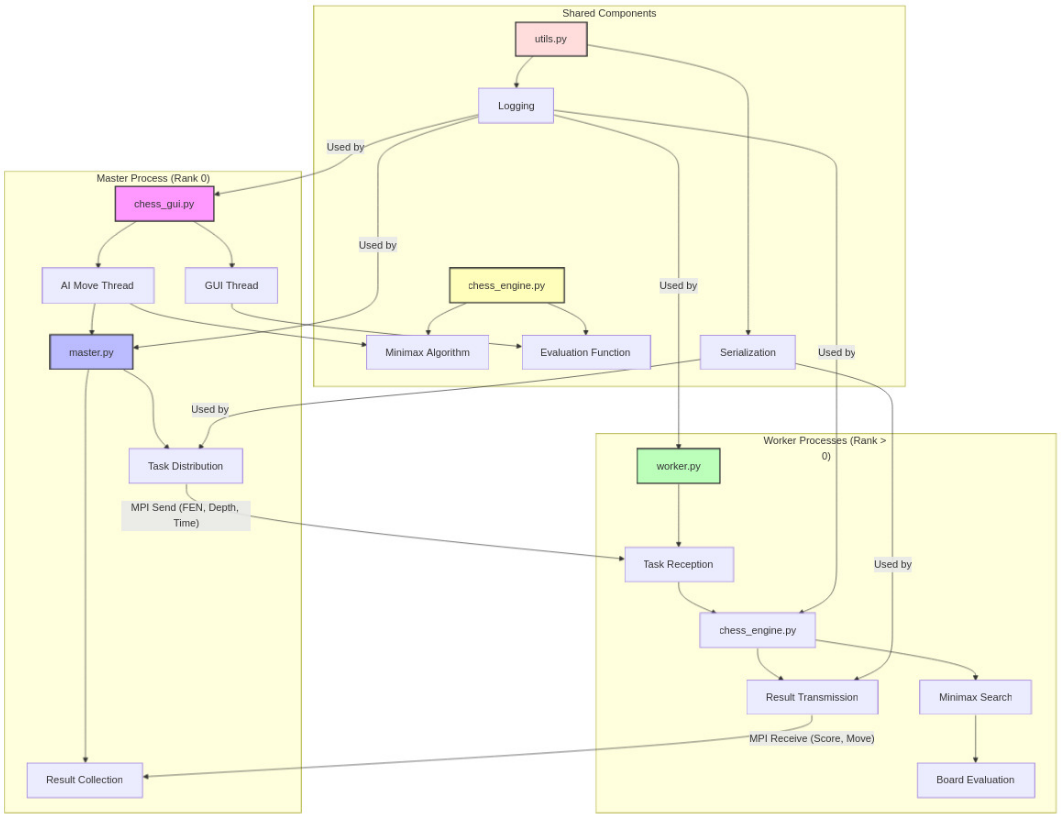

Figure 1: System Architecture Diagram

2.3 Explanation of the Diagram

- **Master Process (Rank 0)**:
  - `chess_gui.py`: The entry point for the master process, handling the GUI and initiating AI moves. It spawns two threads:
  - **GUI Thread**: Manages user interaction and board rendering.
  - **AI Move Thread**: Triggers the AI computation by invoking `master.py`.
  - `master.py`: Handles task distribution and result collection:
  - **Task Distribution**: Serializes and sends tasks (board FEN, depth, time limit) to workers.
  - **Result Collection**: Receives and aggregates scores and moves from workers to select the best move.
- **Worker Processes (Rank ¿ 0)**:
  - `worker.py`: Runs on each worker process, responsible for:
  - **Task Reception**: Receives tasks via MPI.
  - **Result Transmission**: Sends computed scores and moves back to the master.
  - `chess_engine.py`: Invoked by workers to perform:
  - **Minimax Search**: Executes the minimax algorithm with alpha-beta pruning.
  - **Board Evaluation**: Computes board scores based on material, mobility, etc.
- **Shared Components**:
  - `chess_engine.py`: Shared logic for minimax and evaluation, used by both master (for local search) and workers.
  - **Minimax Algorithm**: Core search logic.
  - **Evaluation Function**: Computes board scores, used by both master and workers.
  - `utils.py`: Provides utility functions:
  - **Serialization**: Used for MPI communication (tasks and results).
  - **Logging**: Used across all components for debugging and tracking.
- **Interactions**:
  - **MPI Send**: The master sends tasks (FEN, depth, time limit) to workers.
  - **MPI Receive**: Workers send results (score, move) back to the master.
  - **Serialization**: Used in task distribution and result transmission.
  - **Logging**: Applied universally for debugging.
  - **Evaluation Function**: Called by the GUI thread (for immediate feedback) and minimax search (for AI moves).

2.4 Notes

- **Shared chess_engine.py**: The diagram places `chess_engine.py` in a shared components section to clarify its use by both master and workers, avoiding duplication in the subgraphs.
- **Styling**: Colors differentiate components (e.g., pink for GUI, blue for master logic, green for workers, yellow for engine, red for utils) for clarity.
- **Connections**: Arrows show data flow, with labels for MPI communications to specify payload (e.g., ”FEN, Depth, Time”).

3 Complexity Analysis

3.1 Time Complexity

The time complexity is driven by the minimax algorithm with alpha-beta pruning, distributed across W workers (where $W = \text{size} - 1$).

- **Single-Threaded Minimax**:
  - Without alpha-beta pruning, the time complexity is $O(b^d)$, where:
  - b: Branching factor (average 35 for chess).
  - d: Search depth.
  - With alpha-beta pruning, the best-case complexity is $O(b^{d/2})$, but typically closer to $O(b^{d/2} \cdot \sqrt{b})$ due to imperfect move ordering.
- **Distributed Minimax**:
  - The master distributes legal moves (up to M, typically 35 at the root) across W workers.
  - Each worker evaluates a subtree of depth $d - 1$.
  - Ideal case (perfect load balancing):
  - Time complexity per depth: $O(b^{d-1}/W)$.
  - Iterative deepening runs from depth 1 to $d_{\max}$, so total time is approximately:

$$O\left(\sum_{k=1}^{d_{\max}} \frac{b^{k-1}}{W}\right) \approx O\left(\frac{b^{d_{\max}}}{W \cdot (b-1)}\right)$$

  - In practice, load imbalance and communication overhead increase this slightly.
  - Time limit (T_{limit}) interrupts deeper searches, capping runtime per move at $O(T_{\text{limit}})$.
- **GUI Overhead**:
  - Rendering and event handling are $O(1)$ per frame, negligible compared to search.

Example:

- For $b = 35$, $d_{\max} = 4$, $W = 7$:
- Sequential: $O(35^4) \approx 1.5 \times 10^6$.
- Distributed: $O(35^4/7) \approx 2.1 \times 10^5$.
- Speedup: 7x (ideal), reduced by communication and imbalance.

3.2 Space Complexity

- **Per Process**:
  - **Board Representation**: $O(1)$ (fixed-size chess board, 64 bytes plus metadata).
  - **Minimax Stack**: $O(d)$ for recursion (storing board states up to depth d).
  - **Move Lists**: $O(b)$ for legal moves at each node (35 moves).
  - **Piece Images (Master Only)**: $O(1)$ (12 images, 1 MB total).
  - **Log Files**: $O(L)$, where L is the number of log entries (grows with runtime).
  - Total per process: $O(d + b + L) \approx O(d)$, as L is amortized.
- **Total Across Processes**:

  - $W + 1$ processes (1 master + W workers).
  - Total: $O(W \cdot d)$, assuming logs are bounded.
  - GUI memory (master only): $O(1)$ additional for `pygame` surfaces.

Example:

- For $d = 4$, $W = 7$:
- Per process: $O(4 + 35) \approx 40$ units (plus logs).
- Total: $O(8 \cdot 40) \approx 320$ units, excluding images (1 MB).

3.3 Communication Complexity

MPI communication occurs between the master and workers for task distribution and result collection.

- **Messages**:
  - **Task Distribution**:
  - For each depth k, the master sends up to M tasks (legal moves).
  - Each task is a serialized tuple (FEN string, depth, time limit).
  - FEN string: 100 bytes.
  - Total per depth: $O(M \cdot S_{\text{task}})$, where $S_{\text{task}} \approx 128$ bytes.
  - Across $d_{\max}$: $O(d_{\max} \cdot M \cdot S_{\text{task}})$.
  - **Result Collection**:
  - Each worker returns a serialized (score, move) pair.
  - Size: 32 bytes (float + UCI string).
  - Total per depth: $O(M \cdot S_{\text{result}})$.
  - Across $d_{\max}$: $O(d_{\max} \cdot M \cdot S_{\text{result}})$.
  - **Control Messages**:
  - "NO_MORE_TASKS" and "STOP": $O(W \cdot d_{\max})$ messages, 16 bytes each.
- **Total Communication**:
  - Per depth: $O(M \cdot (S_{\text{task}} + S_{\text{result}}) + W)$.
  - Total: $O(d_{\max} \cdot (M \cdot (S_{\text{task}} + S_{\text{result}}) + W))$.
  - Assuming $M \approx 35$, $S_{\text{task}} \approx 128$, $S_{\text{result}} \approx 32$, $W \approx 7$, $d_{\max} = 4$:
  - Per depth: $35 \cdot (128 + 32) + 7 \approx 5607$ bytes.
  - Total: $4 \cdot 5607 \approx 22.4$ KB.
- **Latency**:
  - Each send/receive has latency L_{MPI} (typically microseconds).
  - Total latency per depth: $O(M + W)$.
  - Total: $O(d_{\max} \cdot (M + W))$.

Example:

- For $M = 35$, $W = 7$, $d_{\max} = 4$:
- Messages: $4 \cdot (35 + 7) = 168$.
- Data: 22.4 KB.
- Latency: Proportional to 168 MPI calls.

4 Implementation Details

4.1 Minimax Algorithm

- **Iterative Deepening**: Searches from depth 1 to d_{max}, stopping if time exceeds T_{limit}.
- **Alpha-Beta Pruning**: Reduces the search tree by pruning branches that cannot affect the outcome.
- **Move Ordering**: Prioritizes previous best moves to maximize pruning.
- **Timeout Handling**: Raises `TimeoutError` if time limit is exceeded.

4.2 Evaluation Function

- **Features**:
  - Material: Standard piece values (e.g., pawn = 100, queen = 900).
  - Piece-Square Tables: Position-specific bonuses (e.g., central pawns score higher).
  - Mobility: Counts legal moves, scaled by 8 points.
  - Pawn Structure: Penalizes doubled pawns (15 points each).
  - King Safety: Bonuses for pawns shielding the king (25 points per pawn).
  - Game Phase: Interpolates between middle and endgame king tables.
- **Random Factor**: Adds ± 2 points to avoid repetition.

4.3 MPI Integration

- **Master**:
  - Distributes moves dynamically using a queue (`deque`).
  - Sends tasks (FEN, depth, time limit) with tag 1.
  - Receives results (score, move) with tag 2.
  - Sends "NO_MORE_TASKS" between depths and "STOP" to terminate.
- **Worker**:
  - Listens for tasks, executes minimax, and returns results.
  - Handles errors (timeouts, exceptions) by sending error messages.

4.4 GUI

- Displays the chessboard with highlights for selected squares, legal moves, and last move.
- Supports pawn promotion via a dialog.
- Allows new games (N key) and undoing moves (U key).
- Shows game state (check, checkmate, stalemate).

5 Performance Considerations

- **Scalability**:
  - Scales well up to $W \approx M$, as each worker handles one move.
  - Beyond M, workers may idle, reducing efficiency.
- **Load Balancing**:
  - Dynamic task assignment mitigates imbalance but depends on move complexity.
- **Fault Tolerance**:
  - Robust handling of timeouts, worker failures, and invalid results to ensure system reliability.
- **Communication Overhead**:
  - Small message sizes (128 bytes/task) minimize bandwidth usage.
  - Latency may impact performance on high-latency networks.
- **Time Limits**:
  - Ensures responsiveness (e.g., 5s for Medium difficulty).
  - May truncate deep searches, affecting move quality.

6 Limitations

- **Move Quality**: Limited by depth (e.g., max 6 for Hard), weaker than professional engines like Stockfish.
- **Communication Bottleneck**: Master must serialize/deserialize all tasks, potentially slowing down with many workers.
- **Error Recovery**: Worker failures are logged but may lead to fallback moves (e.g., random move).
- **GUI Simplicity**: Lacks advanced features like move history or analysis.

7 Conclusion

The Distributed Chess Engine demonstrates effective use of MPI for parallelizing a chess minimax search. It achieves significant speedup over a sequential implementation, with robust error handling and a user-friendly GUI. The complexity analysis highlights efficient resource usage, though scalability is limited by the number of legal moves and communication overhead. Future work could enhance move quality and scalability, making it a competitive educational tool.

Repository For future updates and changes, visit the GitHub Repository