Parallel HITORI Solver

High Performance Computing for Data Science Project 2024/2025

Simone Brentan
239959
simone.brentan@studenti.unitn.it
University of Trento
Trento, Trentino Alto-Adige, Italy

Matteo Costalonga
239960
matteo.costalonga@studenti.unitn.it
University of Trento
Trento, Trentino Alto-Adige, Italy

Alex Reichert
239961
alex.reichert@studenti.unitn.it
University of Trento
Trento, Trentino Alto-Adige, Italy

ABSTRACT

Hitori is a Japanese logic puzzle played on an $N \times N$ grid of numbers, resembling Sudoku but with slightly different rules. The goal of this game is to shade certain cells so that no row or column contains duplicated unshaded numbers, while ensuring that all unshaded cells form a single connected area. Therefore, solving the Hitori puzzle efficiently involves searching large solution spaces, making it well-suited for parallelization. This paper investigates different strategies for the implementation of the Hitori solver using MPI, OpenMP and a hybrid approach that integrates both paradigms. We begin by introducing the fundamental concepts of the game and explaining its rules before presenting our Hitori-solving algorithm. Next, we detail the implementation of each parallelization approach, discussing the strengths and challenges encountered during development. Finally, we evaluate performances and compare key metrics such as speedup, efficiency and scalability across the three different implementations.

KEYWORDS

C programming, Parallelization, Hitori puzzle, High-Performance Computing, Constraint Solving, MPI, OpenMP, Pruning, Backtracking, Speedup, Efficiency, Scalability

1 INTRODUCTION

Hitori is a popular Japanese logic puzzle that first appeared in Puzzle Communication Nikoli in March 1990 [5]. It is typically played on an $N \times N$ numbered grid, where the objective is to mark some of the cells such that no row or column contains duplicated unshaded numbers. In addition, the shaded cells cannot be adjacent to each other and the unshaded cells must form one large connected area. Problems like Hitori belong to the class of **NP-Complete problems**, which require a vast exploration of different solution spaces to correctly get to plausible configurations that satisfy all the constraints. As the size of the input increases, the complexity of solving this kind of problems grows exponentially, making it computationally challenging to find a solution within a short timeframe. Specifically, Hitori becomes increasingly difficult to solve as the grid size grows, with the number of possible configurations expanding, but with still only one solution that actually works and satisfies all the rules.

Several algorithmic approaches have already been proposed for solving the Hitori puzzle, including techniques based on **brute-force** search, **backtracking** and **constraint propagation**. While all these methods possibly guarantee a correct solution given enough time and resources, their computational complexity becomes prohibitive for large grids, as the complexity grows exponentially with the grid size.

By the early 2000s, parallel algorithms emerged as a viable solution for tackling computationally expensive problems like Hitori. In fact, parallelization allows for the distribution of tasks across multiple processors, overcoming the limitations imposed by the sequential algorithms while offering more efficiency and scalability.

This paper explores **parallel strategies** for solving Hitori, focusing on the strengths of the following approaches:

- **MPI**-based approach, as it allows the workload distribution across different processes in multiple nodes of a cluster, thus increasing the overall scalability and bandwidth
- **OpenMP**-based approach, as it leverages multithreading and shared memory within a single node to boost the overall computational performance, prioritizing execution speed at the cost of reduced scalability
- **Hybrid**-based approach, as it integrates both MPI and OpenMP to achieve a balance between scalability and computational efficiency, optimizing resource utilization across both cluster nodes and process threads.

Despite parallelization offers significant benefits, it also comes with its own challenges and drawbacks, particularly in terms of increased code complexity, synchronization overhead and resource management. To address these issues, several solution refinements were made during the development, gradually adjusting towards the final solution. To proper evaluate the correctness and the performances of the proposed algorithm, key metrics such as execution time, speedup, efficiency and scalability factors were taken into consideration and carefully analyzed.

This study aims to present a comprehensive evaluation of a possible parallel Hitori solver algorithm. By demonstrating effective parallelization strategies, the goal is to reduce computational times and improve the overall scalability and efficiency of solving Hitori puzzles, offering valuable insights for similar constraint-based problems.

2 RELATED WORK

NP-complete problems, including Hitori, have been widely studied in the field, with the goal of developing efficient algorithms and techniques to solve these computationally complex problems more effectively. In this section, key literature in this area will be reviewed and analyzed, highlighting significant approaches, advancements and challenges specifically related to solving the Hitori puzzle.

One of the earliest approaches for solving the Hitori involves formulating the puzzle as a Constraint Satisfaction Problem (CSP) or translating it into a Boolean Satisfiability (SAT) problem. Gander and Hofer (2006) [4] explored both ways. In particular, they presented a dedicated algorithm to treat the Hitori as a CSP, which is then converted into a SAT instance that is finally resolved using a SAT solver like MiniSat. This work demonstrated the effectiveness and versatility of constraint programming and SAT solvers to solve combinatorial problems like Hitori. Although very efficient, this approach has limitations when dealing with larger grid sizes. In fact, the encoding process from CSP to SAT can introduce additional complexity and can lead to poor performance of SAT solvers if the generated clauses aren't well structured. Furthermore, this approach is not well-suited for scalability, especially for large-scale problems, highlighting the need for alternative strategies.

Other works, instead, focused on rule-based systems and logical deduction. These approaches, as stated by the work of Yen et al. (2009) [8], leveraged pattern matching and local search to efficiently prune the search space, facilitating the solution exploration and decreasing the computational complexity by a lot. Yet et al. demonstrated that the effectiveness of their approach by comparing its performance against other efficient methods, such as SAT solvers, showcasing significant speed improvements. Their work highlights the importance of combining rule-based and pruning techniques alongside local search for efficient Hitori solving.

Recognizing the computational intensity of solving these large scale problems, researchers have also explored parallelization strategies. Yao and Hajratwala (2023) [6] investigated different parallelization paradigms, including divide-and-conquer and cube-and-conquer, to distribute the computational workload across multiple processors using Haskell. Despite the initial challenges caused by an unbalanced workload, which resulted in less-than-ideal speedups, the authors view this approach as a promising and viable solution for solving these types of problems.

Despite the progress made in developing efficient Hitori solvers, several challenges remain. Most existing methods either struggle with scalability on large grid sizes or require significant computational resources. Parallelization, while promising, introduces synchronization overhead and load-balancing issues, which can limit its effectiveness. Additionally, heuristic and rule-based approaches may not guarantee optimality, requiring a trade-off between accuracy and speed. However, hybrid approaches that combine both parallelization and heuristic optimization could offer a balanced solution, leveraging the computational power of parallel processing while maintaining the efficiency of heuristic pruning techniques.

3 PROBLEM INTRODUCTION

Hitori (*Hitori ni shite kure*; literally "leave me alone") is a logical puzzle from Japan, which is played on an $N \times N$ matrix filled with numbers. The only thing the player must do is to shade the right cells such that all the rules are satisfied. Trivially, larger boards means higher difficulty, as the number of possible configurations grows exponentially, leading to a significant increase in computational complexity. Solving larger Hitori grids, such as a 20×20 board, can be considered challenging, although a real veteran player can solve it in more or less 10 minutes. In computational terms, however, this problem becomes much more difficult to solve without the use of specialized algorithms or heuristics.

In this section, the fundamental rules of Hitori are introduced, followed by an explanation of the pruning techniques employed to reduce the overall combinatorial complexity of the puzzle.

3.1 Rules

Hitori is very easy to learn, thanks to its simple set of rules:

(1) No unshaded cells (white) appears in a row or column more than once.
(2) Shaded cells (black) do not touch each other vertically or horizontally.
(3) When completed, all unshaded cells create a single continuous area.

These three rules form the core of the game and ensure that the puzzle is both challenging and solvable. As already said before, the goal of the player is to strategically shade cells while following these constraints, carefully balancing the placement of black and white cells.

3.2 Solving techniques

As stated by Yen et al. (2009) [8], pattern matching techniques can be employed to significantly reduce the computational complexity by initially pruning the search spaces. To achieve this, the proposed solution heavily relies on some of the techniques proposed on the Menneske website [3], with a focus on those that work particularly well in parallel scenarios.

In other words, the selected techniques were specifically chosen for their applicability to rows and columns, as opposed to those requiring more complex manipulations of larger areas of the board. This approach not only ensures independent processing on different parts of the grid at the same time, but also guarantees huge reductions in the search space, making the overall solution more scalable and effective. By applying these heuristics early in the solving process, we can quickly eliminate invalid configurations and make substantial progress toward the solution. This is especially valuable for solving larger puzzles like a 20×20 grid, where a brute-force approach would otherwise be much slower.

Following, the selected set of rules is explained to ensure a clear understanding of their functioning.

Must-Paint cases. The following rules apply when a cell is marked as white or black:

- **Set White.** When a cell is marked as white, then all the other cells with the same number in the row or column can be painted as black.
- **Set Black.** When a black cell is marked, then all the cells around it must be white.

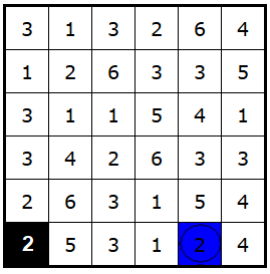


Figure 1: Set White

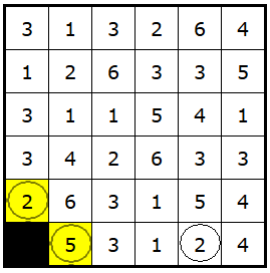


Figure 2: Set Black

Uniqueness case. If a number appears only once in its respective row and column, it must be marked as white.

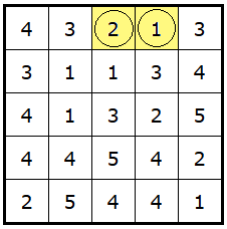


Figure 3: Uniqueness

Sandwich cases. The following rules apply when identical numbers "sandwich" another number:

- **Sandwich Pair.** If two similar numbers are placed on either side of a different number in a row or column, the middle number must be marked as white.
- **Sandwich Triple.** If three similar numbers are placed consecutively in a row or column, the two outermost numbers must be marked as black.

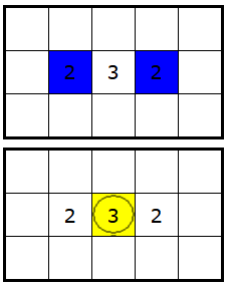


Figure 4: Sandwich Pair

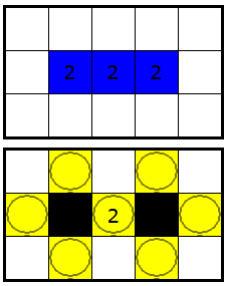


Figure 5: Sandwich Triple

Isolation cases. The following rules apply when certain numbers are isolated from a group within the same row or column:

- **Pair Isolation.** If two similar numbers are placed consecutively in a row or column and there is a third identical number, then the third can be marked as black.
- **Flanked Isolation.** If you find two doubles packed, singles within the same row or column must be black.

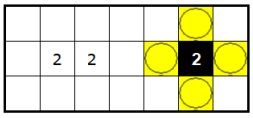


Figure 6: Pair Isolation

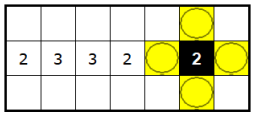


Figure 7: Flanked Isolation

Corner cases. The following rules apply when numbers form specific patterns in the corners of the grid:

- **Quad Corner.** If a corner has four similar numbers or two and two similar numbers, then the diagonal must be marked as black.
- **Triple Corner.** If a corner has three similar numbers, then the middle one must be marked as black. The same is true if the three numbers are set the other way.
- **Double Corner.** If a corner has two similar numbers, a white cell can be marked as white. The same is true for all combinations.
- **Corner Close.** If a corner has a cell marked black, then the other must be marked as white.

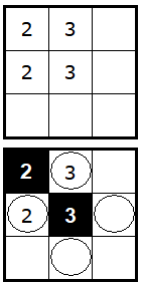


Figure 8: Quad Corner

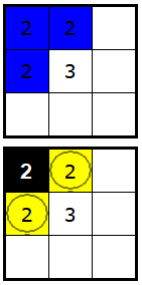


Figure 9: Triple Corner

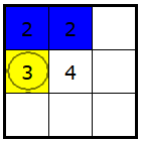


Figure 10: Double Corner

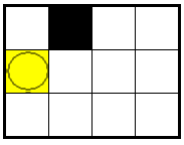


Figure 11: Corner Close

4 ALGORITHM ANALYSIS

Before moving on to the implementation aspect of the three different Hitori parallelization approaches, in this section there will be a general introduction on the **core concepts** behind the algorithm. Everything that is explained here is at the base of the Hitori puzzle solving, and is therefore valid for each individual implementation that will be explained later.

4.1 DATA TYPES DEFINITION

The Hitori board is loaded through an apposite IO method from an input file and then stored as an *Integer* matrix.
However, in addition to the Hitori grid of numbers, the algorithm must also manage the state of each cell. In particular, this means it needs to be able to set each cell value to *BLACK* or *WHITE*, depending if the cell needs to be shaded or not.

For this reason, the date type **CellValue** has been defined, consisting of an Enum of three values: *WHITE*, *BLACK* or *UNKNOWN*. While the meaning of the first two values is rather straightforward, the *UNKNOWN* value is used during the solving process to mark the cells whose value is not yet known. A matrix of cell values (which in the code we called **solution_matrix**) is then allocated and used during the resolution of the puzzle.

Another useful data type defined is what we call the *Board Control Block* (**BCB**), which basically is a Struct that is used to keep track of the current solution that is being analyzed. Each BCB contains the cell values (*WHITE*, *BLACK*, or *UNKNOWN*), as well as the matrix size and a couple of support matrices containing the information about the **unknown cells** present in the solution. These support matrices are useful to speed up the iteration of the matrix. More details about this structure can be found in the following *Solution Backtracking* section.

4.2 SOLUTION STRATEGY

We imagined the full number of solutions that have to be checked as a **BINARY TREE**, where each branch split is dictated by the cell value set as *WHITE* or *BLACK*. Each **leaf** of the tree will then correspond to a solution, which is a matrix full of *WHITE* or *BLACK* cell values that should be checked for **validity**. If the leaf is valid, it means that the matrix is the solution for the Hitori Puzzle.

During the computation of the solution, the task that requires the most resources is the one that needs to check for the validity of all the leafs. For this reason, reducing the number of wrong leaves, that for sure do not correspond to the solution, is essential to improve the computation time.
We call this process **Cells Pruning**, as the main objective is to "prune" the bad leaves of the tree in favor of the good leaves. With the number of *UNKNOWN* cells reduced drastically, a **Solution Backtracking** algorithm is then run to find the correct valid solution among the remaining leaves.

Thus, first the number of leaves is *pruned* by setting known cell values, and only then a *backtracking* algorithm is run. However, even after this improvement, we still wanted to increase the efficiency of the algorithm a step more.
For this reason, we introduced a sort of **Solution Space Fragmentation** before the main backtracking algorithm.

Algorithm 1: Main function

```
board ← read_board(input_path);
pruned_solution ← cells_pruning(board);
spaces ← fragmentation(pruned, SOLUTION_SPACES);
solution ← solution_backtracking(spaces);
write_solution(solution)
```

CELLS PRUNING

Due to the computational complexity involved in solving the challenging problem, improving the basic **Backtracking** technique is fundamentally necessary. This can be achieved by reducing and avoiding useless computations, the ones which we already know beforehand that do not lead to anything. In order to do this, different **Hitori solving techniques** have been leveraged and applied to the initial grid:

- *Uniqueness Rule*
- *Sandwich Rule*
- *Pair Isolation Rule*
- *Flanked Isolation Rule*

Additionally, the four corners of the grid are analyzed and checked for some corner-specific rules.
On top of all that, the final resulting matrix is then fed as input to a method that marks:

- A cell to white if a connected cell is black
- A cell to black if a cell with the same number that is located in the same row or column is white

We called all these operations **Cells Pruning**, as the total tree of solutions is pruned in favor of the most important cells.

Algorithm 2: Cells Pruning

```
solution ← read_board(input);
techniques ← [...];
for technique in techniques do
    solution ← apply_technique(technique, solution);
```

SOLUTION SPACE FRAGMENTATION

We noticed that the process of finding the solution changed a lot depending on the **index** of the solution leaf. This index basically identifies the amount of leaves that needs to be checked for validity before the current leaf is checked.

The backtracking process, by definition, starts setting cell values from the matrix beginning, but, whenever a non-valid cell is found, the last updated cell value will be changed. This means that the actual computation goes bottom-up. If the algorithm starts analyzing

from the top-left corner of the matrix and ends to the bottom-right corner, the value that is set for the first encountered cells influences a lot the total computation time.
For example, if the algorithm starts by setting the first *UNKNOWN* cell at position (1,1) to *WHITE*, but instead the solution requires it to be black, all the remaining cells are **checked for nothing**. And the value of the first *BLACK* cell will be changed to *WHITE* only after the remaining unknowns are changed in all possible ways.

While this computation cannot be avoided completely, as we do not know it beforehand as well as the pruned bad leaves, this process can still be improved by trying to **normalize** the computation time.
This normalization process becomes mostly useful when code parallelization is performed, however it is still beneficial even without multi-processing. By standardizing the mean amount of time that the algorithm takes to process a generic solution, it makes the computation **less probabilistic** and **more controllable**.

SOLUTION SPACES GENERATION. In multi-processing, basically, the **fragmentation** takes place by making the different processes start by different **leaf indexes** and therefore creating different **Solution Spaces**.
Each solution space will have different cell values as the first unknown cells. For example, the division in two solution spaces is done by setting the first unknown cell to *WHITE* for the first solution space and to *BLACK* for the second solution space.
These cell values, set during solution space generation, are made to be immutable so that the solution spaces do not overlap between each other. During the **Solution Backtracking** computation, then, only the remaining cell values are modified to look for the solution.

In a single process, instead, the operation becomes a bit more difficult. To achieve the same result of making the leaves computation in alternating order, a **queue** must be employed.
In particular, the queue should allow for keeping track of the leaf that was last computed for multiple solution spaces. As explained better later, this is carried out by defining a **Queue of BCB**, where each BCB corresponds to a different solution space.

While *SOLUTION_SPACES* could be implicitly declared by simply taking the number of processes that are available, this will make the algorithm more **probabilistic** depending on the number of processes working on it. This also allows the program to manage multiple solution spaces even when running in serial mode.
For such reasons, *SOLUTION_SPACES* has been set as a **parameter** to the algorithm and kept constant during the analysis of the computation time of a single test.

Algorithm [3] shows how the initial cell values are assigned depending on a certain *SOLUTION_SPACE_ID*.
"*solution_space_unknowns*" is a support matrix that keeps track of which cells have been selected during solution space definition, so that they can not be changed later on during backtracking. This support matrix is saved inside the **BCB** block as well as the current solution being checked.
"**is_cell_valid**" is a simple function that checks whether the cell is valid for its row and column, depending if its cell value is either *WHITE* or *BLACK*.

Algorithm 3: Solution space initialization for a certain SOLUTION_SPACE_ID

```
1  ssid ← SOLUTION_SPACE_ID;
2  spaces ← SOLUTION_SPACES − 1;
3  for i=0; i<rows; i++ do
4      for j=0; j<cols; j++ do
5          cell_value ← solution[i][j];
6          if cell_value is not UNKNOWN then
7              continue
8          cell_choice ← ssid%2;
9          if not is_cell_valid(i, j, cell_choice, solution) then
10             cell_choice ← abs(cell_choice − 1);
11             if not is_cell_valid(i, j, cell_choice, solution)
                   then
12                 continue
13         solution[i, j] ← cell_choice;
14         solution_space_unknowns[i, j] ← true;
15         if ssid > 0 then
16             ssid ← ssid/2
17         if spaces > 0 then
18             spaces ← spaces/2
19         if spaces == 0 then
20             break
21     if spaces == 0 then
22         break
```

SOLUTION BACKTRACKING

The backtracking process is the core part of the algorithm. It is executed after an initial pruning of the input board and the fragmentation into multiple solution spaces.
It is tasked with:

- **Managing multiple solution spaces**
- **Exploring each solution space**
- **Checking the solutions validity**

In order to keep track of multiple solution spaces, a special **Queue of BCB** has been implemented. Through the continuous operation of *enqueue* and *dequeue* of the same blocks in the queue(belonging to different solution spaces), it is possible to manage multiple solution spaces even for a single process. For each block, then, the algorithm first needs to find the next valid leaf and after that it checks whether that leaf is the solution for the Hitori puzzle.

The backtracking process has been fundamentally divided into three core functions:

- **build_leaf**: responsible for exploring the Hitori board and setting *UNKNOWN* cells to *WHITE* or BLACK.

- **next_leaf**: tasked with finding the next leaf, corresponding to a possible solution, starting from the last analyzed one.
- **check_hitori**: through a dfs exploration of the solution matrix, determines if it is the solution of the Hitori or not.

BUILD LEAF. It takes as input a solution grid with cell values that could be *WHITE*, *BLACK* or *UNKNOWN* and has the objective to fill all the cells such that no *UNKNOWN* cell is present. It needs to do so in a way such that the resulting solution matrix will contain a **possible** solution, meaning that all *BLACK* and *WHITE* cells are valid with respect to their row and column.
This recursive function should either return **true** if a leaf has been built correctly or not. A positive result meaning that the leaf could be later checked for *validity* with the **check_hitori** function. A negative result, instead, defines that no leaf could be built with the passed cell values in the solution matrix.
The *build_leaf* function advances from *top-left* to *bottom-right* and sets the cell value:

- to *WHITE*: if the cell was *UNKWNOWN*
- to *BLACK*: if *WHITE* is not valid for the cell
- to *UNKNOWN*: if neither *BLACK* nor *WHITE* are valid values. In this case, a negative value is returned to notify that a cell value in the previous positions should be changed.

A special attention is put with regards to the **Solution Space Unknowns**, which are those cells that have been defined during **Solution Space Fragmentation** and which mark the boundaries of each solution space. Those cells, in fact, should not be modified and can only be taken as they are. In addition, the cells that have been marked as wither *WHITE* or *BLACK* during the **Cells Pruning** phase are of course left as is.

Algorithm 4: build_leaf(BCB, x, y)

```
1  x, y ← get_next_unknown_cell(x, y);
2  if reached last cell then
3      return true
4  cell_state ← BCB[x, y];
5  is_solution_space_unknown ←
     BCB.solution_space_unknowns[x, y];
6  if is_solution_space_unknown is false then
7      if cell_state is UNKNOWN then
8          cell_state ← WHITE;
9  for i=0; i<2; i++ do
10     if is_cell_valid(BCB, x, y, cell_state) then
11         BCB.solution[x, y] ← cell_state;
12         if build_leaf(BCB, x, y then
13             return true
14         if is_solution_space_unknown then
15             break
16     cell_state ← BLACK;
17 if not is_solution_space_unknown then
18     BCB.solution[x, y] ← UNKNOWN;
19 return false
```

NEXT LEAF. In opposition to the **build_leaf** function, the **next_leaf** function has the objective to find the next valid leaf by taking as input the last computed leaf of the solution space tree. In fact, the solution matrix passed as input should only contain *WHITE* or *BLACK* cell values.
Starting from the bottom-right corner of the matrix and going top-left, the functions tries to change the cell value:

- to *BLACK*: if it was *WHITE*. It then tries to build a leaf with **build_leaf** with this configuration.
- to *UNKNOWN*: if *BLACK* is not valid. It then goes to the previous cell and leaves the future **build_leaf** function to fill the *UNKNOWN* cells.

No recursion is needed as the top-down exploration is done by calling the **build_leaf** method to fill the gaps.

Algorithm 5: next_leaf(BCB)

```
1  for i=rows-1; i>=0; i- - do
2      for j=cols-1; j>=0; j- - do
3          if cell at [i,j] is pruned then
4              continue
5          cell_state ← BCB[i, j];
6          if BCB.solution_space_unknowns[i, j] is true then
7              return false
8          if cell_state is WHITE then
9              if is_cell_valid(BCB, i, j, BLACK) then
10                 BCB.solution[i, j] ← BLACK;
11                 if build_leaf(BCB, i, j+1) then
12                     return true
13         BCB.solution[i, j] ← UNKNOWN
14 return false
```

CHECK HITORI. Similarly to the **next_leaf** function, **check_hitori** takes as input a full solution matrix with no *UNKNOWN* cell values. However, its task is to scan the matrix to verify if the defined configuration is the solution for the *Hitori* puzzle.
Cell validity, with respect to row and column, is already checked during the leaf building phase. This means that the only thing remaining to be verified is the **full connection of the white cells**. As this algorithm is trivial and could be easily implemented with a recursive DFS or BFS function, the pseudo code for it is not shown.

To put it all together, a solution queue is used to fetch the current block (**BCB**) to analyze, and then the **next_leaf** function is called on that cell. If a valid leaf has been found, than the **check_hitori** function can be called with the last found solution matrix.
If the result is positive, then it means the algorithm found the solution, otherwise it enqueues again the block in the queue and continues to the next loop.
Algorithm [6] shows the pseudo code of the iteration for the **Solution Backtracking** computation.

Algorithm 6: solution_backtracking(queue)

```
while queue is not empty do
    block ← dequeue(queue);
    new_leaf ← next_leaf(block);
    if check_hitori(new_leaf) then
        return new_leaf
return false
```

To summarize, the strategy employed for solving the Hitori problem involves:

- **Cells Pruning**: Initially, the grid matrix is analyzed and basic Hitori solving techniques are applied. This lets us **drastically reduce** the number of unknown cells to approximately 10% of the total cells.
- **Solution Space Fragmentation**: After the initial grid has been simplified, the next step involves the *fragmentation* of all the possible solutions to be analyzed in different groups of about equal size.
- **Solution Backtracking**: In alternating order, each *Solution Space* is then iterated over and every possible solution is checked for validity.

5 IMPLEMENTATION

The implementation of the algorithm has been carried out using the **C Language**, leveraging its parallelization capabilities offered by the **MPI** and **OpenMP** libraries.

Initially, the first version of the application was developed using *MPI* [2] to provide multi-processing. In this implementation, due to the fact that each process has **independent memory access**, a lot of focus was put in managing data sharing and message passing between processes.
In the second phase, the *OpenMP* [1] library has been used to create a second version of the application where the parallelization capabilities depend on the number of threads allocated. In contrast to multi-processing, threads share the same memory allocation, therefore careful considerations had to be made regarding **critical** portions of the code and **data access**.
The final **hybrid** version of the application was developed putting the two approaches together, trying to get the best out of the two libraries. The complexity in some aspects of the individual versions were adjusted and some methods were changed in favor of a more **solid implementation**.

5.1 Data generation

One of the main challenges in our study was the lack of large Hitori puzzle boards available online. Even Menneske [3], which hosts one of the largest puzzle databases, only provides grids up to 20×20, with the highest difficulty level described as *pretty easy* or *super easy* according to the website. This limitation created the necessity for a dedicated puzzle generator. However, generating valid Hitori puzzles is a highly complex task, as every puzzle only have just one correct solution.

During our research, we found only one publicly available generator, developed by Schmoller [7], but it proved to be somewhat unreliable as it relies heavily on random shading and other similar heuristics. Due to these constraints, we were unable to conduct extensive testing on larger boards, especially since developing a new generator from scratch would require significant additional time. Nevertheless, the results obtained still demonstrate the effectiveness of our algorithm.

5.2 Parallelization capabilities

As detailed in the algorithmic analysis section [4], the algorithm was designed keeping in mind the parallelization capabilities of the resulting application. In particular, both **Cells Pruning** and **Solution Backtracking** are made to be parallelizable and leverage the multi-processing computation power.
In addition, thanks to the **fragmentation** in multiple solution spaces, each process is able to perform backtracking computations by avoiding to interfere with other processes and only small communications are required.

However, while both **Cells Pruning** and **Solution Backtracking** have been developed parallelizable, the one that influences the most the outcome is naturally the second one. Of course, this happens because in algorithms of such nature the computation time rises exponentially depending on the input. On the other hand, the pruning phase takes much less time to compute and is more deterministic. In fact, the amount of time the pruning takes to perform the computation is almost **negligible**.

More details can be found in the specific approach sections, but in general the **backtracking** parallelization was made such that the amount of communication needed to perform the computation was minimal. Thanks to the definition of the solution space boundaries, "simply" sharing the **BCB** blocks to compute is enough to make the processes start working in parallel.

5.3 MPI-based approach

In algorithmic analysis [4] is explained why the number of solution spaces has to be defined as a parameter (*SOLUTION_SPACES*) in order to make the solving of the Hitori puzzle not probabilistic. However, this means it is important to assign each worker process to one or more solution spaces.
In particular:

- If the number of processes is equal to the number of solution spaces, then trivially every process takes one solution space
- If the number of processes is less than the number of solution spaces, then each process takes the solution spaces such that $process\_rank = solution\_space\_id \% num\_processes$
- If the number of processes is bigger than the number of solution spaces, there is a need to implement a **solution space sharing technique** (Better explained in the following dedicated section [5.3])

PRUNING PARALLELIZATION. As previously defined, the first stage of the algorithm is the **Cells Pruning** process. While this phase takes much less time than the **Solution Backtracking** part, it was still implemented in a parallel way.

At this stage, a dedicated **MPI communicator**, referred to as (*PRUNING_COMM*), has been introduced. This is a core aspect in the MPI framework, as it defines a group of processes capable of communicating together. Given the high scalability of this multi-process approach, the number of initial processes can be significantly large. While this can be beneficial for handling very large inputs, it can also introduce significant performance bottlenecks. Since splitting the workload across many processes introduces significant message passing and synchronization overhead, sometimes exceeding the execution time of a serial approach, the new communicator was designed to **limit** the number of workers based on a predefined constant (*PRUNING_WORKERS*).

By leveraging the *MPI* library, during each pruning technique, the rows and columns of the board have been divided and assigned to different processes using the *MPI_Scatterv* method. After all processes compute their rows and columns, the results are then retrieved by the *MANAGER_PROCESS* using the *MPI_Gatherv* and combined together. This process is iteratively repeated until all pruning techniques are applied.

Solution Space Sharing

The *SOLUTION_SPACES* parameter has been defined as a fixed value in the algorithm. However, this requires the algorithm to handle cases where the number of processes differs from the amount of solution spaces. In particular, when there are more processes than solution spaces, the workers need to **coordinate** and share the computation for a single solution space. To achieve this, the serial implementation of **Solution Backtracking** requires some adjustments to support this form of synchronization. The adopted approach introduces two additional variables:

- *processes_in_solution_space*: Defines the number of processes sharing the solution space. It is decided based on the actual processes allocated.
- *solutions_to_skip*: This variable is used to mark how many solution leaves a process should skip before calling *check_hitori*. Its value should decrease to 0 at every leaf encountered and then be set equal to $processes\_in\_solution\_space - 1$

Through this simple computation, even when multiple processes are working on the same solution space and are exploring the same solution space tree, they **do not interfere** with each other.

This technique only parallelizes the computation time to validate the leaves when calling the **check_hitori** functions. In fact, the algorithm does nothing to parallelize the time spent to explore the solution tree for sharing processes. However, considering the amount of time spent for exploration is much lower than the time spent to validate a leaf, we consider it an acceptable compromise.

Algorithm [7] shows the edited portion of the function. "*skips*" correspond to the solutions to skip and is passed by reference, as it needs to be updated inside the recursive function. "*processes*", instead, is the total number of processes sharing this solution space. This value is used to reset *skips* whenever it reaches 0 and a leaf is checked for validity.

Algorithm 7: build_leaf_mpi(BCB, x, y, processes, *skips)

```
1 x, y ← get_next_unknown_cell(x, y);
2 if reached last cell then
3     *skips = *skips − 1;
4     if *skips is -1 then
5         *skips = processes − 1;
6         return true;
7     return false;
8 ...The remaining part of the function stays the same
```

A small adjustment is needed for the **next_leaf** function too. However, as it trivially just needs to accept the two new parameters and pass them to the **build_leaf** function, its updated pseudo-code is not reported.

Message Passing Architecture

The updated backtracking algorithm allows for an effective management of all the available processes computation power, as each process is assigned to a shareable solution space. However, the core algorithm shown in the algorithm analysis section [4] does not include details about how processes can share the solution spaces among each other.
For this reason, a **Message Passing Architecture** has to be defined in order to leverage the inherent parallelization capabilities of the code.

MASTER-WORKER RELATIONSHIP. The **MPI** library offers apposite *API* functions to make the processes communicate with each other. But, in order to effectively control the processes, the architectural nature of the message passing system must be taken in considerations. While initially there has been an attempt to let all the processes communicate with each other almost freely, that came out as a failure.
The complexity of managing messages for all the processes increased exponentially until it went out of control. For this reason, we opted for a more centralized approach, were a **MASTER** process has the task to intermediate between the remaining **WORKERS**. Basically forming a **Star Topology**, each *WORKER* can send a message to the *MASTER* by creating an apposite **Message** Struct. This data type contains:

- *message_type*: An *Enum* value defining how to treat the request and what the data signifies.
- *data1* and *data2*: Two integer variables that each request can use to pass information.
- *invalid*: A boolean flag dictating if the request should be marked as invalid. This is useful during block(**BCB**) sharing.

While these messages are mainly used to allow *WORKERS* to communicate with the *MASTER* and vice versa, there are also some exchanges happening between *WORKERS* directly.
Without going into much details, the most important messages exchanged are:

(1) **Termination**: when a process found a solution and needs to notify the other processes to terminate. A *TERMINATE* message is first sent to the *MASTER* which in turn sends it to all the other *WORKERS*.
(2) **Status Update**: to keep the *MASTER* updated on the status of the processes, when a *WORKER* ends a solution space, it sends a message to the *MASTER*, so that it can effectively decide how to manage the workload.
(3) **Ask For Work**: when a process ends its solution spaces, it requests the *MASTER* for more work, which contacts the most fit *WORKER* and tells it to share its solution space. Afterwards, the two workers communicate together to synchronize the **BCB** and the parameters for skipping the solutions.

Within the *MPI* implementation, the **Message** Struct has been committed as a *MPI_Type* so to facilitate the data being exchanged. However, when WORKERS needs to exchange the current block to work on, a stream of *Integers* is used due to the fact that no MPI_Type could be created from a dynamic Struct.
Additionally, a lot of time was spent in the definition of *MPI_Request* data types, because the *MPI* library has specific limitations on their definition and usage. Without proper management, it leads to **uncontrollable errors**, in particular when dealing with the *MPI_Test* function to check whether messages have been received or not. Due to this reason, different channels for communication have been implemented, such as:

- **M2W_MESSAGE**: Master to worker messages
- **W2M_MESSAGE**: Worker to master messages
- **W2W_MESSAGE**: Worker to worker messages

In order to not lose computational power, the *MASTER* process behaves also as a *WORKER*, thus it is able to send and receive messages to itself but in a different role. Without carefully setting the proper channel, messages would overlap with each other leading to unpredictable and uncontrollable states.

PROCESSES INITIALIZATION. With this type of message passing architecture, it becomes almost straightforward the **initialization** of the different processes. In particular:

- If $rank < SOLUTION\_SPACES$ then the process takes solution spaces with $rank = solution\_space\_id \,\%\, processes\_size$
- Otherwise, the process sends an *ASK_FOR_WORK* message to the *MASTER* so that it instructs a selected *WORKER* to **share** part of its solution space.

5.4 OpenMP-based Approach

As for the MPI implementation, the OpenMP approach also follows the same base algorithm [4] described before. However, since OpenMP operates as a single process in a shared-memory environment, it doesn't require the management of inter-process communication. Instead, it utilizes multithreading and shared-memory communication. This can be seen as an advantage over MPI due to the lack of communication overhead and the potential for increased computational speeds, but it also brings potential drawbacks, such as:

- **Limited scalability**, as the application needs to be run on a single node, sacrificing its potential scaling factor
- **Resource contention**, as concurrent access to shared resources may lead to performance bottlenecks
- **False sharing**, as threads require constant cache synchronization

False Sharing Issue

This problem tipically arises when multiple threads inadvertently compete for access to the same cache line. In fact, modern CPUs fetch and store data in cache lines instead of individual variables. Therefore, it causes:

- Frequent **cache invalidations** due to unnecessary synchronization between the cores
- Increased memory latency and reduced performance

To address this issue, several solutions are available, such as using private copies of variables whenever possible. However, fully avoiding shared variables is quite challenging. For this reason, we instead opted for a **task-based approach**, where each task has almost all the necessary information to perform its part of the computation, thus reducing shared data access by distributing the workload across independent tasks.

OpenMP Tasks

An *OpenMP* task is a unit of work that can be executed **asynchronously** by a thread within a parallel region of the code. With this approach, rather than distributing shared variables across threads, the workload is divided in a way that:

- it ensures efficient distribution among workers
- it reduces the need for synchronization of shared variables
- it minimizes cache contention

Task Spawning. In the proposed solution, each task is spawned within a parallel region by a randomly chosen *MASTER* thread, which places it into a sort of task pool. These tasks are then taken by the first available *WORKER* thread and executed asynchronously.

Algorithm 8: task spawning

```
1 #pragma omp parallel
2   #pragma omp single
3     #pragma omp task firstprivate(...params)
4     task_function_to_execute(...params)
```

Task-Based Pruning and Backtracking. By leveraging *OpenMP* tasks, the *MPI* version of pruning and backtracking can be translated into a multithreading-based approach that minimizes critical access to shared variables, thereby reducing false sharing issues and memory contentions. In particular:

PRUNING. Instead of splitting the original board into sets of rows and columns among processes to compute one technique at a time in parallel, each pruning technique is now assigned to a different thread and executed as a task in parallel. This boosts a bit the performance by eliminating the overhead of inter-process communication.

BACKTRACKING.

- **Build solution space:** Instead of having each process calculating all the initial solution spaces, it is now divided into multiple tasks and assigned to different threads. As shown in algorithm [9], the found leaves are enqueued in a global block queue, which is shared among all the threads. This queue is later used by the master thread to split the workload among the other threads. As explained in section [13], this operation is a bit different from its *MPI* counterpart.
- **Find new solutions:** Similar to *MPI*, each thread has its own queue and can operate independently from the others, with the only need to access the shared memory to initially setup the queue and to set the termination flag when a solution is found. As shown in algorithm [10], initially each thread-assigned task initializes a local queue and fetches from the global queue its assigned blocks. By doing this small operation, we make sure each thread has an independent local queue which can use without interfering with the other threads.

Algorithm 9: task_build_leaf(solution_space_id)

```
1  block ← malloc(sizeof(BCB));
2  init_solution_space(&block, solution_space_id);
3  leaf_found ← next_leaf(&block);
4  if leaf_found then
5      solution_found ← check_hitori(&block);
6      if solution_found then
7          #pragma omp critical
8              terminated ← true;
9      else
10         #pragma omp critical
11             enqueue(&block_queue, &block);
```

Algorithm 10: task_find_solution(id,threads,skip,block)

```
1  local_queue ← initQueue();
2  foreach block in leaf_queues[id] do
3      #pragma omp critical
4          current ← dequeue(&leaf_queues[id]);
5          enqueue(&local_queue, &current);
6  while not terminated do
7      block ← dequeue(&local_queue);
8      leaf_found ← next_leaf(&block, threads, skip);
9      if solution_found then
10         #pragma omp critical
11             terminated ← true;
12     else
13         enqueue(&local_queue, &block);
```

WORKLOAD BALANCING. In *MPI*, whenever a process remained without a solution space to work on, it simply contacted the *MASTER* process with an *ASK_FOR_WORK* request. However, given that the *OpenMP* library does not support message passing, this method had to be changed.
In *OpenMP* implementation, the methodology for **solution space sharing** remains the same except for how the processes communicate with each other. When assigning a solution space to work on, in fact, a block (**BCB**) is allocated and assigned together with *solutions_to_skip* and *processes_in_solution_space*.
Instead of waiting for a process to remain without workload before assigning a solution space to share, in this specific implementation we decided it was better to **split the workload evenly** from the beginning. What this basically means is that, when creating the tasks to assign to each thread, the correct blocks with their related parameters (*solutions_to_skip* and *processes_in_solution_space*) have to be initialized properly. These blocks are enqueued in special queues by the *MASTER* thread and are later fetched by each thread individually in the task function.

5.5 Hybrid-based approach

After experimenting with the two individual approaches, using respectively only **multi-processing** and **multi-threading**, a combined approach has been implemented. With the need to merge the utilities offered by the *MPI* and the *OpenMP* library, a new version of the algorithm was developed by taking advantage of both their useful features.
However, considering the natural complexity of the libraries even when used alone, the development of the Hybrid approach has been more difficult than initially predicted.

MPI CHALLENGES. The *MPI* library offers useful tools for the management of inter-process communication and message sharing. However, due to the intricate requirements of the library and how the C compiler manages the memory allocations, the implementation of the *MPI* approach took several **trial and errors** to fix unexpected problems.
These problems mostly rose from a bad management of how **Message** and *MPI_Request* data types were allocated across the application. For example, if a Message variable was deallocated or reused in the *WORKER* before the *MASTER* process could receive the message, then the value of the message was lost.
These problems have been mostly fixed for the first approach, but at the end of the day we decided to avoid these problems a priori by **reducing the amount of messages** exchanged between the various processes. This decision has been taken after carefully considering how threads could impact the management of such an architecture.

OPENMP CHALLENGES. The *OpenMP* library allows for an effective management of inter-process communication and shared memory between different threads. However, the algorithm has been implemented in such a way that it allows threads to have a minimum amount of communication, without the need to constantly share data.

Other attempts have been carried out in testing, for example, how threads would behave when partially sharing the **check_hitori** computation (therefore sharing the dfs exploration to check if all white cells are connected), however they resulted in failure. This happened because the overhead introduced by such a sharing-intensive mechanism did not permit to efficiently scale up the application. In addition to that, the problem of **False Sharing** appeared and the spent computation time became completely random.

HYBRID DESIGN CHOICES

As shown later in the apposite evaluation section [6], both the approaches chosen for the *MPI* implementation and the one for the *OpenMP* implementation have shortcomings. After a certain increase in computation power, the scalability performance decreases substantially. Even though this phenomenon is natural for all parallel programs, there is still room for improvement.
After considering **pros and cons** of both the approaches, an hybrid version of the application has been implemented by mixing the functionalities from the two systems together.
The key features of the hybrid approach include:

- Drastic **reduction** in the number of inter-process messages
- **Extended Star Topology** message passing architecture
- Leveraging the **workload balancing** of the OpenMP approach

SIMPLIFIED MESSAGE TYPES. In the *MPI* implementation, several message types have been defined for information sharing. When reconsidering its design to decide the hybrid approach message passing methodology, it came out that the amount of types defined was excessive. With the further addition of multi-threading as well, this system could become uncontrollable.
For this reason, it was decided to reduce and simplify the types of messages exchanged between the *WORKER* and the *MASTER* processes.
As a result, **only the TERMINATION message** has been used.
Due to the improved workload balancing system (as detailed in the following section [5.5]) this system is able to effectively share the computation time without the need for *ASK_FOR_WORK* messages. Such a drastic reduction in message types allowed for more stable computation times, reducing the overhead each process had to manage. While this design choice has of course some drawbacks, we leave it to future refinements in case it becomes a useful feature to re-introduce.

EXTENDED STAR TOPOLOGY ARCHITECTURE. The best system found for the multi-process implementation was a *Star Topology Architecture*. When compared to the issues faced when no *MASTER* process was present, it was decided that such a resilient architecture **should be maintained**.
However, with the addition of multi-threading, it alone is not enough. Every process, in fact, has to spawn several threads but has also to be able to communicate with the remaining processes, having to manage their available threads too.
In order to keep **centralization** as a key feature of the system, we decided that an **Extended Star Topology** would be the best design choice.

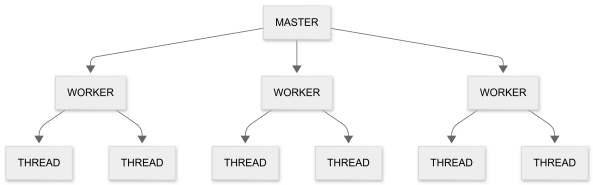


Figure 12: Extended Star Topology

In particular:

- Every *WORKER* process spawns several threads, however only a **MASTER_THREAD** can send messages.
- The threads can only communicate with their managing thread, which in turn exchanges messages with the *MASTER_PROCESS* if necessary.
- A *MASTER_PROCESS* remains in charge of intermediating between the different *WORKERS*.

The huge advantage that was introduced with the *OpenMP* implementation was its **workload balancing system**. In particular, each thread was allocated from the beginning with its copy of the solution space block (**BCB**) as well as its exploration parameters (for *solution space sharing*). While a system of this type was necessary for the multi-thread implementation, it performed so well that it was decided it could be implemented similarly in the Hybrid approach too. The performance scale up of the *OpenMP* implementation is, in fact, the most **stable**, and by reusing this system we aimed to make the Hybrid approach more stable as well.

Algorithm 11: Hybrid workload balancing

```
1  queue ← fragmentation(pruned);
2  for i=0; i<threads; i++ do
3      blocks_per_thread ← len(queue)/threads;
4      if len(queue)%threads > i then
5          blocks_per_thread++
6      threads_per_block = min(blocks_per_thread, 1);
7      threads_per_block ← threads/len(queue);
8      if threads%len(queue) > i then
9          threads_per_block++
10     threads_per_block = min(threads_per_block, 1);
11     for j=0; j<threads_per_block; j++ do
12         block ← dequeue(queue);
13         enqueue(leaf_queue[i], block);
14     skips ← i/len(blocks);
15     threads_sharing ← threads_per_block;
16     if SIZE > SOLUTION_SPACES then
17         threads_sharing ← STARTING_THREADS;
18         skips ← STARTING_SKIPS;
19     processes_in_solution_spaces[i%len(blocks)] ←
        threads_sharing;
20     #pragma omp task
        task_find_solution(i, threads_sharing, skips, blocks_per_thread);
```

With a little revisitation, the hybrid workload balancing is shown in algorithm [11]. *SOLUTION_SPACES* is the usual fixed parameter, **STARTING_THREADS** and **STARTING_SKIPS** are instead two values calculated during solution space initialization. For each process, they are defined based on the number of processes working on the same solution spaces, keeping in consideration the amount of threads each process has available.

HYBRID APPROACH CONSIDERATIONS. As shown afterwards in the evaluation section, the hybrid approach is the one with the best performances. Its features allows to maintain a **good scalability**, both horizontally and vertically.
Thanks to how the system has been devised, the application scales similarly even when in both cases, when increasing the number of processes or the number of threads. This is due to the fact that an **efficient workload balancing** algorithm has been developed.

Although good results have been achieved, the developed application has still room for improvements. For starters, the exchanging of inter-process messages could be restored in a similar fashion to how it has been implemented in the *MPI* version. The mechanism behind the **ASK_FOR_WORK** message is deemed to be useful even in the hybrid implementation, but due to timing and complexity constraints it has been left out.

In addition, the Hybrid approach has an hidden requirement of having the number of processes be < **SOLUTION_SPACES** * **2**. This requirement has been introduced to avoid over complicating the workload balancing system and break it. Considering both the number of threads and *SOLUTION_SPACES* can scale up as well, we think it should not become problem, but we leave it as a future improvement.

6 EXPERIMENTAL EVALUATION

This section will present various experiments that aim to showcase the efficacy of the proposed algorithm. The primary goal is to compare the results with the serial run of the algorithm to show how the speedup times and efficiency vary with varying parameters. In order to to that a baseline of the hardware on which all these tests have been done will be described.

6.1 Hardware

The system on which all the experiments are done is the HPC cluster offered by University of Trento. The cluster relies on the cluster managment software Altaris PBS Professional. The main specifications of the cluster are:

- ***Number of nodes***: 126
- ***CPU cores***: 6092
- ***RAM***: 53 TB
- ***Storage***: 15 TB
- ***Connectivity***: All nodes are interconnected with 10Gb/s network, some also have either a 40Gb/s Infiniband connectivity or even a 100Gb/s Omnipath connection.

6.2 Experimental Parameters Setup

During testing, one of the most critical parameters to select was the ***SOLUTION SPACES***. This value was empirically determined during the evaluation phase and was **set to 8**. This choice effectively minimizes the algorithm's probabilistic behavior, leading to more consistent and reliable results. Changing the amount of *SOLUTION SPACES* leads to different computation time. This is due to the probabilistic nature of the algorithm. In fact, changing the amount of *SOLUTION SPACES* makes each process perform its part of the computation in a different way.
Other important parameters to set up were the **number of threads** and the **number of processes**, respectively for OpenMP and MPI, which directly influence the parallel execution and overall performance of the algorithm.

The tests were made on a vast number of inputs, with varying size and complexity. Notably, bigger sizes **not always equals** to higher complexities, as the pruning techniques usually have more cells to work on, thus having the possibility reducing the number of unknowns to be processed later on.

6.3 PBS Directives

At the evaluation stage, job submissions were managed using Bash files, where PBS directives were employed to allocate efficiently the resources offered by the cluster. These directives define essential parameters for the job execution, such as the number of chunks and cores, memory requirements and CPU placement. A separate script was created for each approach, ensuring that the job executes as intended. Notably, one of the most important directive is **CPU placement**, as it instructs the scheduler on how to allocate the different chunks of resources.

6.3.1 MPI. Since this approach relies on the multi-process communication, we opted for a distributed execution strategy, using the **scatter:excl** placement strategy. With this setup, multiple chunks of resources are created and scattered across different nodes in the cluster. Each chunk is assigned exclusively to a separated node, thus diminishing the overall inter-process communication speed in favor of a higher bandwidth, which was one of the primary focus while thinking for a possible implementation of a hybrid approach.

```
#!/bin/bash
#PBS -l select=$PROCESSES:ncpus=1:mem=4gb
#PBS -l place=scatter:excl
#PBS -l walltime=0:02:00
#PBS -q short_cpuQ
#PBS -o ./output/
#PBS -e ./output/

cd $PBS_O_WORKDIR
module load mpich-3.2
mpirun.actual -n $PROCESSES ./build/main.out $INPUT
```

6.3.2 OpenMP. While *MPI* relies on message passing between processes, *OpenMP* works as a single process utilizing multiple threads in a shared-memory environment. Therefore, since it is not inherently scalable across multiple nodes, we opted to allocate

all resources within a single node, using the **pack:excl** placement strategy. By explicitly declaring the exclusivity clause, we ensure that no other jobs compete for memory within the same node.

```
#!/bin/bash
#PBS -l select=1:ncpus=$THREADS:mem=4gb
#PBS -l place=pack:excl
#PBS -l walltime=0:02:00
#PBS -q short_cpuQ
#PBS -o ./output/
#PBS -e ./output/

cd $PBS_O_WORKDIR
export OMP_NUM_THREADS = $THREADS
./build/main.out $INPUT
```

6.3.3 Hybrid. The hybrid approach combines the strengths of both *MPI* and *OpenMP*. For this reason, we decided to strike a balance between the number of chunks created and the number of processors dedicated to each chunk. By using a **scatter placement strategy**, we still distribute the chunks across multiple nodes, ensuring both scalability and fast shared-memory communication within same chunks. However, due to how the hybrid workload balancing has been designed, there is little difference in computation time between scaling vertically or horizontally. We consider this feature a strong point of the hybrid implementation, as it allows for a bigger increase of parallelization capabilities.

```
#!/bin/bash
#PBS -l select=$PROCESSES:ncpus=$THREADS:mem=4gb
#PBS -l place=scatter:excl
#PBS -l walltime=00:05:00
#PBS -q short_cpuQ
#PBS -o ./output/
#PBS -e ./output/

cd $PBS_O_WORKDIR
module load mpich-3.2
export OMP_NUM_THREADS = $THREADS
mpirun.actual -n $PROCESSES ./build/main.out $INPUT
```

6.4 Parallelization results

The following table shows the average execution time of a medium difficulty 20x20 hitori using between 1 and 64 threads, to show the Average Execution Time archived.

Average execution time			
Thread Count	OpenMP	MPI	Hybrid
1	2,628819	2,628819	2,626922
2	1,5080156	1,4515056	1,3092918
4	0,7718278	0,685069	0,6534918
8	0,4153622	0,3652946	0,3257788
16	0,3200178	0,3461716	0,1702484
32	0,2671876	0,1482134	0,0945632
64	0,2226826	0,1408404	0,0555868

As the number of threads increases, a reduction in execution time is observed. However, to quantify this improvement more precisely, three key metrics are introduced: Speedup, Efficiency and Scalability.

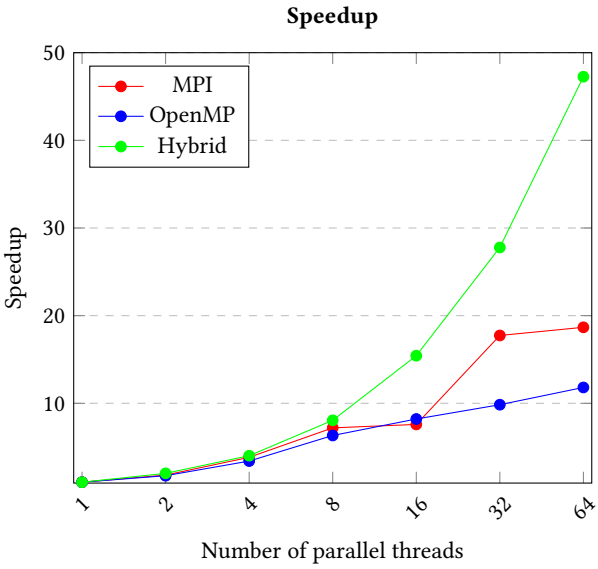
Speedup represents the ratio between the execution time of a sequential algorithm and that of its parallel counterpart. This metric indicates how much faster a parallel implementation performs as the number of processors increases. A higher Speedup value signifies better performance, demonstrating that parallelization effectively reduces execution time.

Efficiency measures how well computational resources are utilized by comparing Speedup to the number of processors used. Ideally, efficiency would reach 100%, but as the number of threads increases, overhead also grows, leading to an inevitable decline.

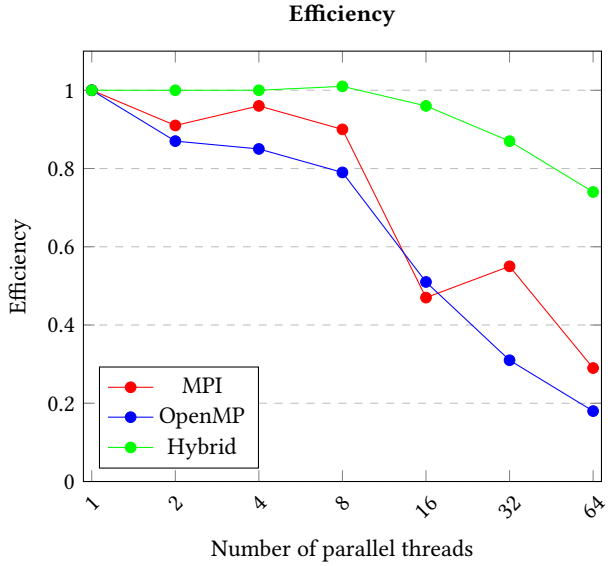
Scalability refers to the ability of a parallel algorithm to maintain efficiency as the number of processors increases. It evaluates how well the performance improves when additional computational resources are introduced. Scalability is typically classified into two categories: strong scalability and weak scalability.

- **Strong scalability** is achieved when a parallel algorithm maintains or improves performance while increasing the number of processors, provided that the problem size remains constant. In an ideally scalable system, doubling the number of processors should result in a near-halving of execution time. However, in practice, communication overhead and synchronization costs often limit this ideal behavior.
- **Weak scalability** assesses how well a parallel algorithm performs when both the number of processors and the problem size increase proportionally. A well-scaling system under weak scalability maintains a steady execution time as workload distribution grows. This form of scalability is particularly relevant for large-scale problems where computational demand increases with available processing power.

The following graphs illustrate the Speedup and Efficiency trends based on the previously presented data.



SPEEDUP. The results of the speedup study highlight the benefits of an optimal range of parallelism for the Hitori solver, balancing performance gains against communication and synchronization overheads. As shown in the speedup plot, the parallel solution exhibits an accelerating rate of speedup when increasing the number of processes from 1 to 32. Beyond this range, additional parallelism provides **diminishing returns**, as indicated by a stabilization or slight decrease in speedup. As it can be seen, the Hybrid implementation consistently demonstrates **superior speedup** compared to either the *MPI* or *OpenMP* methods



EFFICIENCY. The efficiency graph provides insight into how effectively computational resources are utilized across different levels of parallelism. Initially, with a lower number of processes (2, 4, or 8), efficiency remains relatively high. This suggests that in this range, additional computation contribute effectively to the computational workload, maintaining strong efficiency.

However, as the number of threads increases beyond 8, a noticeable **decline in efficiency** is observed. The reduction in efficiency at these levels suggests that the **overhead** introduced by communication, synchronization, and workload distribution outweighs the benefits of increased parallelism.

Among the different parallelization strategies, the Hybrid method demonstrates consistently higher efficiency compared to MPI and OpenMP alone. This suggests that the combined approach better mitigates some of the inefficiencies that arise from the individual approaches.

SCALABILITY. The efficiency graph provides insight into the scalability of the solution by illustrating how effectively computational resources are utilized as the number of computational power increases. At lower counts, efficiency remains close to or even exceeds the ideal level in the Hybrid method, demonstrating **strong scalability**. This indicates that additional processes or threads effectively reduce execution time with minimal overhead, maximizing parallel performance.

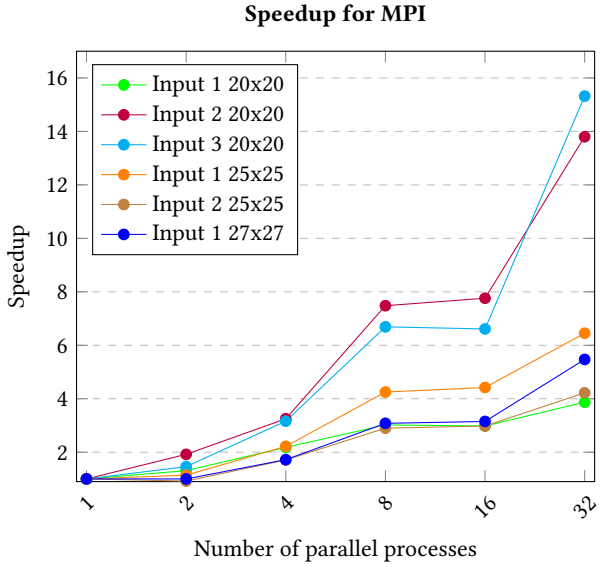
Regarding **weak scalability**, it is very difficult to scale the problem with the number of threads, as every problem can be much harder than another in the same size, and it is really difficult to find a good dataset with reliable data to utilize for this test.

6.5 Showcase of the approaches

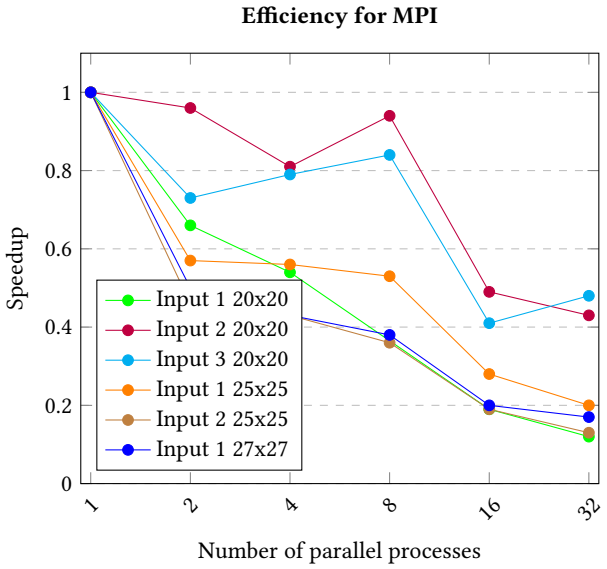
In this section more data of multiple inputs of varying difficulty will be shown for each algorithm, in order to show more specific metrics.

Average Execution Time in seconds (MPI)						
Processes	Input 1 20x20	Input 2 20x20	Input 3 20x20	Input 1 25x25	Input 2 25x25	Input 1 27x27
1	0,0609	21,058	2,632	2,043	0,0589	0,250
2	0,0464	10,945	1,806	1,798	0,0640	0,131
4	0,0279	6,4718	0,834	0,920	0,0344	0,0765
8	0,0201	2,8138	0,393	0,481	0,0203	0,0428
16	0,0204	2,7120	0,397	0,462	0,0198	0,0418
32	0,0157	1,5264	0,171	0,316	0,0139	0,0241

MPI The table above presents the execution times for both serial and parallel runs using MPI. It highlights the relationship between the number of processes used and the time required by the application to solve hitori puzzles of different sizes. While multiple execution times were evaluated during testing, only the total execution times are reported in the table. As the number of processes increases from 1 to 32, execution times consistently decrease, demonstrating the effectiveness of parallelization.



This graph shows how the MPI implementation performs on different test cases. While it can be seen how it significantly improves performance as the number of processes increases, it is not ideal. The results show no increase in performance in the switch from 8 to 16 processes, but the rest of the values show a great performance scale up. This is probably due to *SOLUTION SPACES* that was set to 8. This means that the workload balancing system works best with an equal number of *SOLUTION SPACES* and workers.

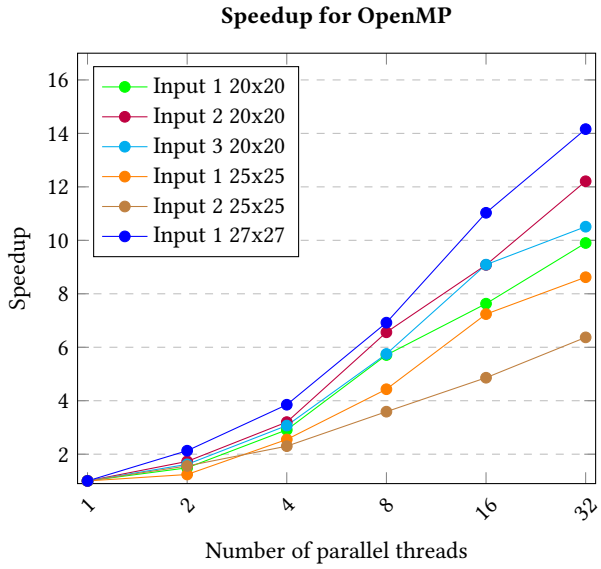


In a similar way to the speedup graph, the efficiency graph for the MPI test cases shows how multiple processes improve the performance of the algorithm. In particular it can be seen 8 processes is the best value, and that is because it is the same number as the *SOLUTION SPACES*, which allows better workload balancing and minimizes communication between processes.

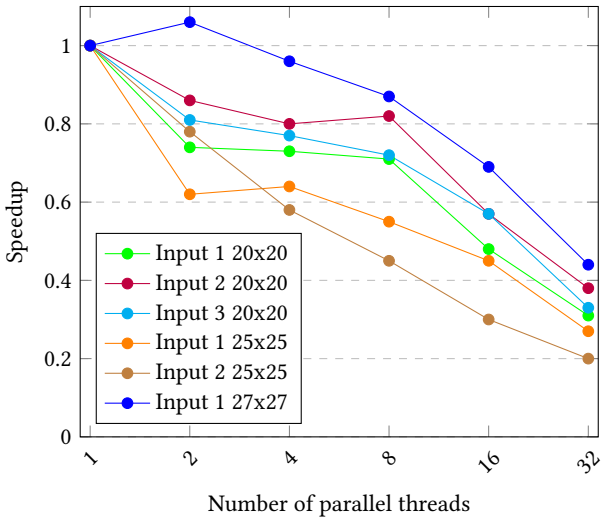
Average Execution Time in seconds (OpenMP)						
Thread	Input 1 20x20	Input 2 20x20	Input 3 20x20	Input 1 25x25	Input 2 25x25	Input 1 27x27
1	0,0609	21,058	2,632	2,043	0,0589	0,250
2	0,0409	12,180	1,626	1,652	0,0378	0,117
4	0,0209	6,585	0,857	0,801	0,0255	0,0651
8	0,0106	3,210	0,457	0,461	0,0163	0,0362
16	0,00798	2,318	0,289	0,282	0,0121	0,0227
32	0,00615	1,724	0,250	0,237	0,00925	0,0177

OpenMP The table above presents the execution times for both serial and parallel runs using OpenMP. Similarly to MPI, as the number of threads increases, execution times decrease, highlighting the benefits of leveraging shared-memory parallelism.

Always referring at **Input 2 (20×20)**, the serial execution time drops from 21 seconds to merely 1.7 seconds with the use of 32 threads. This significant reduction demonstrates the efficiency of OpenMP in distributing the workload across different threads.



As it can be seen in the speedup graph for the *OpenMP* test cases analysis, when compared to the *MPI* one, its shows how the performance improvement is **more stable**. In fact, the increasing in performance is more linear with respect to the less linear results of MPI. This is due to the fact that, without the need for processes to communicate, there is overall less overhead and the results are more deterministic.

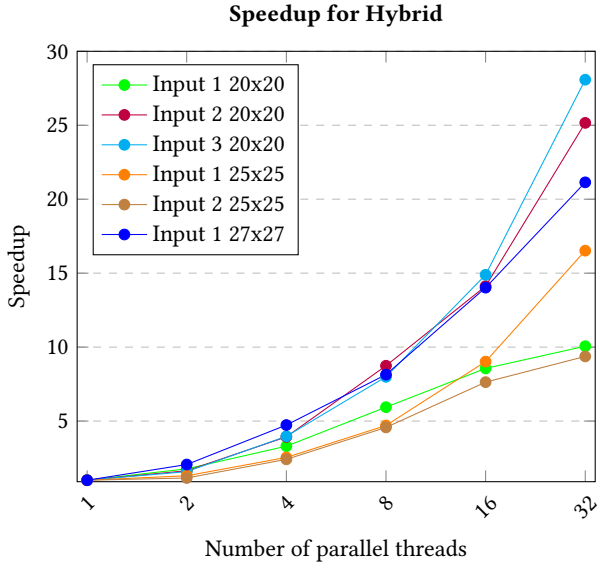


In a similar way to the speedup graph, the efficiency graph displays a better scalability of multi-threading. However, after a certain point, the overhead associated with thread synchronization and memory contention begins to outweigh the performance gains from additional parallelization.

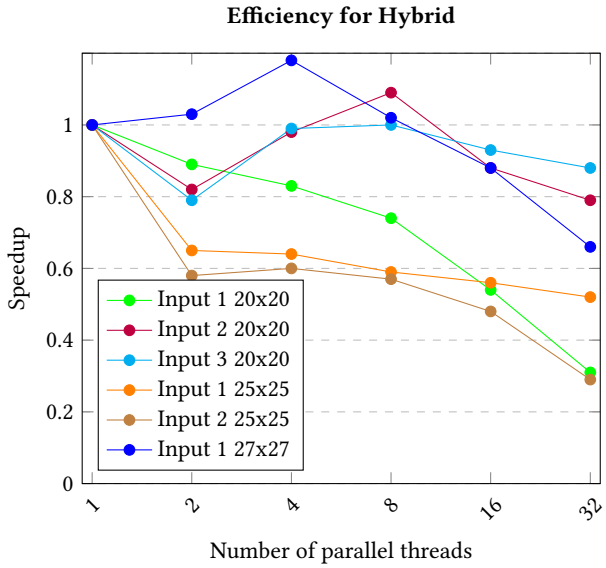
Average Execution Time in seconds (Hybrid)						
Processes X Threads	Input 1 20x20	Input 2 20x20	Input 3 20x20	Input 1 25x25	Input 2 25x25	Input 1 27x27
1	0,0609	21,058	2,632	2,043	0,0589	0,250
2	0,0344	12,805	1,665	1,576	0,0510	0,121
4	0,0184	5,360	0,663	0,804	0,0244	0,0529
8	0,0102	2,408	0,329	0,434	0,0128	0,0307
16	0,00711	1,491	0,176	0,226	0,00772	0,0178
32	0,00605	0,837	0,0937	0,123	0,00629	0,0118

Hybrid The execution time results for the hybrid approach, which combines MPI and OpenMP, demonstrate significant reductions in computation time as parallelism increases. By leveraging both inter-node (*MPI*) and intra-node (*OpenMP*) parallelism, the Hybrid approach ensures efficient workload distribution. This efficiency is maintained up to 32 parallel units, where execution times for all input cases reach their **minimum recorded values**.

One key observation is that the hybrid model scales particularly well for **larger and more computationally demanding inputs**. While inputs with smaller computational workloads (such as the simplier inputes) show diminishing reductions in execution time at higher levels of parallelism, larger inputs continue to experience substantial speedups. This suggests that the hybrid model mitigates some of the inefficiencies seen in pure *OpenMP* or *MPI* implementations, particularly for workloads with sufficient computational granularity to justify the added complexity of hybrid parallelization.



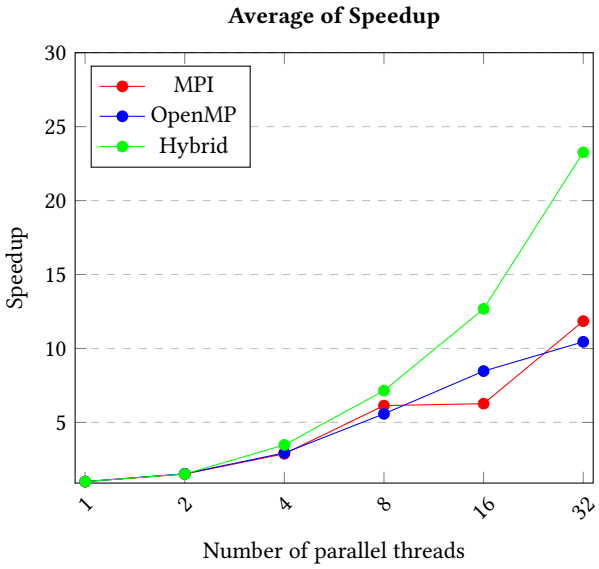
The speedup results for the hybrid approach demonstrate a strong improvement in performance as the number of parallel threads or processes increases. For larger inputs, speedup scales well, reaching values significantly higher than those observed in the MPI or OpenMP-only implementations. This suggests that the hybrid strategy effectively leverages both **inter-node** and **intra-node** parallelism.



The efficiency graph for the hybrid approach highlights how well computational resources are utilized as parallelism increases. For larger inputs, efficiency remains relatively high, even at higher processes count, indicating that the hybrid model **effectively balances workload distribution**. Some cases even show **superlinear** efficiency, likely due to how the workload is better balanced when the workers reach the amount of *SOLUTION SPACES*.

However, for smaller inputs, efficiency declines more rapidly as overheads start to outweigh the benefits of parallelization. This suggests that while the hybrid approach is powerful, it is most effective when applied to sufficiently large problem sizes where the computational workload justifies the additional complexity of managing both *MPI* and *OpenMP*.

6.6 Showcase of the approaches



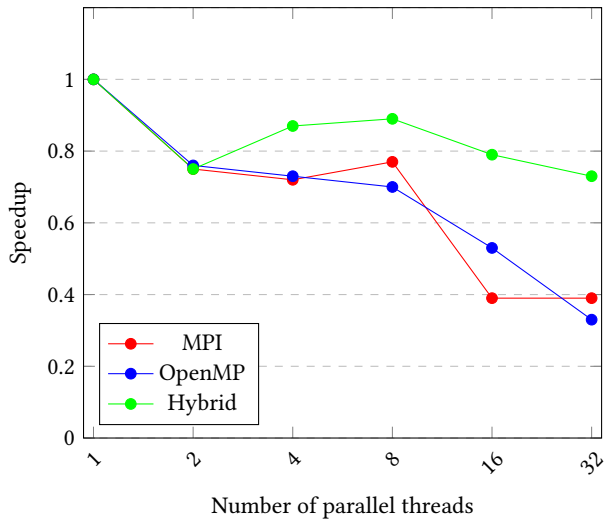
The Average Speedup graph provides a clear comparative analysis of how each approach scales with increasing parallel resources. Initially, all three methods exhibit similar speedup trends for lower thread counts, indicating comparable performance benefits from parallel execution at this stage. However, as the number of processes or threads increases, **distinct differences** emerge.

The **hybrid** approach demonstrates the most substantial speedup, particularly at higher levels of parallelism. Its performance continues to scale efficiently. This suggests that the hybrid strategy effectively leverages both shared and distributed memory paradigms, mitigating some of the limitations observed in the standalone *OpenMP* and *MPI* implementations. The more pronounced speedup gain indicates **improved workload distribution** and **reduced synchronization overhead**, making it the most scalable approach among the three.

MPI initially performs competitively but begins to show diminishing returns beyond 8 threads. This trend suggests that communication overhead between processes starts to counteract the benefits of increased parallelism, limiting further scalability. While MPI still achieves respectable speedup values, its growth rate slows down relative to the hybrid approach.

OpenMP follows a similar trajectory to MPI but demonstrates slightly better scaling at 16 threads, likely due to its lower inter-thread communication overhead.

Average of Efficiency



The efficiency graph highlights the hybrid approach's superior scalability, maintaining higher efficiency across all thread counts. While MPI stabilizes around four threads, its efficiency drops sharply at higher levels due to communication overhead. OpenMP, in contrast, declines more gradually, likely due to increasing thread synchronization costs. The hybrid method **balances these trade-offs**, outperforming the others at larger thread counts, though diminishing returns remain inevitable.

6.7 Limit testing Hybrid approach

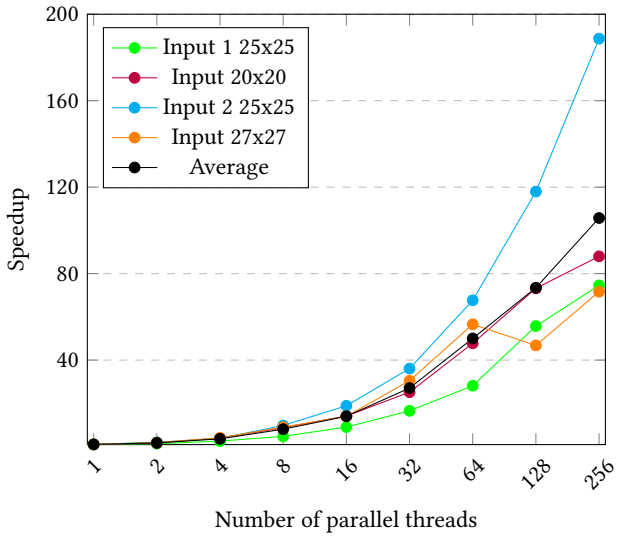
This section presents additional data using significantly more complex inputs to better illustrate the full potential of the Hybrid approach. By removing limitations imposed by scheduling inefficiencies and overhead present in smaller or simpler workloads, these tests provide a clearer picture of how the Hybrid method scales **under more demanding conditions**. The goal is to assess its performance in scenarios where computational resources are fully utilized, revealing its advantages in handling larger problem sizes and demonstrating its efficiency compared to purely *MPI* or *OpenMP* implementations.

Average Execution Time in seconds (Hybrid - hard)

Processes X Threads	Input 1 25x25	Input 20x20	Input 2 25x25	Input 27x27
1	2,0439405	21,058	5431,14	13318,9
2	1,576821	12,805	2743,11	7182,32
4	0,804294	5,360	1436,62	3308,71
8	0,434903	2,408	556,85	1485,03
16	0,226509	1,491	287,75	953,48
32	0,123703	0,837	150,55	437,39
64	0,072644	0,440901	80,27	235,5
128	0,036531	0,287701	46,05	116,12
256	0,027421	0,23932	28,78	75,8

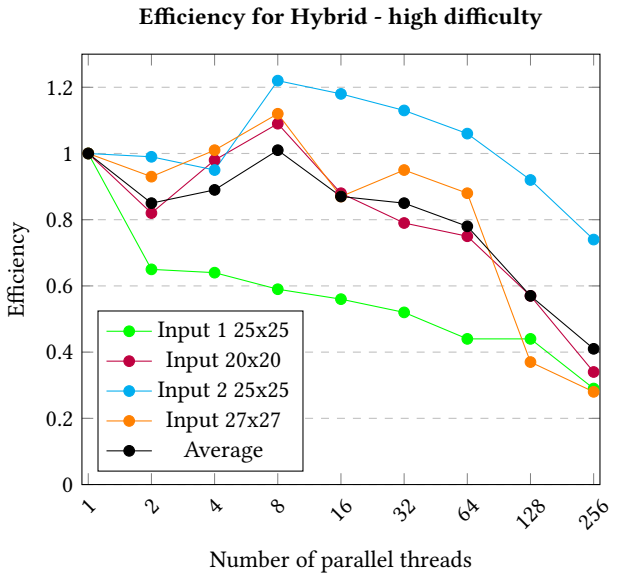
The execution time table clearly demonstrates the Hybrid approach's ability to scale effectively with increasing processes and threads counts. The most significant reductions occur at lower counts, with **diminishing returns** becoming evident at higher levels of parallelism, particularly beyond 128 threads.

Speedup for Hybrid - hard



The speedup graph highlights the substantial performance gains achieved with harder inputs. Unlike previous tests where scalability was hindered, the Hybrid method now achieves **remarkable speedup values**, especially for Input 2 (25x25), which reaches nearly 189x at 256 threads. This trend confirms that when given sufficiently large and computationally intensive workloads, the Hybrid

approach **maximizes resource utilization** more effectively than OpenMP or MPI alone. However, some variations among the inputs indicate that the method's efficiency depends on problem size and structure, with some workloads benefiting more from parallel execution than others.



The efficiency graph further reinforces these observations. At lower thread counts, efficiency remains relatively high across all inputs, with values exceeding 0.9 in multiple cases. Interestingly, certain inputs, such as Input 2 (25x25), exhibit efficiency values, suggesting a level of *superlinear* speedup, likely due to how *SOLUTION SPACES* have been divided. However, efficiency gradually declines as processes and threads count increases, while still maintaining very high and desirable levels.

7 CONCLUSION

This report presents the development and evaluation of three parallel approaches for a Hitori solver, using *MPI*, *OpenMP*, and a *Hybrid* method. The algorithm was designed with parallelism in mind, requiring **minimal communication** between processes and benefiting from an effective **workload balancing system**. This design enabled the solver to scale efficiently both horizontally across multiple nodes and vertically within shared-memory systems.

Experimental results confirmed that all three implementations led to significant **performance improvements** over the serial approach. The hybrid solution proved particularly effective, combining the distributed capabilities of *MPI* with the flexibility of *OpenMP*, addressing some limitations encountered in the individual methods.

In conclusion, the parallel Hitori solver demonstrated **good scalability** and resource efficiency, with the hybrid approach standing out as a robust and versatile solution. This work highlights the importance of proper load balancing and communication minimization in developing high-performance parallel algorithms.

8 FUTURE WORKS

While the developed application has proven to be promising, several areas of improvement can be made in optimizing its performance and flexibility. One of these is the optimization of the workload balancing system under the hybrid approach. The addition of a more dynamic **work-stealing mechanism**, like the message exchange strategy under the *MPI* version, can reduce idle times and make it more efficient.

Additionally, the current hybrid solution imposes a limitation that restricts the maximum number of processes at twice the value of *SOLUTION_SPACES*. Relaxing these constraint in later releases would provide more flexibility and scalability with increasing problem sizes and computational resources available.

Finally, the usage of resources, particularly **memory handling**, can be improved. Memory release and memory usage, in general, have been a bit neglected, and addressing these would serve to improve performance and reliability, especially for larger problem instances.

These directions have more work that could improve the solver's robustness and flexibility so that it is even better positioned to handle tough parallel workloads.

CODE

All code is located in our GitHub repository: https://github.com/sbrentan/hitori-solver/

REFERENCES

[1] OpenMP Architecture Review Board. 2011. OpenMP Application Program Interface. https://www.openmp.org/wp-content/uploads/OpenMP3.1.pdf.
[2] Message Passing Interface Forum. 2015. MPI: A Message-Passing Interface Standard. https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.
[3] Vegard Hanssen. [n. d.]. Puzzle database. https://menneske.no/hitori/eng/.
[4] Christian Hofer Matthias Gander. 2006. *Hitori Solver Bachelor Thesis.* Ph. D. Dissertation. University of Innsbruck.
[5] Puzzle Communication Nikoli. March 1990. Hitori. https://en.wikipedia.org/wiki/Hitori.
[6] Ava Hajratwala Peter Yao. 2023. *Hitori Final Report.* Technical Report. Barnard College of Columbia University.
[7] Schmoller. [n. d.]. Java Hitori. https://github.com/Schmoller/Hitori.
[8] Shi-Jim Yen, Tsan-Cheng Su, and Shih-Yuan Chiu. 2009. Hitori solver. In *Game Programming Workshop 2009.* 83–86.