# High-Performance Parallel Solver for the Futoshiki Puzzle: A Multi-Paradigm Approach using MPI, OpenMP, and Hybrid Parallelization

*High Performance Computing for Data Science Project 2024/2025*

Wendelin Falschlunger
*Mat. 249562*
*University of Trento, DISI*
38123 Povo TN, Italy
w.falschlunger@studenti.unitn.it

Lorenzo Dongili
*Mat. 247204*
*University of Trento, DISI*
38123 Povo TN, Italy
lorenzo.dongili@studenti.unitn.it

Stefano Dal Mas
*Mat. 247201*
*University of Trento, DISI*
38123 Povo TN, Italy
stefano.dalmas@studenti.unitn.it

***Abstract*—The Futoshiki puzzle, an NP-Complete variant of the Latin Square Completion Problem, presents significant computational challenges that depend not only on grid size but also on constraint placement and density. Building upon the list coloring approach proposed by Şen and Diner [1], this work presents a parallel computing framework for solving Futoshiki puzzles. We first implement their sequential algorithm that uses constraint propagation to reduce search space, then develop three distinct parallel implementations: (1) MPI distributed-memory solver with dynamic master-worker load balancing, (2) OpenMP shared-memory solver using task-based parallelism, and (3) hybrid MPI+OpenMP solver that combines both paradigms for enhanced scalability. Our adaptive work generation algorithm dynamically adjusts parallelism granularity based on available resources and puzzle characteristics. Experimental evaluation on an HPC cluster demonstrates that constraint propagation provides substantial speedup over naive backtracking, while our parallel implementations show distinct performance characteristics: single-paradigm solutions excel within their resource domains, whereas the hybrid approach enables scaling to larger core counts with performance dependent on task factor tuning and puzzle characteristics.**

***Index Terms*—Futoshiki, Constraint Satisfaction, List Coloring, OpenMP, MPI, Hybrid Parallelization, High-Performance Computing, Task-Based Parallelism**

## I. Introduction

Combinatorial search problems are fundamental to computer science and artificial intelligence, with applications in logistics, scheduling, bioinformatics, and puzzle solving. The Futoshiki puzzle, a Japanese constraint satisfaction problem, serves as an excellent benchmark for evaluating algorithmic approaches to these challenges.

Futoshiki requires filling an N×N grid with numbers from 1 to N while satisfying two constraint types: the Latin Square property (each number appears exactly once per row and column) and inequality constraints between adjacent cells. These inequality constraints, represented by $<$ and $>$ symbols, create dependencies that distinguish Futoshiki from standard Latin Square puzzles.

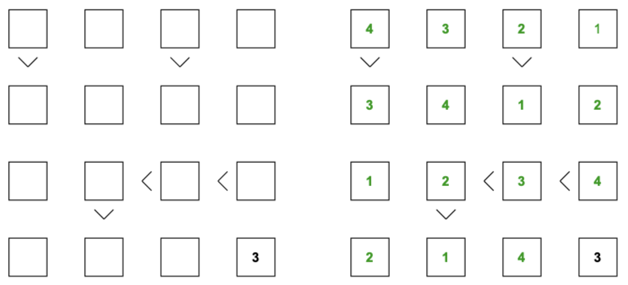

Figure 1. Example of a Futoshiki puzzle and its solved counterpart [1]

The computational challenge lies not only in Futoshiki's NP-Complete complexity but also in the irregular constraint patterns that make difficulty unpredictable from basic parameters like grid size alone. The arbitrary placement of inequality constraints creates variable search space sizes even for puzzles with identical dimensions and constraint counts.

Recent work by Şen and Diner [1] demonstrated that transforming Futoshiki into a list coloring problem enables polynomial-time constraint propagation that significantly reduces search space. However, even with these optimizations, challenging puzzles require substantial computational resources, making them suitable targets for parallel computing approaches.

This paper presents a parallel computing framework exploring three paradigms: distributed-memory parallelism using MPI, shared-memory parallelism using OpenMP, and a hybrid approach combining both. Our implementation contributes a dynamic work generation strategy that adapts to puzzle characteristics and available parallelism, along with comprehensive performance analysis across different computational scales.

### A. *Outline of the paper*

The remainder of this paper is organized as follows. Section II reviews relevant background on constraint satisfaction and parallel puzzle solving. Section III describes our sequential baseline implementation and three parallel approaches. Section IV presents experimental methodology and performance analysis across different puzzle characteristics. Section V and Section VI conclude with findings and future research directions.

## II. Related Work

This section establishes the theoretical foundation for our parallel Futoshiki solver, focusing on the list coloring approach and constraint propagation techniques that form the basis of our implementation.

### A. Futoshiki as a List Coloring Problem

Futoshiki belongs to the family of NP-Complete problems, with naive backtracking algorithms exhibiting exponential time complexity that renders them impractical for large grids. Şen and Diner [1] demonstrated that Futoshiki can be transformed into a list coloring problem, enabling more sophisticated solution strategies.

Their approach introduces a pre-coloring phase that propagates constraints to reduce possible values for each cell before recursive search begins. While this constraint propagation does not change the fundamental NP-Complete nature of the problem, it achieves substantial practical improvement by reducing the search space in polynomial time. This preprocessing eliminates numerous branches from the backtracking tree before exploration begins, dramatically reducing total execution time despite the unchanged theoretical complexity.

Theorem 1 in their work formally establishes the equivalence between Futoshiki instances and List k-Precoloring Extension problems. The list coloring transformation works by representing cells as graph vertices with adjacency based on row/column relationships, encoding inequality constraints as restrictions on possible color lists, and applying iterative constraint propagation to eliminate invalid possibilities. This approach typically reduces the effective search space by 70-90%, transforming an intractable exponential search into a manageable computational problem.

### B. Parallel Constraint Satisfaction

Parallel constraint satisfaction has been extensively studied, with approaches ranging from parallel constraint propagation to distributed search space exploration. Pacheco [2] provides comprehensive coverage of parallel programming patterns, including the master-worker paradigm that handles dynamic load balancing for irregular workloads.

Most existing work in parallel puzzle solving uses static work distribution, where the search space is divided equally among processors at the start of computation. This approach works well for problems with uniform computational difficulty but suffers from load imbalance when work units have varying complexity.

Futoshiki presents particular challenges for parallel implementation due to its irregular constraint patterns. Unlike Sudoku, where constraint placement follows regular patterns, Futoshiki's arbitrary inequality placement creates variable search space sizes that are difficult to predict statically. This motivates the need for dynamic work generation strategies that can adapt to both puzzle characteristics and available computational resources.

### C. Hybrid Parallel Programming

The MPI standard [3] and OpenMP specification [4] provide complementary approaches to parallel computing. MPI excels at distributed-memory communication across nodes, while OpenMP efficiently manages shared-memory parallelism within nodes.

Rabenseifner et al. [5] analyze hybrid MPI/OpenMP programming on multi-core clusters, examining the benefits and challenges of combining both paradigms. Their work demonstrates that careful consideration of task granularity and communication patterns is crucial when implementing hybrid approaches for irregular problems.

Our work builds upon these foundations by developing dynamic work generation strategies that adapt to puzzle characteristics while efficiently utilizing both distributed and shared-memory parallelism across different computational scales.

### D. Dynamic Load Balancing

Load balancing for irregular problems remains challenging due to the difficulty of predicting work unit complexity. Static approaches suffer from load imbalance, while purely dynamic approaches can incur significant communication overhead.

Our approach addresses this by generating sufficient work units upfront based on analysis of puzzle structure, then using on-demand distribution to handle runtime variability. This strategy provides a middle ground between static and fully dynamic approaches, maintaining efficiency while adapting to irregular workload patterns characteristic of constraint satisfaction problems.

## III. Methodology and Implementation

We implement the sequential algorithm described by Şen and Diner [1] and develop three distinct parallel approaches. Our methodology follows a systematic progression: first implementing their list coloring-based sequential solver, then developing MPI, OpenMP, and hybrid parallelizations that leverage distributed-memory, shared-memory, and combined paradigms respectively.

### A. Sequential Algorithm Implementation

We implement the list coloring transformation detailed by Şen and Diner [1]. Translating their theoretical approach into working code required several key implementation decisions that significantly impact the subsequent parallelization strategies.

*1) Pre-coloring: Search Space Reduction:* The pre-coloring phase, implemented in `compute_pc_lists`, computes a “possible color list” (`pc_list`) for each cell through iterative constraint propagation:

1) **Initialization:** Empty cells receive `pc_lists` containing all values 1 to N. Pre-filled cells contain only their given value.
2) **Inequality Filtering:** The `filter_possible_colors` function removes values that violate inequality constraints. For instance, if cell A $>$ cell B and A’s `pc_list` = $\{1, 2\}$, then B cannot contain values $< 2$.
3) **Uniqueness Propagation:** When a cell’s `pc_list` reduces to a single value, `process_uniqueness` removes that value from all other cells in the same row and column. To continue with our example, as B can have value 1 and 2, and as A can have $n < 2$, it would mean that A is going to hold 1 as it is the only integer strictly lower than 2, and B therefore would hold 2, as its pc_list had two values and we need to respect the *Latin Square* property.
4) **Iteration:** The previous two steps repeat until no further reductions occur, ensuring maximal constraint propagation.

This implementation typically achieves the 70-90% search space reduction reported in the paper, creating the foundation for effective parallelization.

*2) Backtracking Implementation:* The core solving algorithm translates the paper’s constrained search into the recursive function `seq_color_g`:

```c
bool seq_color_g(Futoshiki* puzzle,
                 int solution[MAX_N][MAX_N],
                 int row, int col) {
    if (row >= puzzle->size) return true;
    if (col >= puzzle->size)
        return seq_color_g(puzzle, solution,
                           row + 1, 0);

    if (puzzle->board[row][col] != EMPTY) {
        solution[row][col] =
            puzzle->board[row][col];
        return seq_color_g(puzzle, solution,
                           row, col + 1);
    }

    for (int i = 0;
         i < puzzle->pc_lengths[row][col]; i++) {
        int color = puzzle->pc_list[row][col][i];
        if (safe(puzzle, row, col,
                 solution, color)) {
            solution[row][col] = color;
            if (seq_color_g(puzzle, solution,
                            row, col + 1))
                return true;
            solution[row][col] = EMPTY;
        }
    }
    return false;
}
```

Listing 1. Sequential backtracking implementation

The implementation explores only values in each cell’s reduced `pc_list`, with the `safe` function validating constraint compliance. This creates the sequential baseline that serves as both the performance benchmark and the building block for parallel implementations.

**Correctness Foundation for Parallelization:** A critical property of Futoshiki puzzles is that each valid instance has exactly one unique solution. This characteristic enables safe parallel implementation with early termination—when any worker discovers the solution, all other workers can immediately halt without risking loss of correctness.

## B. Parallelization Strategy

The sequential implementation reveals parallelization opportunities in the backtracking phase. Since constraint propagation completes quickly in polynomial time, we focus on parallelizing the exponential backtracking search, following Amdahl’s Law principles [6] by targeting the computationally dominant phase.

All three parallel implementations share a common challenge: how to distribute the irregular search tree effectively across available workers. We address this through a dynamic work generation framework that creates appropriate work units based on available parallelism.

*1) Dynamic Work Generation Framework:* A key implementation challenge was developing a work distribution strategy suitable for all three parallelization paradigms. Rather than static partitioning, we implemented an adaptive approach in `parallel.c` that analyzes the search tree to generate appropriate work units.

```c
int calculate_distribution_depth(
    Futoshiki* puzzle, int num_workers) {
    int empty_cells[MAX_N * MAX_N][2];
    int num_empty = find_empty_cells(
        puzzle, empty_cells);

    for (int d = 1; d <= num_empty; d++) {
        long long job_count =
            count_valid_assignments_recursive(
                puzzle, solution, empty_cells,
                num_empty, 0, d);

        if (job_count > num_workers) {
            log_info("Depth %d generates %lld units"
  ,
                      d, job_count);
            return d;
        }
    }
    return num_empty;
}
```

Listing 2. Dynamic depth calculation for work generation

The algorithm explores progressively deeper levels of the search tree until generating sufficient work units. Each work unit represents a partial solution path, ensuring that workers receive meaningful chunks of computation while maintaining load balance. This approach addresses the irregular nature of Futoshiki’s constraint patterns that make static distribution ineffective.

*2) MPI Implementation: Master-Worker Pattern:* For distributed-memory parallelization, we implemented a master-worker paradigm suitable for cluster environments. The design applies core message-passing principles:

**Master Process (Rank 0):**

- Performs precoloring and fills out a basic version of the puzzle
- Based on the configuration factor given, decides the ratio of jobs to CPUs by evaluating the depth of the backtracking approach and opens the port to listen to the workers
- Sends 1 job to each worker via the `TAG_WORK_ASSIGNMENT`
- If a worker does not solve the job, it receives the message and sends a new job via the same tag. If the solution is found, it broadcasts to every other worker to stop via `TAG_TERMINATE`. This leverages the unique solution property of Futoshiki puzzles for correctness
- Collects the solution found and sends it to the user

**Worker Processes (Ranks 1 to P-1):**

- Poll master by asking for a job to solve via `TAG_WORK_REQUEST`
- Try to solve the precolored puzzle given. If successful, send `TAG_SOLUTION_FOUND` followed by `TAG_SOLUTION_DATA`, otherwise send another `TAG_WORK_REQUEST`
- Upon receiving `TAG_TERMINATE` from the master, gracefully shut down as this indicates a solution was found

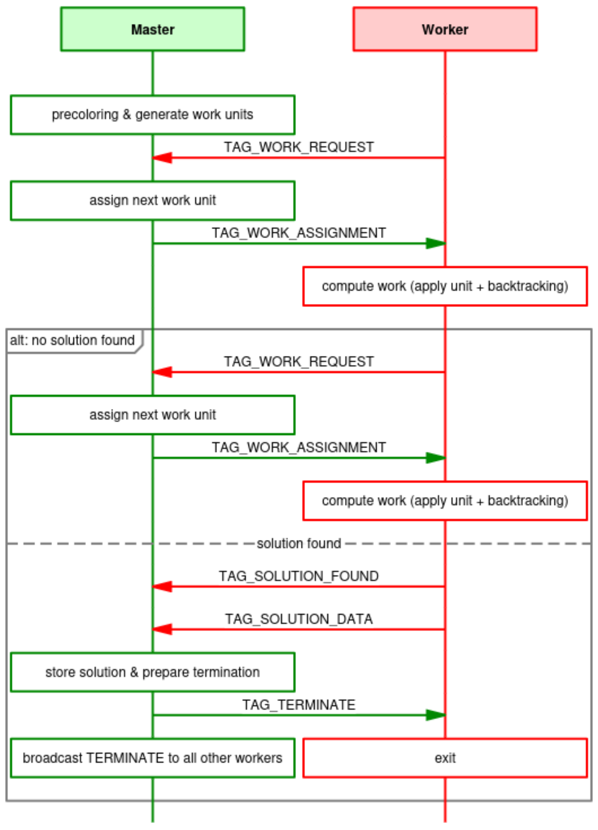

Figure 2. MPI Master-Worker communication sequence

**Implementation Note:** When only one MPI process is available, the implementation automatically falls back to the sequential algorithm as only a master and no worker would be available.

This design enables dynamic load balancing as workers request work only when ready, automatically handling heterogeneous performance and variable work unit difficulty typical in constraint satisfaction problems.

*3) OpenMP Implementation: Task-Based Parallelism:* For shared-memory parallelization, we implemented task-based parallelism using OpenMP. After generating work units, the master thread spawns OpenMP tasks that are dynamically scheduled across available threads.

From a high-level perspective, once the precoloring phase is performed, the master thread gives each worker a puzzle with a partial solution. When any worker finds the complete solution, it sets the `found_solution` flag to true, effectively stopping other threads from continuing unnecessary work.

```
#pragma omp parallel
{
    #pragma omp single
    {
        for (int i = num_work_units - 1;
             i >= 0 && !found_solution; i--) {
            #pragma omp task firstprivate(i) \
                        shared(found_solution)
            {
                if (!found_solution) {
                    int local_solution[MAX_N][MAX_N
    ];
                    apply_work_unit(puzzle,
                        &work_units[i],
    local_solution);

                    if (seq_color_g(puzzle,
                        local_solution,
                        start_row, start_col)) {
                        #pragma omp critical
                        {
                            if (!found_solution) {
                                found_solution =
    true;
                                memcpy(solution,
                                    local_solution,
                                    sizeof(
    local_solution));
                            }
                        }
                    }
                }
            }
        }
        #pragma omp taskwait
    }
}
```

Listing 3. OpenMP task-based implementation

**Key Implementation Features:**

- **Dynamic Scheduling:** OpenMP runtime automatically balances tasks across threads, adapting to varying work unit difficulty without explicit load balancing code.
- **Early Termination:** The shared `found_solution` variable enables early termination across all threads. This provides substantial speedup since probabilistically, the solution is unlikely to be found in the very last work unit processed.

- **Configurable Task Factor:** The dynamic task generation factor allows tuning the ratio between available threads and generated jobs, enabling control over the computational "stress" applied to the system.

**Single-Thread Fallback Design Decision:** An implementation detail worth noting is our fallback to the sequential algorithm when only one thread is available. While OpenMP can run with a single thread, we chose this approach for two reasons:

- **Overhead Avoidance:** Single-threaded OpenMP incurs code expansion and metadata overhead during compilation that provides no benefit when parallelism isn't utilized.
- **Interface Consistency:** To enable meaningful performance comparisons with MPI, we maintain similar design patterns across implementations.

**Shared Memory vs. Message Passing:** Unlike the MPI implementation, this approach leverages shared memory rather than message passing. Variables are explicitly declared as shared or private using OpenMP clauses, and early termination occurs through the shared variable mechanism rather than broadcast messages.

**Data Dependency Analysis**

Data dependency analysis in parallel programming identifies relationships between different parts of code that affect execution order. This analysis ensures parallel tasks operate without conflicts, preventing race conditions and maintaining correctness while enabling efficient parallelization.

The analysis focuses on the core parallel loop where tasks process work units concurrently. Table I presents the complete dependency analysis for the critical shared variables.

Table I
DATA DEPENDENCIES IN OPENMP TASK IMPLEMENTATION

| Variable | Earlier Statement | | Later Statement | | Loop | Dep. |
|---|---|---|---|---|---|---|
| | L# | Op | L# | Op | Carried | Type |
| found_solution | 20 | R | 21 | W | N | anti |
| found_solution | 21 | W | 6 | R | Y | flow |
| found_solution | 21 | W | 10 | R | Y | flow |
| found_solution | 21 | W | 20 | R | Y | flow |
| found_solution | 21 | W | 21 | W | Y | output |
| solution | 22 | W | 22 | W | Y | output |

**Dependency Analysis Results:**

1) **Anti-dependency (Line 20→21, same task):** Within each task, the read of `found_solution` at line 20 followed by potential write at line 21 creates an anti-dependency. This is benign since it occurs sequentially within the same thread.
2) **Flow dependencies (Lines 21→6, 21→10, 21→20):** When one task writes `found_solution = true`, other tasks must see this update to terminate early. Without proper synchronization, tasks might continue processing after a solution is found, causing inefficiency but not incorrectness.
3) **Output dependencies (Lines 21→21, 22→22):** Multiple tasks finding solutions simultaneously could write to the same memory locations concurrently, causing undefined behavior. This is not possible by the definition of a Futoshiki puzzle.

**Mitigation Strategies Applied:**

The implementation employs several OpenMP synchronization mechanisms to address these dependencies:

- `firstprivate(i)`: Ensures each task has its own copy of the loop variable, preventing interference between tasks
- `shared(found_solution)`: Explicitly declares shared access to the termination flag for memory coherence
- `#pragma omp critical`: Provides mutual exclusion for the check-then-act pattern, ensuring only one task can update shared state atomically
- `#pragma omp taskwait`: Prevents premature resource deallocation while tasks are still executing

The critical section implements a double-check pattern: tasks first check `found_solution` outside the critical section for efficiency, then check again inside for correctness. This minimizes time spent in the critical section while ensuring race-free updates to both the solution flag and the solution array.

*4) Hybrid Implementation: Combining MPI and OpenMP:* The hybrid approach represents the combination of multiple parallel programming paradigms for a single problem. Using hierarchical parallelization principles, we combine MPI's distributed-memory capabilities with OpenMP's shared-memory efficiency in a two-layer architecture.

**Two-Layer Design Philosophy:** Our hybrid implementation follows a systematic layering approach to maximize the benefits of both paradigms:

- **Outer Layer (Inter-node):** MPI distributes coarse-grained work units across cluster nodes, handling the master-worker coordination and work distribution logic identical to the pure MPI implementation.
- **Inner Layer (Intra-node):** OpenMP parallelizes the solving of each work unit within individual nodes, applying the task-based approach from our OpenMP implementation.

This design abstracts the two approaches while maintaining common APIs across implementations, enabling direct performance comparisons and modular development.

**Implementation Strategy:** The hybrid approach reuses the established MPI communication protocol while substituting the sequential solving step with our OpenMP solver. This demonstrates how MPI and OpenMP paradigms can be composed hierarchically:

```
static void hybrid_worker(Futoshiki* puzzle) {
    WorkUnit work_unit;
    MPI_Status status;

    while (true) {
        // MPI layer: request work from master
        int request = 1;
        MPI_Send(&request, 1, MPI_INT, 0,
                 TAG_WORK_REQUEST, MPI_COMM_WORLD);
        MPI_Recv(&work_unit, sizeof(WorkUnit),
                 MPI_BYTE, 0, MPI_ANY_TAG,
                 MPI_COMM_WORLD, &status);

        if (status.MPI_TAG == TAG_TERMINATE) break;

        // Apply work unit to create sub-puzzle
        Futoshiki sub_puzzle;
        memcpy(&sub_puzzle, puzzle, sizeof(Futoshiki
    ));
        apply_work_unit(&sub_puzzle, &work_unit,
                       sub_puzzle.board);

        // OpenMP layer: solve using task-based
    parallelism
        int local_solution[MAX_N][MAX_N];
        if (omp_solve(&sub_puzzle, local_solution))
    {
            // MPI layer: report solution back to
    master
            int found_flag = 1;
            MPI_Send(&found_flag, 1, MPI_INT, 0,
                    TAG_SOLUTION_FOUND,
    MPI_COMM_WORLD);
            MPI_Send(local_solution, MAX_N * MAX_N,
                    MPI_INT, 0, TAG_SOLUTION_DATA,
                    MPI_COMM_WORLD);
            break;
        }
    }
}
```

Listing 4. Hybrid worker combining MPI and OpenMP

**Design Benefits and Trade-offs:** The hybrid approach demonstrates several key principles from parallel computing:

1) **Hierarchical Decomposition:** Work is decomposed at two levels—coarse-grained distribution via MPI and fine-grained task parallelism via OpenMP, matching the hardware hierarchy of clusters with multi-core nodes.
2) **Communication Minimization:** By using OpenMP within nodes, we reduce the number of MPI processes needed, decreasing inter-node communication overhead which is typically orders of magnitude more expensive than intra-node synchronization.
3) **Resource Utilization:** The approach can better exploit modern cluster architectures where each node contains multiple cores, using MPI for scalability across nodes and OpenMP for efficiency within nodes.
4) **Implementation Modularity:** By maintaining common interfaces, the hybrid approach leverages existing MPI and OpenMP implementations without requiring complete redesign.

**Configuration Flexibility:** The hybrid model supports various process-thread combinations, enabling exploration of different resource allocation strategies. For a fixed number of computational units, one can configure:

- **MPI-heavy:** More processes, fewer threads per process (e.g., 8 processes × 2 threads)
- **OpenMP-heavy:** Fewer processes, more threads per process (e.g., 2 processes × 8 threads)
- **Balanced:** Equal distribution (e.g., 4 processes × 4 threads)

This configurability allows performance tuning based on problem characteristics and hardware topology, demonstrating practical application of parallel programming trade-offs.

The hybrid implementation showcases how multiple parallelization paradigms can be systematically combined, providing a comprehensive parallel programming approach to constraint satisfaction problems.

## IV. Experimental Evaluation

In this section we are going to present the results gotten from our experimental evaluation. We start off by giving a brief overview of how we can(not) determine the complexity of a Futoshiki problem instance a priori in Section IV-A, we then move onto presenting the actual HW used to perform such tests in Section IV-B.

We then present some common ground strategies used across every problem that we have run: Precoloring in Section IV-C and Factor Section IV-D.

We move towards presenting our parallel solutions in Section IV-E, starting with MPI and OMP in Section IV-E1, we give an overview of the Speedup and Efficiency of our solution in Section IV-E2 and finally we present the results for our Hybrid implementation in Section IV-E3.

We wrap up the section by performing the strong scalability analysis in Section IV-F: MPI first in Section IV-F1, then OMP in Section IV-F2 and finally Hybrid one in Section IV-F3.

In order to get the reader accustomed with what we are evaluating, we present only a subset of the whole family of instances that we have evaluated for this project. We specifically center our focus towards a 9x9 instance, so that the reader can understand how different solutions compare given the same problem to solve. In case you are more interested, you can find more plots in our Google Drive Folder [7].

As we have chosen to show only a specific 9x9 in order to not let the reader get confused with the parameters under which we are showcasing our solution performs, the conclusion we are going to track are relative to this instance. As we are going to see that stating the difficulty of an instance *a priori* is an hard task, findings for our 9x9 instance might not hold for another 9x9 instance. Let's say we are finding that MPI in this case outperforms the hybrid solution: this does not mean that the MPI is better *overall*, but that for the specific instance considered it is better and might be worse in another instance of the problem. If we wanted to have an omni comprehensive study this would require to analyze each instance on its own, which would imply analyzing the results for each problem instance. This does not mean that the analysis is useless, instead this shall serve as a rule of thumb to showcase the capabilities when taking into account the parallelization of Futoshiki solvers.

If the reader is interested in seeing how the solution performs under different instances, but without having a detailed analysis, he can find some extra results in the aforementioned drive folder.

## A. Can we evaluate difficulty of a Futoshiki before solving it?

It is important to stop for a moment and assess a crucial factor: as we have already explained in Section III-A2 how Futoshiki has dynamic placed constraints, which in turn implies that the assumption we can make on the input are less strict w.r.t. Sudoku. This means that by receiving as input only the number of constraints, without knowing where they are placed and how they are intertwined one to another, it is extremely hard to define the actual "difficulty" of the problem. It is easy to see that with a simple manipulation of the constraints presented in Figure 1, one could not have easily understood the chain between values 1-4. Another factor to take into account is where the initial numbers are placed and how many we have. It is true as the number of constraints that we have increases, the more variables we can fix and therefore the less big the search spaces becomes, but if those constraints are placed in a "bad way", one could not find the solution easily.

We can also say that the "size matters" in this case it not strictly true. One could imagine a puzzle of a specific size and one of a slightly bigger size: let's say 9x9 and 10x10. Given those as inputs, as the correlation among constraint is so important that just like for the amount of numbers placed, they might not lead to an "easier" or "harder" puzzle by default. To wrap up this consideration, we can safely say that if we are only given number of constraints, number of initialized cells and size of the problem it is not possible to give a precise estimate over a metric to evaluate the "difficulty" of the problem.

For this reason the *Weak Scalability analysis* cannot be performed in our specific case, because in order to devise the entries for which we evaluate we cannot just say 5x5, ... 9x9, 10x10 and so on, as they are not the only discriminant factor.

## B. Experimental Setup

We now move onto some first consideration on the underlying Hardware, to have an idea of what the numbers that we are going to see actually mean.

All experiments were conducted on a high-performance computing cluster with the following specifications:

- **Hardware:** 126 nodes with Intel Xeon processors.
- **Network:** 10Gb/s Ethernet with Infiniband/Omnipath options.
- **Software:** Linux CentOS 7, GCC 9.1, MPICH 3.2.
- **Test Cases:** Various puzzles from 5×5 to 11×11, including "hard" instances.

## C. Impact of Pre-coloring Optimization

As we have already discussed in Section III-A1, the precoloring procedure helps us at doing some pre computation, which in turn lets us reduce the search space of our solution. In Figure 3 we see how by performing precoloring gives us better results w.r.t. the standard brute force approach. In the X axis we have 3 different 9x9 problems to be solved, and in the Y axis the total time measured in seconds for *the solving time of our algorithm only*.

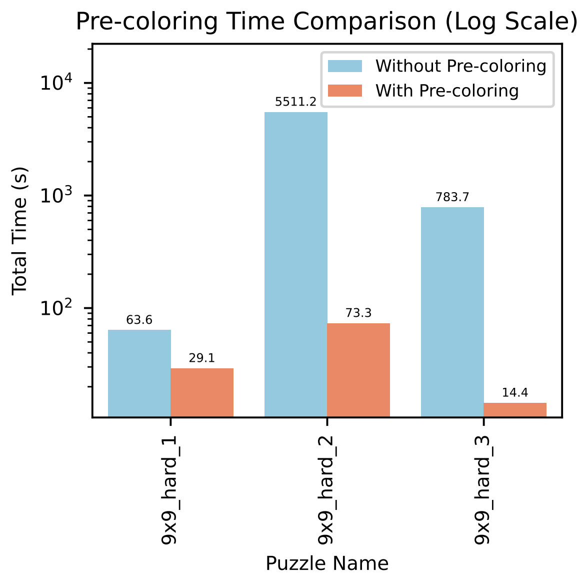

Figure 3. Comparison of execution times for some 9x9 futoshiki problems

This figure lets us understand that by employing this technique we can see up to 54x speedup, like in the case of the instance *9x9_hard_3*. This means that even if we have a little bit of overhead to apply the list coloring solution, we can still see great improvements over the baseline. For this reason we chose to employ precoloring on all of the following case studies, so that we both reduce the amount of variables in our comparison while introducing several fold speedup in our proposed solution.

## D. What the factor are we talking about?

We have already mentioned that our solutions have a parameter presented as *Configurable Factor*. We said that this factor lets us tweak the ratio between the jobs being scheduled and the underlying computational unit (either a thread for the OpenMP case, or a CPU for the MPI one). From an high level point of view we can think it as a knob that lets us choose at our will the amount of "pressure" that we are putting the underlying system in.

From a more formal point of view, given a factor F and the number of underlying computational units C, our preprocessing algorithm aims at going deeper and deeper into the search space by exploring the backtrack tree until it does not find an amount of puzzles to solve P such that:

$$F * C \leq P$$

This lets us play around to find the best possible ratio, which therefore lets us find the maximum amount of "stress" to put our computational units at before we have generated too many jobs for them.

In fact, we have run several tests to assess which factor could be the best one to choose from, so we now present our *Factor Analysis* that we have devised to pick it.

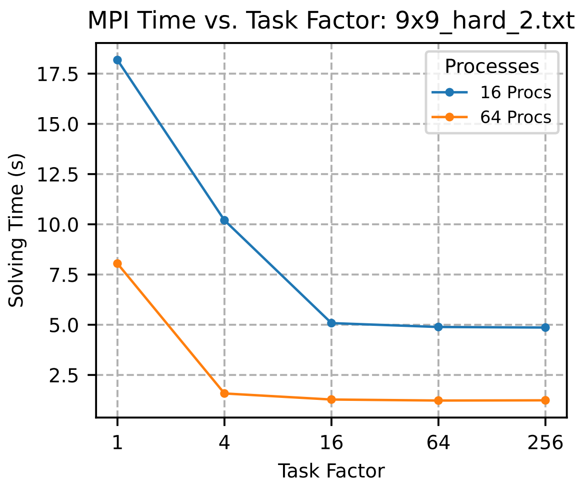

Figure 4. factor analysis for MPI

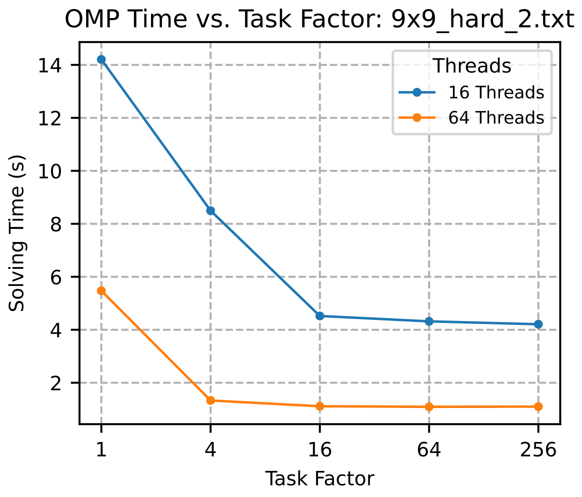

Figure 5. factor analysis for OMP

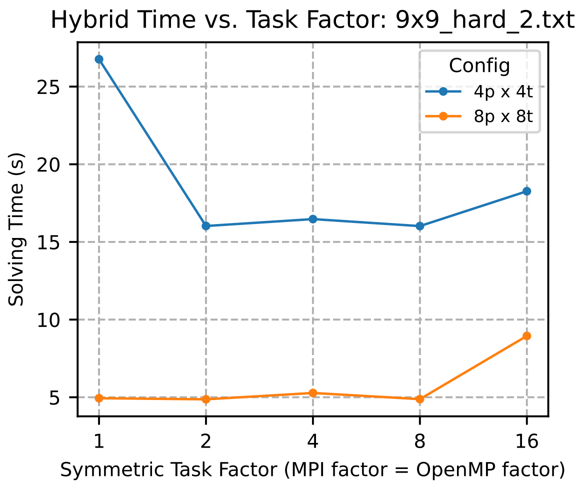

Figure 6. factor analysis for hybrid solution

From Figure 4 and Figure 5 we can see that we have a good slope in the decreasing of the solving time of our solution with 16 processors/threads up until 64, and then we start to stabilize.

After running other experiments, we have seen that 64 was most of the times the elbow point at which we would see diminishing returns and for this reason we picked such value.

As for the hybrid, we have faced the issue on "how can we evaluate MPI and OMP solutions"? In order to provide some interesting results, we have decided to abstract away from the actual implementation of the worker (MPI or OMP), and we have therefore decided to treat a processor in MPI as the same as the threads for OMP. This modeling simplification is going to help us in Section IV-E3 when evaluating how the hybrid solution works under several different configurations.

After having taken this necessary detour, we can move towards finding the best factor for the hybrid one: we can see from Figure 6 that if we consider the symmetric factor 8, which means setting MPI Factor to 8 and Openmp Factor to 8, if we take this approximation into account (considering processors = threads), we can see how 8*8=64 is also a fairly good factor in this case.

For this reason, we have decided to set for the following runs MPI and OMP factor to 64, and for the hybrid the MPI Factor to 8 and its symmetric counterpart to 8, so that we would have a factor of 64 "computational units" across the board.

The concept of oversubscription, it being creating more tasks than workers is well-established in parallel computing, and as we have seen with a factor of 64 we can see up to **3.5x**.

### E. Parallel Performance Analysis

After having understood how hard it is to evaluate the difficulty of our problem *a priori*, and having set the ground for our base configuration (hardware used, pre coloring algorithm, factor selected), we can finally move into the evaluation of how our parallel solutions work.

*1) Solving times for MPI and OMP:* We start off by showing how our MPI and OMP solutions work with a 9x9_hard instance and a 10x10_hard instance (note that the "hard" concept comes from how the website from which we have taken the problems from ranked them. Here are the 2 main sources which we downloaded from [8], [9] and converted images to compatible representation via a custom parser which we have devised in order to gather data).

Please note that as the cluster has a maximum amount of thread per node set to 64, we cannot gather data in the case of 128 threads due to HW constraints. This is the main reason why in the following plots there is not going to be an entry for OMP with 128 Computational units.

We start off by showing a 9x9 instance in Figure 7. On the x axis we have the number of computational units that we are using, and on the Y axis the time it required for our algorithms to find a valid permutation of numbers to solve the puzzle.

We can see that, as we expect, as the number of computational units increase, the solving time for our problem decreases as we can extract more and more parallel work.

The red dotted line serves as a baseline, as it is the time required for our sequential algorithm to find a solution and it is going to be present in the next graphs also, so that one can easily understand how a given parallel implementation compares w.r.t. the sequential one. Please note that, as also the execution of the sequential algorithm is depending on the hardware, what we are presenting is an average over the execution times of the sequential solution.

We can see that for 1 and 2 we have values which are in the range of the sequential algorithm minus some seconds of delta in the measurement and this is because, as we have already presented in the Section III we have decided to opt for the sequential algorithm to remove the overhead given by OMP and MPI.

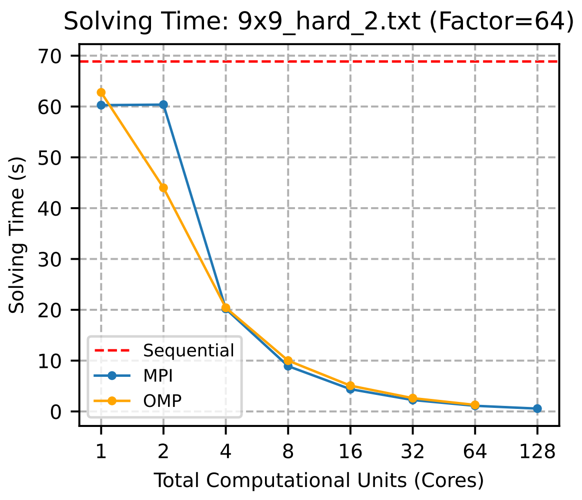

Figure 7. Solving time comparison for a 9x9 instance

We now move towards a 10x10 instance. In Figure 9 we see something strange: every parallel implementation is consistently worse w.r.t. solving time compared to the sequential solution (minus some cases like the 4,8 and 16 CU which have little to no actual increase in performance).

This is very interesting, because by looking at the absolute value of the solving time we can note that even the sequential algorithm is taking less than 0.01 second to find the solution. This lets us draw two necessary conclusions:

1) As we have stated in Section IV-A, the size alone is not a factor which directly implies the difficulty of the problem. We can see that the absolute time to solve the 9x9 and 10x10 are vastly different, even if someone might at a first glance say "if we have more cells we should have higher execution times". This does not always hold if we do not consider all of the aforementioned variables.
2) Parallel solutions introduce some overhead, either the message passing infrastructure needed for the MPI framework to process messages, or the locks needed to achieve high efficiency shared memory reading and writing when accessing critical parts of code. Such overhead is usually mitigated by the fact that the overall *speedup* in finding a solution is higher than the overhead introduced by parallelizing the work. This is not the case for these specific families of problems: as the sequential solution is already extremely fast, it does not make sense to parallelize it, and therefore we can conclude that this family of instances are not tailored or our parallel solution analysis.

For such reasons, we are going to move to analyze only the family of 9x9 instances, as they are more interesting and actually give us some interesting data and points to reason about w.r.t. just saying "for the 10x10 instance we have a degradation in performance".

We have decided to insert an image of the Futoshiki problem taken into consideration, which is the 9x9_hard_2 and it can be seen in Figure 8

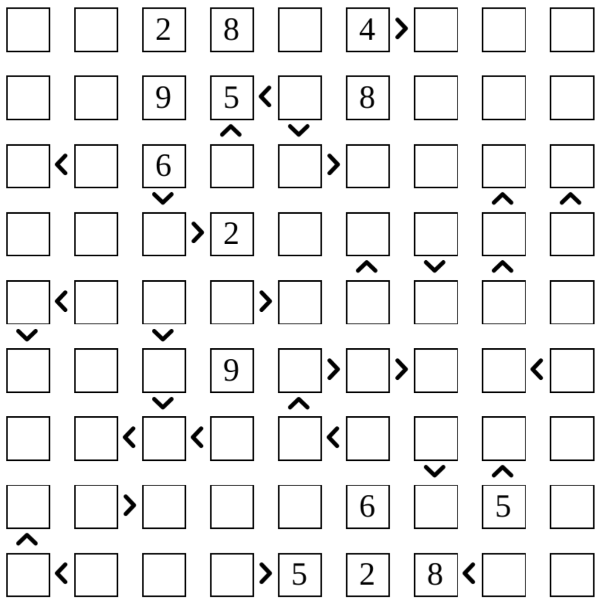

Figure 8. Our problem instance taken into consideration for evaluation purposes.

*2) How well does parallel solution perform compared to sequential?:* Now that the reader has understood how configurations and difficulty of the puzzles impact the overall execution time, he might ask himself "is there a metric to evaluate how much better the parallel solutions are compared to the sequential one?"

Turns out that we have such metrics. We know that if we have a task and we parallelize it, we can use the *Speedup* concept to understand how much better it gets, and it is presented as:

$$S = \frac{Execution\_time\_sequential}{Execution\_time\_parallel}$$

This means that the lower the execution time is when parallelized, the higher the speedup is going to be. In Figure 10 we

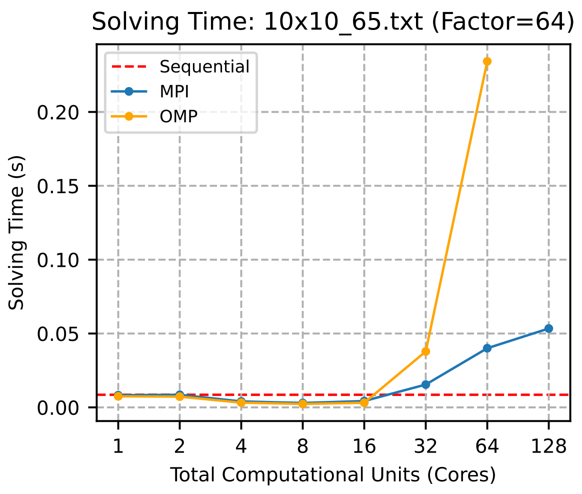

Figure 9. Solving time comparison for a 10x10 instance

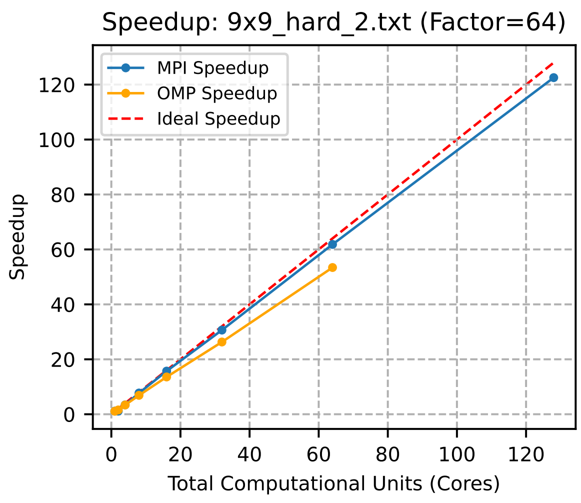

Figure 10. Speedup of parallel solutions

show the speedup that we can reach with our solution. On the x axis w.h. the computational units, whereas in the Y axis we have the speedup value. The red line represents the theoretical speedup that we can reach, so assuming that the parallelization of the task introduces absolutely 0 overhead.

We can see that, while for MPI the speedup is close to the ideal one, for the OMP case it is not. Of course, again, we do not have the 128 computational unit present for OMP as we are capped at 64 threads.

We can also conclude something interesting:

1) As the number of computational units increase, we introduce higher overhead due to the message passing or the shared variables accesses, which is not much if we have "few" computational units, but as the number goes up the overhead is getting a little bit bigger, therefore impacting the overall speedup. For this reason we see that the theoretical line and the real one strand further and further as the number of the CU increase.
2) Another kind of overhead to consider is the one introduced by *Resource Contention*, which is happening due to increasing the amount of threads and cpus utilizing the same memory, therefore reducing the effective memory bandwidth or saturating caches.
3) The fact that the speedup does not grow as fast as the ideal one does not mean that the solution gets slower: It is still faster than the sequential solution.

We can now move into the second metric that we can use to evaluate our parallel algorithm w.r.t. the sequential one: the *Efficiency*. This is a derivative parameter which is dependent on the speedup, and it can be expressed as:

$$E = \frac{Speedup}{\#computational\_units}$$

This parameter is basically relating the speedup (how well the parallel solution is w.r.t. the sequential one) with the number of units available. This parameter is therefore going to tell us how well the available cores are being used.

Figure 11 we can see how efficiency is evaluated in our solutions.

We can make some considerations here:

1) MPI is strictly dominating the OMP efficiency plot, and this tells us that MPI overhead due to the shared memory access is higher than the message passing approach used by MPI, which is fairly interesting.
2) We see a degradation in efficiency for MPi with 2 CUs. This is due to the fact that for MPI with 2 workers we are basically running the sequential under the hood: 1 master and 1 slave with the MPI message passing overhead, so it is perfectly expected.

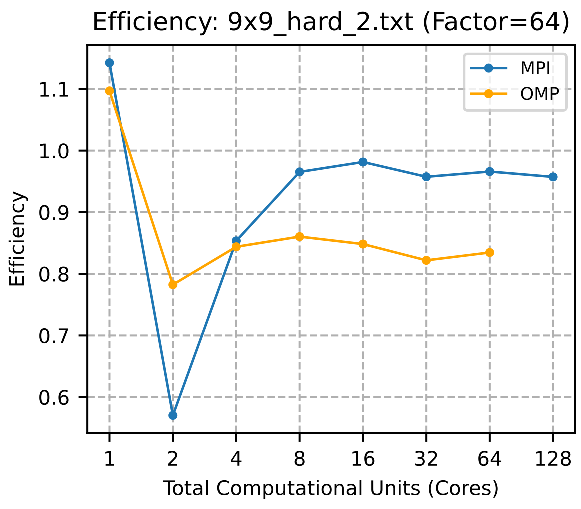

Figure 11. Efficiency of parallel solutions

*3) How does the Hybrid solution perform?:* Now that we have presented the MPI and OMP solution we can move to our hybrid solution. For this one we have devised 3 scenarios: one in which we would balance the number of CPU for MPI and threads for OMP, one in which we relied more on the MPI rather than OMP and the latter one viceversa.

This means that if we have for example 16 available computational units, we would have the balanced setting with 4 cpus for MPI and 4 threads for each MPI slave, whereas for the other cases we would have 2 cpus and 8 threads or 8 cpus and 2 threads respectively.

Note that we have already fixed the factor to 8 and 8 in Figure 6, which is not to be confused with the available CPUS/Threads that we are selecting.

The main reason is to assess if for the tasks given and the architecture of our solution, it would be better to rely more on MPI or OMP – or maybe a balanced approach would be the best...

This is what we are trying to answer with Figure 12. We present the same instance of 9x9 which we are evaluating the parallel solutions in order to have less variables to consider. The structure is the same: we have the red line representing the average solving time for our sequential time and we color coded the balanced, MPI Heavy and OMP Heavy accordingly. Over the Y axis the solving time measured in seconds.

We can take several notes:

- As expected, as the number of CU increase the solving time reduces.
- The sweet spot to get a performance increase lies in the 8-32 zone and after that for the law of diminishing returns we can see that actually we are not improving by a lot in terms of absolute values.
- we can see how actually the choice of relying on one or another one does not influence too much the overall execution time if the number of units increase, no matter whether they are threads or cpus.
- We can see an outlier with MPI Heavy and 2 Computational units. This is due to the fact that we are running the sequential solution under the hood with the overhead of message passing from MPI.

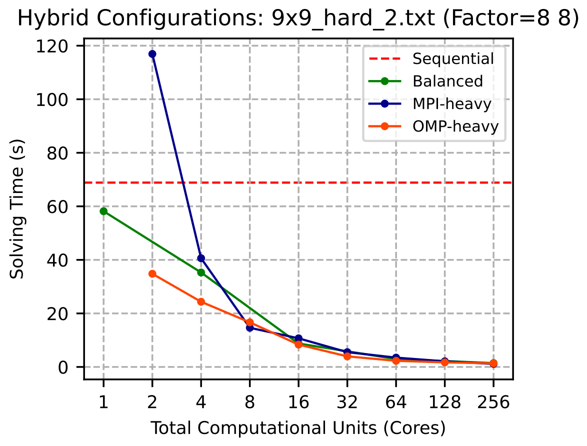

Figure 12. Solving time for hybrid solution

We can now move onto the comparison among hybrid and sequential, just like we did for the normal solutions: we start off with the Speedup in Figure 13.

Also here we can draw some conclusions:

- The overall speedup is not as close as the ideal one just like for the MPI and OMP only cases. This is actually expected, as we have seen how the overhead added from the underlying technology (locks to access shared memory in OMP, waiting algorithms used in MPI solution to sync messages) actually add an overhead, and the more computational units we have, the more they compound.
- We can see that up to 128 computational units the OMP heavy solution has a little bit more of an advantage compares to the other ones, but at 256 the MPI heavy solution surpasses the other ones (which can also mean, it does not degrade as much as the other ones).

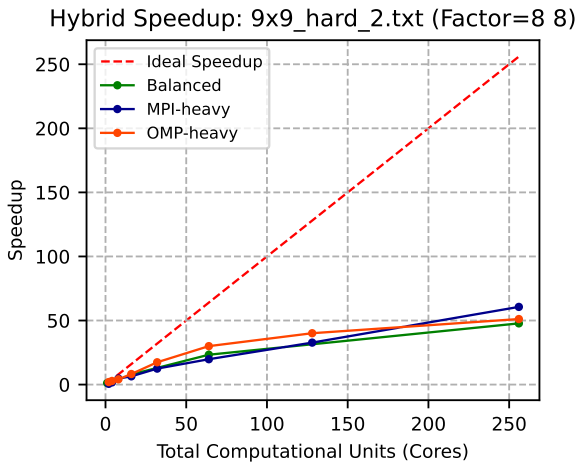

Figure 13. Speedup of hybrid solution

We now move onto the last metric, it being the *Efficiency*. From Figure 14 we can see that:

- The overall efficiency is less compared to the standard MPI and OMP only approaches, and this is due to the aforementioned overhead from the underlying technologies used.
- There is a strict trend in this case: as the number of computational units increase, the overall efficiency decreases. This is expected as the more Computational units we have, the less each one is contributing towards the final result w.r.t the solving time. It would be interesting to play around with the factor to overload more the computational units to see how the efficiency performs.
- As the efficiency value is basically a mirror over the speedup, as we have seen that the speedup was not on the same level as the speedup for the classic solutions, we were actually expecting that the efficiency would decline as the computational units increase.

## *F. Strong Scalability: MPI, OMP and Hybrid*

In this subsection we are going to show our strong scalability analysis given MPI, OMP and Hybrid solutions. These

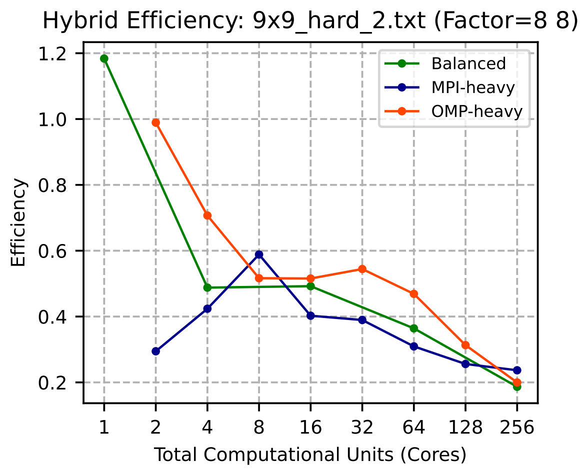

Figure 14. Efficiency of hybrid solution

are going to be presented in a tabular fashion to give an easier way of seeing absolute values compared to the trends that we have already seen up until now with the plots.

*1) MPI Scalability:* Table II presents MPI performance across multiple nodes:

Table II
MPI STRONG SCALING (9×9 HARD PUZZLE)

| Processes | Time (s) | Speedup | Efficiency |
|---|---|---|---|
| 1 | 60.27 | 1.00 | 100.0% |
| 2 | 60.37 | 1.14 | 57.0% |
| 4 | 20.17 | 3.41 | 85.4% |
| 8 | 8.92 | 7.72 | 96.5% |
| 16 | 4.38 | 15.70 | 98.2% |
| 32 | 2.25 | 30.64 | 95.8% |
| 64 | 1.11 | 61.82 | 96.6% |
| 128 | 0.56 | 122.54 | 95.7% |

MPI shows a "quasi linear" scaling in speedup terms. We see how with 1 and 2 the execution time is practically the same (minus noise given by underlying hardware, scheduling of resources, etc), but then it quickly reduces. We can see how the overall efficiency values are higher than OMP, suggesting that MPI is better suited to stress the underlying hardware, which is exactly what we have seen in Figure 11.

*2) OpenMP Scalability:* Table III shows OpenMP performance with increasing thread counts on a single node:

The OMP solution shows a good improvement as the number of thread increase. Up until 16 threads the solving time almost halves each time. It is worth noting that the efficiency remains extremely high also for 64 threads, which means that the solution holds very well on shared memory systems.

*3) Hybrid Scalability:* The hybrid solver demonstrates superior scalability by combining both paradigms:

Here we presented the hybrid solution by increasing first the threads. We can see how the efficiency overall reduces due to the MPI and OMP overhead, which means that the synchronization overhead basically compounds as the number of computational units increase. Overall this table shows how we can still achieve high performances increase, but at the cost of a worse efficiency, which would suggests us that the best setting if we want to maximize this metric is the MPI solution.

Table III
OPENMP STRONG SCALING (9×9 HARD PUZZLE)

| Threads | Time (s) | Speedup | Efficiency |
|---|---|---|---|
| 1 | 62.78 | 1.00 | 100.0% |
| 2 | 44.01 | 1.56 | 78.2% |
| 4 | 20.40 | 3.38 | 84.4% |
| 8 | 10.00 | 6.88 | 86.0% |
| 16 | 5.07 | 13.57 | 84.8% |
| 32 | 2.62 | 26.30 | 82.2% |
| 64 | 1.29 | 53.41 | 83.5% |

Table IV
HYBRID SCALING (9×9 HARD PUZZLE)

| Processes | Threads | Time (s) | Speedup | Efficiency |
|---|---|---|---|---|
| 1 | 1 | 58.17 | 1.00 | 100.0% |
| 1 | 2 | 34.81 | 1.98 | 98.9% |
| 1 | 4 | 24.35 | 2.83 | 70.7% |
| 2 | 2 | 35.29 | 1.95 | 48.8% |
| 2 | 4 | 16.66 | 4.13 | 51.7% |
| 2 | 8 | 8.35 | 8.25 | 51.6% |
| 4 | 1 | 40.62 | 1.70 | 42.4% |
| 4 | 2 | 14.63 | 4.71 | 58.9% |
| 4 | 4 | 8.74 | 7.88 | 49.2% |
| 4 | 8 | 3.95 | 17.42 | 54.4% |
| 4 | 16 | 2.29 | 30.01 | 46.9% |
| 8 | 2 | 10.69 | 6.44 | 40.3% |
| 8 | 4 | 5.52 | 12.47 | 39.0% |
| 8 | 8 | 2.95 | 23.30 | 36.4% |
| 8 | 16 | 1.72 | 40.09 | 31.3% |
| 8 | 32 | 1.35 | 51.15 | 20.0% |
| 16 | 4 | 3.48 | 19.80 | 30.9% |
| 16 | 8 | 2.10 | 32.76 | 25.6% |
| 16 | 16 | 1.44 | 47.77 | 18.7% |
| 32 | 8 | 1.14 | 60.63 | 23.7% |

## V. CONCLUSION

We presented a comprehensive high-performance computing implementation for solving the Futoshiki puzzle, demonstrating the feasibility of solving an instance of an NP-hard by employing several different techniques, starting from the search space reduction via precoloring, and later on moving towards MPI, OMP and Hybrid solutions.

We have to note that when gathering data from the runs, we have noticed some discrepancies w.r.t. the execution times, and this is most likely dependent from the fact that older machine with the same node category (c-nodes) might have different hardware, or there is some wear and tear damage performed. Another factor is also due to the fact that the scheduler and the timing of when some jobs are scheduled might incur in some variance over the expected finishing time of our solutions.

We have also noted that there were some differences in the time required to get results since the changes done in august to the hpc, which caused some processes which were able to finish before this update was rolled out to not finish at times.

### A. Key Achievements

Our implementation achieves significant performance improvements at multiple levels:. As always, we are evaluating those results in the specific instance proposed in the evaluation section, so these numbers are for this case, but they shall serve as a rule of thumb for how much the speedup can be.

1) **Algorithmic Optimization:** We have seen that precoloring phase, based on list coloring theory, reduces the search space by 70-90%, providing a 59× speedup over naive backtracking. This optimization is crucial for making puzzles which have a bigger search space tractable also with a sequential solution, and therefore we opted to apply in every subsequent experiment.
2) **Parallel Scalability:** We successfully parallelized the solver using three distinct paradigms. Here are the execution time findings:
   - **OpenMP** achieves up to 53.4× speedup on 64 threads with minimal code complexity compared to the sequential solution.
   - **MPI** demonstrates 123× speedup when using 128 processes. From our Speedup analysis it also slightly beat the OMP solution and therefore was also the best for the efficiency parameter. This means that, while the OMP solution is the easier one to implement compared to the MPI version, it also suffers more from memory contention and mutex or lock-like structures needed to perform syanchronization.
   - **Hybrid MPI+OpenMP** reaches 60.4× speedup when using 32 CPUs and 8 threads, which showcases how leveraging MPI is actually the best solution compared to the other hybrid configurations. This was expected as the MPI only solution is already slightly better than OMP. An interesting finding is that while the Hybrid does not perform as good as the MPI only overall, there are still cases in which its configurability can lead to better results compared to our test under analysis.
3) **Configurable Performance:** The task generation factor allows fine-tuning for specific hardware configurations and puzzle characteristics, with 64 factor leading up to **3.5x** speedup over the baseline (factor set to 1, which means we are not oversubscribing anything).
4) **Scalability Analysis**: the Strong analysis lead us to understanding that MPI achieves almost linear speedups and retains higher efficiency, making it the best solution if we are aiming at maximizing that parameter. We have also seen that, due to the fact that it is hard to determine a priori how complex a puzzle can be to solve given only its size, it does not make sense to proceed with the Weak scalability analysis.

Overall we can therefore say that the speedup performed by the list coloring algorithm in order to precolor the solution is extremely important, that in our evaluated instance MPI outperforms OMP and the hybrid solution, while at times it introduces some overhead due to the message passing and shared memory synchronization, therefore decrementing the efficiency at higher computational units selected, still serves as a fairly interesting solution due to its extreme configuratbility. This does not mean that the OMP solution is bad as a whole, what we are stating is that the other two solutions outperform it, but still is an extremely useful entry point due to its ease of implementation, as it requires little work to transform some sequential algorithms to parallel ones.

Other than the Futoshiki instance itself, one could easily see how this can be adapted to a Sudoku Solver, a generalized version for Graph Coloring, Scheduling problems and of course, even with some overhead to convert the problem instance, also for general SAT solving or SMT solving strategies.

## VI. Future Work

During the course of this report we have presented how our framework compares w.r.t. a standard sequential solution when it comes to the instance of Futoshiki solving.

There are still some areas to explore in order to shed some light into. Here are presented in a bullet point fashion:

- **Variable Ordering Heuristics:** Implement most-constrained-variable (MCV) and minimum-remaining-values (MRV) heuristics to select cells more intelligently during backtracking.
- **Optimizing search space reduction:** We have seen how the precoloring algorithm already does some of the heavy lifting when it comes to reducing the search space, but there are still some techniques which can be applied. The first one would be to Implement *clause learning* to avoid redundant exploration of similar subtrees, which can happen quite frequently considering the class of search space. The latter technique is the *Symmetry Breaking*.
- **Parameter tweaking**: We have seen how the Hybrid configuration enables some configuratbility by defining how much the solution can rely on MPI or OMP, we can therefore move in this direction to tweak these parameters and see how hybrid solution works in different scenarios. Another *factor* to play around with is the configuration factor itself. As it determines the amount of over subscription, playing around with this value could lead to interesting findings in its relation with the *efficiency*.
- **Reducing communication overhead** via non-blocking MPI message passing strategies, which would lead to a decrease of idle-time in the single CPUs.
- **A new metric**: we have stated that it is not easy to determine the complexity of a problem based on single variables alone (e.g. size, number of constraints, etc). As this goes beyond the scope of exploring the parallelization of this problem, we have still decided to include this bullet point as by finding a single metric which takes into account all of the variables of the problem in one go,

one could design a configuration finder such that, given the puzzle, it would find the best configuration (either in the hybrid zone, or just picking OMP or MPI for some specific tasks) to enhance the performance of the whole solution.

- **Distributed Precoloring**: while we have stated that due to the law of deminishing returns it does not make much sense to distribute the first precoloring phase, given a huge search space the overhead of parallelizing also this problem could be a smaller factor compared to the time taken for our current sequential precoloring algorithm, so this might be worth exploring in the future.
- **More dynamic structure**: by collecting runtime data, one could implement a more dynamic work distribution.

These extensions would further improve performance, increment the applicability area of our solution, and make the framework more accessible to researchers working on constraint satisfaction problems or instances which can be easily reduced to CSPs.

## References

[1] B. B. Şen and Öznur Yaşar Diner, "List coloring based algorithm for the futoshiki puzzle," *An International Journal of Optimization and Control: Theories & Applications*, vol. 14, no. 4, pp. 294–307, 2024.

[2] P. S. Pacheco, *An Introduction to Parallel Programming*. Morgan Kaufmann, 2011.

[3] Message Passing Interface Forum, "MPI: A message-passing interface standard, version 4.0," Message Passing Interface Forum, Tech. Rep., June 2021. [Online]. Available: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[4] OpenMP Architecture Review Board, "OpenMP application programming interface, version 5.1," OpenMP Architecture Review Board, Tech. Rep., November 2020. [Online]. Available: https://www.openmp.org/spec-html/5.1/openmp.html

[5] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes," in *Parallel, Distributed and Network-Based Processing, 17th Euromicro International Conference on*. IEEE, 2009, pp. 427–436.

[6] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring). New York, NY, USA: Association for Computing Machinery, 1967, p. 483–485. [Online]. Available: https://doi.org/10.1145/1465482.1465560

[7] "Our google drive," https://drive.google.com/drive/folders/1yoTW_qEyay7eqQgOsUIRt-v5sbG6yubK?usp=sharing.

[8] "Puzzle-futoshiki.com," https://it.puzzle-futoshiki.com/daily-futoshiki.

[9] "puddelbee.com," https://www.puddelbee.com/puzzles/futoshiki/index.html.