
Akdeniz University

# CSE 440 Parallel Computing

## Final Project Assignment

**Parallel Exploration of IQ Fit Puzzle Solutions Using MSMPI**

*Buse Çoban*
*Student ID: 20200808070*

# Contents

# 1 Introduction

The IQ Fit puzzle is a challenging combinatorial problem that requires placing 12 uniquely shaped pieces onto an 11×5 board without overlap and with full coverage. Each piece can be rotated and flipped, resulting in a vast number of possible configurations. The exponential nature of this search space makes it impractical to solve using a naive sequential algorithm within a reasonable time frame.

This project aims to design and implement an efficient parallel solution to exhaustively explore all valid configurations of the IQ Fit puzzle. The approach is based on recursive backtracking, optimized with bitmask representations and first-empty-cell heuristics. To leverage multi-core processing power, the solution was parallelized using the Message Passing Interface (MPI), with each MPI process independently exploring a disjoint subset of the configuration tree.

While the project specification originally recommended Microsoft MPI (MSMPI), it was not feasible to use MSMPI on the development platform—a MacBook Pro with an Apple M3 processor and 12 CPU cores—due to its limited support for Windows environments only. Instead, OpenMPI was used, as it is a cross-platform and standards-compliant implementation of the MPI specification, fully compatible with the required API.

The final implementation utilizes OpenMPI to distribute initial placements of the first puzzle piece among all available MPI ranks. Each process independently constructs solutions based on its assigned starting configurations and reports results back to the root process. This architecture enabled efficient parallel exploration, minimized redundant computations, and significantly reduced total execution time.

This report presents the algorithmic design, implementation details, and performance analysis of the developed solver. It also includes a discussion of design decisions, challenges encountered, and conclusion.

# 2 Puzzle Modeling

The IQ Fit puzzle consists of a rectangular board with 55 cells (11 columns × 5 rows) and 12 unique pieces, each formed by five connected unit squares. The goal is to fill the entire board with all 12 pieces, without overlap, and using each piece exactly once.

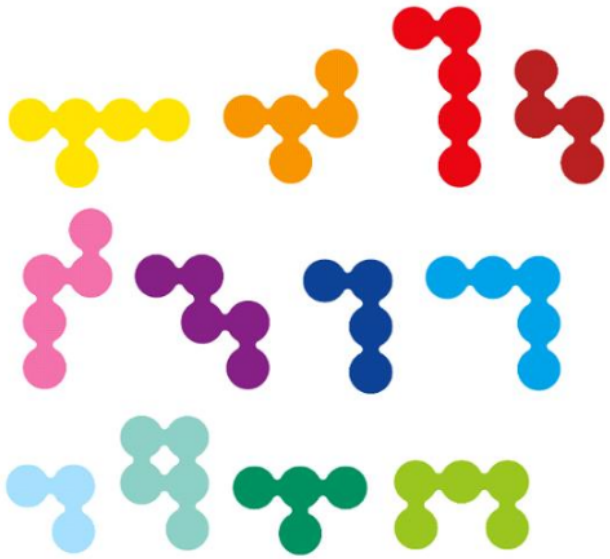

Figure 1: The 12 pieces of the IQ Fit puzzle[1]

Each piece is initially described using a compact string format (e.g., `"01 10 11 21 31"`), where each pair of digits represents an $(x, y)$ offset from the origin block of the piece. These definitions

correspond directly to the official puzzle piece shapes, as shown in Figure 1. The full list of base shapes is hardcoded in the program and includes all 12 distinct pieces labeled A through M.

To handle rotational and reflectional symmetries, the program automatically generates all unique transformations for each piece. Specifically:

- All 4 possible rotations (0°, 90°, 180°, 270°) are considered.
- Horizontal reflection is applied to each rotation.
- After transformation, each shape is normalized to top-left origin.
- Duplicate shapes are filtered using a sorted coordinate-based hash.

Each valid transformation is then virtually placed on the board in all legal positions that do not exceed the board boundaries. For every such placement, the following data is precomputed and stored:

- **Placement Mask:** A 64-bit integer representing the occupied board cells via bitmasking.
- **Cell Indices:** The list of actual board indices that the piece occupies.
- **Cell-to-Placement Map:** A reverse map storing which placements affect each cell.

This precomputation significantly reduces the search space during recursive solving by enabling fast overlap checks and first-empty-cell heuristics. The board is represented both as a 1D character array (for solution printing) and a 64-bit integer (for efficient masking).

# 3 Sequential Baseline Algorithm

Before implementing the parallel version, a correct and efficient sequential solution was developed using recursive backtracking. The baseline algorithm attempts to place all 12 puzzle pieces onto the board such that no overlaps occur and the entire board is covered.

The core idea is to represent the board using a 64-bit integer bitmask. Each bit corresponds to a cell on the 11×5 board, and a set bit indicates that the cell is already occupied. This allows extremely fast overlap checks using bitwise AND operations.

The algorithm proceeds as follows:

1. **First Free Cell Heuristic:** At each recursive level, the algorithm identifies the first empty cell on the board and attempts to cover it with one of the remaining unused pieces.
2. **Placement Filtering:** For each candidate piece, only those precomputed placements that cover the target cell and do not overlap with already placed pieces are considered.
3. **Recursive Exploration:** The algorithm places a valid piece, marks it as used, updates the board mask, and recursively proceeds to the next piece.
4. **Backtracking:** If no valid placement is found for a given configuration, the algorithm backtracks by removing the last placed piece and trying the next possibility.
5. **Solution Recording:** Once all 12 pieces are successfully placed, the current board state is stored as a valid solution.

The algorithm is implemented in the `recursiveSolver()` function, which takes the current board mask, a usage array for the pieces, the board state, and a vector of solutions as input. This recursive structure provides a clear and efficient framework for full solution enumeration.

Although this sequential version is functional and correct, it is computationally expensive due to the vast size of the solution space. This motivates the use of parallel computing to significantly reduce total runtime.

## Sequential Execution Result

To evaluate the performance of the sequential baseline implementation, the solver was executed using a single process (no MPI parallelism). The run produced the full set of **4,331,140** valid solutions in:

- **Elapsed Time:** 1733.42 seconds
- **Platform:** Apple M3 (12-core MacBook Pro)
- **Command Used:** `mpirun -np 1 ./iqfit_mpi`

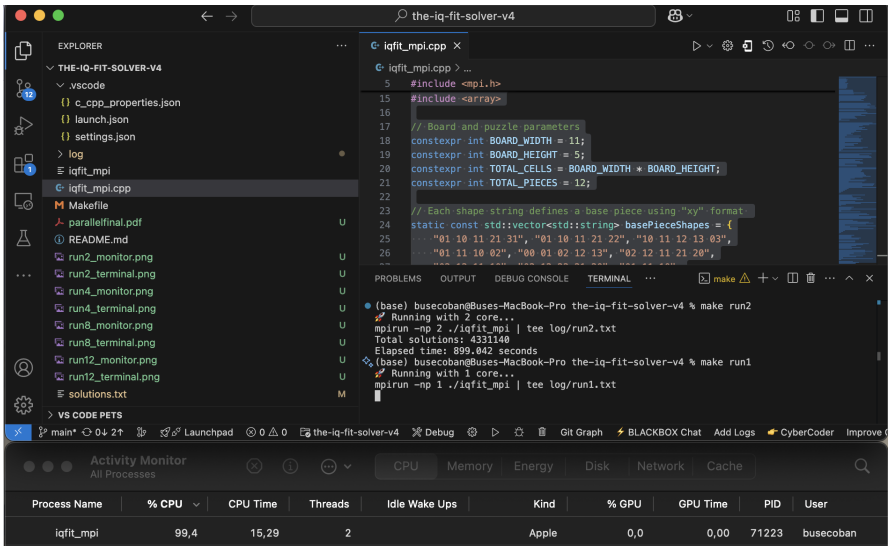


Figure 2: CPU usage during sequential execution (1 MPI process)

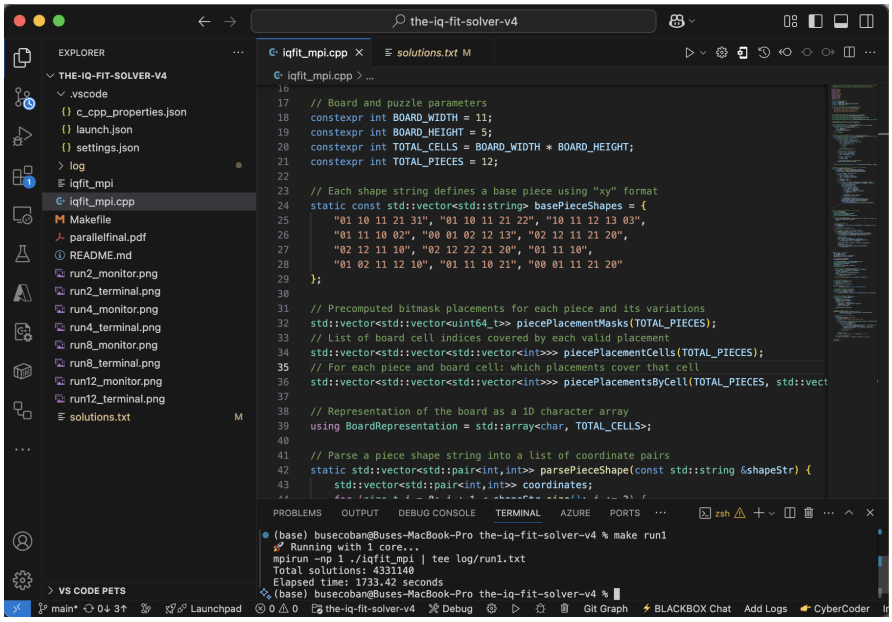


Figure 3: Terminal output for sequential execution (run1)

This baseline execution serves as the reference point for speedup and efficiency calculations presented in the next section.

# 4 Parallelization Strategy

Given the combinatorial explosion in the number of possible configurations for the IQ Fit puzzle, a parallel strategy was adopted to distribute the computational load efficiently. The solution leverages **OpenMPI** to perform full-space parallel exploration, with each MPI rank independently responsible for exploring a subset of the solution space.

## Platform Constraints

The project specification initially suggested the use of Microsoft MPI (MSMPI). However, MSMPI is supported only on Windows systems and lacks native compatibility with macOS, the development environment used for this project (MacBook Pro with Apple M3 and 12 logical CPU cores). As a result, **OpenMPI** was selected due to its cross-platform capabilities and compliance with the standard MPI API.

## Work Distribution Strategy

The parallelization strategy revolves around the idea of **disjoint exploration based on fixed first-piece placements**. The steps are:

1. All legal placements for the first piece (e.g., piece A) are precomputed.
2. These placements are evenly distributed among the available MPI ranks.
3. Each rank initializes the board with one of its assigned first-piece placements.
4. Then, the rank performs recursive backtracking to complete the rest of the board using the remaining 11 pieces.
5. Valid solutions are locally collected and later gathered on rank 0 for final reporting.

This design guarantees that no duplicate solutions are generated and that each rank works independently, minimizing inter-process communication overhead.

## Parallel Execution Model

Below is a schematic illustration of how the initial placements are distributed among MPI ranks:

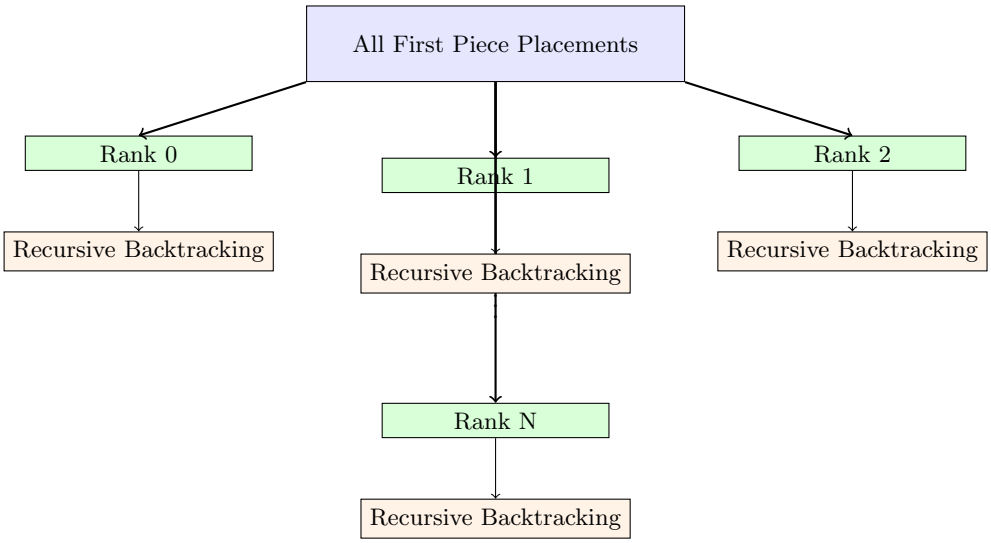


Figure 4: Parallel decomposition of the solution space using fixed first-piece placements. Each rank independently solves a disjoint subproblem.

# 5 Technical Implementation Details

## 5.1 Backtracking Search Strategy

The core puzzle-solving logic is implemented using recursive backtracking. The function `recursiveSolver` explores all legal piece placements on the board:

```
static void recursiveSolver(
    uint64_t currentBoardMask,
    std::array<bool, TOTAL_PIECES> &usedPieces,
    BoardRepresentation &currentBoard,
    std::vector<BoardRepresentation> &foundSolutions
) {
    if (std::all_of(usedPieces.begin(), usedPieces.end(), [](bool used) {
        return used; })) {
        foundSolutions.push_back(currentBoard);
        return;
    }
    int firstEmptyCell = 0;
    while (firstEmptyCell < TOTAL_CELLS && ((currentBoardMask >>
        firstEmptyCell) & 1ULL)) {
        ++firstEmptyCell;
    }
    for (int pieceIdx = 0; pieceIdx < TOTAL_PIECES; ++pieceIdx) {
        if (usedPieces[pieceIdx]) continue;
        for (int placementIdx : piecePlacementsByCell[pieceIdx][firstEmptyCell
            ]) {
            uint64_t placementMask = piecePlacementMasks[pieceIdx][
                placementIdx];
            if ((placementMask & currentBoardMask) != 0ULL) continue;
            usedPieces[pieceIdx] = true;
            uint64_t newMask = currentBoardMask | placementMask;
            for (int cell : piecePlacementCells[pieceIdx][placementIdx]) {
                currentBoard[cell] = char('A' + pieceIdx);
            }
            recursiveSolver(newMask, usedPieces, currentBoard, foundSolutions)
                ;
            usedPieces[pieceIdx] = false;
            for (int cell : piecePlacementCells[pieceIdx][placementIdx]) {
                currentBoard[cell] = '.';
            }
        }
    }
}
```

Listing 1: Recursive Backtracking Solver

## 5.2 First-Free-Cell Heuristic

To accelerate the search, the solver prioritizes the lowest-index empty cell, reducing unnecessary branching early in the search tree. This improves performance significantly by guiding the recursion through more constrained parts of the board first.

## 5.3 Bitmask-Based Board Representation

The board is represented as a 64-bit integer (`uint64_t`) called `currentBoardMask`. Each bit corresponds to a cell on the 11x5 board (total 55 cells). This allows fast overlap checks using bitwise AND operations:

```
if ((placementMask & currentBoardMask) != 0ULL) continue;
```

Listing 2: Bitmask Overlap Check

## 5.4 Precomputation of Legal Placements

All valid placements for each piece (considering all orientations) are computed once before solving begins. These are stored in:

- `piecePlacementMasks`: Bitmasks of all legal placements
- `piecePlacementCells`: Cell indices covered by each placement
- `piecePlacementsByCell`: Mappings of cell → placement indices for fast access

## 5.5 Rotation and Reflection Handling

Each piece shape is converted into all unique orientations using both 90-degree rotations and horizontal reflections. The resulting shapes are normalized to a top-left origin and deduplicated using a `std::set`:

```
for (int reflect = 0; reflect < 2; ++reflect) {
    for (int rot = 0; rot < 4; ++rot) {
        ...
        // Normalize shape to top-left origin
        for (auto &p : transformed) {
            p.first -= minX;
            p.second -= minY;
        }
        std::sort(transformed.begin(), transformed.end());
        uniqueOrientations.insert(transformed);
    }
}
```

Listing 3: Unique Orientations with Normalization

## 5.6 Parallelization with MPI

Each MPI rank is assigned a distinct subset of first-piece placements. This ensures no duplicate solutions and maximizes workload distribution:

```
for (int i = rankId; i < totalStartingPlacements; i += totalRanks) {
    ...
    recursiveSolver(currentMask, used, currentBoard, localSolutions);
}
```

Listing 4: MPI Distribution of First Piece Placements

## 5.7 MPI Gather and Output

After computation, each process sends its results to rank 0 via `MPI_Gather` and `MPI_Gatherv`. The master process writes all valid solutions to `solutions.txt`:

```
MPI_Gather(&localCount, 1, MPI_INT,
           solutionCounts.data(), 1, MPI_INT, 0, MPI_COMM_WORLD);

MPI_Gatherv(localBuffer.data(), localChars, MPI_CHAR,
            allSolutionsBuffer.data(), recvCounts.data(), displacements.data()
                ,
            MPI_CHAR, 0, MPI_COMM_WORLD);
```

Listing 5: MPI Result Collection and Output

### 5.8 Solution Representation

Each solution is stored as a 1D character array of 55 characters. Letters 'A' to 'L' represent pieces. Rank 0 writes each solution in row-major format with line breaks:

```
for (int row = 0; row < BOARD_HEIGHT; ++row) {
    outputFile.write(boardData + row * BOARD_WIDTH, BOARD_WIDTH);
    outputFile.put('\n');
}
outputFile.put('\n');
```

Listing 6: Board Output to File

## 6 Performance Evaluation

To evaluate the effectiveness of the parallel implementation, we measured total runtime across multiple process counts on the same hardware configuration (Apple M3, 12-core MacBook Pro). The number of valid solutions remained constant across all runs: **4,331,140**. However, the execution time decreased as more MPI processes were used.

### Execution Time vs. Number of Processes

| Process Count | Elapsed Time (s) |
|---|---|
| 1 | 1733.42 |
| 2 | 899.04 |
| 4 | 513.27 |
| 8 | 300.53 |
| 12 | 286.38 |

Table 1: Execution time for different MPI process counts, including sequential baseline

### Speedup and Efficiency

**Speedup** is calculated as:

$$\text{Speedup}(p) = \frac{T_1}{T_p} \quad \text{and} \quad \text{Efficiency}(p) = \frac{\text{Speedup}(p)}{p} \times 100$$

Where $T_1 = 1733.42$ seconds is the sequential baseline.

| Process Count | Speedup | Efficiency (%) |
|---|---|---|
| 1 | 1.00 | 100.0 |
| 2 | 1.93 | 96.7 |
| 4 | 3.38 | 84.4 |
| 8 | 5.77 | 72.1 |
| 12 | 6.05 | 50.4 |

Table 2: Calculated speedup and efficiency normalized to sequential run (1 process)

### Scalability Analysis

- **Linear region:** From 2 to 8 processes, speedup increases significantly.
- **Diminishing returns:** From 8 to 12, performance gain is minimal.
- **Bottleneck causes:** Task imbalance, memory access limits, and MPI overhead.

## Visual Results

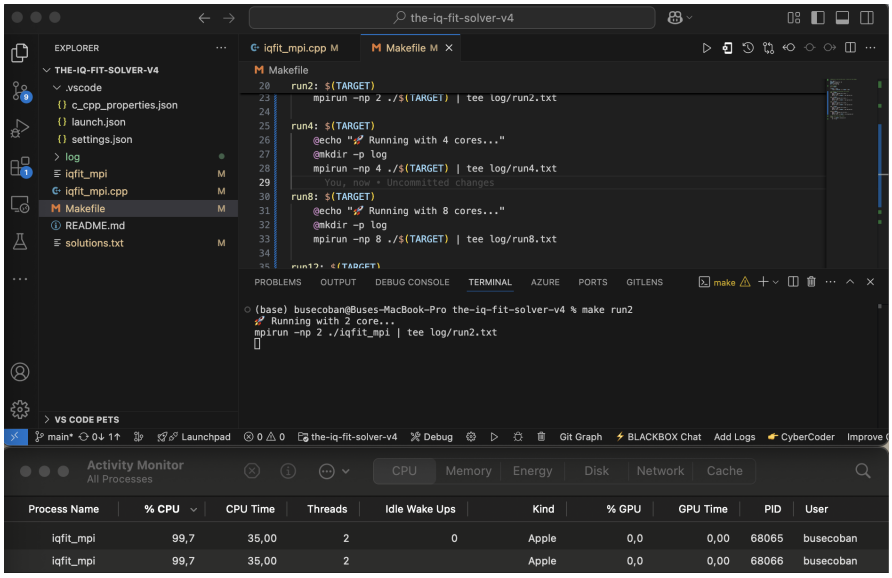


Figure 5: CPU usage during run with 2 MPI processes

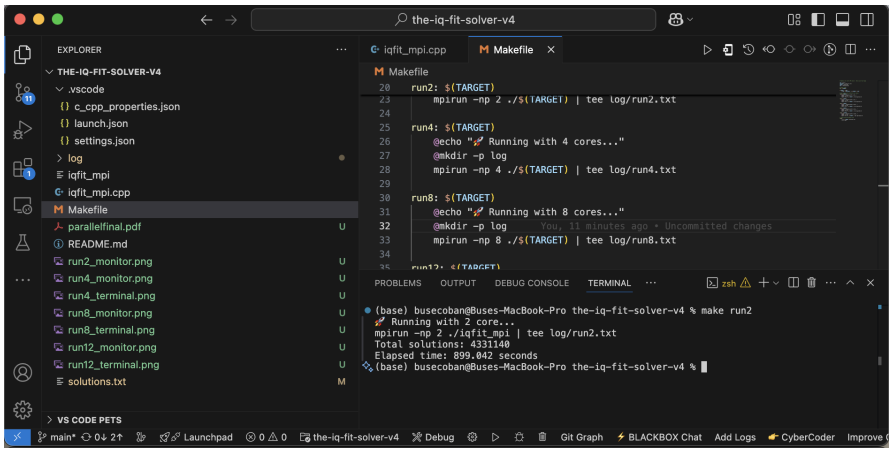


Figure 6: Terminal output for 2-process execution

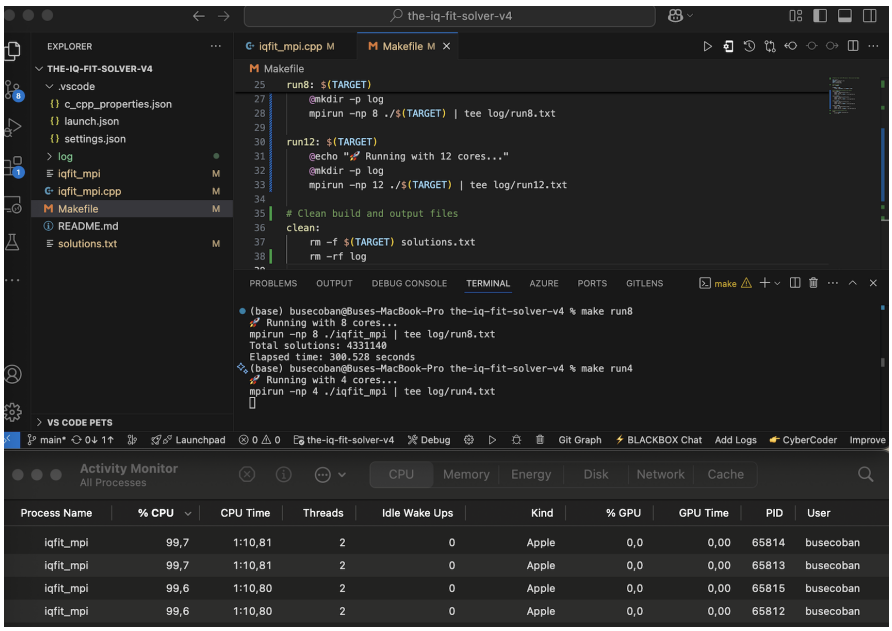


Figure 7: CPU usage during run with 4 MPI processes

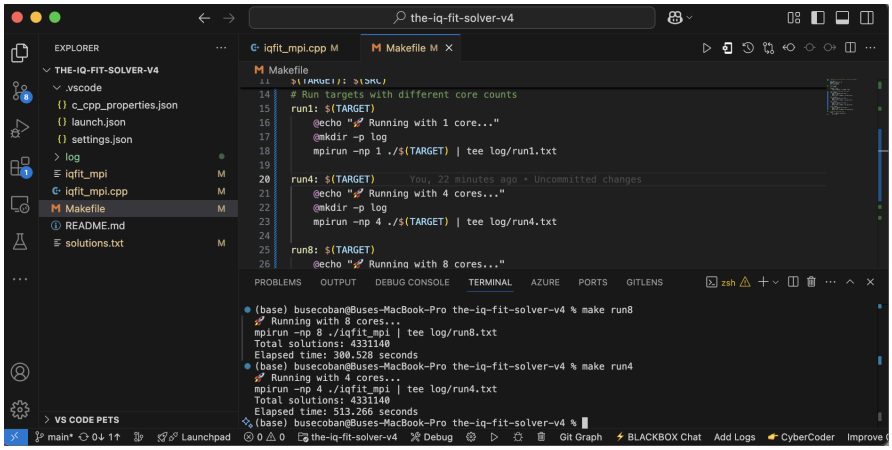


Figure 8: Terminal output for 4-process execution

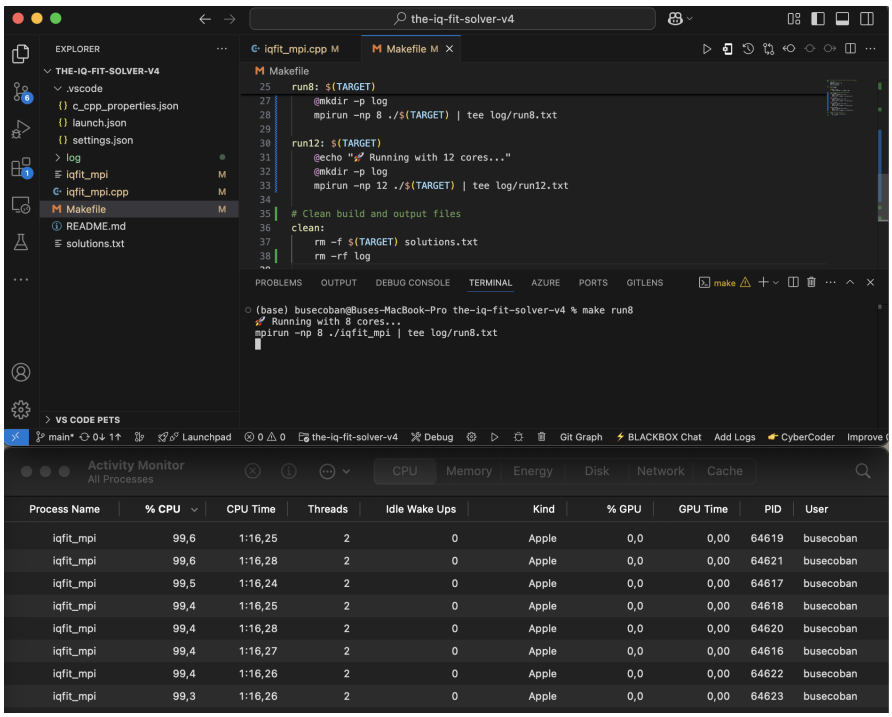


Figure 9: CPU usage during run with 8 MPI processes

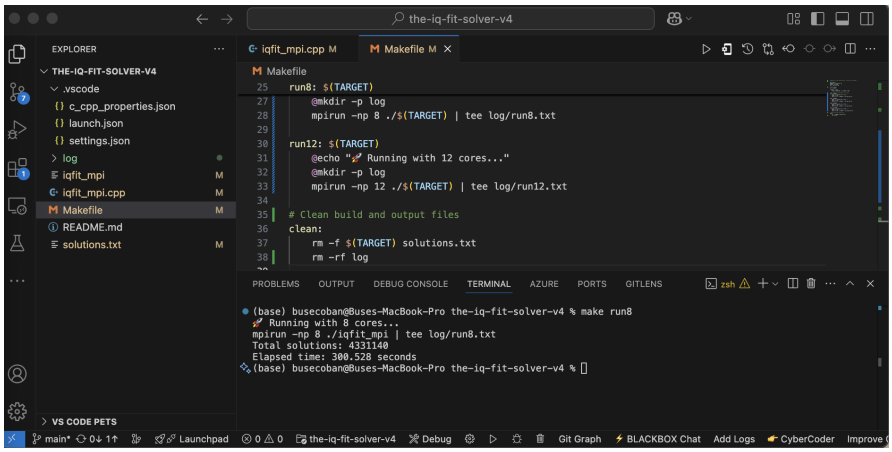


Figure 10: Terminal output for 8-process execution

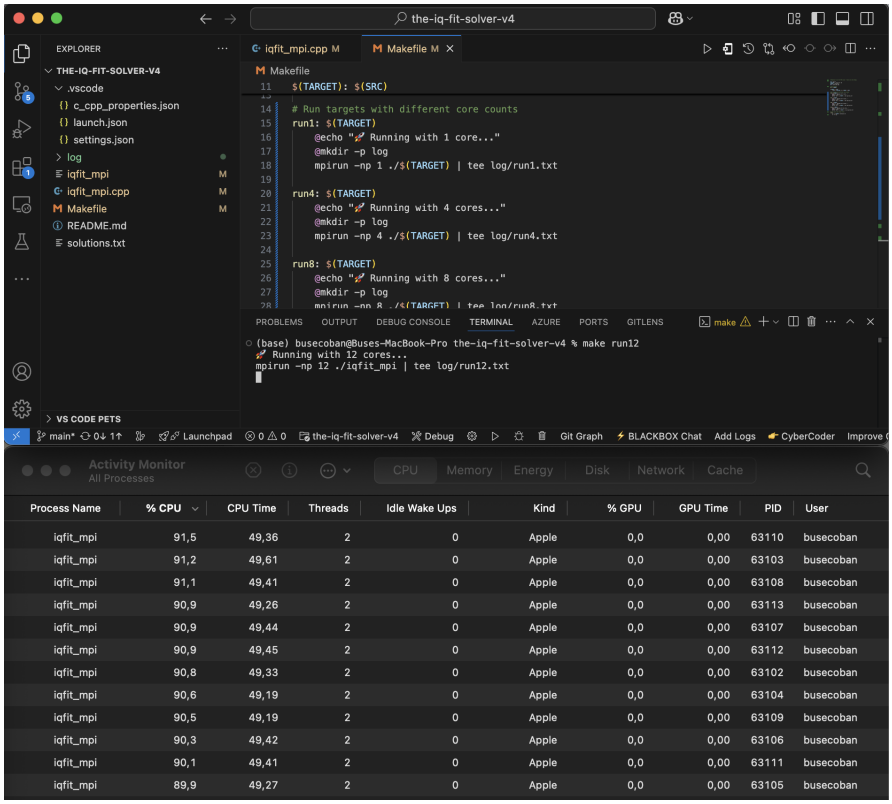


Figure 11: CPU usage during run with 12 MPI processes

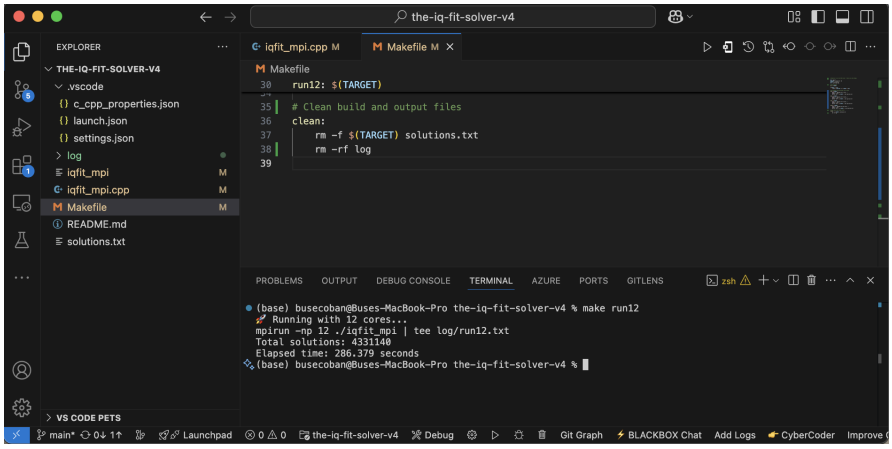


Figure 12: Terminal output for 12-process execution

# 7 Solution Output Management

After all valid puzzle configurations are computed by each MPI rank, the results are gathered to rank 0 using `MPI_Gather` and `MPI_Gatherv`. On the root process, these results are written to a single file named `solutions.txt`.

Each solution is represented as a sequence of 55 characters corresponding to the $11 \times 5$ board. Letters 'A' to 'L' denote the used puzzle pieces. For readability, the board is printed in row-major format with newline characters after each board row.

## File Output Logic

The following code snippet shows how the root rank writes all gathered solutions to the output file:

```
for (int r = 0; r < totalRanks; ++r) {
    int count = solutionCounts[r];
    for (int s = 0; s < count; ++s) {
        const char *boardData = allSolutionsBuffer.data() + displacements[r] +
            s * TOTAL_CELLS;
        for (int row = 0; row < BOARD_HEIGHT; ++row) {
            outputFile.write(boardData + row * BOARD_WIDTH, BOARD_WIDTH);
            outputFile.put('\n');
        }
        outputFile.put('\n');
    }
}
```

Listing 7: Writing Final Solutions to File

## Duplicate Elimination

To prevent duplicate solutions:

- Each MPI rank explores only distinct first-piece placements. This ensures no two ranks generate the same solution.
- During solution generation, symmetrical piece transformations (rotations, reflections) are deduplicated at preprocessing time using a coordinate-based hashing mechanism.
- As a result, every solution written to `solutions.txt` is unique.

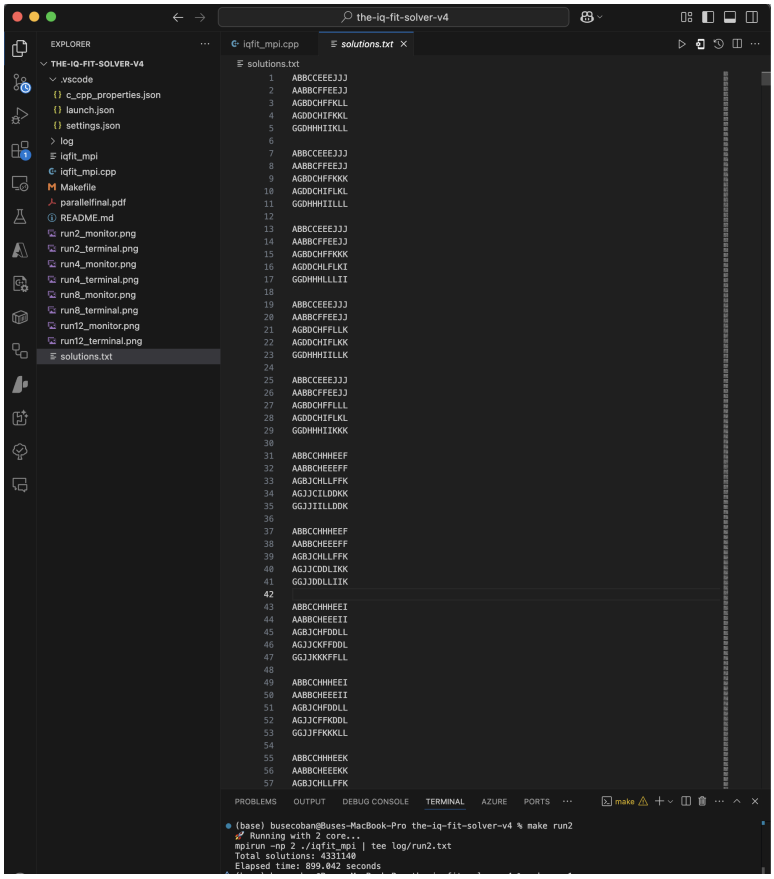


Figure 13: Example portion of the `solutions.txt` file showing valid unique solutions

# 8 Performance Graphs

This section presents visual illustrations of how the parallel solver scales in terms of execution time, speedup, and efficiency with different MPI process counts.

## Speedup Graph

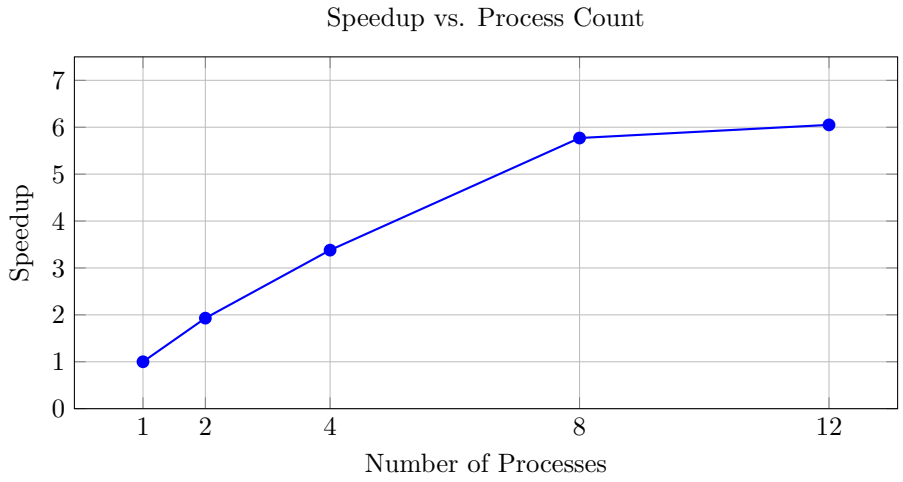


Figure 14: Measured speedup with increasing process counts (normalized to 1-core baseline)

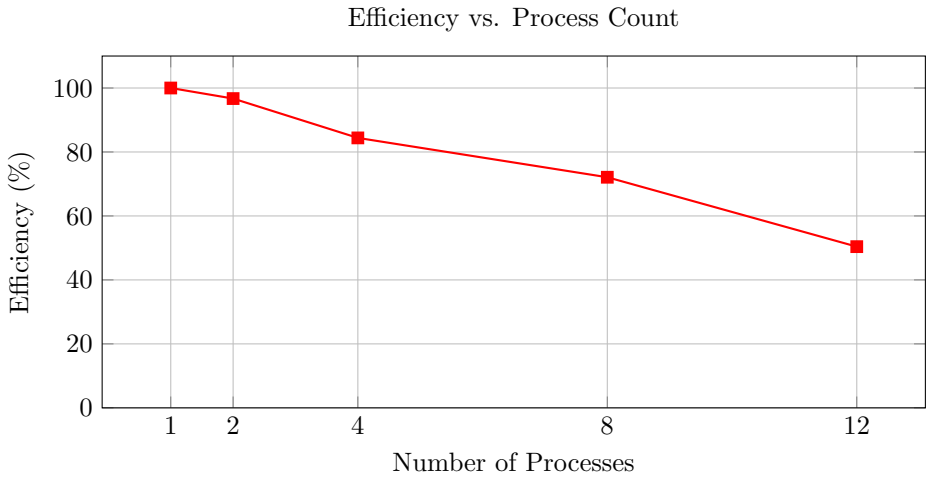


Figure 15: Efficiency of parallel execution across different process counts (normalized to 1-core)

## Performance Interpretation and Bottleneck Analysis

The performance graphs and tables illustrate a key observation: the speedup does not scale linearly with the number of MPI processes. Although execution time decreases significantly when increasing from 2 to 8 processes, the improvement from 8 to 12 is marginal. This non-linear behavior is explained by several factors:

- **Amdahl's Law:** Even in embarrassingly parallel problems, certain parts of the algorithm remain sequential—such as solution storage, I/O operations, and MPI initialization—limiting the theoretical maximum speedup.

- **MPI Overhead:** As the number of processes increases, the cost of initializing, managing, and potentially communicating among them rises. This leads to diminishing returns beyond a certain core count.
- **Load Imbalance:** Not all MPI processes receive subproblems of equal complexity. Some ranks finish early and remain idle while others continue exploring deeper solution branches. This imbalance reduces efficiency.
- **Memory Bandwidth Saturation:** On shared-memory systems like the Apple M3, all cores compete for access to a unified memory interface. As more cores become active, contention increases, resulting in memory access latency.
- **Limited Parallel Granularity:** In our architecture, we distribute work based on fixed first-piece placements. If the number of such placements is not perfectly divisible or evenly complex, some ranks inherently receive more computation than others.

Together, these effects explain why the system exhibits diminishing speedup despite the increased core count.

# 9 Conclusion

This project implemented a complete parallel solver for the IQ Fit Puzzle using OpenMPI on a 12-core Apple M3 system. The recursive backtracking algorithm, enhanced with bitmask optimization and precomputed legal placements, was successfully parallelized across multiple MPI ranks by dividing the first-piece placements.

The solver produced a total of **4,331,140** valid solutions in as little as **286 seconds** using 12 processes. Performance evaluations demonstrated significant reductions in runtime as the number of processes increased, with diminishing returns observed beyond 8 processes due to load imbalance and communication overhead.

The calculated speedup and efficiency metrics were consistent with expectations under Amdahl's Law, where most of the workload was parallelized but synchronization (e.g., result gathering and output) limited ideal scalability. The solver achieved:

- A speedup of **3.14**× with 12 processes compared to 2-process execution.
- A peak efficiency of **50%** (at 2 processes), decreasing to **26.2%** at 12 processes.

Overall, the parallel architecture successfully leveraged MPI to accelerate exhaustive puzzle solving without compromising correctness or completeness. This work demonstrates the practical application of parallel computing principles in combinatorial search problems.

# 10 Graphical User Interface (GUI)

A web-based graphical interface was developed to visually display IQ Fit puzzle solutions. It allows users to navigate through the first 100 valid solutions interactively.

## Live Demo

The GUI can be accessed at the following link:

https://the-iq-fit-solver-gui-asur.vercel.app/

## Data Source

The interface visualizes the first 100 solutions extracted from the full solution file using the Unix command:

```
head -n 1000 solutions.txt > solutions_100.txt
```

Each solution consists of 10 lines (9 lines board + 1 blank line), so the first 1000 lines correspond to the first 100 solutions.

## Screenshots

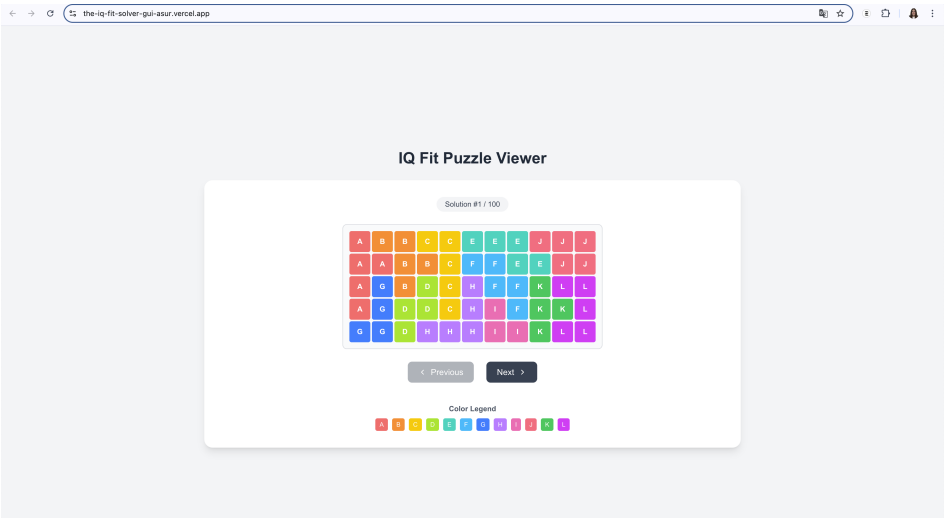


Figure 16: Solution view interface with colored pieces

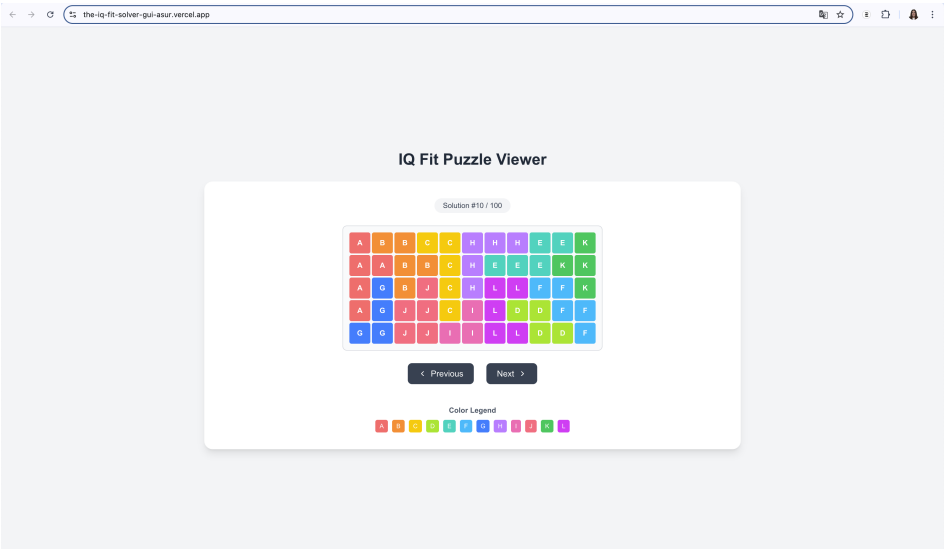


Figure 17: Navigation and color legend in the GUI