HPC Projet : Meet-in-the-Middle Attack

Qu'est-ce qu'on a fait ?

Nous allons ici expliquer la logique de notre algorithme puis dans la partie suivante, nous discuterons de la différence avec le code séquentiel.

Dans un premier temps, nous avons effectué un sharding de la hashtable (*clé-valeur*) qui contient la clé et sa valeur associée. Chaque processus insère dans sa hashtable à l'aide de la fonction *dict_insert* uniquement les clés dont la division euclidienne par le nombre de processus est égale à son rang. Les autres clés sont stockées dans un tableau unidimensionnel *send_buffer* afin que chacune d'entres elles soient envoyées au processus qui est censé l'insérer selon la règle citée précédemment. Afin d'économiser de la mémoire, nous allons allouer dans chaque processus ,un *send_buffer* de taille égale aux nombres d'éléments qu'il va envoyer et une hashtable de taille 1.125 fois le nombre de *clés_valeurs* qu'il va insérer dans sa hashtable. Pour ça, on va parcourir l'ensemble des clés-valeurs dans chaque processus et compter combien chacun va envoyer à chacun des processus. Puis, à l'aide de la communication Alltoall nous allons envoyer ce compteur à chacun des processus. Maintenant que la hashtable et le *send_buffer* sont créés, on va commencer à remplir le *send_buffer* afin d'envoyer les *clé_valeurs* aux processus censées les recevoir. Pour cet envoi, on va utiliser les communications MPI, plus spécifiquement des communications collectives Alltoall et Alltoallv. On effectue le *Alltoallv* pour que chaque processus reçoit les clés-valeurs à insérer dans sa hashtable. Une fois les données reçues, chaque processus va insérer dans sa hashtable les *clés_valeurs* reçues.

Dans un second temps, maintenant que chacun des processus contient sa propre hashtable, on va pouvoir rechercher dans chacun d'entre eux toutes les valeurs associées à une clé spécifique. On va procéder comme pour le remplissage des hashtables, si une clé modulo size est égale au rang du processus, il va rechercher dans son dictionnaire à l'aide de la fonction *dict_probe* sinon il va stocker la clé-valeur dans le *send_buffer* afin de l'envoyer au processus qui contient le dictionnaire les clés de même modulo size que celle-ci.

Quelle est la différence avec le code séquentiel ?

Pour la partie remplissage du dictionnaire et pour la recherche dans le dictionnaire, chaque processus traite environ N/size valeurs au lieu de traiter N valeurs. Cette répartition de charge entraîne une réduction significative du temps d'exécution, que nous détaillerons dans la section suivante sur les résultats.

Comment avons-nous testé ?

Nous avons constaté que la mémoire se remplissait rapidement lorsqu'on tentait de résoudre pour $n > 27$ avec un nombre limité de processus. C'est pourquoi nous avons décidé d'exécuter le code sur un plus grand nombre de processus. Cependant, avec des valeurs élevées de n, certaines limitations sont apparues, notamment une saturation de la mémoire.

Résulats :

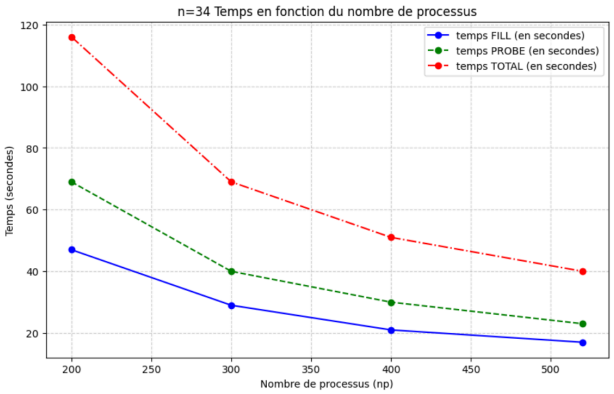
Les graphiques sont tracés grâce à un script python sur Jupyter Notebook.

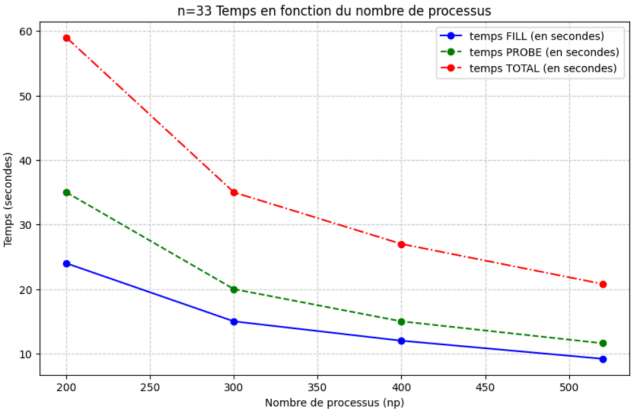
Conditions expérimentales :

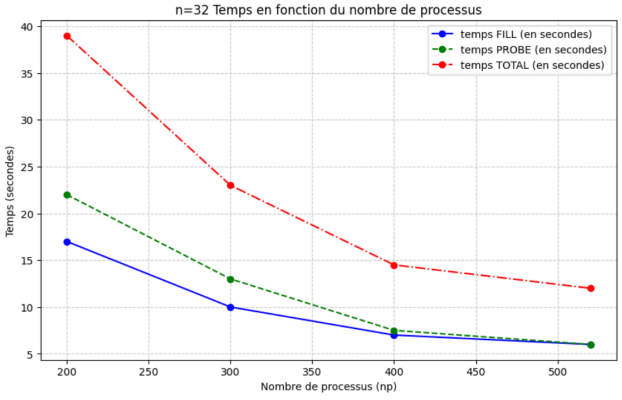
Compilation :
mpicc -Wall -O3 Finalmitm.c -o mitm

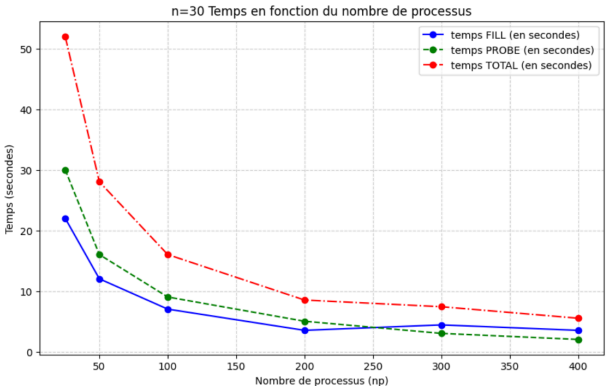
Réservation :
Cluster de Rennes
oarsub -l host=20/core=26 -I

Exécution : Les fichiers script.sh peuvent etre trouvés dans le fichier script.zip, on écrit le n correspondant au titre du graphique, exemple pour n=30 :
script30.sh

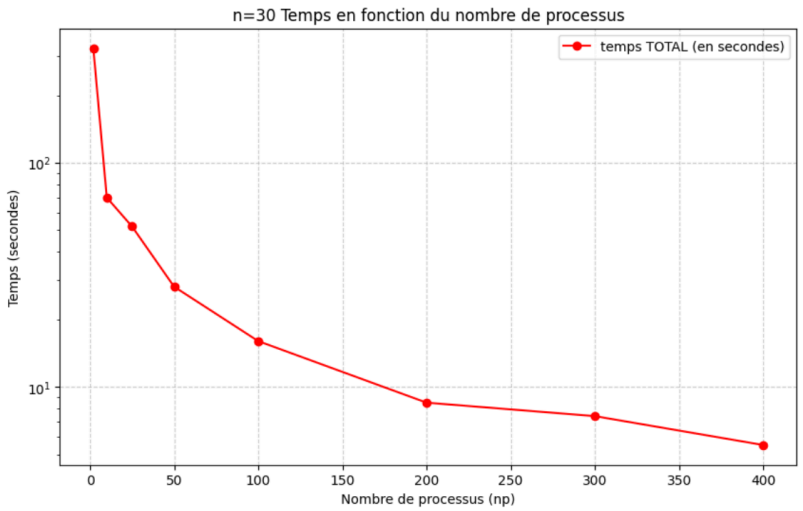


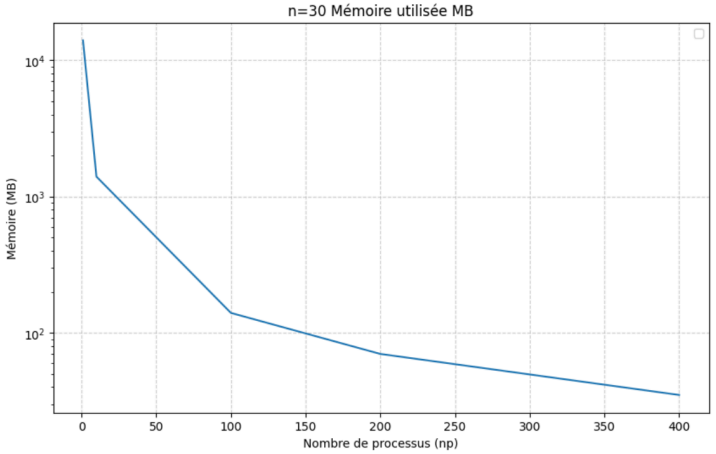




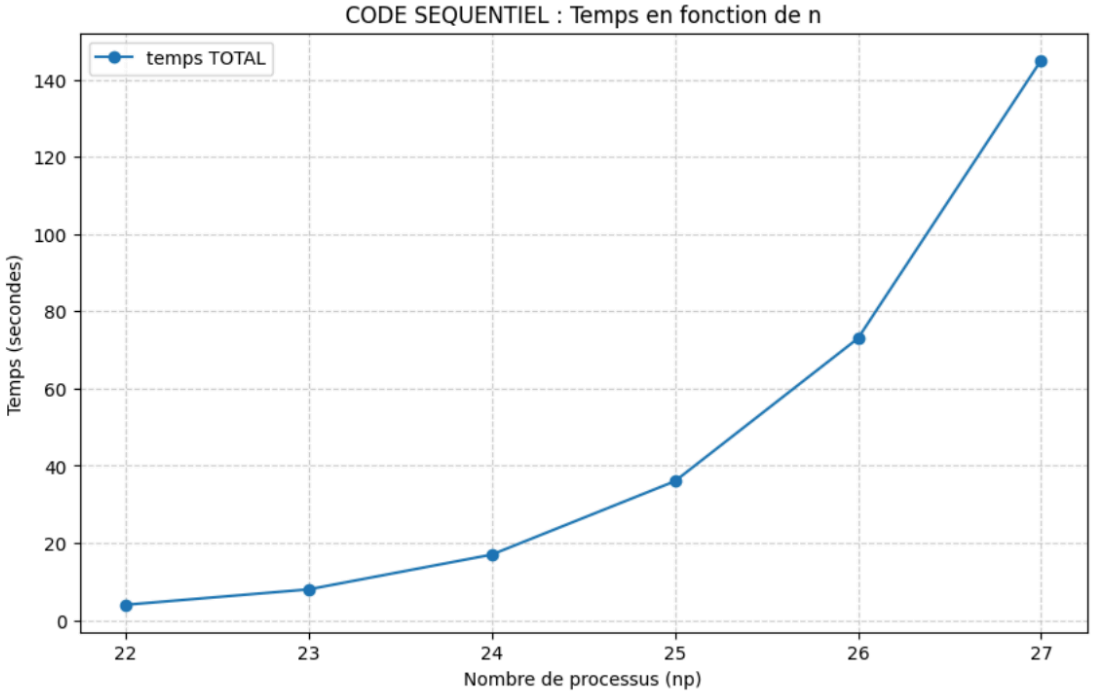


Sur ces 4 graphiques, on voit clairement que pour un n donné, en augmentant le nombre de processus on diminue le temps d'exécution de manière exponentielle décroissante (surtout ici n=30). On peut conjecturer qu'en multipliant par 2 le nombre size, on divise par 2 le temps d'exécution. Cela montre que la complexité est distribuée et qu'il devient particulièrement avantageux d'utiliser un grand nombre de processus pour des valeurs élevées de n, tandis que pour les petites valeurs de n, l'effet est moins marqué.

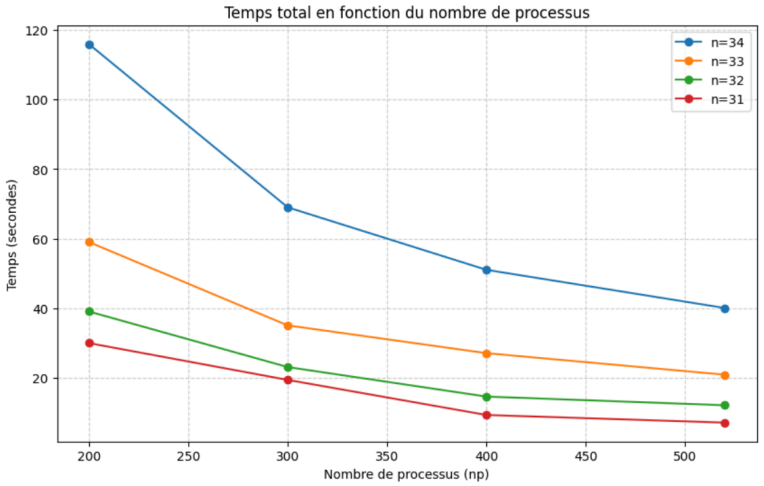




Pour n=30 on voit que la mémoire utilisée décroît de manière 1/size. Si le code tournait sur 1 processeur, la mémoire utilisée par ce dernier vaudrait 14GB. Pour n=36 sur 1 processeur la mémoire serait de 910GB (IMPOSSIBLE sur grid5000 !), c'est pour cela qu'il est important de distribuer sur plusieurs processus.



On observe bien que le temps est multiplié par 2 a chaque itération de n. Si on pouvait exécuter le code séquentiel sur n=36, cela prendrait environ 72 000 secondes ou 20h.



On distribue la complexité : il devient très intéressant d'utiliser beaucoup de processus pour des grandes valeurs de n (cela est moins important au départ).

Complexité du code : elle est principalement due aux fonctions dict_probe et golden_claw_search

	version séquentiel	version MPI
golden_claw_seach	$O(2^n)$	$O(2^n/size)$
dict_probe	$O(2^n)$	$O(2^n/size)$

Les appels aux communications collectives Alltoallv et Alltoall effectuent des échanges de données entre tous les processus. La complexité dépend de la taille des messages échangés, soit proportionnelle à O(total_send_count) et O(total_recv_count): qui sont en moyenne proportionnels à $O(2^n/size)$.

On se rend bien compte que pour des grandes valeurs de size de l'ordre 10^2 ou 10^3 .La version parallélisée a une complexité beaucoup plus faible.

Performance maximale du code :

Le plus grand 'n' pour lequel notre code fonctionne est n=36. Nous avons exécuter le code avec 650 process à l'aide de ces conditions :

Compilation :
mpicc -Wall -O3 Finalmitm.c -o mitm

Réservation :
Cluster de Rennes
oarsub -l host=25/core=26 -I

Exécution :
script36.sh

Username :
Ghali
Le temps d'exécution est au total de 128 secondes avec 45 secondes pour Fill et 83 secondes pour Probe

Etude de la mémoire :

Au départ nous exécutons le code en local sur notre ordinateur pour le tester, nous avons utilisé les logiciels htop pour mesurer la mémoire et valgrind pour trouver les fuites de mémoire et les fonctions associées. C'est grâce à htop que nous avons mis en lumière les bottleneck de mémoire, on observait que toute notre RAM était utilisée pour n=27. Nous avons aussi pu vérifier que plusieurs processeurs étaient en train de travailler.

Limitations et améliorations :

Les 'bottlenecks' rencontrés ont été majoritairement des problèmes de saturation de la mémoire.

-On avait initialement un tableau 2D de taille [size][2*N/size] soit 2*N*(u64int_t) dans chaque processus où stockaient les clés valeurs puis que l'on copiait dans un tableau unidimensionnel à l'aide de la fonction de 'memcpy' de la bibliothèque 'string' ce qui implique que l'on allouait 4*N*sizeof(u64int_t) en mémoire par processus sans prendre en compte le reste ce qui est énorme. Sachant qu'on ne remplissait ce tableau uniquement de 2*N/size entier. Pour y remédier, on a donc directement stocker les clés_valeurs dans un tableau de taille 2*N/size, ce qui nous a permis de gagner beaucoup de mémoire et d'exécuter jusqu'à n=31.

-On s'est rendu compte qu'on allouait pour chaque processus une hashtable de taille N alors qu'en réalité, on insérait dans chacune d'elles environ N/size paires clés_valeurs.
On a donc optimisé cela en calculant le nombre de valeurs que chacun des processus allait insérer dans sa hashtable et alloués une hashtable égale 1.125 fois le nombre de paires à insérer. Grâce à ça, nous avons pu aller jusqu'à n=36 !

Lors de l'exécution du code parallélisé pour n=37, nous obtenons un signal "9 killed", indiquant un problème de mémoire. Une amélioration possible serait de mieux gérer la mémoire.
Une autre solution pourrait être de réserver plus de nœuds que les 25 actuellement utilisés, afin d'avoir accès à plus de mémoire et de réduire les besoins mémoire de chaque processus.
Enfin, tester l'exécution du code en mode non interactif pourrait également permettre d'optimiser l'utilisation des ressources.