Direct Meet-in-the-Middle Attack

Julian Mouthon, Mario Razafinony

29 décembre 2025

Résumé

Ce rapport présente la parallèlisation d'un code d'attaque *Meet-in-the-Middle* (MitM) en utilisant MPI et OpenMP.

Table des matières

1 Introduction

L'attaque *Meet-in-the-Middle* est une technique qui exploite la possibilité de diviser un chiffrement en deux parties indépendantes. Le but est de chercher des paires de clés (k_1, k_2) telles que :

$$\mathrm{Enc}_{k_1}(\mathrm{Enc}_{k_2}(P_1)) = C_1 \quad \text{et} \quad \mathrm{Enc}_{k_1}(P_0) = \mathrm{Dec}_{k_2}(C_0)$$

où (P_0, C_0) et (P_1, C_1) sont deux paires texte clair-chiffré connues.

2 Parallèlisation

Pour maximiser les performances, nous combinons plusieurs niveaux de parallélisme :

— **MPI** : parallélisme multi-processus
— **OpenMP** : parallélisme multi-thread
— **Sharding** : distribution du dictionnaire entre processus
— **Vectorisation** : accélération de l'initialisation des dictionnaires locaux

2.1 Dictionnaire Partagé

Dans la version séquentielle, toutes les valeurs $f(x)$ étaient stockées dans un dictionnaire global unique. Ici, ce dictionnaire est distribué (*shardé*) entre les processus.

Le processus propriétaire d'une clé intermédiaire z est déterminé par :

$$\text{shard}(z) = z \bmod P$$

où P est le nombre total de processus.

Chaque processus maintient uniquement son dictionnaire local, ce qui permet :
— de réduire la mémoire utilisée par processus
— d'améliorer la localité mémoire
— de paralléliser les insertions et les recherches

Nous avons ajouté trois fonctions pour gérer le dictionnaire shardé :
— `shard_dict_setup` pour l'initialisation,
— `shard_dict_insert` pour l'insertion,
— `shard_probe_local` pour la recherche locale.

2.2 MPI

Chaque processus MPI est responsable :
— d'une fraction de l'espace de recherche des clés,
— d'un **shard** du dictionnaire global
— du traitement local des collisions correspondant à son shard.

La répartition ne dépend pas d'un processus maître, ce qui évite les goulots d'étranglement.

2.2.1 Découpage de l'espace de recherche

L'espace des clés $\{0,1\}^n$ est réparti entre les processus MPI selon leur rang :

$$x \equiv \text{rank} \pmod{\text{P}}$$

Ainsi, chaque processus traite exactement $\frac{2^n}{P}$ clés, où P est le nombre total de processus.
Ce découpage permet :
— une charge de travail équilibrée,
— l'absence de recouvrement entre processus,
— aucune synchronisation nécessaire pendant le calcul local.

2.2.2 Phase 1 : Construction distribuée du dictionnaire

Chaque processus calcule $z = f(x)$ pour ses valeurs locales de x :
— si le processus courant est propriétaire de z, l'entrée (z, x) est insérée directement dans le dictionnaire local ;
— sinon, la paire est stockée dans un buffer temporaire destinée au processus propriétaire.
Les communications sont ensuite regroupées :
— échange des tailles à l'aide de `MPI_Alltoall`,
— envois et réceptions asynchrones (`MPI_Isend` / `MPI_Irecv`),
— insertion locale après réception.
Ceci permet d'éviter un envoi MPI à chaque itération, ce qui aurait été extrêmement coûteux.

2.2.3 Phase 2 : Recherche distribuée des collisions

— chaque processus calcule $y = g(z)$ pour ses valeurs de z,
— si le shard correspondant est local, on peut chercher dans le dictionnaire directement,
— sinon, le couple (y, z) est envoyé au processus propriétaire.
Les collisions candidates sont ensuite validées localement avec le second couple clair–chiffré (P_1, C_1).

2.2.4 Phase 3 : Agrégation des résultats

Chaque processus conserve uniquement ses résultats locaux. À la fin de l'exécution :
— le nombre de solutions locales est collecté avec `MPI_Gather`,
— les clés (k_1, k_2) sont rassemblées sur le processus 0 via `MPI_Gatherv`.
Cette étape est peu coûteuse car le nombre de solutions est très faible comparé à la taille de l'espace de recherche.

2.3 OpenMP

À l'intérieur de chaque processus, le parallélisme est renforcé à l'aide d'OpenMP. L'objectif est d'exploiter le parallélisme multi-cœur local tout en gardant une compatibilité avec les communications MPI.

Les boucles principales des phases 1 et 2 sont parallélisées à l'aide de `#pragma omp parallel for` avec un ordonnancement statique. Chaque thread traite ainsi une partie disjointe des clés locales, sans recouvrement.

Chaque thread dispose de buffers privés pour stocker temporairement les paires clé–valeur destinées aux autres processus MPI. Ces buffers locaux sont ensuite fusionnés dans des buffers globaux dans des sections critiques, avant les phases de communication MPI.

Les insertions dans le dictionnaire shardé sont sûres : elles sont effectuées soit par un seul thread après la réception MPI, soit en parallèle sur des clés distinctes lors de la construction locale. De même, chaque thread valide les collisions localement, et les résultats sont stockés dans des tableaux partagés protégés par des sections critiques pour éviter toute condition de concurrence.

2.4 Vectorisation

La vectorisation est utilisée lors de l'initialisation des dictionnaires locaux (`shard_dict_setup`). Grâce à la directive `#pragma omp simd`, plusieurs entrées du dictionnaire sont initialisées en parallèle, ce qui accélère la phase de préparation.

3 Algorithme principal (version simplifiée)

Algorithm 1 Algorithme golden_claw_search

```
1: procedure GOLDEN_CLAW_SEARCH(maxres, k1[], k2[])
2:     Phase 1 : Construction du dictionnaire
3:     for x tel que x ≡ rank (mod world_size) do
4:         z ← f(x)
5:         shard ← z mod world_size
6:         if shard = rank then
7:             SHARD_DICT_INSERT(z, x)
8:         else
9:             Ajouter (z, x) dans un buffer pour les envoyer plus tard shard
10:        end if
11:    end for
12:    Échanger les buffers entre tous les processus
13:    Phase 2 : Recherche de collisions
14:    for z tel que z ≡ rank (mod world_size) do
15:        y ← g(z)
16:        shard ← y mod world_size
17:        if shard = rank then
18:            X ← SHARD_PROBE_LOCAL(y)
19:            for x ∈ X do
20:                if IS_GOOD_PAIR(x, z) then
21:                    Ajouter (x, z) aux résultats locaux
22:                end if
23:            end for
24:        else
25:            Ajouter (y, z) dans un buffer pour les envoyer plus tard shard
26:        end if
27:    end for
28:    Échanger et traiter les buffers
29:    Phase 3 : Agrégation des résultats
30:    Collecter tous les résultats sur le processus 0
31:    return solutions trouvées
32: end procedure
```

4 Résultats

5 Difficultés rencontrées

La parallélisation s'est révélée plus complexe que prévu. Avant d'aboutir à la version finale présentée dans ce rapport, plusieurs approches ont été explorées et abandonnées en raison de performances insuffisantes.

5.1 Approche Boss-Workers

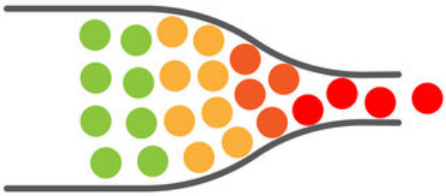
Une première implémentation reposait sur un schéma *boss-workers* :

— un processus maître distribuait dynamiquement des blocs de clés aux travailleurs,
— les travailleurs effectuaient les calculs puis renvoyaient leurs résultats au maître.

Cette approche s'est révélée être contre-productive pour plusieurs raisons :

— le processus maître devenait rapidement un **goulot d'étranglement**.
— les communications étaient très fréquentes et de petite taille, ce qui augmentait la latence MPI
— la charge de messages et de synchronisations annulaient complètement les bénéfices du parallélisme.

Le Boss avait besoin de beaucoup de mémoire RAM, et il y a avait de nombreux workers. Cela créait un scénario de 'botttleneck'.



Cette version parallèle était bien **plus lente que le code séquentiel**.

5.2 Communications à chaque itération

Une deuxième tentative était proche de la version finale, mais avec une stratégie de communication différente :

— chaque paire (z, x) ou (y, z) était envoyée immédiatement à son processus propriétaire à l'aide de `MPI_Send`,
— les communications étaient donc déclenchées à chaque itération de boucle

Cette version a rapidement montré ses limites :

— elle générait un **nombre extrêmement élevé de messages MPI** et de petites tailles.
— le coût de gestion des communications dominait le temps de calcul
— la saturation du réseau dégradait les performances

Cette version était aussi bien **plus lente que le code séquentiel**.