Rapport du Projet HPC

Résolution du problème Exact Cover

M1 Informatique - Spécialité SAR et SFPN

Achaibou Radia, Chetti Amel, Kaci Inas

Année universitaire 2020-2021

Table des matières

I Introduction

Le but du projet est de paralléliser un programme séquentiel fourni qui effectue la résolution d'une instance du problème de la couverture exacte. Ce problème consiste en un ensemble d'objets et d'options constituées d'un ou plusieurs objets. Le but est de choisir un sous-ensemble d'options tel que chaque objet soit contenu dans une et une seule des options sélectionnées.
Notre stratégie est la suivante : Dans un premier temps, nous avons fait de la programmation multithread sur une seule machine multi-coeur grâce à OpenMP. On a ensuite fait de la programmation répartie sur un cluster grâce à l'envoi de messages sous MPI. Enfin, nous avons jumelé les deux approches en utilisant MPI+OpenMP sur un cluster.

II Parallélisation avec OpenMP sur une machine multi-coeurs

Caractéristiques des exécutions
nombre de coeurs : 8
modèle de la machine : Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz
nombre de processus sur lesquelles le programme parallèle s'execute : 8
commande de compilation : gcc -g -fopenmp -o prog prog.c

1 Approche 1 : Parallélisation des fonctions terminales

Nous appelons fonctions terminales, les fonctions qui ne font pas d'appels à d'autres fonctions
Dans un premier temps, nous avons voulu paralléliser les portions de code les plus simples, nous avons donc cherché les fonctions qui ne font pas d'appels récursifs et qui ont des boucles for afin de les paralléliser avec *pragma omp parallel*

conclusion : Le traitement fait par ces petites fonctions est petit et peu coûteux donc ne requiert pas d'être parallélisé et sa parallélisation n'améliore pas les performances du calcul global.

2 Approche 2 : Pralléliser la fonction solve avec omp parallel for

Nous décidons de changer de stratégie et de paralléliser la fonction initiatrice des traitements "solve". Comme la fonction solve est composée d'une boucle for et à chaque tour de boucle un appel récursif est effectué, nous avons essayé de paralléliser la boucle for pour que les appels récursifs se répartissent sur les processus de la machine.
Le code source de cette approche se trouve dans l'archive Version$_MPI$ dans un fichier nommé $exact\_cover\_OMP1.c$.

Résultats
Nous avons exécuté notre fonction sur 9 instances et nous avons constaté que la fonction renvoie la bonne solution mais en terme de performance , elle est beaucoup plus coûteuse que le code

séquentiel.
Les résultats obtenus sont attendus car solve est appelé pour chaque option contenant l'objet choisi alors que dans le code séquentiel, on avait une condition "if" qui permettait d'appeler solve uniquement si le solve précédent n'avait pas atteint la solution maximale.

Instances/Temps d'exécution(secondes)	Séquentiel	Parallèle	solutions trouvées
$Bell12$	3.6	4.8	4213597
Bell13	24.8	32.5	27644437
Bell14	179.7	231	190899322
matching8	0.8	1.6	2027025
matching9	14.9	28.0	34459425
matching10	286.7	541.8	654729075
$pentomino - 6 - 10$	25.5	25.7	9356
$pento - plus - tetra2 \times 4 \times 10$	1635	3068	895536
$pento - plus - tetra8 \times 8$	$>$ 15 heures	>18 heures	1207328

Conclusion
La parallélisation de la boucle for de solve avec pragma omp for est très coûteuse. En examinant de près les affichages , nous avons constaté que la boucle for du premier appel à solve prenait beaucoup de temps.

3 Approche 3 : Utilisation de OMP TASK

La deuxième solution qui nous vient intuitivement en premier, étant donné que "solve" est une fonction récursive, est d'utiliser la directive "pragma omp task" pour créer une tâche associée à chaque appel récursif. Le code source est dans l'archive $Version\_OMP$ dans un fichier nommé $exact\_cover\_OMP2.c$.

Problème 1 : Ceci induit des accès concurrents de différents threads sur la variable de contexte. Pour résoudre ce problème, nous pouvons :

- faire un contrôle des accès concurrents pessimiste grâce à la création de sections critiques sur les champs du contexte modifié par les différentes tâches. Ceci pourrait se faire avec l'utilisation de lock (avec les directives "pragma omp critical", "pragma omp atomic"), sauf que l'on voit vite que cette solution n'est pas adéquate à notre problème car nos instances grandissant très vite, il y a beaucoup d'accès concurrents. L'utilisation de lock rend les accès plus lents, séquentialise le code et induit des attentes entre les threads. Cette solution limite le passage à échelle et n'est donc pas adaptée à notre problème. Nous avons jugé qu'il était inutile de faire cette implémentation
- faire une copie du contexte avant chaque appel récursif

Problème 2 : Pour l'instance la plus simple : "bell12.ec", nous calculons 8427195 appels récursifs en tout. Donc une copie de contexte par appel récursif fait vite exploser l'utilisation de la mémoire et du CPU. Ceci s'observe dans la figure suivante capturée lors de l'exécution du programme.

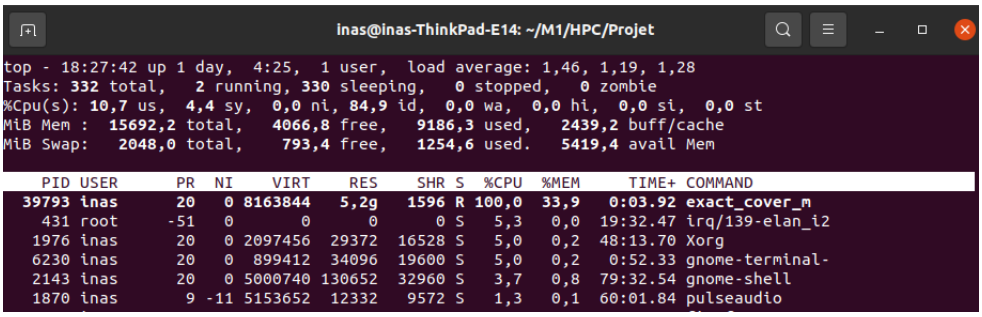


FIGURE 1 – capture de la commande top lors de l'exécution du programme

Solution : Pour résoudre ce problème, nous avons pensé à une approche similaire à celle que nous avons proposé en TME sur la parallélisation de Fibonnaci. Celle-ci consiste à diminuer le nombre de tâches et à ne créer de tâche que pour les appels récursifs les plus longs. Dans notre cas, plus on avance en profondeur dans les appels récursifs, moins il y a d'objets à couvrir et donc d'options disponibles. Nous avons donc pensé à créer des tâches pour les premiers niveaux d'appels récursifs uniquement et ainsi, nous avons moins de copies de contexte et résolvons le problème précédant. Cependant en terme de temps, nous n'obtenons pas d'amélioration notable.
Ci-dessous un tableau de comparaison de performances entre l'exécution séquentielle et l'exécution de la solution ci-dessus. Chaque valeur est une moyenne de 5 exécutions d'une instance sur le programme donné.

Instances/Temps d'exécution(secondes)	Séquentiel	Parallèle	solutions trouvées
*Bell*12	3.78	$3,82$	4213597
Bell13	26.3	26.64	27644437
Bell14	195,04	$192,34$	190899322
matching8	0.96	$0,94$	2027025
matching9	16.34	$16,02$	34459425
matching10	322.5	316.6	654729075
$pentomino - 6 - 10$	28.22	$27,84$	9356

4 Meilleure approche : Paralléliser la boucle for du premier appel à solve avec omp task

Pour effectuer cette parallélisation ,nous affectons à chaque processus une tâche , cette dernière correspond à une affectation d'un paquet d'options dédié à chaque processus initiée par un processus maître, et pour chaque option le processus doit appeler la fonction récursive solve pour trouver le nombre de noeuds à couvrir à partir de celle-ci. Le processus maître qui a lancé ces tâches doit sommer l'ensemble des solutions retournées par les autres processus pour arriver à un résultat final. Le code source de cette approche se trouve dans l'archive Version_OMP dans un fichier nommé $exact\_cover\_OMP.c$.

Résultats obtenus sur une machine à 8 coeurs

Instances/Temps d'exécution(secondes)	Séquentiel	Parallèle	solutions trouvées
$Bell12$	3.6	1.3	4213597
Bell13	24.8	9.0	27644437
Bell14	179.7	67.7	190899322
matching8	0.8	0.3	2027025
matching9	14.9	4.3	34459425
matching10	286.7	88.5	654729075
$pentomino - 6 - 10$	25.5	7.2	9356
$pento - plus - tetra2 \times 4 \times 10$	1635	704.2	895536
$pento - plus - tetra8 \times 8$	$>$ 15 heures	45852.1 $\approx$ 12 heures	1207328

Conclusion
Cette approche permet de paralléliser notre code séquentiel et de bien équilibrer les charges. L'exécution de ce code parallèle s'effectue en un temps réduit comparé au code séquentiel, nous gagnons un facteur deux pour chaque instance exécutée.

III Parallélisation MPI sur un cluster

Caractéristiques des exécutions sur Grid5000
Site : Nancy
nombre de coeurs : 16
commande de compilation : mpicc -o nom_de_executable programme.c
mpiexec –hostfile $OAR_NODEFILE ./prog

1 MPI sans équilibrage de charge

1.1 Attribuer des paquets d'options consécutives à chaque processus esclave

Cette approche permet d'attribuer aux processus esclaves au plus les $\frac{nombre\_d'options}{nombre\_de\_processus}$ options consécutives dans la liste des options en s'assurant que chaque processus ait une liste d'options différente de toutes les autres.
Pour chaque option , le processus résout récursivement le problème.Le code source de cette approche se trouve dans l'archive $Version\_MPI$ sous le nom $exact\_cover\_MPI\_2.c$

Résultats obtenus

En réservant 4 coeurs sur le cluster nancy, nous obtenons :

Instances/Temps d'exécution(secondes)	Séquentiel	Parallèle	solutions trouvées
$Bell12$	3.6	4.8	4213597
Bell13	24.8	13.3	27644437
Bell14	179.7	90.6	190899322
matching8	0.8	0.5	2027025
matching9	14.9	6.4	34459425
matching10	286.7	152.4	654729075
$pentomino - 6 - 10$	25.5	953.9	9356
$pento - plus - tetra2 \times 4 \times 10$	1635	704.2	895536
$pento - plus - tetra8 \times 8$	$>$ 15 heures	45852.1 $\approx$ 12 heures	1207328

En réservant 4 noeuds sur le cluster nancy, nous obtenons

Instances/Temps d'exécution(secondes)	Séquentiel	Parallèle	solutions trouvées
hline $Bell12$	3.6	4.9	4213597
Bell13	24.8	36.1	27644437
Bell14	179.7	267.5	190899322
matching8	0.8	1.5	2027025
matching9	14.9	105.5	34459425
matching10	286.7	104.7	654729075
$pentomino - 6 - 10$	25.5	3.5	9356
$pento - plus - tetra2 \times 4 \times 10$	1635	192.2	895536
$pento - plus - tetra8 \times 8$	$>$ 15 heures	45852.1 $\approx$ 12 heures	1207328

1.2 Attribuer des paquets d'options non consécutives à chaque processus esclave

Cette approche permet d'attribuer à chaque processus esclave au plus $\frac{nombre\_d'options}{nombre\_de\_processus}$ options. Les numéros d'options sont attribués au processus esclave avec un pas = $nombre\_de\_processus$. Ce pas représente l'écart entre les numéro d'options de chaque processus. Nous nous assurons que les sous-ensembles d'options attribués à chaque processus ne s'intersectent pas
Pour chaque option , le processus résout récursivement le problème. Le code source de cette approche se trouve dans l'archive Version$_MPI$ dans un fichier nommé $exact\_cover\_MPI.c$.

Résultats obtenus

En réservant 12 coeurs sur le cluster nancy, nous obtenons :

Instances/Temps d'exécution(secondes)	Séquentiel	Parallèle	solutions trouvées
$Bell12$	3.6	1.9	4213597
Bell13	24.8	13.2	27644437
Bell14	179.7	13.5	190899322
matching8	0.8	0.3	2027025
matching9	14.9	6.5	34459425
matching10	286.7	109.6	654729075
$pentomino - 6 - 10$	25.5	10.6	9356
$pento - plus - tetra2 \times 4 \times 10$	1635	986.4	895536
$pento - plus - tetra8 \times 8$	$>$ 15 heures	45852.1 $\approx$ 12 heures	1207328

En réservant 4 noeuds sur le cluster nancy, nous obtenons :

Instances/Temps d'exécution(secondes)	Séquentiel	Parallèle	solutions trouvées
$Bell12$	3.6	1.3	4213597
Bell13	24.8	8.5	27644437
Bell14	179.7	57.0	190899322
matching8	0.8	0.1	2027025
matching9	14.9	3.1	34459425
matching10	286.7	53.0	654729075
$pentomino - 6 - 10$	25.5	3.8	9356
$pento - plus - tetra2 \times 4 \times 10$	1635	191.3	895536
$pento - plus - tetra8 \times 8$	$>$ 15 heures	45852.1 $\approx$ 12 heures	1207328

2 MPI avec équilibrage de charge , maître esclave

Pour mieux paralléliser notre programme,nous avons implémenté la stratégie maître esclave où le processus maître s'occupe d'envoyer à chaque processus esclave n'ayant pas de tâche à exécuter, une option non traitée tant qu'il en reste pour résoudre le problème récursivement à partir de cette dernière. Le processus maître met à jour la solution après chaque réception d'une solution partielle par le processus esclave. Le code source de cette approche se trouve dans l'archive Version_MPI dans un fichier nommé $exact\_cover\_MasterSlave.c$.

— **Les modifications apportées**

La principale fonction que nous avons modifié est la fonction solve, autrement dit, le premier appel à solve. Il est d'abord question d'ajouter les tags dont nous nous servirons dans les communications. Un tag pour l'envoi des numéros d'options à traiter, un autre pour la réception des options, un tag indiquant la terminaison des processus et enfin le tag fin s'il n'y a plus de blocs à traiter.
— int TAG_REQ = 1 ;
— int TAG_DATA = 2 ;
— int TAG_END = 0 ;
— int FIN = -1 ;
On communique ensuite le numéro d'option à traiter à tous via un MPI_Send. Évidemment, seul le processus maître a besoin de faire cette distribution :

if($my\_rank == 0$)

/* des lignes de codes manquantes*/
MPI_Send(cpt,1,MPI_INT,dest,TAG_REQ,MPI_COMM_WORLD)

/* des lignes de codes manquantes*/

1. Le processus 0 envoi un message aux autres processus dest.
2. Le premier argument désigne le numéro d'option à traiter.
3. MPI_INT pour dire que nous envoyons un entier.
4. TAG_REQ sert à différencier les messages reçus par la suite en comparant les tags.

Chaque processus reçoit un numéro d'option approprié :

$MPI\_Recv(bloc, 1, MPI\_INT, 0, MPI\_ANY\_TAG, MPI\_COMM\_WORLD, status)$;

Les esclaves effectuent leurs tâches assignées en appelant solve avec l'option reçue et le résultat sera stocké et renvoyé au maître avec MPI_Send pour que ce dernier mette à jour la solution en ajoutant les solutions reçues aux solutions du contexte précédemment calculées.

Lorsqu'il n' y a plus de tâches à exécuter par les processus. Le processus maître enverra TAG_END à tous les processus pour qu'ils arrêtent leurs exécutions.

— **Résultats**

En réservant un noeud et 8 coeurs sur le cluster nancy de Grid5000, nous avons obtenu :

Instances/Temps d'exécution(secondes)	Séquentiel	Parallèle	solutions trouvées
$Bell12$	3.6	0.9	4213597
Bell13	24.8	6.1	27644437
Bell14	179.7	41.4	190899322
matching8	0.8	0.3	2027025
matching9	14.9	4.2	34459425
matching10	286.7	72.1	654729075
$pentomino - 6 - 10$	25.5	6.7	9356
$pento - plus - tetra2 \times 4 \times 10$	1635	464.4	895536
$pento - plus - tetra8 \times 8$	$>$ 15 heures	/	1207328

En réservant 10 coeurs, 12 coeurs puis 24 coeurs, nous avons eu une meilleure optimisation pour $pento - plus - tetra2 \times 4 \times 10$.

Instances/Temps d'exécution(secondes)	8 coeurs	10 coeurs	12 coeurs	24 coeurs
$pento - plus - tetra2 \times 4 \times 10$	$644.4s$	494	344.3	240

En réservant 4 noeuds sur le cluster nancy de Grid5000, nous avons obtenus :

Instances/Temps d'exécution(secondes)	Séquentiel	Parallèle	solutions trouvées
$Bell12$	3.6	0.9	4213597
Bell13	24.8	6.1	27644437
Bell14	179.7	45.1	190899322
matching8	0.8	0.5	2027025
matching9	14.9	1.6	34459425
matching10	286.7	27.1	654729075
$pentomino - 6 - 10$	25.5	2.2	9356
$pento - plus - tetra2 \times 4 \times 10$	1635	190.4	895536
$pento - plus - tetra8 \times 8$	$>$ 15 heures	4noeuds=2h	1207328

Conclusion

Nous avons testé la stratégie maître esclave sur Grid5000 sur plusieurs instances, en réservant plusieurs nœuds et plusieurs coeurs sur le cluster nancy . L'utilisation de MPI nous a permis de paralléliser notre code séquentiel et de bien équilibrer les charges car nous avons obtenu des temps d'exécutions plus performants que le version MPI sans équilibrage de charge et nettement plus performant que le séquentiel .

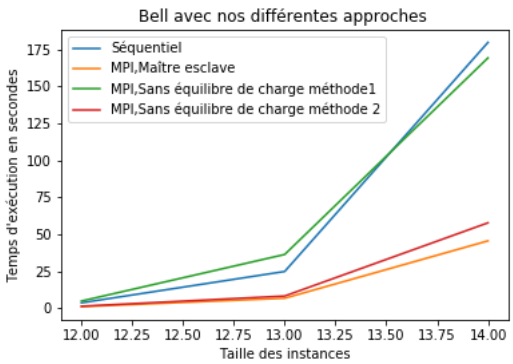
Nous constatons qu'il y a gain énorme pour les grandes instances , telle $pento - plus - tetra2 \times 4 \times 10$, nous passons de 1635 secondes pour la version séquentielle à 190.4 secondes pour celle de MPI, ce qui n'est pas négligeable.

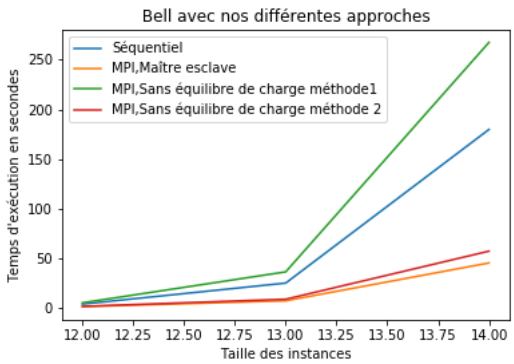
Nous remarquons également que le fait d'incrémenter le nombre de processus ou le nombre de noeud réservés sur le cluster affecte négativement le temps d'exécution des petites instances telle que Bell12 ,Bell13, matching8 car elle ne prennent pas beaucoup de temps pour s'exécuter et répartir les options de ces instances sur énormément de processus fait que ralentir leurs exécutions. Ceci s'explique par le temps de traitement qui devient négligeable par rapport aux temps de communications

IV Résultats et performances

1 INSTANCES BELL

— En réservant 2 noeuds(figure gauche) puis 4 noeuds(figure droite) sur le cluster nancy de Grid5000 , nous avons obtenu ces résultats :



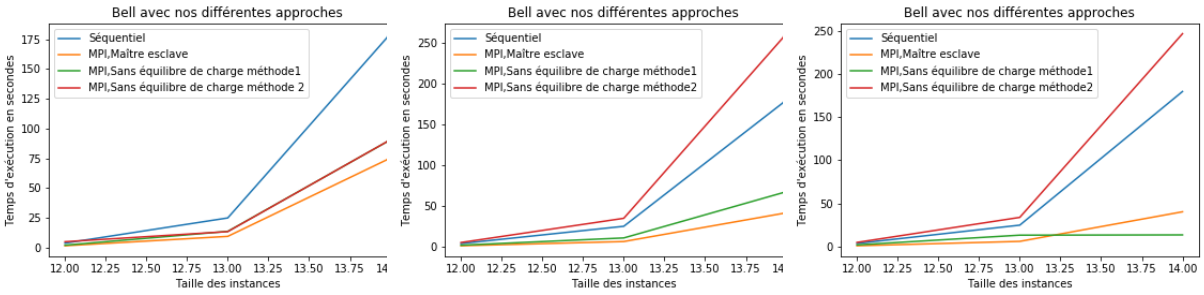


Remarque :

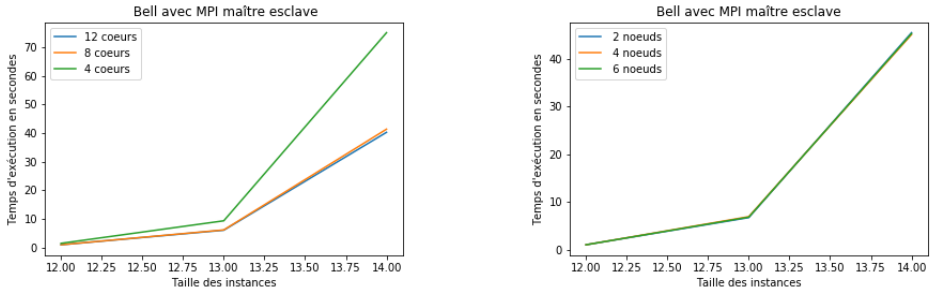
Notre parallélisation MPI maître esclave est plus performante que les autres approches

et ce résultats est attendu car cette stratégie répartie la charge sur les processus de manière équitable.

— En réservant 4,8 et 12 coeurs sur le cluster nancy de Grid5000 , nous avons obtenu ces résultats :



— Comparaison des résultats obtenus sur les instances Bell avec maître esclave en réservant 4,8,12,14 coeurs (figure de gauche) puis 2,4,6,8 noeuds(figure de droite) sur le cluster nancy de Grid5000 :



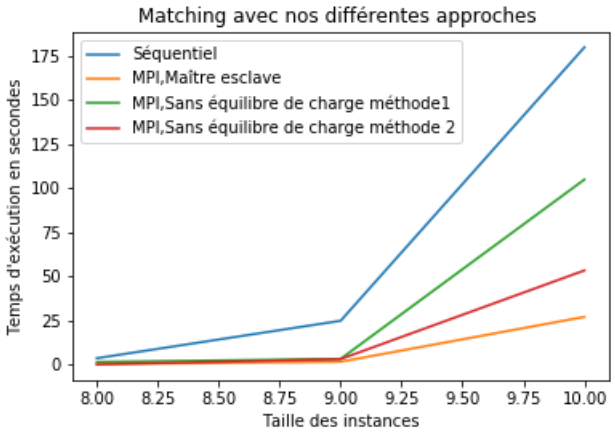
Remarques

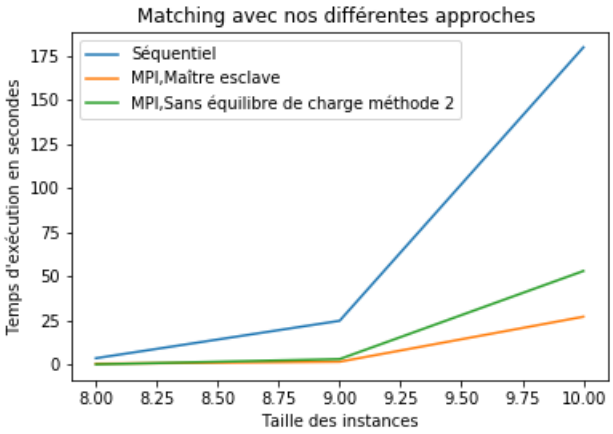
Nous avons obtenu un temps d'exécution performant à partir de coeur=8 sur les instances Bell, et au delà ce nombre la parallélisation n'est pas optimale et affecte de manière négative le temps d'exécution de ces instances car la taille de ces instances n'est pas assez grande pour la répartir sur énormément de processus.

De même pour les noeuds, comme la taille des instances est assez petites , réserver plus de noeuds n'apporte rien à cette parallélisation . Le temps de communications entre les processus et de création des tâches induit par l'ajout de nouveaux noeuds ou processus influence négativement le temps d'exécution après un certain seuil. Plus il y a des communications à faire, plus on est occupé à communiquer et pas à calculer et donc plus on perd du temps de calcul.

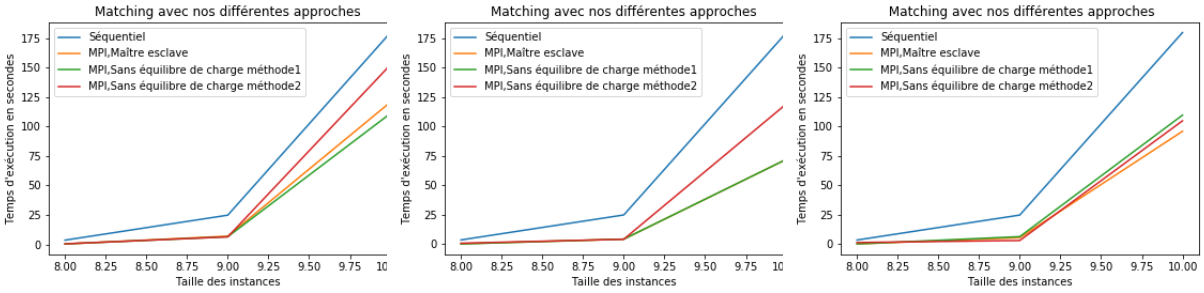
2 INSTANCES MATCHING

— **En réservant 2 noeuds(figure gauche) puis 4 noeuds(figure droite) sur le cluster nancy de Grid5000 , nous avons obtenu ces résultats :**





— **En réservant 4,8 et 12 coeurs sur le cluster nancy de Grid5000 , nous avons obtenu ces résultats :**

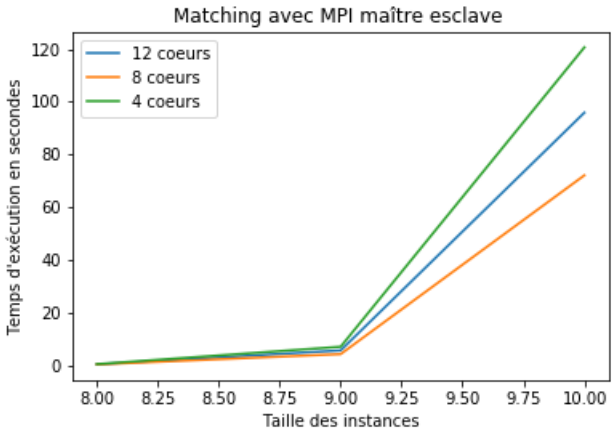


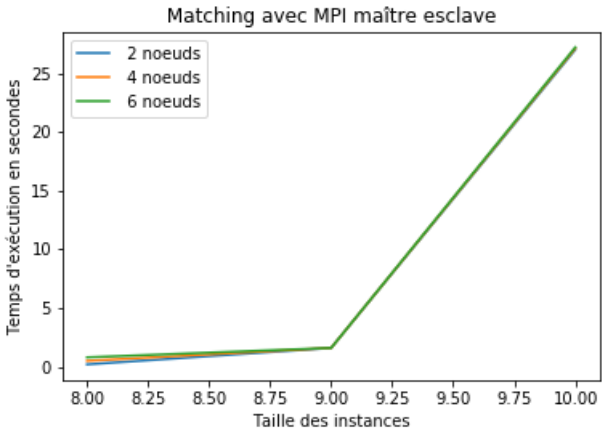
— **Comparaison des résultats obtenus sur les instances Matching avec maître esclave en réservant 4,8,12 coeurs (figure de gauche) puis 2,4,6 noeuds(figure de droite) sur le cluster nancy de Grid5000 :**

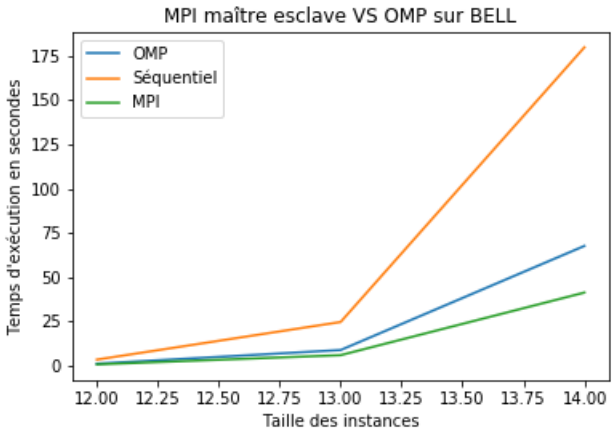
Ce résultat est bien le résultat attendu, le gain en temps des solutions parallèles est remarquable. De plus, nous avons constaté qu'en augmentant le nombre de cœurs jusqu'à 12 cette amélioration MPI est de plus en plus efficace, car les taille des instances Matching sont plus ou moins grandes et l'augmentation du nombre de cœurs permet la répartition équitable de la charge entre un grand nombre de processus.

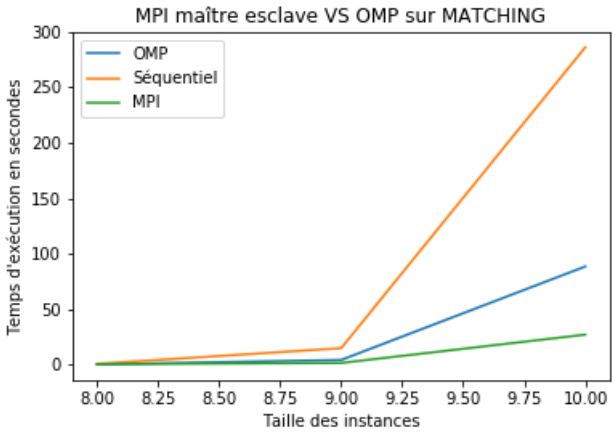
— **Comparaison des résultats obtenus avec OMP et MPI sur les instances Bell et Matching**

Les résultats obtenues affirment que les différentes approches de parallélisation que nous avons proposées nous permettent de gagner en temps ; un facteur de 4 pour les instances Bell et un facteur de 12 pour les instances matching. Nous confirmons que la parallélisation des programmes séquentiels est très optimale et affecte de manière positive le temps d'exécutions lorsque la taille des instances est assez grande.









V Parallélisation avec MPI + OpenMP

Pour effectuer cette parallélisation , nous avons repris le même code MPI sans équilibrage de charge où chaque processus reçoit du maître un paquet d'options à traiter.Nous avons transformé la boucle for qui permet de traiter ces options en un code parallélisable avec pragma omp task , en créent une tâche pour chaque moitié du paquet reçu et résoudre ensuite le problème récursif solve de la boucle for en prenant en compte que cette moitié du paquet .Le code source de cette approche se trouve dans l'archive Version_MPI_OMP.
Nous n'avons pas réussi à avoir des résultats performants par rapport au code séquentiel.

VI Conclusion

L'objectif fixé a été atteint, nous avons réussi à paralléliser le programme séquentiel qui effectue la résolution d'une instance du problème de la couverture exacte. Nous avons eu recours à plusieurs approches de parallélisation, commençant par la parallélisation des fonctions terminales en utilisant OMP qui n'améliore les performances global car le traitement effectué par ces fonctions est petit. Par la suite nous avons eu recourt à l'utilisation de l'approche OMP task avec équilibrage de charge qui nous a permis de gagner un facteur 2.
Dans un second temps, nous avons eu recourt à l'utilisation des directives MPI, nous avons été surpris par l'amélioration obtenu grâce à l'utilisation de l'approche maître esclave avec équilibrage sur des grandes instances. Par contre, la combinaison des deux approches de pa-

rallélisation OMP et MPI nous donne des résultats défavorables,cela pourrait être du au coût des communications élevé et au coût de création des tâches (copie de contexte d'exécution, allocation de pile ...) .
Nous avons noté deux points difficiles qu'il est important de prendre en compte afin de paralléliser au mieux un code séquentiel. Le premier étant qu'il est important d'identifier les parties qu'il n'est pas possible de paralléliser, sinon, les résultats obtenus sont incohérents. Le second point est qu'il faut se poser la question : "Est-ce que le coût de la communication vaut l'amélioration ?".
Pour conclure, il est indéniable que paralléliser un code correctement fait un gagner un temps précieux à l'exécution.