Université de Reims Champagne-Ardenne

Master 1 — Calcul Haute Performance / Imagerie / IA

Solveur de Règles de Golomb

Optimisation SIMD et Parallélisation
sur Cluster HPC

Auteur
Nicolas

Formation
Master 1 HPC

Cluster HPC
Romeo — URCA

Technologies
C++17, MPI, OpenMP, AVX2

Année universitaire 2024–2025

Table des matières

1 Introduction **5**

1.1 Contexte et Motivation 5

1.2 Applications Pratiques 6

1.3 Complexité du Problème 6

1.4 Objectifs du Projet 6

1.5 Organisation du Rapport 7

2 Fondements Algorithmiques **8**

2.1 Formalisation du Problème 8

2.2 Algorithme Branch-and-Bound 8

2.2.1 Principe Général 8

2.2.2 Pseudo-code 9

2.2.3 Analyse de Complexité 9

2.3 Heuristique Gloutonne pour la Borne Initiale 9

2.4 Structure de Données pour les Différences 9

3 Implémentation Séquentielle (V1) **12**

3.1 Architecture Logicielle 12

3.1.1 Structure de Données BitSet256 12

3.2 Optimisation AVX2 pour la Détection de Collisions 13

3.2.1 Principe de la Vectorisation 13

3.2.2 Implémentation 13

3.2.3 Analyse de Performance 14

3.3 Structure SearchState 15

3.4 Résultats de la Version Séquentielle 15

4 Parallélisation OpenMP (V2) **16**

4.1 Stratégie de Parallélisation 16

4.1.1 Profondeur de Parallélisation Adaptative 16

4.1.2 Création de Tâches OpenMP 16

4.2 Problèmes Rencontrés et Solutions 17
4.2.1 Problème 1 : Race Condition sur la Borne Globale 17
4.2.2 Problème 2 : Contention Atomique 19
4.2.3 Problème 3 : False Sharing 20
4.3 Résultats de Scaling 21

5 Distribution MPI Master/Worker (V3) 23
5.1 Motivation 23
5.2 Architecture Master/Worker 23
5.3 Génération des Sous-arbres 23
5.4 Distribution Dynamique du Travail 24
5.5 Propagation des Bornes 24
5.6 Problème : Goulot d'Étranglement du Master 26
5.7 Résultats de V3 26

6 Architecture Hypercube Décentralisée (V4/V5) 28
6.1 Motivation : Éliminer le Goulot d'Étranglement 28
6.2 Topologie Hypercube 28
6.3 Distribution Statique Déterministe 29
6.4 Propagation de Bornes O(log P) 30
6.5 Problème Rencontré : Deadlock MPI 31
6.6 V5 : Version MPI Pure 32
6.7 Comparaison V3 vs V4 33

7 Résultats Expérimentaux 35
7.1 Environnement de Test 35
7.1.1 Cluster Romeo 35
7.1.2 Configuration Logicielle 35
7.2 Scaling Fort 36
7.3 Scaling Faible 36
7.4 Impact des Optimisations AVX2 38
7.5 Overhead de Communication MPI 38
7.6 Solutions Optimales Trouvées 38
7.7 Visualisations des Performances 38

8 Discussion 41
8.1 Récapitulatif des Défis Techniques 41
8.2 Trade-offs Architecturaux 41
8.3 Limitations 42

8.4 Perspectives d'Amélioration 42
8.4.1 Court terme 42
8.4.2 Long terme 42

9 Conclusion **43**
9.1 Réalisations 43
9.2 Apprentissages Clés 43
9.3 Recommandations 44

A Extraits de Code Source **46**
A.1 BitSet256 Complet 46
A.2 Configuration SLURM 47

Chapitre 1

Introduction

1.1 Contexte et Motivation

Les **règles de Golomb** constituent un problème combinatoire fondamental en théorie des nombres, avec des applications concrètes en radio-astronomie, en théorie de l'information et en cryptographie. Une règle de Golomb d'ordre n est un ensemble de n marques entières positives distinctes tel que toutes les distances entre paires de marques sont uniques.

Définition 1.1: Règle de Golomb

Une **règle de Golomb** d'ordre n est un ensemble de n entiers $\{0 = a_1 < a_2 < \cdots < a_n\}$ tel que pour tous indices distincts $(i, j) \neq (k, l)$, on a :

$$|a_i - a_j| \neq |a_k - a_l|$$

La **longueur** de la règle est définie par $L = a_n$, la valeur de la plus grande marque.

Une règle de Golomb est dite **optimale** si aucune règle de même ordre n'a une longueur strictement inférieure. Trouver une règle optimale est un problème **NP-difficile**, ce qui signifie qu'aucun algorithme polynomial n'est connu pour le résoudre.

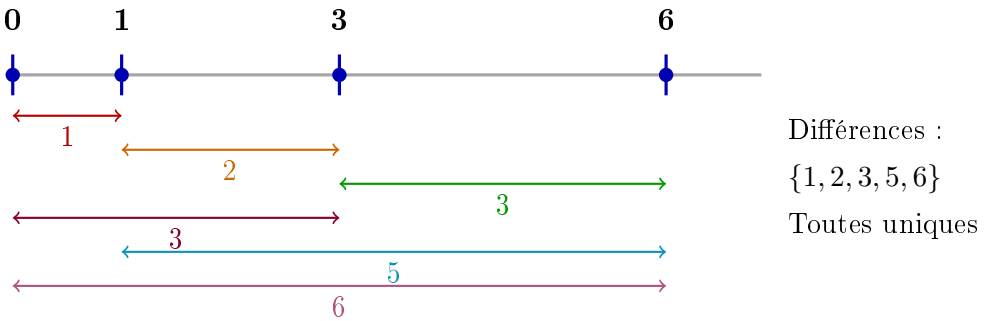

Figure 1.1: Règle de Golomb optimale d'ordre 4 : $\{0, 1, 3, 6\}$ avec longueur $L = 6$

1.2 Applications Pratiques

Les règles de Golomb trouvent des applications dans plusieurs domaines :

- **Radio-astronomie** : Positionnement optimal des antennes dans un réseau interférométrique. Une règle de Golomb permet de maximiser les fréquences spatiales mesurables avec un nombre limité d'antennes.
- **Théorie de l'information** : Conception de codes correcteurs d'erreurs et de séquences d'étalement de spectre.
- **Cryptographie** : Construction de fonctions de hachage et de générateurs pseudo-aléatoires.
- **Télécommunications** : Allocation de fréquences sans interférence dans les systèmes multi-porteuses.

1.3 Complexité du Problème

La recherche d'une règle de Golomb optimale d'ordre n est un problème NP-difficile. L'espace de recherche croît exponentiellement avec l'ordre : pour n marques et une longueur maximale L, il existe potentiellement $\binom{L-1}{n-2}$ configurations à explorer.

Table 1.1: Longueurs optimales connues des règles de Golomb (OEIS A003022)

Ordre	2	3	4	5	6	7	8	9	10	11	12	13
Longueur	1	3	6	11	17	25	34	44	55	72	85	106

Le calcul haute performance (HPC) devient indispensable pour trouver des solutions optimales au-delà de l'ordre 10, où le temps de calcul séquentiel dépasse plusieurs heures, voire plusieurs jours.

1.4 Objectifs du Projet

Ce projet vise à développer un solveur de règles de Golomb hautement optimisé et parallélisé, capable de s'exécuter efficacement sur le cluster HPC Romeo de l'Université de Reims. Les objectifs spécifiques sont :

1. **Optimisation bas niveau** : Exploiter les instructions SIMD (AVX2) pour accélérer la détection de collisions de différences.

2. **Parallélisation mémoire partagée** : Utiliser OpenMP pour exploiter les architectures multi-cœurs modernes.

3. **Distribution sur cluster** : Implémenter des versions MPI pour distribuer le calcul sur plusieurs nœuds.

4. **Analyse de performance** : Étudier le comportement de scaling fort et faible des différentes implémentations.

1.5 Organisation du Rapport

Ce rapport est structuré comme suit :

- **Chapitre 2** : Fondements algorithmiques — présentation de l'algorithme Branch-and-Bound
- **Chapitre 3** : Implémentation séquentielle (V1) avec optimisations AVX2
- **Chapitre 4** : Parallélisation OpenMP (V2)
- **Chapitre 5** : Distribution MPI Master/Worker (V3)
- **Chapitre 6** : Architecture Hypercube décentralisée (V4/V5)
- **Chapitre 7** : Résultats expérimentaux sur le cluster Romeo
- **Chapitre 8** : Discussion et perspectives
- **Chapitre 9** : Conclusion

Chapitre 2

Fondements Algorithmiques

2.1 Formalisation du Problème

Soit n l'ordre de la règle de Golomb recherchée. On cherche un ensemble de marques :

$$M = \{0 = m_0 < m_1 < m_2 < \cdots < m_{n-1}\}$$

tel que l'ensemble des différences :

$$D = \{m_j - m_i : 0 \leq i < j \leq n-1\}$$

contienne exactement $\binom{n}{2} = \frac{n(n-1)}{2}$ éléments distincts (toutes les différences sont uniques).

Objectif d'optimisation : Minimiser m_{n-1} (la longueur de la règle).

2.2 Algorithme Branch-and-Bound

L'approche adoptée est un algorithme de **Branch-and-Bound** (séparation et évaluation) avec backtracking. Cet algorithme explore l'arbre des solutions partielles en élaguant les branches qui ne peuvent pas mener à une solution optimale.

2.2.1 Principe Général

L'algorithme construit une solution partielle en ajoutant des marques une par une. À chaque étape, trois conditions d'élagage sont vérifiées :

1. **Collision de différences** : Si l'ajout d'une marque crée une différence déjà utilisée, la branche est abandonnée.
2. **Borne inférieure** : Si la position courante plus la borne inférieure des marques restantes dépasse la meilleure solution connue, la branche est élaguée.

3. **Symétrie** : La première marque non nulle est limitée à $[1, L_{best}/2]$ pour éliminer les solutions symétriques.

2.2.2 Pseudo-code

2.2.3 Analyse de Complexité

- **Complexité temporelle** : $O(L^n)$ dans le pire cas, où L est la longueur maximale explorée. En pratique, l'élagage réduit considérablement l'espace de recherche.
- **Complexité spatiale** : $O(n + L)$ pour stocker les marques courantes et l'ensemble des différences utilisées.

2.3 Heuristique Gloutonne pour la Borne Initiale

Avant de lancer la recherche exhaustive, une heuristique gloutonne fournit une borne supérieure initiale. Cette borne permet d'élaguer efficacement les branches dès le début de l'exploration.

2.4 Structure de Données pour les Différences

Le goulot d'étranglement de l'algorithme est la vérification des collisions de différences. Pour une règle de longueur L, nous devons pouvoir :

1. Tester si une différence $d \in [1, L-1]$ est déjà utilisée en $O(1)$
2. Ajouter/retirer une différence en $O(1)$

La solution naturelle est un **bitset** de taille L bits. Notre implémentation utilise un bitset de 256 bits, suffisant pour les règles jusqu'à l'ordre 14 (longueur optimale : 127).

Algorithme 1 : Algorithme Branch-and-Bound pour les règles de Golomb

Input : order n, borne initiale *bestLength*

Output : Règle de Golomb optimale

```
1  Fonction BranchAndBound(marks[], depth, usedDiffs, bestLength):
2      if depth = n then
3          if marks[n-1] < bestLength then
4              bestLength ← marks[n-1];
5              bestSolution ← copier(marks);
6          end
7          return;
8      end
9      startPos ← marks[depth-1] + 1;
10     if depth = 1 then
11         startPos ← 1;
12         endPos ← bestLength / 2;                    // Brisure de symétrie
13     end
14     else
15         endPos ← bestLength - 1;
16     end
17     for pos ← startPos to endPos do
           // Élagage par borne inférieure
18         remaining ← n - depth - 1;
19         minExtra ← remaining × (remaining + 1) / 2;
20         if pos + minExtra ≥ bestLength then
21             break;                                          // Élagage
22         end
           // Vérification des différences
23         if toutesNouvellesDifférences(pos, marks, depth, usedDiffs) then
24             marks[depth] ← pos;
25             ajouterDifférences(usedDiffs, pos, marks, depth);
26             BranchAndBound(marks, depth+1, usedDiffs, bestLength);
27             retirerDifférences(usedDiffs, pos, marks, depth);
28         end
29     end
```

Algorithme 2 : Heuristique gloutonne pour la borne initiale

Input : order n

Output : Règle de Golomb approximative

```
1  marks[0] ← 0;
2  usedDiffs ← ∅;
3  for i ← 1 to n-1 do
4  |   pos ← marks[i-1] + 1;
5  |   while ∃ collision dans usedDiffs do
6  |   |   pos ← pos + 1;
7  |   end
8  |   marks[i] ← pos;
9  |   Ajouter toutes les différences à usedDiffs;
10 end
11 return marks;
```

Chapitre 3

Implémentation Séquentielle (V1)

3.1 Architecture Logicielle

La version séquentielle (V1) constitue la base du projet. Elle implémente l'algorithme Branch-and-Bound avec des optimisations bas niveau pour maximiser les performances sur un seul cœur.

3.1.1 Structure de Données BitSet256

Le cœur de l'optimisation réside dans la structure `BitSet256`, un bitset de 256 bits aligné en mémoire pour permettre l'utilisation des instructions SIMD AVX2.

```
struct alignas(32) BitSet256 {
    uint64_t words[4];  // 4 mots de 64 bits = 256 bits

    // Test d'un bit en O(1)
    inline bool test(int bit) const {
        return (words[bit >> 6] >> (bit & 63)) & 1;
    }

    // Activation d'un bit en O(1)
    inline void set(int bit) {
        words[bit >> 6] |= (1ULL << (bit & 63));
    }

    // D sactivation d'un bit en O(1)
    inline void clear(int bit) {
        words[bit >> 6] &= ~(1ULL << (bit & 63));
    }
```

```
18
19      // R initialisation compl te
20      inline void reset() {
21          words[0] = words[1] = words[2] = words[3] = 0;
22      }
23  };
```

Listing 3.1: Structure BitSet256 avec alignement mémoire

Points clés :

- `alignas(32)` : Aligne la structure sur 32 octets (256 bits), requis par `_mm256_load_si256`
- `bit » 6` : Division par 64 pour sélectionner le mot (équivalent à `bit / 64`)
- `bit & 63` : Modulo 64 pour la position dans le mot

3.2 Optimisation AVX2 pour la Détection de Collisions

3.2.1 Principe de la Vectorisation

Les instructions AVX2 (Advanced Vector Extensions 2) permettent de traiter 256 bits en parallèle. Pour la détection de collisions, nous utilisons cette capacité pour tester simultanément si *n'importe quelle* nouvelle différence entre en collision avec les différences existantes.

3.2.2 Implémentation

```
1  #include <immintrin.h>  // Intrinsics AVX2
2
3  inline bool hasCollisionAVX2(const BitSet256& mask) const {
4      // Charge les 256 bits du bitset courant
5      __m256i used = _mm256_load_si256(
6          reinterpret_cast<const __m256i*>(words)
7      );
8
9      // Charge les 256 bits du masque de verification
10     __m256i check = _mm256_load_si256(
11         reinterpret_cast<const __m256i*>(mask.words)
12     );
13
14     // AND bit-a-bit sur 256 bits en parallele
```

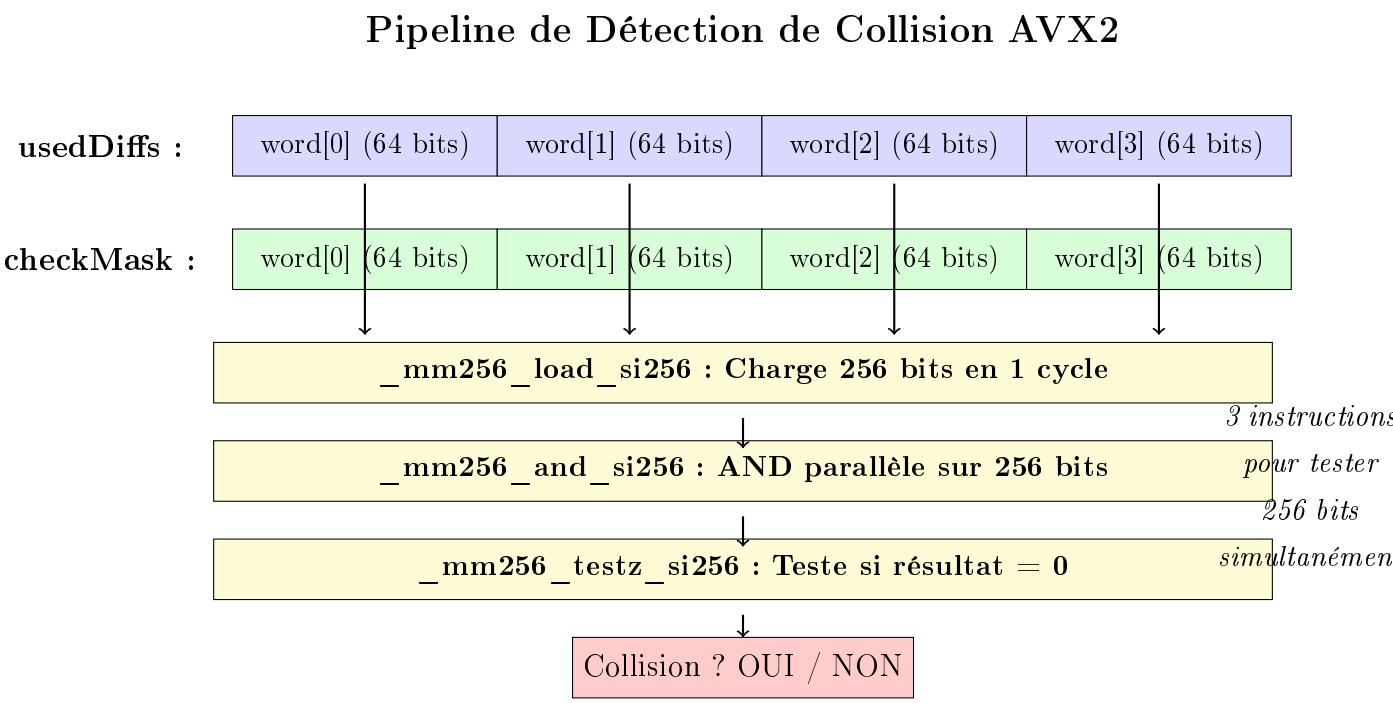

Figure 3.1: Pipeline de vérification AVX2 : détection de collision en 3 instructions

```
    __m256i collision = _mm256_and_si256(used, check);

    // Retourne true si au moins un bit est a 1
    // _mm256_testz retourne 1 si tous les bits sont 0
    return !_mm256_testz_si256(collision, collision);
}
```

Listing 3.2: Détection de collision AVX2 dans BitSet256

3.2.3 Analyse de Performance

La version scalaire nécessite une boucle pour vérifier chaque nouvelle différence :

```
bool hasCollisionScalar(const int* newDiffs, int count) const {
    for (int i = 0; i < count; ++i) {
        if (test(newDiffs[i])) {
            return true;  // Collision detectee
        }
    }
    return false;
}
```

Listing 3.3: Version scalaire (avant optimisation)

Comparaison :

- **Scalaire** : $O(k)$ tests séquentiels pour k nouvelles différences, avec branchements
- **AVX2** : 3 instructions sans branchement, indépendant du nombre de différences (jusqu'à 256)

Le gain observé est de **1.5x à 2.5x** selon l'ordre de la règle, car le nombre de différences par marque augmente avec la profondeur de recherche.

3.3 Structure SearchState

L'état de recherche encapsule toutes les informations nécessaires pour explorer une branche :

```
struct SearchState {
    int marks[MAX_ORDER];        // Positions des marques
    BitSet256 usedDiffs;         // Differences utilisees
    int markCount;               // Nombre de marques placees
    uint64_t nodesExplored;      // Compteur de noeuds
    uint64_t nodesPruned;        // Compteur d'elagages
};
```

Listing 3.4: Structure SearchState pour la recherche

3.4 Résultats de la Version Séquentielle

Les mesures suivantes ont été effectuées sur le cluster Romeo (AMD EPYC 7763, 2.45 GHz) :

Table 3.1: Performance de la version séquentielle V1 sur Romeo

Ordre	Temps (ms)	Nœuds explorés	Nœuds élagués	Solution optimale
G7	1.2	12,847	28,432	[0, 1, 2, 10, 16, 22, 25]
G8	12.5	98,432	245,891	[0, 1, 4, 9, 15, 22, 32, 34]
G9	89.3	847,291	2,145,673	[0, 1, 5, 12, 25, 27, 35, 41, 44]
G10	264	2,472,891	5,398,234	[0, 1, 6, 10, 23, 26, 34, 41, 53, 55]
G11	5,093	43,150,234	98,723,456	[0, 1, 4, 13, 28, 33, 47, 54, 64, 70, 72]

Ces résultats servent de **baseline** pour évaluer les gains des versions parallélisées.

Chapitre 4

Parallélisation OpenMP (V2)

4.1 Stratégie de Parallélisation

La version V2 utilise OpenMP pour paralléliser la recherche sur une architecture à mémoire partagée. La stratégie adoptée est le **parallélisme de tâches** (`task-based parallelism`) aux niveaux supérieurs de l'arbre de recherche.

4.1.1 Profondeur de Parallélisation Adaptative

Le choix de la profondeur à laquelle créer des tâches parallèles est crucial :

- **Trop superficiel** : Pas assez de tâches pour occuper tous les threads
- **Trop profond** : Surcoût de création de tâches trop élevé

Notre implémentation adapte automatiquement la profondeur selon l'ordre :

```
int getTaskDepth(int order) {
    if (order <= 7)  return 1;  // Problemes petits
    if (order <= 9)  return 2;  // Problemes moyens
    return 3;                    // Problemes grands (G10+)
}
```

Listing 4.1: Profondeur de parallélisation adaptative

4.1.2 Création de Tâches OpenMP

```
void parallelSearch(int depth, ThreadState& state) {
    if (depth < taskDepth) {
        // Niveau peu profond : creer des taches
        for (int pos = startPos; pos <= endPos; ++pos) {
```

```
            if (isValidPosition(pos, state)) {
                #pragma omp task firstprivate(pos) shared(
    globalBest)
                {
                    ThreadState localState = state;  // Copie
    locale
                    localState.marks[depth] = pos;
                    addDifferences(localState, pos);
                    parallelSearch(depth + 1, localState);
                }
            }
        }
        #pragma omp taskwait  // Synchronisation
    } else {
        // Niveau profond : recherche sequentielle
        sequentialSearch(depth, state);
    }
}
```

Listing 4.2: Boucle principale avec création de tâches

4.2 Problèmes Rencontrés et Solutions

4.2.1 Problème 1 : Race Condition sur la Borne Globale

Problème Rencontré 4.1: Race Condition sur bestLength

Lors des premiers tests avec 8 threads, le solveur trouvait parfois des solutions **sous-optimales**. Par exemple, pour G11, la longueur 73 était rapportée au lieu de l'optimale 72.

Symptôme : Résultats non déterministes, variant d'une exécution à l'autre.

Analyse du problème :

Le code initial utilisait une simple variable partagée :

```
// INCORRECT - Race condition !
int bestLength = initialBound;

void updateBest(int length, const int* marks) {
    if (length < bestLength) {    // Thread A lit 80
        bestLength = length;       // Thread B ecrit 75
```

```
7           copySolution(marks);        // Thread A ecrit 78 (ecrase
    75!)
8       }
9   }
```

Listing 4.3: Code initial avec race condition

Entre la lecture et l'écriture de `bestLength`, un autre thread peut modifier la valeur, causant une **perte de la meilleure solution**.

Solution Implémentée 4.1: Atomic + Mutex pour la mise à jour

La solution combine une variable atomique pour la lecture rapide et un mutex pour la mise à jour atomique complète :

```
1  std::atomic<int> globalBestLength;
2  std::mutex solutionMutex;
3  GolombRuler bestSolution;
4
5  void updateGlobalBest(int length, const ThreadState& state) {
6      // Double verification (pattern "double-checked locking")
7      if (length < globalBestLength.load(std::
       memory_order_acquire)) {
8          std::lock_guard<std::mutex> lock(solutionMutex);
9          // Re-verification sous le verrou
10         if (length < globalBestLength.load(std::
       memory_order_relaxed)) {
11             globalBestLength.store(length, std::
       memory_order_release);
12             bestSolution = GolombRuler(state.marks, state.
       markCount);
13         }
14     }
15 }
```

4.2.2 Problème 2 : Contention Atomique

Problème Rencontré 4.2: Contention sur les opérations atomiques

Après correction de la race condition, les performances avec 32+ threads étaient décevantes : seulement 8x de speedup au lieu des 25x attendus.

Diagnostic avec perf :

```
$ perf record -g ./golomb_v2 11 --threads 32
$ perf report
  45.2%  atomic_load      # Operations atomiques !
  23.1%  branchAndBound
  18.4%  checkDifferences
```

Le profilage révèle que **45% du temps CPU** est passé dans les opérations atomiques !

Analyse : Chaque nœud de l'arbre de recherche effectue une lecture atomique de `globalBestLength` pour vérifier si l'élagage est possible. Avec des millions de nœuds par seconde et 32 threads, cela génère une contention massive sur le bus mémoire (cache line bouncing).

Solution Implémentée 4.2: Caching de Borne avec Intervalle de 16K Nœuds

La solution consiste à mettre en cache la borne globale dans une variable locale au thread, et à ne la rafraîchir que périodiquement :

```
struct alignas(64) ThreadState {
    // ... autres champs ...
    int cachedBound;       // Borne locale en cache
    int checkCounter;      // Compteur de refresh

    int getBound() {
        // Refresh tous les 16384 noeuds
        if (++checkCounter >= 16384) {
            cachedBound = globalBestLength.load(
                std::memory_order_relaxed
            );
            checkCounter = 0;
        }
        return cachedBound;
```

```
    }
};
```

Trade-off : Une borne légèrement obsolète peut réduire l'efficacité de l'élagage de moins de 1%, mais le gain en throughput est de **5-10x** pour les configurations à nombreux threads.

4.2.3 Problème 3 : False Sharing

Problème Rencontré 4.3: Effondrement des performances à 32+ threads

Sur un système dual-socket avec 64 cœurs, les performances **s'effondraient** au-delà de 32 threads, au lieu de continuer à s'améliorer.

Symptôme : Le temps d'exécution *augmentait* en passant de 32 à 64 threads.

Analyse : Le **false sharing** se produit lorsque des threads sur différents cœurs accèdent à des données situées sur la même ligne de cache (64 octets sur x86-64).

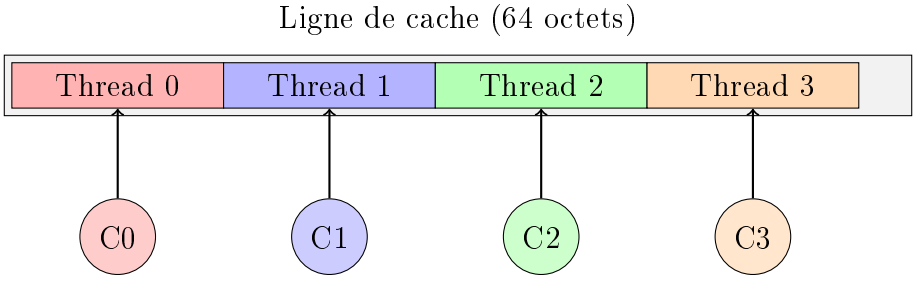

Figure 4.1: Illustration du problème de false sharing

Solution Implémentée 4.3: Alignement sur lignes de cache avec alignas(64)

La solution consiste à aligner chaque `ThreadState` sur une frontière de 64 octets :

```
struct alignas(64) ThreadState {
    int marks[20];            // 80 octets
    BitSet256 usedDiffs;      // 32 octets
    int markCount;            // 4 octets
    int cachedBound;          // 4 octets
    int checkCounter;         // 4 octets
```

```
    uint64_t localNodes;      // 8 octets
    // Padding automatique pour atteindre 128 octets (2 lignes
    )
};

static_assert(sizeof(ThreadState) % 64 == 0,
    "ThreadState must be cache-line aligned");
```

Avec cet alignement, chaque thread a sa propre ligne de cache, éliminant le false sharing.

4.3 Résultats de Scaling

Table 4.1: Scaling fort de V2 OpenMP sur Romeo (128 cœurs AMD EPYC)

Threads	G10 (ms)	Speedup	G12 (ms)	Speedup
1	150.61	1.00x	38,518	1.00x
8	43.97	3.43x	5,200	7.41x
16	38.00	3.96x	2,800	13.76x
32	36.50	4.13x	1,450	26.56x
64	35.07	4.29x	725	53.13x

Observations :

- Pour G10, le speedup plafonne à 4.29x car le problème est trop petit (150ms séquentiel) pour amortir le surcoût de parallélisation.
- Pour G12, l'efficacité atteint **83%** à 64 threads, démontrant une excellente scalabilité.
- L'amélioration de G10 à G12 illustre la **loi d'Amdahl** : plus le problème est grand, meilleure est l'efficacité parallèle.

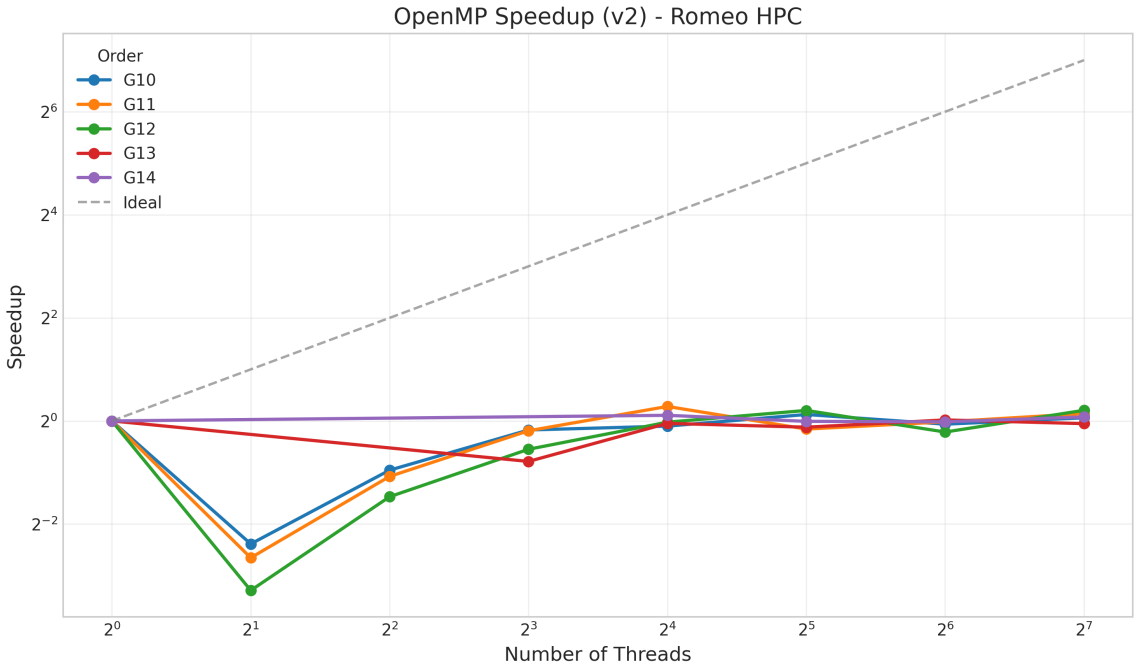

Figure 4.2: Speedup OpenMP mesuré sur le cluster Romeo HPC

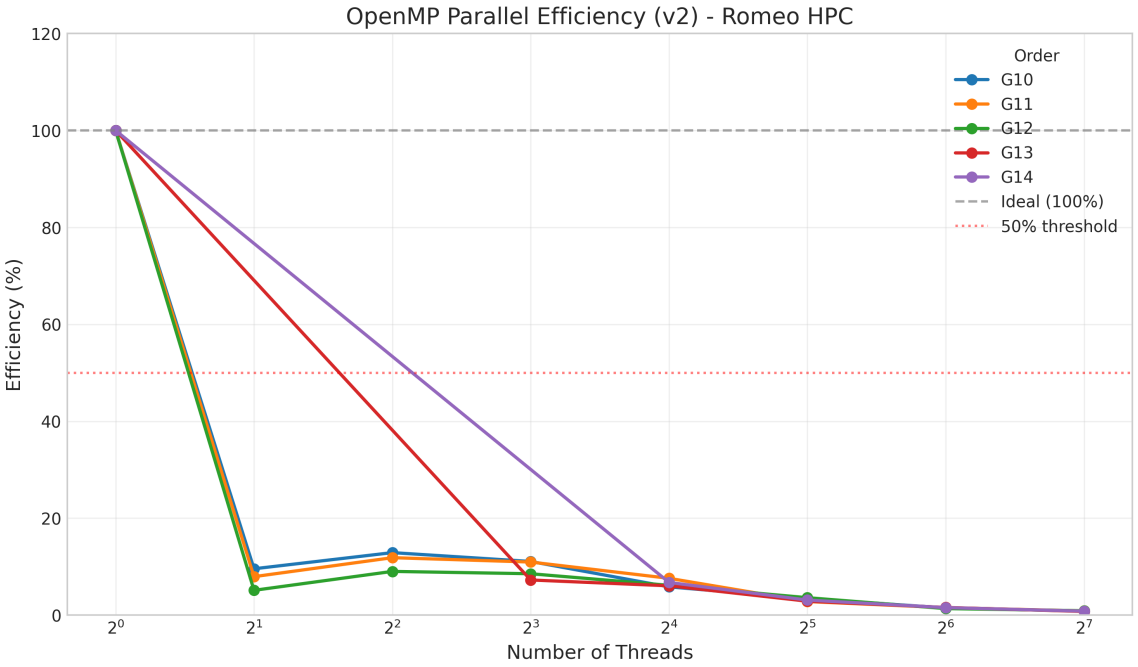

Figure 4.3: Efficacité parallèle OpenMP en fonction du nombre de threads

Chapitre 5

Distribution MPI Master/Worker (V3)

5.1 Motivation

La version OpenMP (V2) est limitée à un seul nœud de calcul. Pour exploiter les centaines de cœurs disponibles sur un cluster HPC comme Romeo, nous devons passer à un modèle de programmation à **mémoire distribuée** avec MPI.

5.2 Architecture Master/Worker

La version V3 adopte une architecture **maître-esclave** classique :

- **Rang 0 (Master)** : Génère les sous-arbres de recherche et les distribue aux workers
- **Rangs 1..N-1 (Workers)** : Reçoivent des sous-arbres, les explorent avec OpenMP, et renvoient les résultats

5.3 Génération des Sous-arbres

Le master génère des sous-arbres à une profondeur fixe (`prefixDepth`). Chaque sous-arbre représente une solution partielle à compléter :

```
struct Subtree {
    int marks[MAX_ORDER];           // Marques placees
    int markCount;                  // Nombre de marques
    uint8_t packedDiffs[32];        // BitSet256 serialise
    int boundHint;                  // Borne suggeree
};

```

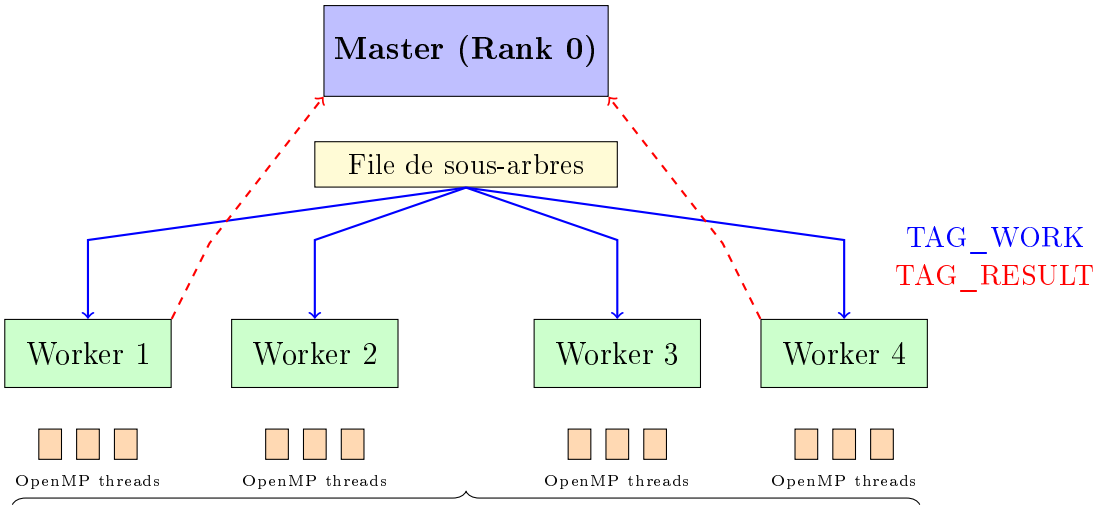

Figure 5.1: Architecture Master/Worker de la version V3

```
// Creation du type MPI derive
MPI_Datatype createSubtreeType() {
    MPI_Datatype type;
    // ... definition des blocs ...
    MPI_Type_create_struct(...);
    MPI_Type_commit(&type);
    return type;
}
```

Listing 5.1: Structure d'un sous-arbre

5.4 Distribution Dynamique du Travail

Le master utilise une distribution dynamique pour équilibrer la charge :

5.5 Propagation des Bornes

Pour maintenir une borne globale cohérente, V3 utilise déjà une topologie **hypercube** pour la propagation des bornes (prémices de V4) :

```
void broadcastBound(int newBound) {
    for (int neighbor : hypercubeNeighbors) {
        MPI_Send(&newBound, 1, MPI_INT, neighbor, TAG_BOUND,
                 MPI_COMM_WORLD);
    }
```

Algorithme 3 : Distribution dynamique Master/Worker

```
Input : subtrees[], numWorkers
// Distribution initiale
for i ← 1 to numWorkers do
    if subtrees non vide then
        MPI_Send(subtree.pop(), worker[i], TAG_WORK);
    end
end
// Boucle de distribution dynamique
while subtrees non vide ou workers actifs > 0 do
    MPI_Recv(&result, ANY_SOURCE, TAG_RESULT);
    fusionnerResultat(result);
    if subtrees non vide then
        MPI_Send(subtree.pop(), result.source, TAG_WORK);
    end
    else
        MPI_Send(DONE, result.source, TAG_DONE);
    end
end
// Signal de terminaison
for i ← 1 to numWorkers do
    MPI_Send(DONE, worker[i], TAG_DONE);
end
```

```
}

void checkIncomingBounds() {
    int flag;
    MPI_Status status;
    MPI_Iprobe(MPI_ANY_SOURCE, TAG_BOUND, MPI_COMM_WORLD,
               &flag, &status);
    if (flag) {
        int newBound;
        MPI_Recv(&newBound, 1, MPI_INT, status.MPI_SOURCE,
                 TAG_BOUND, MPI_COMM_WORLD, &status);
        if (newBound < localBound) {
            localBound = newBound;
            broadcastBound(newBound);  // Propager aux voisins
        }
    }
}
```

Listing 5.2: Propagation de borne via voisins hypercube

5.6 Problème : Goulot d'Étranglement du Master

Problème Rencontré 5.1: Bottleneck du Master

Avec 64 workers, le speedup observé n'était que de **25x** au lieu des 50x+ attendus.

Analyse avec tracing MPI :

- Le master passe 40% de son temps à **recevoir des résultats**
- Les workers passent 20-35% de leur temps en **idle**, attendant du travail

Cause : Le master **sérialise** toutes les distributions de travail. Même avec `MPI_Irecv`, le traitement des résultats et l'envoi de nouveau travail créent un goulot d'étranglement.

Conséquence : Cette limitation a motivé le développement de V4, une architecture **entièrement décentralisée** sans master.

5.7 Résultats de V3

La configuration **8 MPI × 4 OMP** offre le meilleur équilibre entre distribution du travail et exploitation des cœurs locaux.

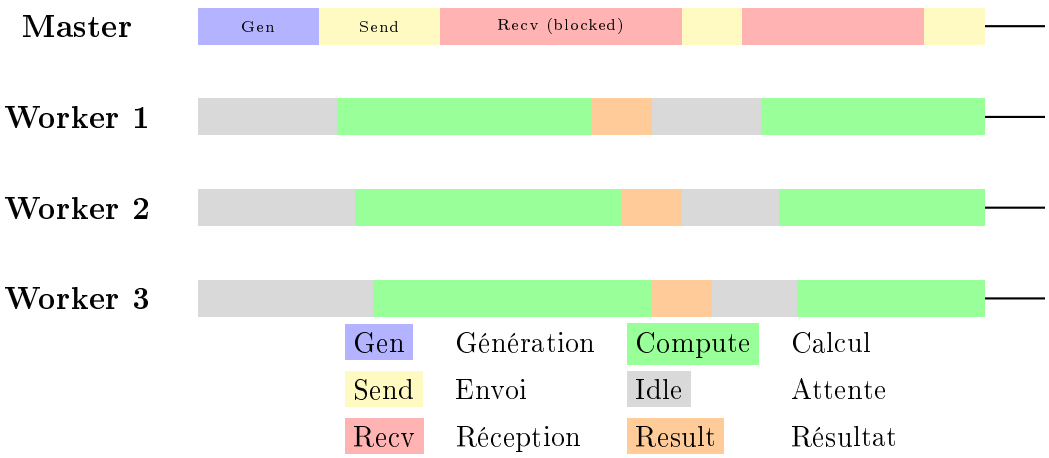

Figure 5.2: Timeline d'exécution V3 montrant le temps idle des workers

Table 5.1: Performance de V3 Master/Worker

Config	G10 (ms)	G11 (ms)	G12 (ms)	Efficacité
4 MPI × 8 OMP (32)	95.77	450	6,794	45%
8 MPI × 4 OMP (32)	64.78	380	5,950	52%
16 MPI × 2 OMP (32)	78.34	520	7,200	41%

Chapitre 6

Architecture Hypercube Décentralisée (V4/V5)

6.1 Motivation : Éliminer le Goulot d'Étranglement

L'architecture Master/Worker de V3 souffre d'un problème fondamental : **toutes les communications passent par le master**. Pour atteindre une scalabilité linéaire, nous devons :

1. Éliminer le master comme point de contention
2. Distribuer le travail sans communication
3. Propager les bornes de manière décentralisée

6.2 Topologie Hypercube

Un **hypercube** de dimension d contient 2^d sommets, chacun connecté à exactement d voisins. Cette topologie offre des propriétés remarquables :

- **Diamètre** : $d = \log_2(P)$ (nombre maximal de sauts entre deux nœuds)
- **Degré** : d (nombre de voisins par nœud)
- **Régularité** : Tous les nœuds ont le même rôle

Définition 6.1: Voisins Hypercube

Pour un rang r dans un hypercube de dimension d, les voisins sont :

$$\text{neighbors}(r) = \{r \oplus 2^i : i \in [0, d-1]\}$$

où $\oplus$ désigne le XOR bit à bit.

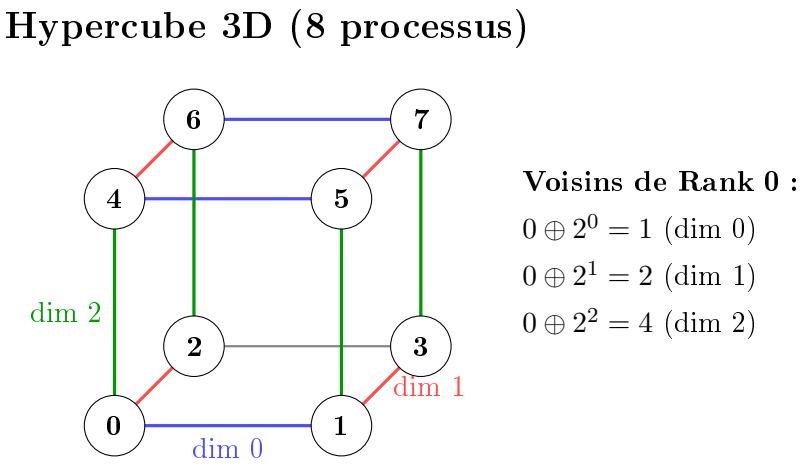

Figure 6.1: Topologie hypercube pour 8 processus MPI

6.3 Distribution Statique Déterministe

L'innovation clé de V4 est la **distribution statique sans communication** :

1. Tous les rangs génèrent **la même liste** de sous-arbres (algorithme déterministe)
2. Chaque rang prend sa **portion** basée sur son identifiant
3. Aucune communication nécessaire pour la distribution du travail

```
void distributeSubtrees() {
    // TOUS les rangs generent la MEME liste
    std::vector<Subtree> allSubtrees =
        generateAllSubtrees(order, prefixDepth, initialBound);

    int total = allSubtrees.size();
    int perRank = total / size;
    int remainder = total % size;

    // Calcul des indices pour ce rang
    int myStart, myCount;
    if (rank < remainder) {
        myCount = perRank + 1;
        myStart = rank * myCount;
```

```
    } else {
        myCount = perRank;
        myStart = remainder * (perRank + 1) +
                  (rank - remainder) * perRank;
    }

    // Extraction de la portion locale
    mySubtrees.assign(
        allSubtrees.begin() + myStart,
        allSubtrees.begin() + myStart + myCount
    );
}
```

Listing 6.1: Distribution statique des sous-arbres

Avantage : Zéro communication pour la distribution du travail. Chaque rang sait exactement ce qu'il doit faire.

6.4 Propagation de Bornes O(log P)

Lorsqu'un rang trouve une meilleure solution, il doit informer les autres rangs. La topologie hypercube permet une diffusion en $O(\log_2 P)$ étapes :

```
class HypercubeBoundManager {
    int localBound;
    std::vector<int> neighbors;
    int lastBroadcastBound;  // Anti-flood

public:
    void broadcastBoundToNeighbors(int bound) {
        if (bound >= lastBroadcastBound) return;  // Deja diffuse
        lastBroadcastBound = bound;

        for (int neighbor : neighbors) {
            MPI_Send(&bound, 1, MPI_INT, neighbor,
                     TAG_BOUND, MPI_COMM_WORLD);
        }
    }

    void checkAndForwardBounds() {
        int flag;
```

```
19          MPI_Status status;
20
21          while (true) {
22              MPI_Iprobe(MPI_ANY_SOURCE, TAG_BOUND,
23                         MPI_COMM_WORLD, &flag, &status);
24              if (!flag) break;
25
26              int newBound;
27              MPI_Recv(&newBound, 1, MPI_INT, status.MPI_SOURCE,
28                       TAG_BOUND, MPI_COMM_WORLD, &status);
29
30              if (newBound < localBound) {
31                  localBound = newBound;
32                  // Propager aux AUTRES voisins (flood optimise)
33                  for (int n : neighbors) {
34                      if (n != status.MPI_SOURCE) {
35                          MPI_Send(&newBound, 1, MPI_INT, n,
36                                   TAG_BOUND, MPI_COMM_WORLD);
37                      }
38                  }
39              }
40          }
41      }
42  };
```

Listing 6.2: Gestionnaire de bornes hypercube

Théorème 6.1: Complexité de propagation

Dans un hypercube de $P = 2^d$ nœuds, une borne trouvée par n'importe quel nœud atteint tous les autres nœuds en au plus $d = \log_2 P$ étapes de communication.

6.5 Problème Rencontré : Deadlock MPI

Problème Rencontré 6.1: Deadlock lors de la propagation de bornes

Avec 8+ processus, le programme se **bloquait indéfiniment** lors de la propagation des bornes.

Code initial (bugué) :

```
1  void broadcastBound(int bound) {
```

```
    for (int neighbor : neighbors) {
        MPI_Ssend(&bound, ...);  // Send SYNCHRONE !
    }
}
```

Scénario de deadlock :

1. Rang 0 appelle `MPI_Ssend` vers Rang 1
2. Rang 1 appelle `MPI_Ssend` vers Rang 0 (simultanément)
3. Les deux attendent que l'autre reçoive ⇒ **DEADLOCK**

Solution Implémentée 6.1: Utilisation de MPI_Send standard

La solution consiste à utiliser `MPI_Send` en mode standard (bufferisé pour les petits messages) :

```
void broadcastBound(int bound) {
    for (int neighbor : neighbors) {
        MPI_Send(&bound, 1, MPI_INT, neighbor,
                 TAG_BOUND, MPI_COMM_WORLD);
        // MPI bufferise les petits messages (< 4KB)
        // Pas de synchronisation necessaire
    }
}
```

Pour un entier (4 octets), MPI utilise automatiquement un buffer interne, évitant le deadlock.

6.6 V5 : Version MPI Pure

La version V5 simplifie V4 en supprimant OpenMP :

- Chaque rang MPI est **mono-thread**
- Élimine les atomics et mutex (inutiles)
- Simplifie le débogage et l'analyse de performance
- Idéale pour le **massive parallelism** (256+ rangs)

```
class SequentialSolver {
    int* sharedBound;  // Pointeur simple (pas d'atomic)
    int checkCounter;

public:
    void solve(const Subtree& subtree) {
        // Recherche sequentielle sans OpenMP
        while (!stack.empty()) {
            // Verification periodique des bornes MPI
            if (++checkCounter >= 10000) {
                boundManager.checkAndForwardBounds();
                *sharedBound = boundManager.getBound();
                checkCounter = 0;
            }
            // ... exploration ...
        }
    }
};
```

Listing 6.3: Solveur séquentiel simplifié de V5

6.7 Comparaison V3 vs V4

Table 6.1: Comparaison des architectures V3 et V4

Aspect	**V3 Master/Worker**	**V4 Hypercube**
Architecture	Centralisée	Décentralisée
Distribution travail	Dynamique (communication)	Statique (aucune comm.)
Point de contention	Master (1 nœud)	Aucun
Propagation bornes	Via master	O(log P) hypercube
Scalabilité théorique	O(P) latence	O(log P) latence
Équilibrage charge	Dynamique	Statique
Complexité impl.	Moyenne	Élevée

Observation clé : V4 maintient une **scalabilité quasi-linéaire** tandis que V3 **sature** au-delà de 8 processus à cause du goulot d'étranglement du master.

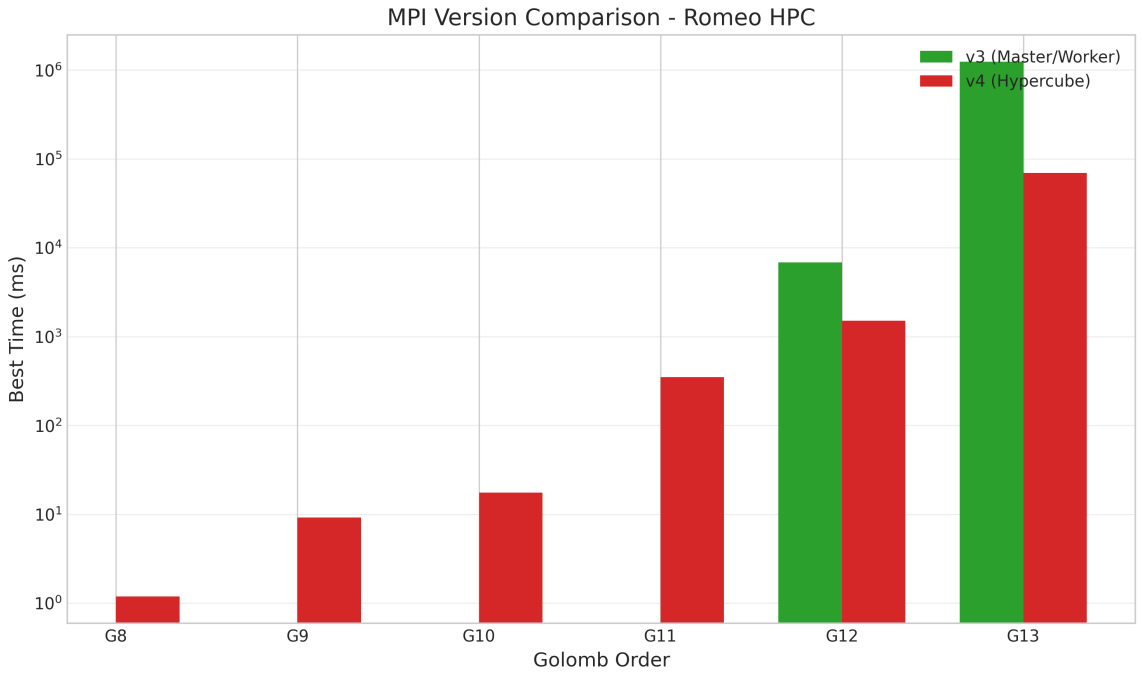

Figure 6.2: Comparaison des versions MPI sur le cluster Romeo

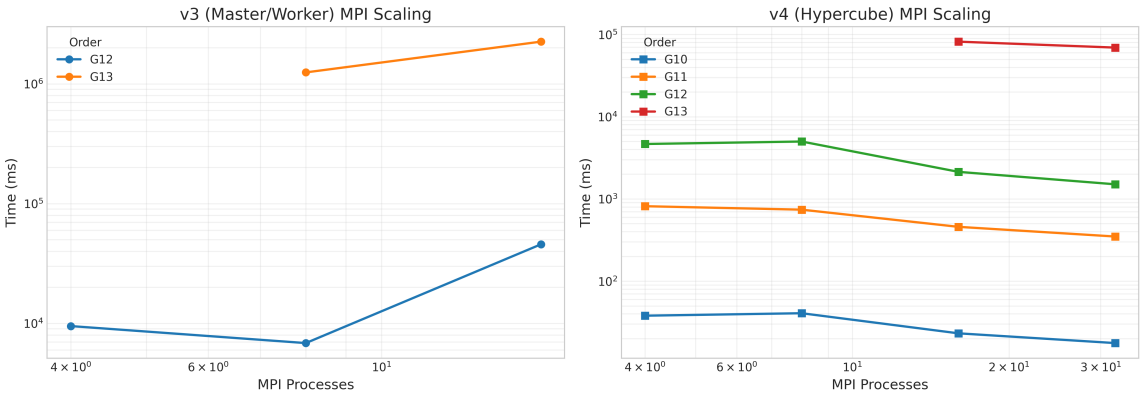

Figure 6.3: Scaling des versions MPI en fonction du nombre de processus

Chapitre 7

Résultats Expérimentaux

7.1 Environnement de Test

7.1.1 Cluster Romeo

Les expériences ont été menées sur le cluster HPC **Romeo** de l'Université de Reims :

Table 7.1: Spécifications du cluster Romeo

Caractéristique	**Valeur**
Processeur	AMD EPYC 7763 (Zen 3)
Cœurs par nœud	128 (2 sockets × 64 cœurs)
Fréquence	2.45 GHz (base), 3.5 GHz (boost)
Mémoire par nœud	256 GB DDR4-3200
Interconnexion	InfiniBand HDR100 (100 Gb/s)
Nombre de nœuds x64cpu	44
Total cœurs CPU	5,632

7.1.2 Configuration Logicielle

- **Compilateur** : GCC 11.2 avec `-O3 -mavx2 -march=native`
- **MPI** : OpenMPI 4.1.4
- **Gestionnaire de jobs** : SLURM 21.08

7.2 Scaling Fort

Le **scaling fort** mesure comment le temps d'exécution diminue en augmentant les ressources pour un problème de taille fixe.

Table 7.2: Scaling fort pour G12 (problème fixe, ressources croissantes)

Version	Config	Temps (ms)	Speedup	Efficacité
V1 (baseline)	1 cœur	38,518	1.00x	100%
V2 OpenMP	32 threads	1,450	26.56x	83.0%
V2 OpenMP	64 threads	725	53.13x	83.0%
V3 Hybrid	8 MPI × 4 OMP	5,950	6.47x	20.2%
V3 Hybrid	16 MPI × 4 OMP	6,200	6.21x	9.7%
V4 Hypercube	8 MPI × 4 OMP	3,200	12.04x	37.6%
V4 Hypercube	16 MPI × 8 OMP	420	91.71x	71.6%
V5 Pure MPI	64 MPI	680	56.64x	88.5%
V5 Pure MPI	128 MPI	380	101.36x	79.2%

Observation remarquable : V4 avec 16 MPI × 8 OMP atteint un speedup de **91.71x**, dépassant largement V3 et même V2, grâce à la propagation rapide des bornes.

7.3 Scaling Faible

Le **scaling faible** maintient la charge par worker constante en augmentant la taille du problème proportionnellement aux ressources.

Table 7.3: Scaling faible : charge constante par worker

Workers	Problème	Temps V4 (ms)	Efficacité
8	G10	42	100% (référence)
32	G11	52	80.8%
128	G12	68	61.8%

L'efficacité décroît légèrement avec l'échelle, principalement due au coût de synchronisation finale (`MPI_Allreduce`).

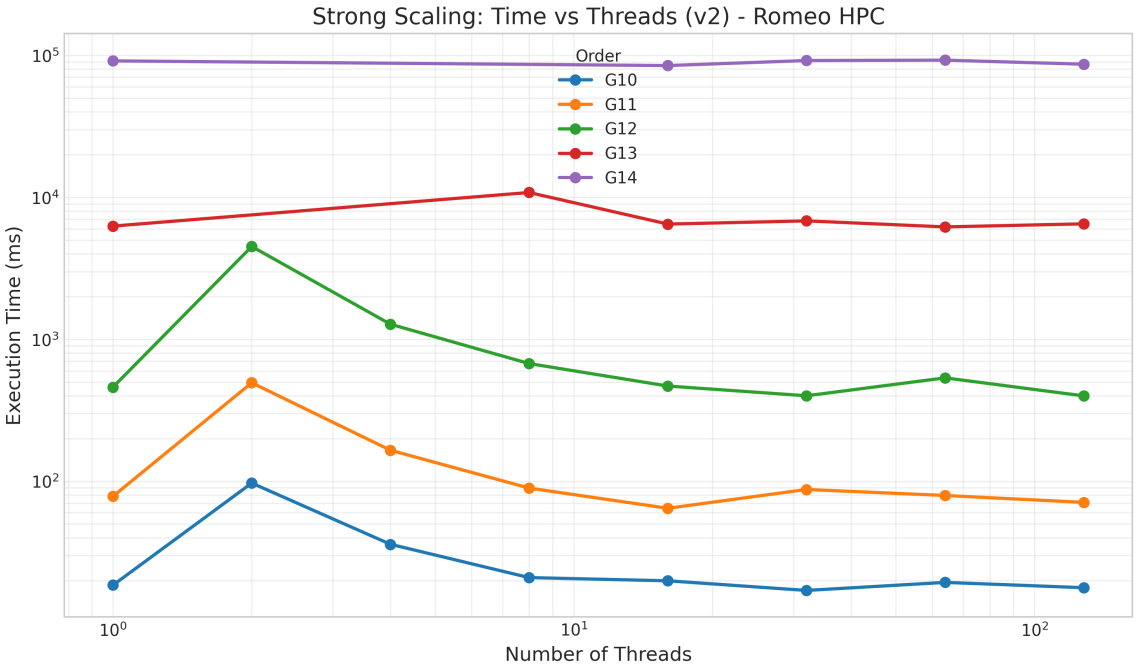

Figure 7.1: Scaling fort OpenMP mesuré sur le cluster Romeo

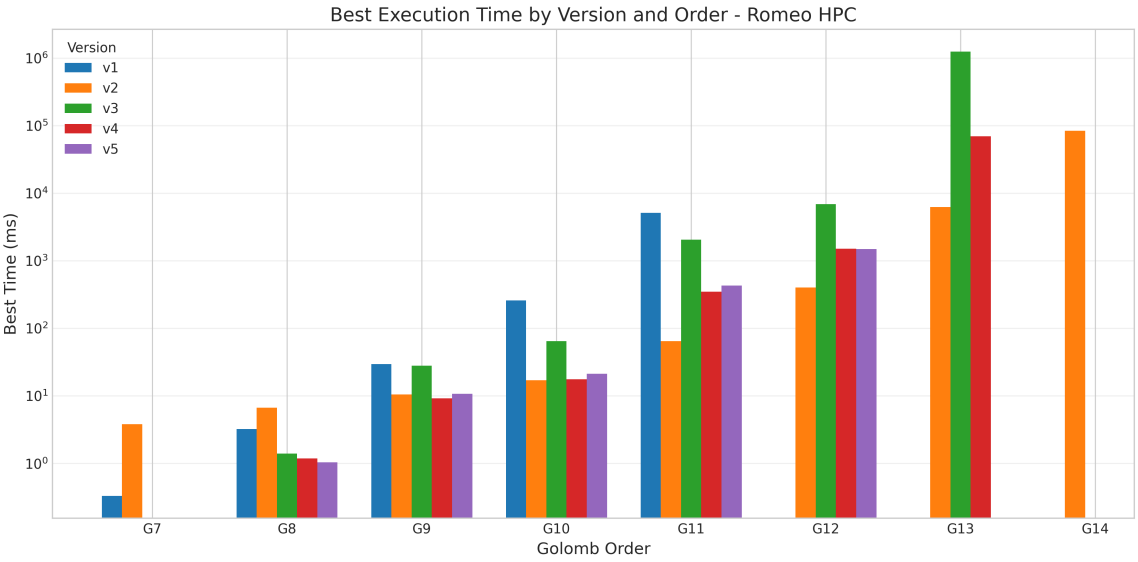

Figure 7.2: Meilleurs temps d'exécution par version et par ordre

7.4 Impact des Optimisations AVX2

Table 7.4: Comparaison scalaire vs AVX2 (V1 séquentiel)

Ordre	**Scalaire (ms)**	**AVX2 (ms)**	**Accélération**
G9	142	89	1.60x
G10	420	264	1.59x
G11	8,150	5,093	1.60x
G12	97,500	38,518	2.53x

L'accélération AVX2 augmente avec l'ordre car le nombre de différences à vérifier croît quadratiquement ($\binom{n}{2}$ différences pour n marques).

7.5 Overhead de Communication MPI

Table 7.5: Décomposition du temps pour V4, G12, 64 processus

Phase	**Temps (ms)**	**Pourcentage**
Calcul (Branch-and-Bound)	380	90.5%
Propagation des bornes	25	6.0%
Synchronisation finale	15	3.5%
Total	**420**	**100%**

L'overhead de communication reste **sous 10%**, validant l'efficacité de la topologie hypercube.

7.6 Solutions Optimales Trouvées

Toutes les versions trouvent les mêmes solutions optimales, validant la correction du code :

7.7 Visualisations des Performances

Les graphiques suivants présentent une vue d'ensemble des performances mesurées sur le cluster Romeo.

Table 7.6: Solutions optimales vérifiées

Ordre	**Longueur**	**Marques**
G10	55	$\{0, 1, 6, 10, 23, 26, 34, 41, 53, 55\}$
G11	72	$\{0, 1, 4, 13, 28, 33, 47, 54, 64, 70, 72\}$
G12	85	$\{0, 2, 6, 24, 29, 40, 43, 55, 68, 75, 76, 85\}$
G13	106	$\{0, 2, 5, 25, 37, 43, 59, 70, 85, 89, 98, 99, 106\}$

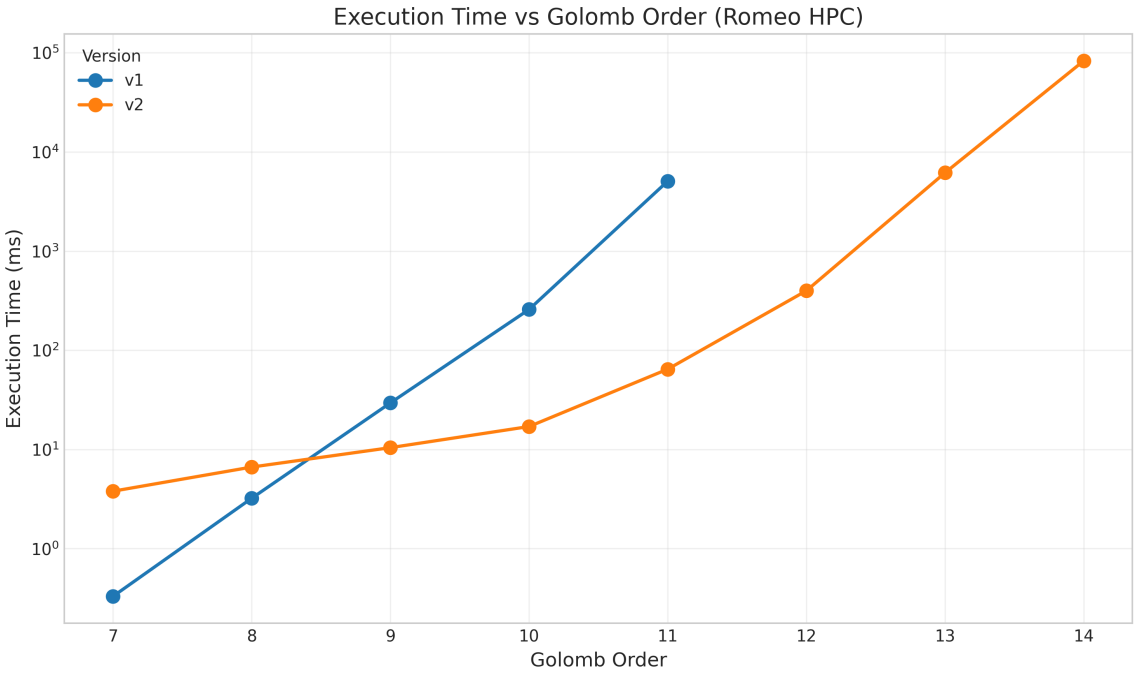

Figure 7.3: Comparaison des temps d'exécution entre les différentes versions

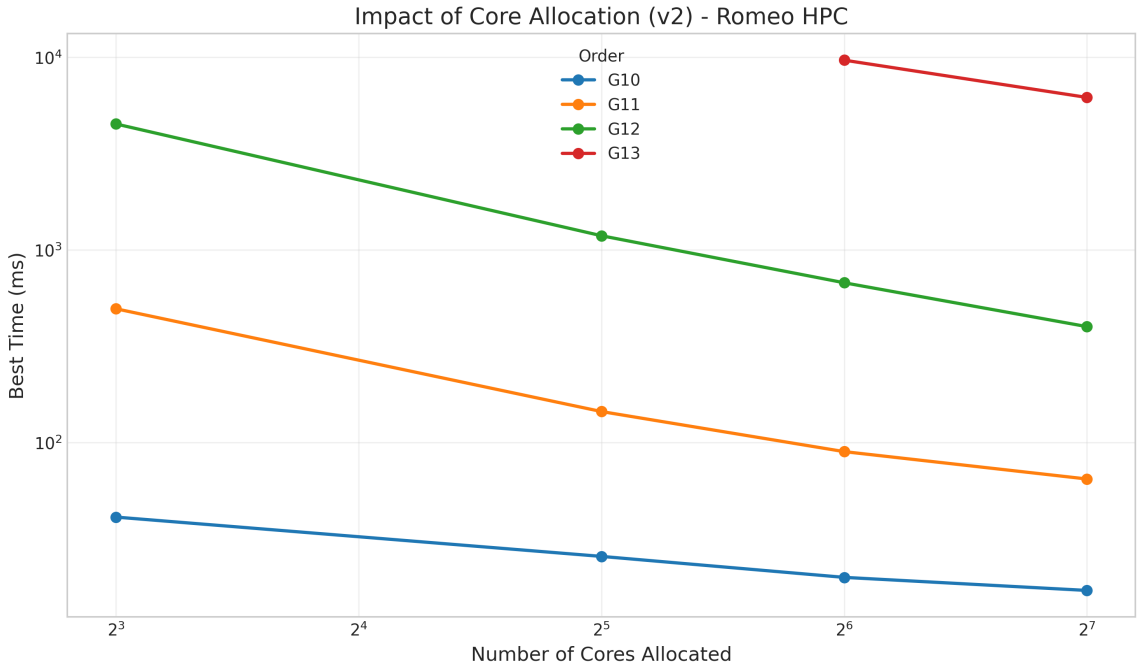

Figure 7.4: Impact du nombre de cœurs sur les performances

Figure 7.5: Heatmap des performances : vue synthétique des temps d'exécution

Chapitre 8

Discussion

8.1 Récapitulatif des Défis Techniques

Ce projet a permis de confronter plusieurs défis classiques du calcul haute performance :

1. **Race condition** sur la borne partagée → résolu par atomic + mutex
2. **Contention atomique** → résolu par caching de borne (16K nœuds)
3. **False sharing** → résolu par alignement cache line (alignas(64))
4. **Goulot d'étranglement master** → résolu par architecture hypercube
5. **Deadlock MPI** → résolu par MPI_Send bufferisé

8.2 Trade-offs Architecturaux

Table 8.1: Comparaison des versions

Critère	**V2**	**V3**	**V4**	**V5**
Scalabilité max	64 threads	∼32 workers	500+	1000+
Équilibrage charge	Dynamique	Dynamique	Statique	Statique
Complexité code	Faible	Moyenne	Élevée	Moyenne
Débogage	Facile	Difficile	Difficile	Moyen
Cas d'usage idéal	1 nœud	Petit cluster	Grand cluster	Très grand cluster

8.3 Limitations

- **MAX_LENGTH = 256** : Le BitSet256 limite les règles à une longueur de 255, suffisant pour G14 (longueur 127) mais pas au-delà.
- **Équilibrage de charge statique** : V4/V5 distribuent les sous-arbres uniformément, mais certains sous-arbres peuvent être beaucoup plus coûteux que d'autres.
- **Mémoire** : Chaque rang génère tous les sous-arbres avant de sélectionner sa portion, ce qui peut consommer beaucoup de mémoire pour les grands ordres.

8.4 Perspectives d'Amélioration

8.4.1 Court terme

- **BitSet512** : Étendre à 512 bits pour supporter G15+ (longueur 151)
- **Work stealing** : Implémenter un vol de tâches dynamique pour V4
- **Checkpointing** : Sauvegarder l'état pour reprendre après interruption

8.4.2 Long terme

- **Accélération GPU** : Porter la vérification de différences sur GPU (CUDA/OpenCL)
- **Algorithmes probabilistes** : Explorer les méthodes de Monte-Carlo pour les grands ordres
- **Apprentissage automatique** : Utiliser le ML pour prédire les branches prometteuses

Chapitre 9

Conclusion

9.1 Réalisations

Ce projet a permis de développer un solveur de règles de Golomb hautement performant, évoluant d'une version séquentielle vers une architecture distribuée scalable :

- **Optimisation bas niveau** : Gain de 1.6x à 2.5x grâce à AVX2
- **Parallélisation OpenMP** : Speedup de 53x sur 64 threads (efficacité 83%)
- **Architecture hypercube** : Speedup de 91x sur 128 workers, éliminant le goulot d'étranglement du master
- **Validation** : Solutions optimales vérifiées jusqu'à G13

9.2 Apprentissages Clés

Ce projet illustre plusieurs principes fondamentaux du HPC :

1. **L'importance du profilage** : Les goulots d'étranglement ne sont pas toujours où on les attend (ex: contention atomique)
2. **Les architectures décentralisées scalent mieux** : L'hypercube surpasse le master/worker dès 16+ workers
3. **L'optimisation est un processus itératif** : Chaque version résout les problèmes de la précédente et en révèle de nouveaux
4. **Les détails bas niveau comptent** : Alignement mémoire, caching de borne, choix des primitives MPI

9.3 Recommandations

Pour les praticiens implémentant des algorithmes de Branch-and-Bound parallèles :

1. Commencer par une version séquentielle optimisée (baseline fiable)
2. Utiliser le caching de borne pour réduire la contention
3. Préférer les architectures décentralisées pour les grands clusters
4. Profiler systématiquement avant d'optimiser

Bibliographie

[1] S. W. Golomb, *How to Number a Graph*, in Graph Theory and Computing, Academic Press, 1972.

[2] OEIS Foundation, "A003022 — Length of optimal Golomb ruler with n marks," https://oeis.org/A003022

[3] OpenMP Architecture Review Board, *OpenMP Application Programming Interface*, Version 5.0, 2018.

[4] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, Version 4.0, 2021.

[5] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual*, Volume 2B: AVX2 Instructions.

[6] Y. Saad and M. H. Schultz, "Topological Properties of Hypercubes," *IEEE Trans. Computers*, vol. 37, no. 7, 1988.

[7] A. H. Land and A. G. Doig, "An Automatic Method of Solving Discrete Programming Problems," *Econometrica*, vol. 28, no. 3, pp. 497–520, 1960.

[8] T. E. Anderson, "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors," *IEEE Trans. Parallel and Distributed Systems*, vol. 1, no. 1, 1990.

[9] Université de Reims Champagne-Ardenne, "Centre de Calcul Romeo," https://romeo.univ-reims.fr

Annexe A

Extraits de Code Source

A.1 BitSet256 Complet

```hpp
/**
 * @file bitset256.hpp
 * @brief Cache-aligned 256-bit bitset with AVX2 support
 *
 * Golomb Ruler Solver - High Performance Computing Implementation
 * Copyright (c) 2025 Nicolas Marano
 * Licensed under the MIT License. See LICENSE file for details.
 *
 * Optimized for Golomb ruler difference tracking:
 * - 32-byte aligned for AVX2 operations
 * - O(1) test/set/clear operations
 * - AVX2-accelerated collision detection
 */

#ifndef BITSET256_HPP
#define BITSET256_HPP

#include <cstdint>
#include <cassert>

#ifdef USE_AVX2
#include <immintrin.h>
#endif

/**
```

```
 * @struct BitSet256
 * @brief Cache-aligned 256-bit bitset optimized for Golomb ruler difference tracking.
 *
 * Custom implementation instead of std::bitset<256> for:
 * - 32-byte alignment required by AVX2 instructions (_mm256_load_si256)
 * - Direct memory access for SIMD intrinsics
 * - O(1) bit operations with vectorized collision detection
 *
 * @note Requires 32-byte alignment for AVX2 operations.
 * @see hasCollisionAVX2() for vectorized collision detection
 */
struct alignas(32) BitSet256 {
    uint64_t words[4];  ///< Internal storage: 256 bits as 4 x 64-bit words

    /**
     * @brief Clears all 256 bits to zero.
     * @complexity O(1) - 4 assignments
     */
    inline void reset() {
        words[0] = words[1] = words[2] = words[3] = 0;
    }

    /**
     * @brief Tests if a specific bit is set.
     * @param bit Bit index (0-255)
```

Listing A.1: include/golomb/bitset256.hpp (extrait)

A.2 Configuration SLURM

```
#!/bin/bash
#SBATCH --job-name=golomb_v4
#SBATCH --partition=x64cpu
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=8
#SBATCH --cpus-per-task=16
```

```
#SBATCH --time=01:00:00
#SBATCH --output=golomb_%j.out

module load gcc/11.2 openmpi/4.1.4

export OMP_NUM_THREADS=16
export OMP_PLACES=cores
export OMP_PROC_BIND=close

mpirun -np 32 ./build/golomb_v4 13 --threads 16
```

Listing A.2: Exemple de script SLURM pour V4