Direct Meet-in-the-Middle Attack

Julian Mouthon, Mario Razafinony

2 janvier 2026

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** : parallélisme au niveau des données

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 :

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

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 :

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

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.

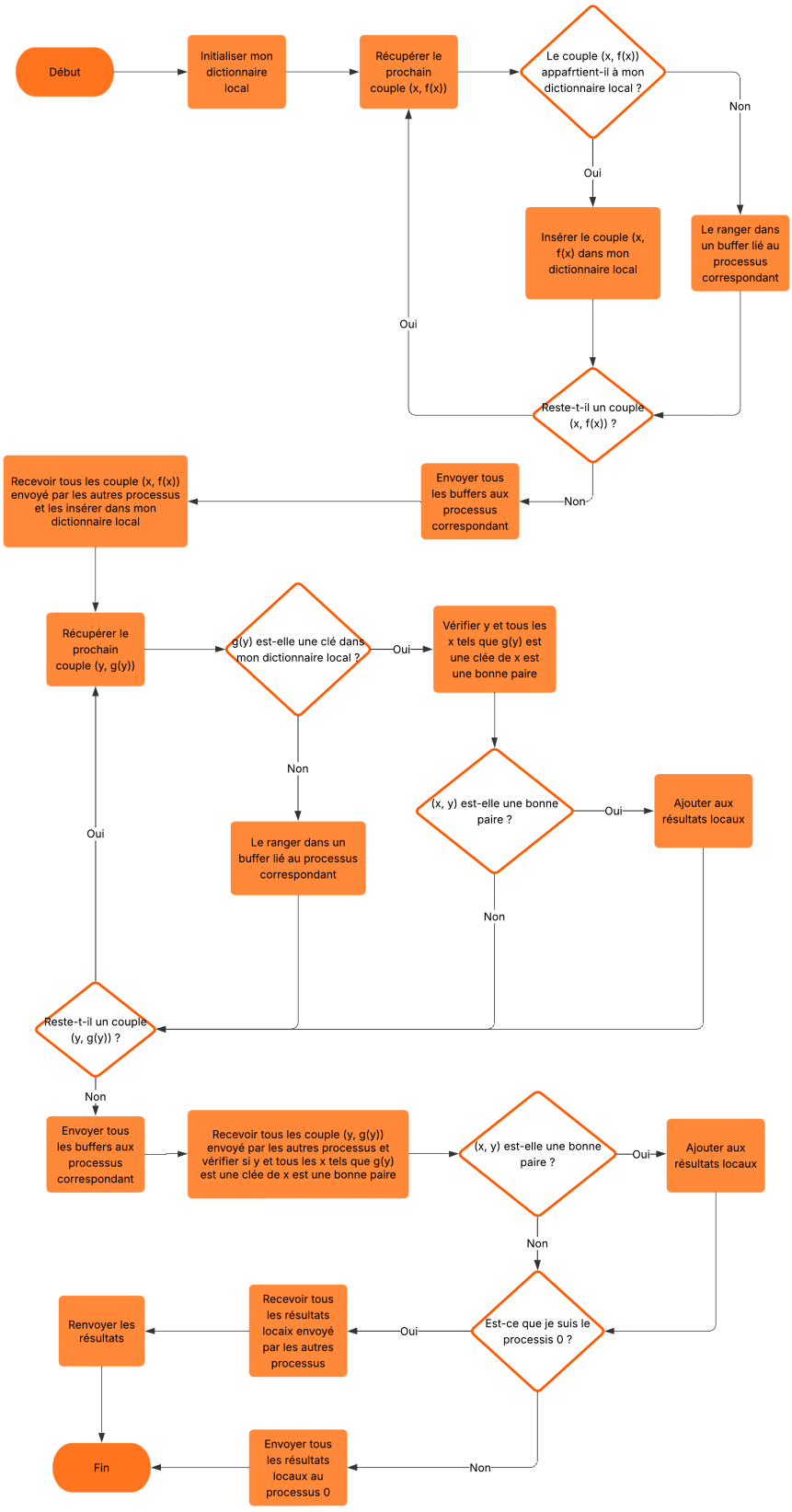
Chaque thread dispose de buffers privés pour stocker temporairement les paires clé–valeur destinées aux autres processus MPI. Ces buffers locaux sont ensuite fusionnés dans des buffers globaux dans des sections critiques, avant les phases de communication MPI.

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 Schéma détaillé



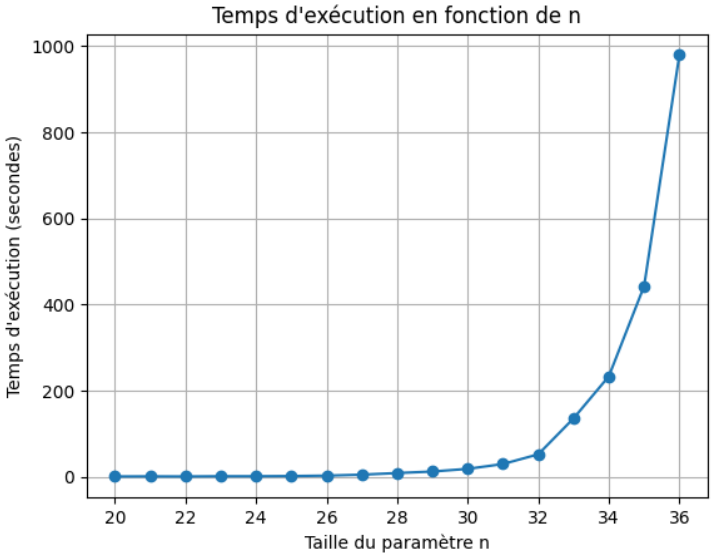
4 Résultats

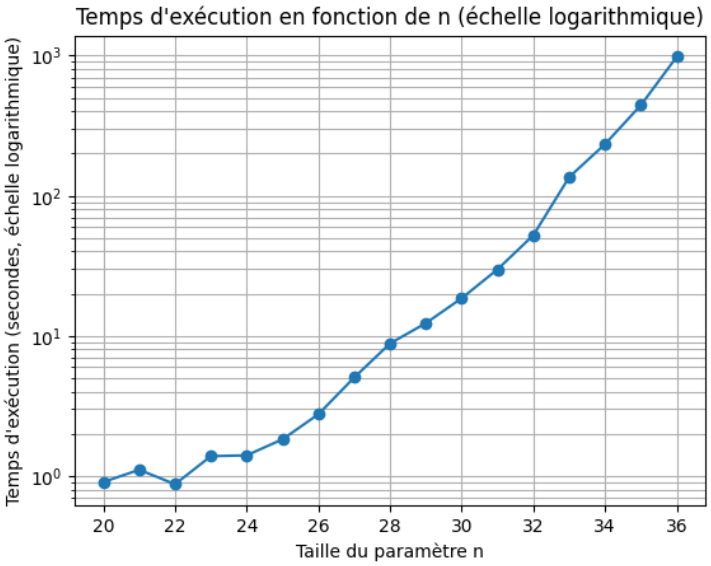
L'ensemble des résultats a été obtenu en utilisant le script `script.sh`, fourni en annexe et joint en fichier. Ce script permet d'automatiser l'exécution de tous les tests en mode interactif en lançant successivement les calculs pour différentes valeurs du paramètre n.

Les instances de test utilisées viennent de `https://ppar.tme-crypto.fr/<username>/<n>` avec :
— <user> = *'test'*
— $< n >$ = 20 . . .36

n	C_0	C_1	Solution trouvée	Temps (s)
20	6a5f3d287fb1b0a6	c1a0679bd25f06c5	(be693, cfc2e)	0.904
21	0ac3cb45892c96d6	cb28af67fc74dcbc	(1a826b, f179c)	1.106
22	cd842501442e84f0	63778751f0e4bedb	(2cdae9, 1b6b5e)	0.872
23	1a3bd0f512dba4fd	1706a8b7d71f3017	(56f292, 13e52a)	1.383
24	e3b8fb27d90bd863	0c094fccd755cde2	(2683b1, 53d2b6)	1.402
25	dea7744bf2755ad9	6e0ebdc050a492ee	(1038e1d, 1588d4d)	1.825
26	18667086664cac57	7ceb2e54e389be66	(f73f86, 2668ca4)	2.755
27	ce2c621789812038	13950eec4d6caf2b	(66e3e4f, 5076457)	5.041
28	0094a07e0d20f0d4	3e29df756df6b567	(1acb623, ec683c1)	8.818
29	19b05757e5e766a4	09a83a6bee5132a1	(1ea8fa9b, b3d61ec)	12.314
30	6b588b953555fdc3	857f62bb3ab86c48	(2cfce03, 7e53f17)	18.551
31	9cbcfd09ac384207	b486d19a8ebf1afa	(37f31175, 3e610dcc)	29.797
32	3458103b639f9f47	9e1499d15be61572	(e498a06f, 945a7522)	52.306
33	4d1b29953931a24f	e0bc6cefe332ac33	(770ef739, 1fa29f82e)	135.823
34	9fb679b7ce303683	adf075f18af8e8fb	(2166cee16, 34368c0ca)	233.493
35	ce7edd87642dc9dd	9a192c412800b2c5	(3b257675e, 33348638)	443.046
36	2ddbc9fe4da17416	fab28111310cbea3	(8c41a4b9b, fcd233151)	979.802

TABLE 1 – Récapitulatif des solutions trouvées





Limites : Les tests n'ont pas été au-delà de $n = 36$ en raison de la consommation mémoire qui est exponentielle. La taille du dictionnaire devient bien trop grande pour notre approche et pour les noeuds réservés.

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

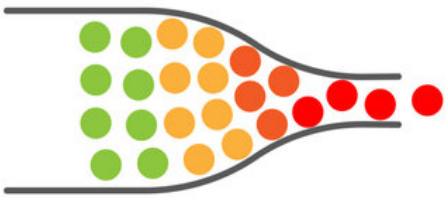
Une première implémentation reposait sur un schéma *boss-workers* :

— 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**.

6 Script d'exécution des tests

```
#!/bin/bash

params=(
  "20 6a5f3d287fb1b0a6 c1a0679bd25f06c5"
  "21 0ac3cb45892c96d6 cb28af67fc74dcbc"
  "22 cd842501442e84f0 63778751f0e4bedb"
  "23 1a3bd0f512dba4fd 1706a8b7d71f3017"
  "24 e3b8fb27d90bd863 0c094fccd755cde2"
  "25 dea7744bf2755ad9 6e0ebdc050a492ee"
  "26 18667086664cac57 7ceb2e54e389be66"
  "27 ce2c621789812038 13950eec4d6caf2b"
  "28 0094a07e0d20f0d4 3e29df756df6b567"
  "29 19b05757e5e766a4 09a83a6bee5132a1"
  "30 6b588b953555fdc3 857f62bb3ab86c48"
  "31 9cbcfd09ac384207 b486d19a8ebf1afa"
  "32 3458103b639f9f47 9e1499d15be61572"
  "33 4d1b29953931a24f e0bc6cefe332ac33"
  "34 9fb679b7ce303683 adf075f18af8e8fb"
  "35 ce7edd87642dc9dd 9a192c412800b2c5"
  "36 2ddbc9fe4da17416 fab28111310cbea3"
)

for p in "${params[@]}"; do
  n=$(echo "$p" | awk '{print $1}')
  C0=$(echo "$p" | awk '{print $2}')
  C1=$(echo "$p" | awk '{print $3}')

  echo "Soumission pour --n $n --C0 $C0 --C1 $C1"

  mpiexec --mca pml ob1 --mca btl tcp,self \
          --hostfile $OAR_NODEFILE \
          -n $(wc -l < $OAR_NODEFILE) \
          final --n "$n" --C0 "$C0" --C1 "$C1"
done
```