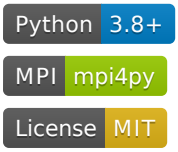
Distributed Chess Engine Using MPI



A distributed chess engine built with Python, leveraging the Message Passing Interface (MPI) for parallel computation of chess moves. The engine uses a minimax algorithm with alpha-beta pruning, distributed across multiple processes, and features a graphical user interface (GUI) powered by Pygame. Play chess against an AI with configurable difficulty levels, visualize moves, and explore parallel computing in action.

Table of Contents

Features

- **Distributed Minimax Search**: Parallelizes chess move computation using MPI, speeding up the minimax algorithm with alpha-beta pruning.
- **Interactive GUI**: Built with Pygame, offering a user-friendly chessboard with move highlights, promotion dialogs, and game status updates.
- **Configurable Difficulty**: Three levels (Easy, Medium, Hard) with adjustable search depth and time limits.
- **Advanced Evaluation**: Considers material, piece-square tables, mobility, pawn structure, king safety, and game phase.
- **Robust Error Handling**: Manages timeouts, worker failures, and invalid moves gracefully.
- **Logging**: Detailed logs for debugging and performance analysis.

PROF

Architecture

The system follows a master-worker architecture using MPI:

- **Master Process (Rank 0)**: Runs the GUI (`chess_gui.py`) and coordinates task distribution (`master.py`).
- **Worker Processes (Rank > 0)**: Compute minimax searches (`worker.py`) using the chess engine (`chess_engine.py`).
- **Shared Utilities**: Serialization and logging (`utils.py`) support communication and debugging.

Below is the architecture diagram (Mermaid syntax):

```
graph TD
    subgraph "Master Process (Rank 0)"
        A[chess_gui.py] --> B[GUI Thread]
        A --> C[AI Move Thread]
        C --> D[master.py]
        D --> E[Task Distribution]
        D --> F[Result Collection]
    end

    subgraph "Worker Processes (Rank > 0)"
        G[worker.py] --> H[Task Reception]
        H --> I[chess_engine.py]
        I --> J[Minimax Search]
        J --> K[Board Evaluation]
        I --> L[Result Transmission]
    end

    subgraph "Shared Components"
        M[chess_engine.py] --> N[Minimax Algorithm]
        M --> O[Evaluation Function]
        P[utils.py] --> Q[Serialization]
        P --> R[Logging]
    end

    E -- "MPI Send (FEN, Depth, Time)" --> H
    L -- "MPI Receive (Score, Move)" --> F
    Q -- "Used by" --> E
    Q -- "Used by" --> L
    R -- "Used by" --> A
    R -- "Used by" --> D
    R -- "Used by" --> G
    R -- "Used by" --> I
    B --> O
    C --> N

    style A fill:#f9f,stroke:#333,stroke-width:2px
    style D fill:#bbf,stroke:#333,stroke-width:2px
    style G fill:#bfb,stroke:#333,stroke-width:2px
    style M fill:#ffb,stroke:#333,stroke-width:2px
    style P fill:#fdd,stroke:#333,stroke-width:2px
```

PROF

Prerequisites

- **Python**: 3.8 or higher
- **MPI Implementation**: OpenMPI or MPICH
- **Python Libraries**:
    - `mpi4py`: For MPI communication

    - `python-chess`: For chess logic
    - `pygame`: For the GUI
- **Operating System**: Linux, macOS, or Windows (with MPI support)

Installation

1. **Clone the Repository**:

```
git clone https://github.com/your-username/distributed-chess-engine.git
cd distributed-chess-engine
```

2. **Install MPI**:

    - **Ubuntu/Debian**:

```
sudo apt-get install openmpi-bin openmpi-common libopenmpi-dev
```

    - **macOS**:

```
brew install openmpi
```

    - **Windows**: Install WSL2 and follow Ubuntu instructions, or use MPICH.

3. **Set Up a Virtual Environment** (recommended):

```
python -m venv myenv
source myenv/bin/activate  # On Windows: venv\Scripts\activate
```

4. **Install Python Dependencies**:

```
pip install mpi4py python-chess pygame
```

5. **Download Chess Piece Images** (optional):

    - Place chess piece images (e.g., `wp.png`, `bp.png`, etc.) in a `chess_pieces` folder.
    - Without images, the GUI falls back to text rendering.
    - Example source: Chess Piece Sprites.

Usage

1. **Run the Game**:
    - Launch the master and worker processes using `mpirun`:

      ```
      mpirun -n 8 --oversubscribe python chess_gui.py
      ```

        - `-n 8`: Runs 1 master + 7 workers.
        - `--oversubscribe`: Allows more processes than CPU cores (optional if cores suffice).
        - **Increase Nodes**: To use more workers, increase `-n`. For example:

          ```
          mpirun -n 16 --oversubscribe python chess_gui.py
          ```

            - This runs 1 master + 15 workers. Optimal performance is typically with 10–35 workers, matching the number of legal moves (~35).
2. **Gameplay**:
    - **Select Difficulty**: Choose Easy, Medium, or Hard at startup.
    - **Make Moves**: Click a piece to select, then click a highlighted square to move.
    - **Pawn Promotion**: Select a piece (Queen, Rook, Bishop, Knight) in the promotion dialog.
    - **New Game**: Press `N` to start a new game.
    - **Undo Move**: Press `U` to undo the last human and AI moves.
    - **Quit**: Close the window or press `Ctrl+C` (sends STOP signal to workers).
3. **Logs**:
    - Check `rank_X.txt` files for logs from each process (master: `rank_0.txt`, workers: `rank_1.txt`, etc.).

Project Structure

PROF

```
distributed-chess-engine/
├── chess_gui.py      # GUI and master process entry point
├── master.py         # Task distribution and result collection
├── worker.py         # Worker process logic
├── chess_engine.py   # Minimax algorithm and board evaluation
├── utils.py          # Serialization and logging utilities
├── chess_pieces/     # (Optional) Chess piece images
├── rank_X.txt        # Log files per process
├── README.md         # This file
└── report.md         # Detailed project report
```

Configuration

Adjust AI difficulty in `chess_gui.py` under `DIFFICULTY_PROFILES`:

```
DIFFICULTY_PROFILES = {
    "Easy": {'max_depth': 3, 'time_limit': 3, 'label': 'Easy',
'use_distributed': True},
    "Medium": {'max_depth': 4, 'time_limit': 5, 'label': 'Medium',
'use_distributed': True},
    "Hard": {'max_depth': 6, 'time_limit': 10, 'label': 'Hard',
'use_distributed': True}
}
```

- `max_depth`: Maximum search depth.
- `time_limit`: Time (seconds) per move.
- `use_distributed`: Enable/disable MPI (set to `False` for local search).

Contributing

Contributions are welcome! To contribute:

1. Fork the repository.
2. Create a feature branch (`git checkout -b feature/your-feature`).
3. Commit changes (`git commit -m "Add your feature"`).
4. Push to the branch (`git push origin feature/your-feature`).
5. Open a pull request.

Please include tests and update documentation as needed.

Ideas for Improvement

- Add transposition tables for faster searches.
- Implement an opening book or neural network evaluation.
- Enhance the GUI with move history or analysis.
- Optimize load balancing for uneven subtrees.

PROF

License

This project is licensed under the MIT License.
Developed as part of an academic project at **IIIT Hyderabad**, for educational and research purposes.

Repository

For future updates and changes, visit the GitHub Repository.