EuroHPC Summit 2025
A branch-and-bound algorithm for graph coloring

Jan E. Marxen[1], **Luca Lamperti**[2], **Mohamed Z.M. Mandour**[3], and **Giacomo Pauletti**[4]

[1]*Sorbonne Université*, jan.marxen@gmail.com
[2]*Politecnico di Milano*, l.lamperti2002@gmail.com
[3]*Politecnico di Milano*, mohamedzeyad.mandour@mail.polimi.it
[4]*Politecnico di Milano*, giacomo.pauletti@mail.polimi.it

February 28, 2025

Abstract *Graph coloring has applications in fields like operational research, computational biology, and compiler optimization. In this report, we present a branch-and-bound algorithm designed to scale efficiently on modern supercomputers. At each step, the algorithm selects two vertices and either merges them or assigns them different colors, using heuristics to refine upper and lower bounds on the chromatic number. We tested different heuristics to improve performance and parallelized the algorithm with MPI and OpenMP. Our implementation runs on the EuroHPC petascale machine Vega, showing strong scalability and efficiency.*

1 Introduction

The *graph coloring problem* is a well-known NP-hard problem, which means that there exists no algorithm which finds an exact solution in polynomial time. Over the years many heurisitcs ([7], [1], [11], [10]) have been developed. A remarkable heuristic is the one Halldórsson [10] have proposed *SampleIS*, which currently has the best-known approximation ratio $|G|(log\ log|G|)^2/log^3|G|$, where G is the input graph and $|G|$ is the number of vertices. However, its time complexity is $O(|G|^3)$. The heuristics do not provide an exact solution but have a polynomial complexity.

Taking from the much broader field of optimization problems, many metaheuristics have been implemented, such as simulated annealing (cite this) or tabu search (cite this).

Many efforts have also been made in developing exact algorithms, which have been overviewed in [3].

In our work, we adapt an existing exact algorithm, originally developed by Zykov [2], to run on a modern supercomputer. In particular, we worked on the EuroHPC petascale machine Vega, having access to XXX of the XXX CPU nodes. [Later on we should give some in depth description of the supercomputer.]

Organization

The rest of the report is organized as follows. Section 2 introduces some basic facts about graph theory; Section 3 presents the original algorithm developed by Zykov and the theoretic aspects that support it. Section 4 shows the implementation details of the algorithm, that is, the updates to Zykov's algorithm, the graph representation, the branching strategies, and the heuristics used. Finally, Section 5 shows the tests we performed with the algorithm and Section 6 sums up the algorithm and what we achieved.

2 Preliminaries

Let $G = (V, E)$ be an undirected graph where $V = \{v_1, v_2, ..., v_n\}$ is the set of vertices and E is the set of edges in G.

Two vertices $v, w \in V$ are considered neighbors or adjacent if $(v, w) \in E$.

Given a sets of colors $C \subset N^*$, a valid coloring for graph G is a map $f : V \to C$ such that if two vertices are neighbors then $f(v) \neq f(w)$.

The chromatic color $\chi(G)$ is the minimum number of colors for a valid coloring of G to exist.

A clique of a graph G is a *complete induced subraph*, i.e. a group of vertices which are all neighbors to each other. Finding the *maximum clique* of a graph is another well-known NP-hard problem.

Another recurrent concept is the degree of a vertex, which is the number of neighbors

3 Zykov algorithm

The original algorithm [2], developed by Zykov in 1962, is based on his recurrence which states that, given a graph G, for any pair of vertices $x, y \in V(G)$ that do not share and edge, an optimal coloring of G can either assign the same color to x and y or not (Theorem 1).

In this section, we first show the basis of Zykov's algorithm, which he called a *Zykov tree*. Then, we discuss the complexity of the algorithm based on [3] and [5].

3.1 Zykov Trees

Zykov [2] has stated that we can obtain two new graphs from G, for a given pair of vertices $x, y \in V(G)$ that are not neighbors. One of these graphs will *contract* x and y into a single vertex, while the other will create an edge between x and y. The chromatic number of G corresponds to the minimum chromatic number of one of these graphs.

Definition 1 *For two vertices $x, y \in V(G)$ that do not share an edge, a contraction in G produces a new graph G'_{xy} given by*

$$V(G'_{xy}) = V(G) \setminus \{x, y\} \cup \{z\}$$

$$E(G'_{xy}) = \begin{cases} \{u, v\} \in E(G) \ : \ x \notin \{u, v\} \ , \ y \notin \{u, v\} \\ \quad \cup \\ \{u, z\} \ : \ \{u, x\} \in E(G) \ \text{or} \ \{u, y\} \in E(G) \end{cases}$$

For two vertices $x, y \in V(G)$ that do not share an edge, an addition in G produces a new graph G''_{xy} given by

$$V(G''_{xy}) = V(G) E(G''_{xy}) = E(G) \cup x, y$$

Theorem 1 *The chromatic number of G is given by the recurrence*

$$\chi(G) = \min\{\chi(G'_{xy}), \chi(G''_{xy})\}$$

such that $x, y \in V(G)$ and $\{x, y\} \notin E(G)$

The recurrence of Theorem 1 builds a binary tree called *Zykov tree*. Its leaves are cliques, particularly complete graphs. Since they are the base case of the recursion defined by Theorem 1, the chromatic number of the original graph is the minimum chromatic number of the cliques, i.e. the size of the smallest clique.

Zykov trees have exctly one branch formed only by constraction operations, where the leaf of this branch is a clique whose size is an upper bound q for the chromatic number. This bound is updated each time a better coloring than the current one is found, i.e. a smaller leaf is found. In a Branch-and-Bound version of Zykov's algorithm, operations of contractions and additions will happen only in graphs that do not have a q-clique on its structure; branches whose graphs do not satisfy this will be pruned, since their chromatic number is no better than the current best guess q. Figure 1 shows an example of Algorithms 2 and 1.

De Lima and Carmo [3] showed that, in general, Zykov's algorithm has a complexity $O(2^{n^2})$ and space $O(n^2(n+m))$, where $n = |V|, m = |E|$. McDiarmid [5] focused on a more precise estimate, stating that, for almost all graphs, Zykov's algorithm has a complexity of $O(e^{cn\sqrt{\log n}})$ where $c > 1$ is a constant.

Algorithm 1 Color(G)

```
1:  procedure COLOR(G)
2:      n ← |V(G)|
3:      if G is a complete graph then
4:          q ← min{n, q}
5:      else if G does not have a q-clique then
6:          Choose x, y ∈ V(G) such that {x, y} ∉ E(G)
7:          COLOR(G'_xy)
8:          COLOR(G'_xy)
9:      end if
10:     return q
11: end procedure
```

Algorithm 2 Zykov(G)

```
1: procedure COLOR(G)
2:     n ← |V(G)|
3:     χ(G) ← COLOR(G)
4:     return χ(G)
5: end procedure
```

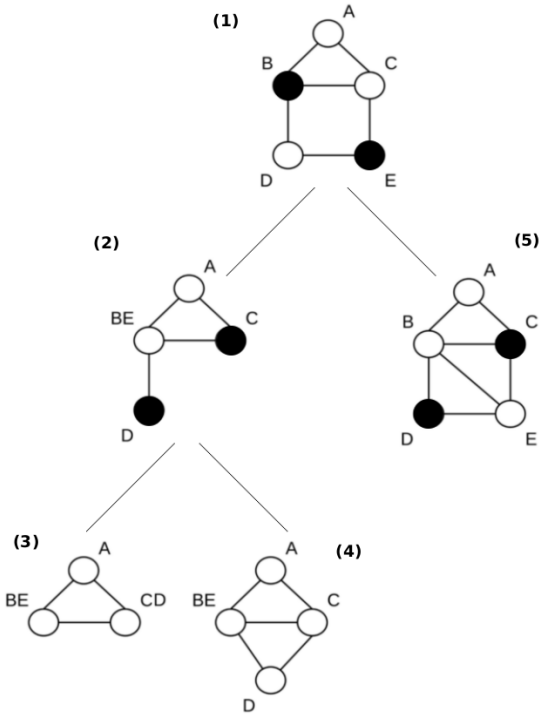

Figure 1: Pruned Zykov tree

4 Implementation

4.1 Graph Representation

We have found 4 possible ways to represent the graph.

Adjacency matrix, where the graph is represented as a boolean symmetric square matrix A of size $|V|$ and $A(v, w) = 1 \iff (v, w) \in E$. The matrix is contiguously stored in memory, so edges operations are very efficient ($O(1)$) but the three operations on the vertices, which require modifying the shape of the matrix, are very expensive ($O(|V|^2)$). Accessing neighbors of a ver-

Table 1: Comparison of Graph Representations

Operation	Adjacency matrix	Edge List	CSR	Adjacency List
Add Edge	$O(1)$	$O(1)$	$O(\|E\|)$ (rebuild)	$O(1)$
Remove Edge	$O(1)$	$O(\|E\|)$	$O(\|E\|)$ (rebuild)	$O(k)$
Add Vertex	$O(\|E\|)$	$O(1)$	$O(1)$	$O(1)$
Remove Vertex	$O(\|E\|)$	$O(\|E\|)$	$O(\|E\|)$ (rebuild)	$O(\|E\|)$
Merge Vertices	$O(\|E\|)$	$O(\|E\|)$	$O(\|E\|)$ (rebuild)	$O(\|E\|)$
Get Neighbors	$O(\|V\|)$	$O(\|E\|)$	$O(1)$)	$O(1)$

tex costs $O(|V|)$ since it is done by reading a row of the matrix. Memory continuity is an advantage, especially for denser graphs where the number of 0)'s in the matrix is low. One last remark is that the space complexity is much higher than the other formats.

Edge vector, where the edges are stored as (v, w) pairs in a unique vector. Edge removal and lookup, as well as neighbors lookup and vertex merging, cost $O(|E|$, since they involve going through the whole array. Instead the edge addition, vertex addition/removal cost $O(1)$.

Compressed Sparse Row (CSR). Inspired to the adjacency matrix but it is represented in a compressed format. [Document this better]

Adjacency lists, where for each vertex there is a vector containing its neighbors. The graph is then represented as a vector of vectors. Vertex addition/removal as well as edge addition, are computed in $O(1)$. Vertex merging costs in the worst case $O(|E|)$ but in practice it is much less. Edge removal and lookup cost $O(|V|)$ and neighbor lookup costs $O(1)$.

In the Table 1 below, we sum op the four representations in terms of their complexity on the most common operations: *vertex addition/removal*, *edge addition/removal*, *vertex merging*[1], *edge lookup*[2] and *accessing neighbours of a vertex*. Note that k is the (mean) number of neighbors for a vertex.

We concluded that the **adjacency list** representation is more suited for our application, since it offers a great performance in modifying the graph. Even though the memory is dislocated in more vectors, if the vectors are big enough (i.e. the graph is enough dense), there is no advantage in having a unique contiguous array.

4.2 Branching Strategy

At each step, the branch-and-bound algorithm *chooses* 2 non-adjacent vertices $v, w \in V$ and creates two new branches: in the first one, v and w are contracted; in the second one the edge (v, w) is added to E.

The strategy with which the vertices are chosen, i.e. the branching strategy, is crucial for an efficient branch-and-bound algorithm.

We first adopt an easy strategy, called the *random branching strategy*, in which the pair v, w is chosen randomly.

[1] the same operation which was discussed in the Zykov's algorithm

[2] seeing if $(v, w) \in E$

A more efficient choice is the one proposed by Brigham and Dutton [4]. The authors aim to choose the vertices v, w that, if contracted, provide the smallest graph (i.e. with fewest edges possible). In other words, vertices v, w have the maximum number of common neighbors among all possible pairs of vertex.

4.3 Heuristics

Another key aspect for the efficiency of the algorithm is the quality of the heuristics used for finding a lower bound (lb) and an upper bound (ub) of the chromatic number.

Good heuristics gives an interval lb, ub which is as tight as possible and so, given two branches, the probability of pruning one branch (i.e. $lb_1 > ub_2$) is much higher.

At the same time, heuristics have to be computationally light.

4.3.1 Coloring heuristics

Coloring heuristics give an upper bound value for the chromatic number: $\chi(G) \leq ub$.
There is a manifold of heuristics developed during the years. A first example is a **greedy** algorithm (Algorithm 3). It iterates through all the vertices assigning them the the lowest color possible (minimum color is 1).

It is well-known that this algorithm performs better if the vertices in the input are sorted by non-increasing degree [12]. An improved solution could be found also by running the greedy algorithm on the vertices ordered by color by a previous solution [9].

Algorithm 3 Greedy coloring

```
1:  procedure GreedyColor(G)
2:      Initialize max_color ← 0
3:      for all v ∈ graph do
4:          for i = 1 → max_color do
5:              if can be assigned color i then
6:                  color[v] ← i break
7:              end if
8:          end for
9:          if not assigned then
10:             max_color ← max_color + 1
11:             color[v] ← max_color
12:         end if
13:     end for
14: end procedure
```

Probably the most used heuristic is **DSatur** [7]. The algorithm is a greedy algorithm which dynamically chooses the vertex with highest saturation degree, i.e. the highest number of unique colors that its neighbors have. Tie are broken by choosing the vertex with highest degree. A pseudo code is shown in Algorithm 4.

The complexity of this algorithm lies on efficiently implementing the function *GetMaxSatDegree*. This operation is efficient thanks to a problem-tailored priority queue. This data structures orders the vertices by its saturation degree and those with same saturation degree with the degree. When a vertex is colored, only its neighbors change color and so only those change their position in the data structure.

The complexity for updating the data structure when a vertex is colored is $O(k * |E|)$ in the worst case, with k being the number of neighbors of the vertex. However this is a very rare case and typically an update is very less expensive. The complexity could be improved to $O(k * log(|E|)$ by using a priority queue based on a balanced tree.

This provides a better strategy than the greedy algorithm since it gives the lowest possible color to the vertices which are most probably going to have a high color.

Algorithm 4 DSatur coloring

```
1: procedure DSaturColor(G)
2:     Initialize max_color ← 0
3:     while G not empty do
4:         v ← GetMaxSatDegree(G)
5:         for i = 1 → max_color do
6:             if can be assigned color i then
7:                 color[v] ← i break
8:             end if
9:         end for
10:        if not assigned then
11:            max_color ← max_color + 1
12:            color[v] ← max_color
13:        end if
14:    end while
15: end procedure
```

There exist methods that can improve the current solution. An example of such an algorithm is the **recolor** algorithm. For any vertex v with the highest color C_{max}, it tries to assign it a lower color C'. To make this new coloring valid, all neighbors of v with color $C'$are recolored with another available color. This might not always be possible; in that case, the algorithm tries to assign a different color C'' to the vertex v. In this way, the maximum color can be decreased by 1 for each iteration of the algorithm. However, it is rare that this algorithm works for a large graph, but it can be successfully applied when the graph has been reduced by many contractions by the branch-and-bound algorithm.

A pseudo code of a recursive implementation of recoloring can be found in Algorithm 5. For any $v \in V$, $C(v)$ is the color assigned to it by a previous coloring algorithm.

Algorithm 5 Recoloring

```
1: procedure Recolor(G, V_max)
2:     v ← pop a vertex from V_max
3:     for k = 1 → max_color − 1 do
4:         C(v) ← k
5:         initializes success ← 1
6:         for all w ∈ Γ(v) do
7:             if C(w) = C(v) then
8:                 success ← ChangeColor(w)
9:                 if success = 0 then
10:                    Revert()
11:                    break
12:                end if
13:            end if
14:        end for
15:        if success = 1 then
16:            success ← Recolor(G, V_max)
17:            if success = 1 then
18:                push back v into V_max
19:                return 1
20:            end if
21:            Revert()
22:        end if
23:    end for
24:    push back v into V_max
25:    return 0
26: end procedure
```

4.4 Clique Heuristics

Clique heuristics are used to approximate the *maximum clique* of a given graph G, providing a *lower bound* for the chromatic number:

$$lb \leq \omega(G) \leq \chi(G)$$

Finding the maximum clique is an *NP-hard problem*, making exact solutions computationally expensive for large graphs. Instead, heuristics provide *fast approximations* with good performance.

4.4.1 FastWClq Algorithm

For this project, we adopt the **FastWClq** heuristic, introduced by Cai and Lin in [6]. FastWClq efficiently approximates the *maximum weighted clique* through *clique construction* and *graph reduction techniques*.

4.4.2 Vertex Selection in FastWClq

A critical step in FastWClq is selecting the best vertex to extend the current clique. We implement a *Best from Multiple Selection* (BMS) heuristic to ensure efficient selection.

4.4.3 Graph Reduction in FastWClq

To optimize performance, FastWClq employs a graph reduction technique that removes vertices that cannot be part of a maximum clique.

Algorithm 6 FastWClq

```
1: procedure FastWClq(G, cutoff)
2:     StartSet ← V(G)
3:     C* ← ∅
4:     k ← k_0
5:     while elapsed time < cutoff do
6:         if StartSet = ∅ then
7:             StartSet ← V(G)
8:             AdjustBMSNumber(k)
9:         end if
10:        u ← pop a random vertex from StartSet
11:        C ← {u}
12:        CandSet ← N(u)
13:        while CandSet ≠ ∅ do
14:            v ← ChooseAddVertex(CandSet, k)
15:            if w(C) + w(v) + w(N(v) ∩ CandSet) ≤
   w(C*) then
16:                break
17:            end if
18:            C ← C ∪ {v}
19:            CandSet ← CandSet \ {v}
20:            CandSet ← CandSet ∩ N(v)
21:        end while
22:        if w(C) > w(C*) then
23:            C* ← C
24:        end if
25:        G ← ReduceGraph(G, C*)
26:        StartSet ← V(G)
27:        if G becomes empty then
28:            return C*
29:        end if
30:    end while
31:    return C*
32: end procedure
```

Algorithm 7 ChooseAddVertex

```
1: procedure ChooseAddVertex(CandSet, k)
2:     if |CandSet| < k then
3:         return the vertex v ∈ CandSet with the
   highest estimated benefit
4:     end if
5:     v* ← random vertex in CandSet
6:     for i = 1 → k − 1 do
7:         v ← random vertex in CandSet
8:         if b(v) > b(v*) then
9:             v* ← v
10:        end if
11:    end for
12:    return v*
13: end procedure
```

Algorithm 8 ReduceGraph

```
1: procedure ReduceGraph(G, C*)
2:     for all v ∈ V(G) do
3:         if UB_0(v) ≤ w(C*) or UB_1(v) ≤ w(C*) then
4:             Add v to RmQueue
5:         end if
6:     end for
7:     while RmQueue ≠ ∅ do
8:         u ← pop vertex from RmQueue
9:         Remove u and incident edges from G
10:        for all v ∈ N(u) do
11:            if UB_0(v) ≤ w(C*) or UB_1(v) ≤ w(C*)
   then
12:                Add v to RmQueue
13:            end if
14:        end for
15:    end while
16:    return G
17: end procedure
```

4.4.4 Performance Considerations

FastWClq is significantly *faster* than traditional *exact algorithms* (e.g., *Bron-Kerbosch*), making it ideal for *large-scale graphs*. The *graph reduction* step helps *prune the search space*, leading to improved efficiency.

By using **FastWClq**, we ensure that our branch-and-bound algorithm has *strong lower bounds*, increasing the effectiveness of the pruning strategy and reducing the number of unnecessary branches.

4.5 Parallelization Strategies

Our branch and bound algorithm is highly parallelizable. We used **MPI** for inter-processor parallelization plus **OpenMP** for intra-processor parallelization.

The main idea is to assign to each MPI process a particular node of the *Zykov's tree*, where a node corresponds to a particular graph obtained through *contraction* and *addition* operations. Each MPI process will explore the subtree generated from the particular node assigned to it. This implies that, frequently, MPI processes share the current best upper bounds found, to help other processes prune their branches. This is a lightweight operation. However, if this operation is performed too frequently, it can have negative effects; it should then be decreased (i.e the period is increased) if the number of MPI processes is increased. We suggest to start from a period of 20s.

When an MPI process finishes exploring its subtree, it *steals* some work from other branches, i.e it steals an unexplored subtree that another MPI process should explore. This involves heavier communications, since graphs are exchanged between MPI processes, however, it is very rare and negligible.

Further communications are used to signal the timeout or when an optimal solution is found.

OpenMP was used to divide communication from computation. The following threads were used:

- *Thread_0_terminator*, which handles code termination when the timeout is reached or an optimal solution is found.
- *Thread_1_Gatherer*, which updates the global solution by sharing the local solutions of all cores using a specific parameter.
- *Thread_2_employer*, which manages the workstealing process as the recipient of task requests.
- *Worker thread*, responsible for handling all branching-related functions.

In the first implementation, the branching was performed by the Master, a process with rank equal to zero, which then distributed the corresponding branches to each worker process to execute the algorithm in parallel, as shown in Figure 2.

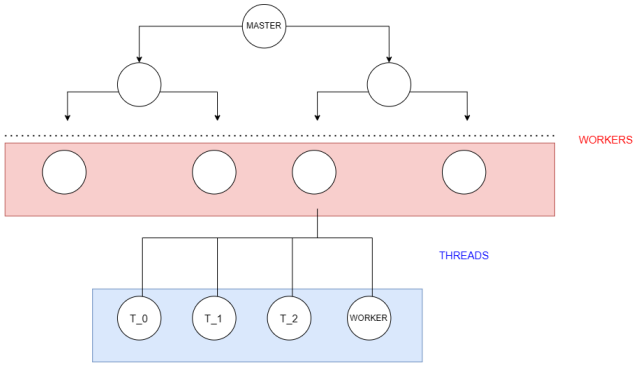

Figure 2: Simple Approach

The most critical aspects that we identified with this approach were:

1. *Process initialization*, ensuring scalability and balancing the workload among workers, avoiding scenarios where one worker has significantly more work than others.
2. *Branch transfer between processes*, both in the initial distribution and during work-stealing.

After evaluating various scenarios and optimizing for the previously mentioned challenges, we decided that each worker should load the initial graph and perform its own specific branching (i.e. *contraction* and *addition* operation) until they reach a unique branch not seen by other workers. In other words, the workers calculate which "red node" in Figure 3 they are assigned and then they build the corresponding graph.

This eliminates the need for initial communication, making the process faster, while still ensuring that each worker operates on an independent branch, avoiding work duplication. This guarantees scalability and data balancing.

We developed two approaches: *balanced branch-and-bound* (Figure 3) and *unbalanced branch-and-bound* (Figure 4). As the name suggests, the first has a better workload balancing property. For the first approach to work properly, we require that the number of MPI processes used is a power of 2. This requisite is aimed to simplify the code but can be easily be updated to consider cases in which the number of MPI processes is not a power of 2.

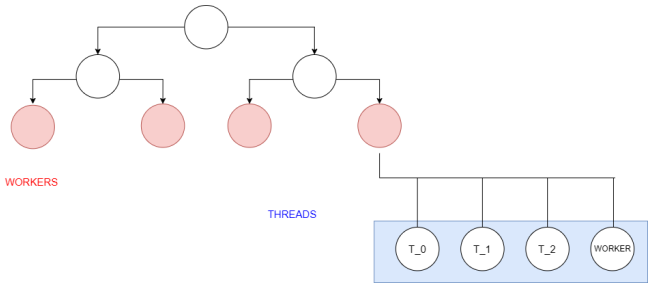

Figure 3: Scalable Approach 1, case 4 cores

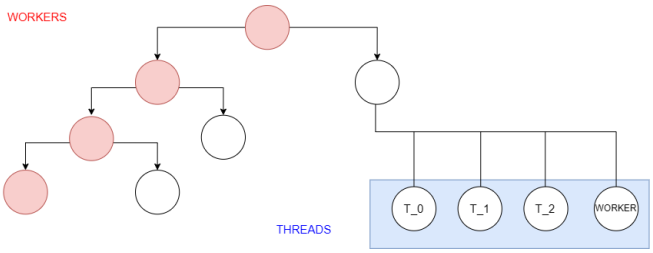

Figure 4: Scalable Approach 2, case 4 cores

As mentioned above, we implemented a workstealing strategy to address potential pruning effects or cases where some branches are easier to process than others. This allows an idle worker to obtain work from others. Notably, the use of a dedicated thread for communication management and initial work distribution choices, see figure 3 reduces the likelihood of needing this technique compared to 4, thanks to improved data balancing.

5 Results

5.1 Supercomputer Architecture

Our experiments were conducted on the supercomputer Vega hosted by the Institute of Information Science - IZUM in Maribor, Slovenia. The CPU partition consists of two AMD Rome 7H12 processors, each with 64 cores, totaling 128 cores per node. Each processor follows a Non-Uniform Memory Access (NUMA) architecture, where cores are grouped into multiple NUMA domains with varying memory access latencies.

To optimize performance, we carefully chose the MPI and OpenMP configuration based on the system architecture. Specifically:

- We use **8 MPI processes per node** (i.e., `#SBATCH --ntasks-per-node=8`) to match the NUMA configuration of the AMD Rome 7H12 processors. Since each processor has four NUMA regions (eight per node in total), this ensures that each MPI process is assigned to a distinct NUMA region, reducing memory contention and improving locality.
- We allocate **4 CPU cores per MPI process** (i.e., `#SBATCH --cpus-per-task=4`), which aligns with our choice of running **4 OpenMP threads per MPI process**. This configuration ensures that each OpenMP region has exclusive access to 4 cores within the assigned NUMA domain.
- The number of nodes is varied (i.e., `#SBATCH --nodes=<variable>`) to study the strong scaling behavior of our algorithm while keeping the MPI and OpenMP configuration fixed.

This setup aims to minimize memory access overhead and maximize compute resource usage.

5.2 Evaluation criteria

We evaluated our algorithm on the *Graph Coloring Instances* database[3].

Due to the exponential complexity of the task, our aim was not to prove the optimality of our solution, i.e fully explore the whole *Zykov's tree*: for many graphs this is too complex to be achieved in less than $10000s$. Instead, for those graphs whose *chromatic number* $\chi(G)$ was given in the database, we aimed to find a valid coloring with $\chi(G)$ colors. When this coloring is found, the algorithm stops, otherwise the algorithm runs until timeout.

For those graphs whose *chromatic number* is not given, we let the algorithm run until convergence (i.e. *Zykov's tree* is fully explored) or timeout.

5.3 Strong Scaling

To evaluate the strong scaling of our algorithm, we fix the problem size (i.e., a specific graph) and vary the number of nodes while maintaining the chosen hybrid parallelism model. The expectation is that as the number of nodes increases, the execution time should decrease proportionally.

The experiment was performed to solve for the chromatic number of the graph `queen7_7.col`.

Nodes	Time (s)	Total Cores	N processors
1	330.91	32	8
2	100.22	64	16
4	70.78	128	32
8	55.89	256	64
16	46.01	512	128
32	0.23	1024	256

Table 2: Strong scaling results for solving the chromatic number of `queen7_7.col`.

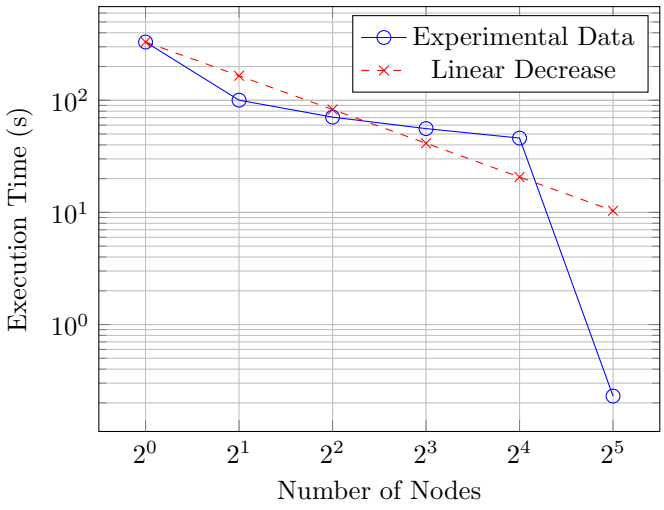

Figure 5: Execution time vs. number of nodes for solving `queen7_7.col`.

With 32 nodes, we see a dramatic drop in execution time to just 0.23 seconds. This happens because one of the 256 MPI processes quickly finds the solution, effectively ending the computation almost immediately. The results show that doubling the number of nodes consistently reduces execution time, making strong scaling highly effective in this case. Exploring many branches in parallel, the likelihood to find a good solution is higher than the sequential algorithm. This implies that in the parallel version, *more branch pruning is performed* and so, as shown in Figure 6, we can sometimes obtain a **superlinear speedup**.

Another experiment was performed to solve for the chromatic number of the graph `queen9_9.col` with balanced strategy and Dsatur coloring.

Nodes	Time (s)	Total Cores	N processors
1	218.86	1	1
4	1.115	512	32

Table 3: Strong scaling results for solving the chromatic number of `queen9_9.col`.

[3] `https://mat.tepper.cmu.edu/COLOR/instances.html`

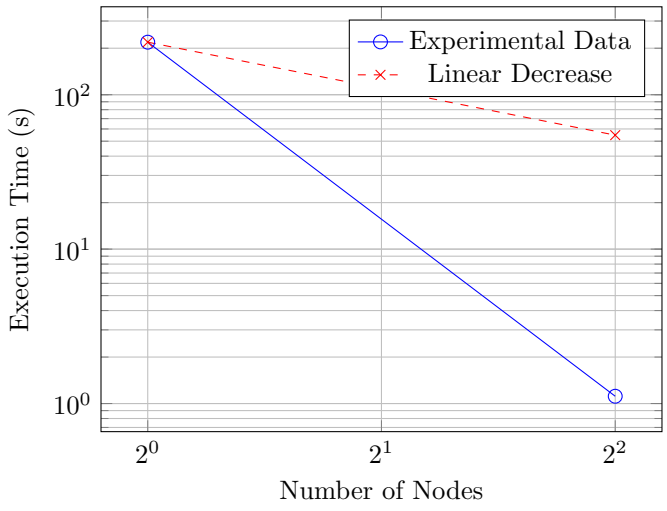

Figure 6: Execution time vs. number of nodes for solving `queen9_9.col`, with comparison with linear decreasing.

In this case, we observe a sub-linear speedup. One reason for this behavior could be the use of a check based on the expected solution instead of waiting for the complete execution of graph.

Since communication overhead is minimal, adding more nodes should continue to improve performance, especially for more complex graphs that require longer search times. This suggests that our approach will scale well as problem difficulty increases.

5.4 Comparisons

The results in Table 4 are obtained by using different options for each graph. We tried both the balanced (B) and the unbalanced branch-and-bound (U), combined with Greedy (G), DSatur (D) or Interleaved Dsatur with DSatur&Recolor (I).

We found that there is *no configuration that outperforms the others* in all graphs. A striking example is *inithx.i.1*: using *balanced branch-and-bound* plus DSatur took about $30s$ to reach the optimal value; instead using *unbalanced branch-and-bound* plus DSatur took just $0.2s$. The reason behind this is that the unbalanced algorithm performs an earlier prune, which is not done by balanced, due to how work is initially distributed.

For graphs *fpsol2.i.1*, *fpsol2.i.2*, *fpsol2.i.3* the sequential algorithm performed better than the parallel one, altough the parallel one did not perform badly (around *2* seconds). The scheduling in this case dominates the complexity of coloring the graph.

Graph	NP	Coloring	Time	Strategy
anna	8	11*	0.012	–
jean	8	10*	0.031	–
fpsol2.i.1	1	65*	0.03	(SEQ)UI
fpsol2.i.2	1	30*	0.013	(SEQ)UI
fpsol2.i.3	1	30*	0.014	(SEQ)UI
games120	8	9*	0.089	BI
homer	8	13*	0.99	BI
huck	8	11*	0.03	BI
inithx.i.1	8	54*	0.025	BD
queen6_6	32	7*	22.931	BD
queen6_6	64	7*	0.671	BI
queen7_7	256	7*	0.23	BI
queen9_9	1	10*	223	(SEQ)UI
queen9_9	512	10*	1.12	BD
queen11_11	1024	13	> 10000	BI
queen16_16	512	19	1000	BI
le450_25a	1024	25*	12.35	BI
le450_25b	1024	25*	14.31	BI
le450_25c	1024	27	> 10000	BD
le450_25d	1024	27	> 10000	BD
le450_5a	512	6	> 10000	BI
le450_5b	512	6	> 10000	BI
le450_5c	8	5*	22.61	BI
le450_5d	1	5*	46.79	(SEQ)UD
le450_15d	1024	22	> 10000	BI
miles250	128	8*	0.063	BI
miles500	8	20*	0.16	BI
miles750	8	31*	0.95	BI
miles1000	8	42*	0.34	BI
miles1500	8	73*	0.34	BI
myciel3	8	4*	0.01	-D
myciel4	8	5*	0.012	-D
myciel5	8	6*	0.029	-D
myciel6	8	7*	0.074	-D
myciel7	8	8*	0.34	-D
zeroin.i.1	8	49*	0.5	BI
zeroin.i.2	8	30*	0.46	BI
zeroin.i.3	8	30*	0.45	BI

Table 4: Some results obtained. [Legend: n^* means that optimum was reached; U=Unbalanced, B=balanced, G=greedy, D=Dsatur, I=interleaved, -=any strategy]

6 Conclusions

We developed a branch-and-bound algorithm for graph coloring that makes use of modern supercomputers. By refining Zykov's algorithm with heuristics and parallelization using MPI and OpenMP, we achieved strong scalability, with parallel execution leading to superlinear speedup in some cases due to more effective branch pruning. Our implementation on the EuroHPC petascale machine Vega showed that this approach works well for many graph instances, though sparser graphs posed more challenges.

Future work could focus on adapting our strategies to handle sparse graphs more efficiently. Further OpenMP parallelization could also help, allowing multiple threads to work on branching simultaneously. Moreover, reduc-

ing communication overhead by replacing the AllGather operation with hierarchical reduction could further improve communication at scale. This would enable more frequent solution exchanges while keeping communication costs low, ensuring better scalability.

Our results show that parallel computing can make a big difference in tackling NP-hard problems like graph coloring. With further refinements, this approach could be used to solve for bigger, sparser graphs.

7 Acknowledgments

We are grateful to the organizers of the EuroHPC Summit Challenge for the opportunity to work on this project and we would also like to thank the team behind the Vega supercomputer for providing access to their HPC system, which was essential for running and testing our parallel implementation at scale.

A special thanks to Professors Janez Konc and Roman Trobec for their role in organizing this challenge and their work described in [8], which helped guide our approach to tackling this problem.

Finally, we appreciate everyone who provided feedback and engaged in discussions that contributed to this project. A special thanks to Professor Arnaud Renard, our mentor throughout the project, for his continuous guidance and support.

References

[1] Hertz A. "A fast algorithm for coloring Meyniel graphs". In: *Journal of Combinatorial Theory* 50 (1990), pp. 231–240.

[2] Zykov A. "On some properties of linear complexes". In: *Mat. Sb.* 24.2 (1962), pp. 418–419.

[3] R. Carmo A.M. de Lima. "Exact Algorithms for the Graph Coloring Problem". In: *RITA* 25.04 (2018), pp. 47–73.

[4] Dutton R. D. Brigham R. D. "A new graph coloring algorithm". In: *The Computer Journal* 24 (1981), pp. 85–86.

[5] McDiarmid C. "Determining the chromatic number of a graph". In: *Comput.* 8.1 (1979), pp. 1–14.

[6] Lin J. Cai S. "Fast Solving Maximum Weight Clique Problem in Massive Graphs". In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence* (2016), pp. 568–574.

[7] Brélaz D. "New methods to color the vertices of a graph". In: *Commun ACM* 22 (1979), pp. 251–256.

[8] M. Depolli et al. "Exact parallel maximum clique algorithm for general and protein graphs". In: *Journal of Chemical Information and Modeling* 53.9 (2013), pp. 2217–2228.

[9] R.M.R Lewis. *Guide to graph colouring. Algorithms and Applications*. Springer, 2021. ISBN: 978-3-030-81053-5.

[10] Halldórsson M. M. "A still better performance guarantee for approximate graph coloring". In: *Information Processing Letters* (1993).

[11] Leighton F. T. "A graph coloring algorithm for large scheduling problems". In: *Journal of Research of the National Bureau of Standards* 84 (1979), pp. 489–503.

[12] Powell M Welsh D. "An upper bound for the chromatic number of a graph and its application to timetabling problems". In: *Comput J* 12 (1967), pp. 317–322.