Accelerating Puzzle Solutions with Parallel Programming

Jun-Wei Chen
National Yang Ming Chiao Tung University
Hsinchu, Taiwan
chenwison0819.cs13@nycu.edu.tw

Yong-Zhe Zhang
National Yang Ming Chiao Tung University
Hsinchu, Taiwan
asd91020907@gmail.com

Ting-Huan Chen
National Yang Ming Chiao Tung University
Hsinchu, Taiwan
tim7658@gmail.com

1 Abstract

This study focuses on accelerating the solution process of combinatorial puzzles, particularly the "A-Puzzle-A-Day" challenge, through advanced parallel programming techniques. The puzzle's computational complexity, arising from its large search space, necessitates efficient algorithms to ensure feasible runtime. We propose a multi-threaded recursive backtracking approach, leveraging OpenMP and pThreads for parallelization. By incorporating task-based decomposition and thread-level optimizations, we achieve significant speedups over sequential solutions. Experimental results demonstrate the scalability and efficiency of the proposed methods across various configurations, providing insights into the trade-offs between complexity, synchronization mechanisms, and performance.

2 Introduction

The A-Puzzle-A-Day from DragonFjord[1] presents a new challenge each day, requiring players to fit polyominoes into a date-specific grid. Solving this puzzle manually or sequentially can be computationally expensive due to the vast number of possible configurations. To address this, we aim to develop a parallelized solution that leverages multi-core architectures to improve efficiency. By breaking down the problem into smaller tasks and distributing them across cores, we explore how parallel computing can significantly reduce the time needed to find valid solutions, making this project relevant for both entertainment and broader optimization problems.

3 Problem statement

The A-Puzzle-A-Day puzzle involves arranging polyominoes to fit a date-specific grid. The exponential growth of the search space makes manual or sequential solutions highly time-consuming and inefficient. This project aims to accelerate the process using parallel programming by dividing the problem into smaller subproblems for faster exploration. The strategy focuses on recursive backtracking and task-based decomposition to improve performance and reduce computation time.

4 Proposed approach

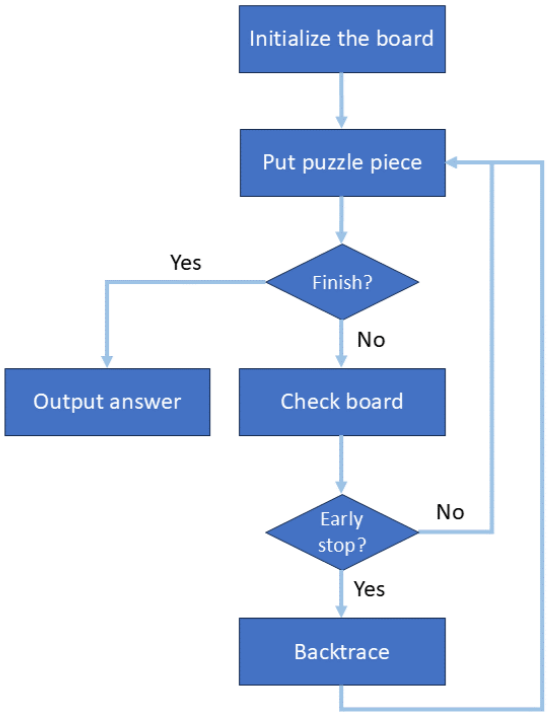

Figure 1. Our flowchart of the parallelized A-Puzzle-A-Day Solver. The process begins with board initialization, followed by iterative puzzle piece placements. At each step, the system checks if the puzzle is complete by evaluating the current board state and determines whether to proceed or backtrack.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Parallel Programming, 2024,
© 2024 Association for Computing Machinery.

5 Methodology

5.1 Overview

The method for solving the *A-Puzzle-A-Day* challenge involves a recursive backtracking approach, enhanced by parallelization techniques. The overall process can be broken down into the following steps:

1. **Initialize the board**: The puzzle board is initialized according to the specific grid layout for a given day, which corresponds to a particular date.
2. **Put puzzle piece**: A polyomino (puzzle piece) is placed onto the board. This placement is part of a recursive backtracking process, where each piece is positioned and checked for validity.
3. **Check if the puzzle is finished**: After placing a piece, the system checks if all pieces have been placed successfully and the board is fully covered. If so, the puzzle is solved, and the solution is output.
4. **Check board**: If the puzzle is not complete, the current configuration of the board is analyzed to determine if it can still be solved with the remaining pieces.
5. **Early stop condition**: If an invalid configuration is detected (i.e., no further valid moves are possible), an early stop is triggered. This prevents the system from exploring unproductive branches of the solution space.
6. **Backtrace**: If the current configuration is invalid or the early stop condition is met, the system backtracks by removing the last placed piece and attempts to find an alternative placement. This process continues recursively until a valid solution is found or all possibilities are exhausted.

5.2 Parallelization Strategy

Given the exponential growth in the number of possible configurations, the solution process is parallelized to improve efficiency. The parallelization strategy focuses on:

- **Task-based decomposition**: The recursive backtracking search space is divided into smaller tasks that can be executed in parallel across multiple processing cores.
- **Parallel execution of branches**: Each branch of the recursive tree (or subproblem) is explored independently, allowing different threads or cores to work simultaneously on different portions of the search space. This significantly reduces the total computation time.

By leveraging multi-core architectures, the time required to solve the puzzle is significantly reduced. This approach not only accelerates the solution process for the *A-Puzzle-A-Day* challenge but can also be extended to other combinatorial optimization problems where search space exploration is computationally intensive.

5.3 Method detail

- **pThread**: This work presents an advanced multi-threaded approach to solving a complex puzzle by arranging predefined shapes on a 10x10 grid to generate calendar-based solutions. Utilizing recursive backtracking algorithms, the method systematically explores all valid placements, ensuring each configuration meets specific calendar constraints. Each thread is assigned a unique rotation of the initial shape, allowing independent and parallel processing that significantly enhances computational efficiency and search speed. As threads operate concurrently, they validate board states and store valid configurations in a structured format indexed by month and day, facilitating easy retrieval and analysis. To ensure data integrity and thread safety, mutex-protected shared storage mechanisms are employed, preventing race conditions and data conflicts during simultaneous access. Additionally, the design optimizes resource utilization and coordination among threads, enhancing the algorithm's scalability and flexibility. Overall, this multi-threaded recursive backtracking technique demonstrates robust performance in generating calendar-compliant puzzle solutions, offering a reliable framework for similar constraint-based problem-solving tasks.
- **One/Two Level task**: The proposed One/Two Level Task approach limits task generation to the initial phase. Once the tasks are partitioned, no new tasks are created throughout the execution process, and the total number of tasks decreases progressively as execution proceeds, avoiding the task explosion observed in conventional methods. The core objective of this design is to balance the granularity of task partitioning with execution efficiency. By reducing the costs associated with memory management and scheduling, our approach further enhances the overall performance of the program. Additionally, this method simplifies the synchronization process, minimizing the overhead typically required for coordinating multiple tasks. As a result, the system achieves greater stability and can handle larger workloads more effectively without compromising on speed or reliability.
- **OpenMP task**: In the implementation of OpenMP tasks. At each recursive step of the backtracking algorithm, when a valid placement for a piece is identified, a new independent task is spawned using OpenMP. This independence allows them to run concurrently without needing synchronization between tasks. Each task represents a unique branch of the search tree, exploring further placements of pieces starting from the current board state. The only dependency occurs when multiple tasks are spawned from the same parent; in such cases, the parent task waits for all its child tasks

to complete before continuing. To ensure that tasks do not interfere with one another, each task operates on a private copy of the board and associated variables.

OpenMP task will dynamically assigns tasks to available threads in the thread pool. This dynamic scheduling ensures efficient utilization of computational resources, as threads that finish earlier are assigned new tasks. To avoid race conditions, shared data structures (such as the solutions container) are accessed within critical sections protected by mutexes. This ensures that concurrent updates to the shared data are safe and consistent.

5.4 Workflow

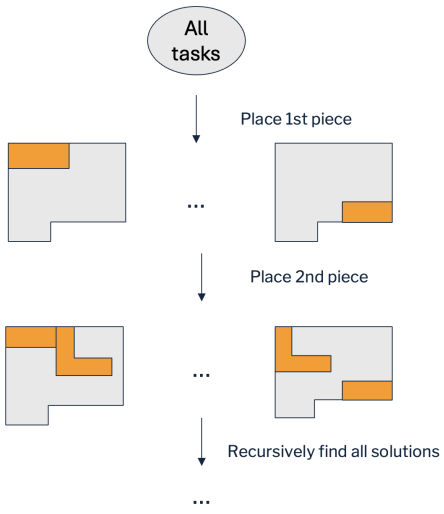

Figure 2. Serial version. Our flowchart of the parallelized A-Puzzle-A-Day Solver. The process begins with board initialization, followed by iterative puzzle piece placements. At each step, the system checks if the puzzle is complete by evaluating the current board state and determines whether to proceed or backtrack.

6 Related work

Parallelization of combinatorial puzzles, such as Sudoku, has been researched, with OpenMP and MPI often used to enhance performance[2]. For instance, previous implementations of large Sudoku solvers have demonstrated significant speedups by distributing tasks across threads or nodes, especially for problems requiring high computational power[3]. Similar strategies have been applied to other combinatorial puzzles, confirming the efficacy of parallel methods in reducing runtime for large, complex problems. These findings underscore the potential of parallel computing to accelerate problem-solving efficiently.

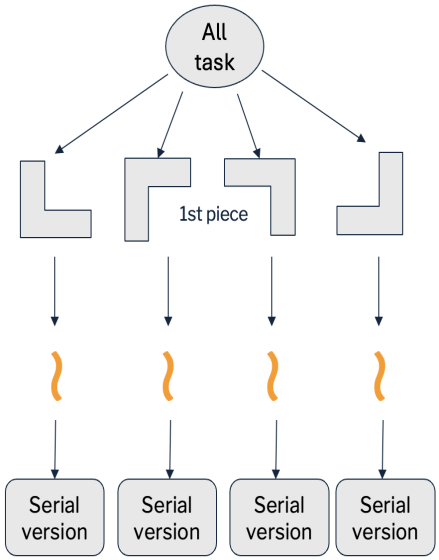

Figure 3. pThread. An illustration of the pthread-based approach to solving the jigsaw puzzle problem. The first piece is fixed in one of its possible orientations, and corresponding threads are spawned to independently compute subsequent solutions.

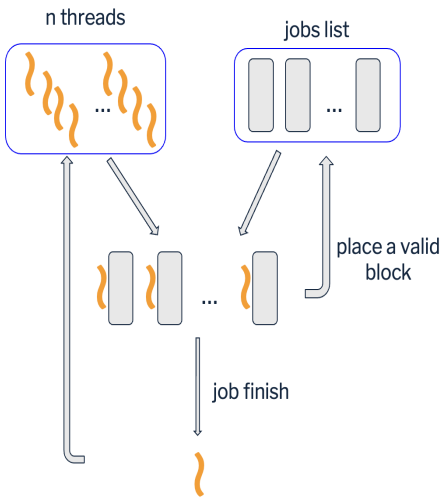

Figure 4. OpenMP task. An illustration of the OpenMP task-based approach for solving the jigsaw puzzle problem. Each board update generates a new task, which is added to a centralized task queue. Threads dynamically retrieve tasks from the queue for execution, enabling efficient load balancing and parallel exploration of the solution space.

7 Experimental Methodology

To evaluate the efficiency of our approach, we measured the speedup averaged over 10 trials. To ensure fairness, all parallel methods generated all possible results for each day.

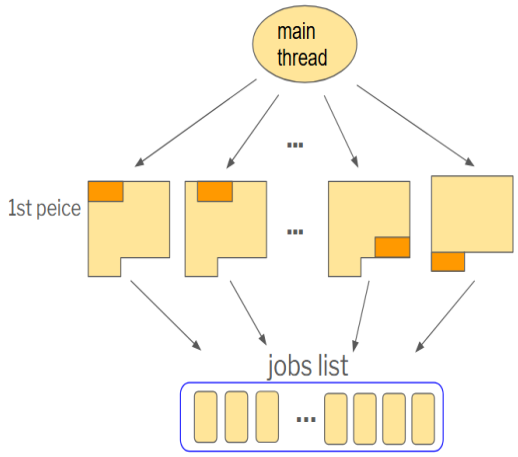

Figure 5. One/Two level task. An illustration of the One/Two Level Task approach using OpenMP tasks for jigsaw puzzle solving. The main thread partitions the search space based on all possible placements of the first piece, generating initial tasks that are added to the job queue. Threads then independently execute these tasks without generating additional tasks, ensuring efficient resource utilization and reducing task management overhead.

The experiments were conducted on Debian 12.6.0 workstations with Intel(R) Core(TM) i5-10500 @ 3.10GHz, Intel(R) Core(TM) i5-7500 @ 3.40GHz processors, and a GeForce GTX 1060 6GB GPU. The setup included g++-12, clang++-11, and CUDA 12.5.1 to support both CPU and GPU-based methods. These configurations ensured a consistent and reliable environment for accurately comparing the performance of the proposed methods.

8 Results

As **Table 1** shows, the Task-Based Approach achieves the highest speedup (5.92x at 6 threads) through fine-grained task generation but suffers from scheduling overhead at higher thread counts. The One-Level Task Approach, with task creation limited to the initial phase, reduces overhead and achieves 5.81x at 6 threads, balancing efficiency at the cost of scalability at lower thread counts. The Two-Level Task Approach strikes a middle ground by generating tasks up to the second recursion level, achieving 5.55x at 6 threads but facing limits in scalability at higher threads.

For synchronization, the Mutex-Based Approach ensures correctness but incurs significant thread contention, limiting performance to 5.47x at 8 threads. In contrast, the No-Mutex Approach avoids synchronization overhead, achieving 5.73x at 8 threads, but risks data inconsistency, making it unsuitable for workloads requiring strict correctness.

8.1 Performance comparison table

Task Performance

Threads	Time (s)	Speedup
1	355.009	1
2	198.554	1.788
3	128.828	2.756
4	90.739	3.912
5	72.685	4.884
6	60.009	5.916

One-Level Performance

Threads	Time (s)	Speedup
1	355.009	1
2	224.074	1.584
3	165.000	2.152
4	92.319	3.845
5	78.036	4.549
6	61.127	5.808

Two-Level Performance

Threads	Time (s)	Speedup
1	355.009	1
2	213.957	1.659
3	154.917	2.292
4	91.122	3.896
5	76.500	4.641
6	63.943	5.552

Mutex Performance

Threads	Time (s)	Speedup
1	355.009	1
2	182.792	1.942
4	90.641	3.917
8	64.959	5.465

No Mutex Performance

Threads	Time (s)	Speedup
1	355.009	1
2	185.813	1.911
4	89.130	3.983
8	61.938	5.732

Table 1: Performance between each method. Table 1 highlighting the varying speedups achieved under different thread configurations.

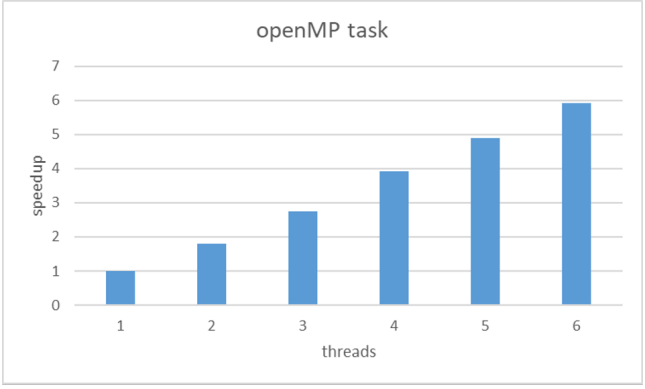

Figure 6. Task Performance. Speedup results of the OpenMP task-based implementation. The speedup achieved by each thread closely approaches the theoretical maximum, demonstrating the efficiency of the parallelization strategy in utilizing available computational resources.

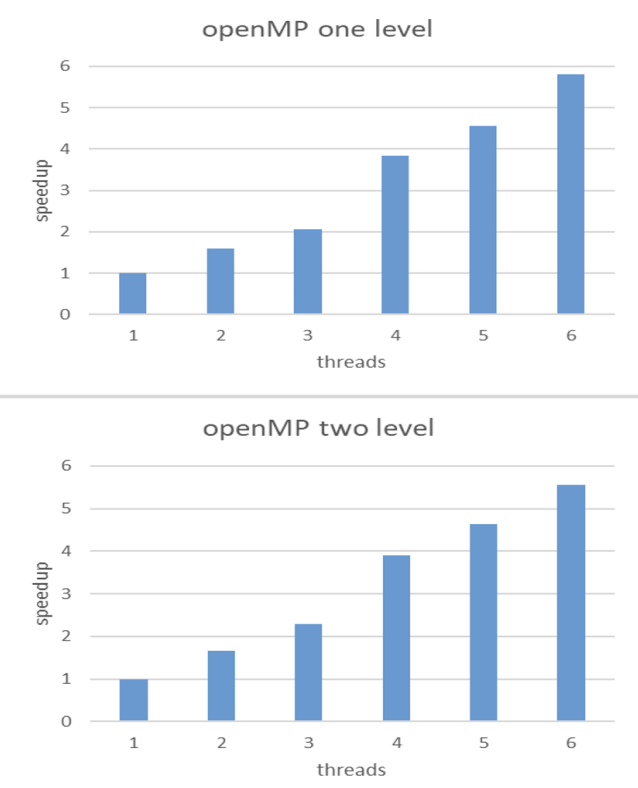

Figure 7. one/two-Level Performance. Speedup results of the OpenMP task-based implementation. The speedup achieved by each thread closely approaches the theoretical maximum, demonstrating the efficiency of the parallelization strategy in utilizing available computational resources.

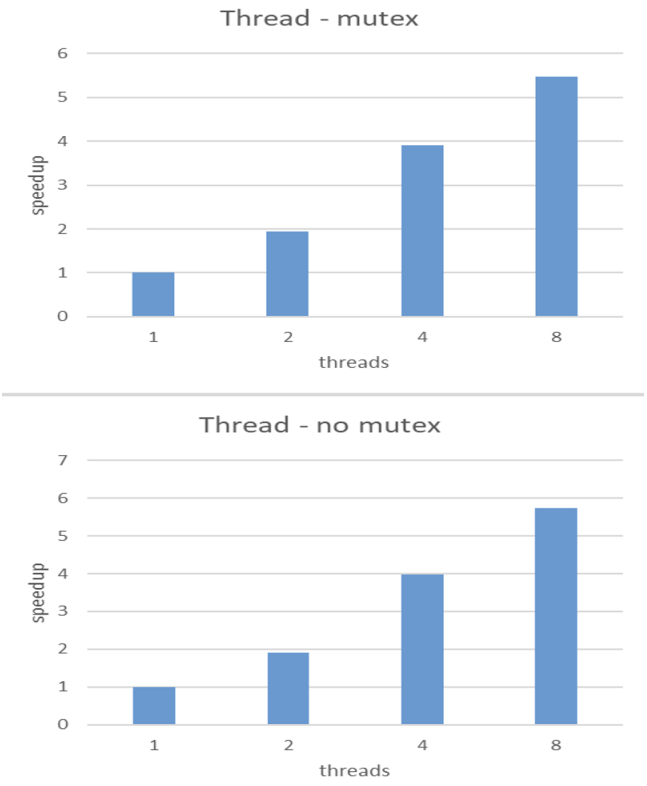

Figure 8. pThread Performance. Speedup results for pthread implementations with and without mutex. Both configurations achieve near-theoretical speedup for thread counts up to 8, with the no-mutex version slightly outperforming the mutex version at 8 threads due to reduced synchronization overhead.

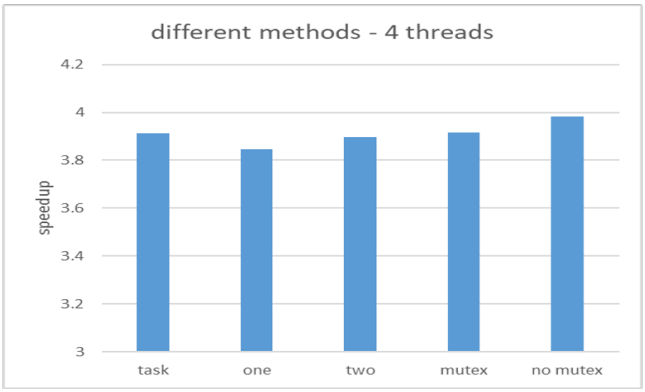

Figure 9. Methods performance comparison. Comparison of speedup across different methods using 4 threads. The One/Two Level approaches show slightly lower performance. Despite this, all methods achieve near-theoretical maximum speedup through effective parallelization.

9 Conclusion

Our work demonstrates the effectiveness of parallel programming in solving complex combinatorial puzzles, reducing

runtime significantly compared to sequential approaches. The experimental results highlight the trade-offs between synchronization strategies and performance, with task-based decomposition and thread-level optimizations offering the best balance of scalability and efficiency. This study not only provides a robust framework for tackling puzzles like "A-Puzzle-A-Day" but also offers insights applicable to other combinatorial optimization problems. Future work may focus on further refining task partitioning and exploring hybrid parallelization strategies to enhance performance across diverse computational environments.

References

[1] Dragonfjord. n.d.. Puzzle A Day. https://www.dragonfjord.com Accessed: YYYY-MM-DD.

[2] Peter Norvig. [n.d.]. Solving Every Sudoku Puzzle. https://norvig.com/sudoku.html

[3] Sruthi Sankar. 2014. Parallelized Sudoku Solving Algorithm Using OpenMP. http://example.com/lecture-notes CSE 633: Parallel Algorithms, Professor Dr. Russ Miller.