HPC

RÉSOLUTION DU PROBLÈME *EXACT COVER*

Quentin Deschamps

Avril-Mai 2021

Table des matières

1 Introduction

Le but du projet est de paralléliser un programme séquentiel qui résout une instance du problème de couverture exacte. Ce rapport vise à expliquer les stratégies de parallélisation retenues, présenter les implémentations et montrer les résultats obtenus.

L'archive du projet est organisée en 7 répertoires :

- `sequential` : programme séquentiel
- `openmp` : parallélisation avec OpenMP
- `mpi` : parallélisation avec MPI
- `hybrid` : parallélisation avec MPI et OpenMP
- `checkpointing` : parallélisation avec *checkpointing*
- `instances` : instances du problème de couverture exacte
- `benchmark` : scripts en Python pour lancer des benchmarks et visualiser les résultats, fichiers de configuration (json) et fichiers de résultats (csv, png)

Les 5 premiers répertoires contiennent un `Makefile` qui permet de compiler les programmes avec la commande `make`. La commande `make clean` permet de supprimer les exécutables créés. Le `Makefile` à la racine de l'archive permet de tout compiler avec la commande `make`, et supprimer tous les exécutables avec `make clean`.

Ce rapport décrit la meilleure stratégie de parallélisation trouvée qui utilise un parcours en largeur (*Breadth-first search*). L'archive contient aussi les programmes correspondant à d'autres stratégies qui ne sont pas décrites par le rapport.

2 Stratégie générale

La résolution séquentielle du problème de couverture exacte consiste à explorer un arbre décisionnel pour trouver les différentes solutions :

- ➟ Un noeud correspond à la couverture d'un objet.
- ➟ Une arête correspond au choix d'une option.

Le but du projet est donc de paralléliser l'exploration de l'arbre.

La stratégie de parallélisation retenue repose sur un **parcours en largeur** de l'arbre. Voici l'idée :

1. On commence l'exploration de l'arbre à sa racine en effectuant un parcours en largeur. On arrête ce parcours lorsque le nombre de noeuds au niveau actuel de l'arbre est supérieur à un certain seuil.
2. Les sous-arbres au niveau où le parcours en largeur s'est arrêté sont répartis entre les processeurs ou threads. **Ils sont explorés en parallèle**.
3. En sommant le nombre de solutions trouvé pour chaque sous-arbre, on peut alors retrouver le nombre de solutions total.

Avec cette méthode, on divise l'exploration de l'arbre en tâches pouvant être réalisées en parallèle. L'utilisation d'un parcours en largeur au début de la résolution permet d'exploiter tous les coeurs à notre disposition en créant assez de tâches.

3 Parallélisation avec OpenMP

Cette partie présente la parallélisation du programme séquentiel avec OpenMP. Le fichier décrit est le suivant :

`openmp/exact_cover_omp_bfs.c`

3.1 Stratégie

La stratégie de parallélisation est la suivante :

1. On définit les sous-arbres à distribuer grâce à un parcours en largeur s'arrêtant à un certain niveau de l'arbre de sorte à en avoir assez pour occuper tous les threads.
2. Les sous-arbres sont explorés en parallèle par les threads qui trouvent le nombre de solutions de ceux-ci.
3. On somme le nombre de solutions trouvé par chaque thread pour obtenir le nombre total de solutions.
4. Le programme affiche le nombre total de solutions.

3.2 Implémentation

Pour implémenter cette stratégie, on a besoin des fonctions supplémentaires suivantes :

- `array_copy` : copie un tableau d'entiers
- `sparse_array_copy` : copie un tableau creux
- `copy_ctx` : copie un contexte
- `sparse_array_free` : nettoie la mémoire pour un tableau creux
- `free_ctx` : nettoie la mémoire pour un contexte
- `free_instance` : nettoie la mémoire pour une instance
- `solve_bfs` : effectue un parcours en largeur de l'arbre s'arrêtant à un certain niveau, puis termine la résolution en parallèle avec la fonction `solve`

De plus, pour effectuer le parcours en largeur, on définit une file avec les variables suivantes :

- `queue` : file de contextes
- `queue_front` : index de tête de file
- `queue_rear` : index de queue de file
- `queue_size` : nombre de contextes dans la file

Les fonctions suivantes permettent de manipuler la file :

- `queue_is_empty` : indique si la file est vide
- `enqueue` : enfile un contexte
- `dequeue` : défile un contexte
- `free_queue` : libère la mémoire occupée par la file

Voici la procédure de résolution avec OpenMP :

1. **Initialisation** : on charge l'instance et on crée un contexte. Le chronomètre est lancé.
2. **Création des tâches** : on exécute la fonction `solve_bfs`. On initialise les variables `count` et `level` permettant de compter le nombre de noeuds de chaque niveau de l'arbre. Le nombre de solutions est mis à zéro, représenté par la variable `solutions`. On alloue suffisamment de mémoire pour la file (nombre d'options au carré pour avoir assez d'espace mémoire) et on enfile le contexte de la racine avec la fonction `enqueue`. On démarre alors le parcours en largeur. Cela fonctionne comme la fonction `solve` en enfilant au lieu d'explorer en profondeur :
   (a) On défile un contexte avec la fonction `dequeue`.
   (b) On couvre un objet sur ce contexte.

(c) Pour chaque option, on crée une copie du contexte avec `copy_ctx`, on choisit l'option sur cette copie, et on l'enfile. **Cette portion est exécutée en parallèle** avec `omp parallel for`, en enfilant avec une portion critique pour éviter les conflits lors de la manipulation de la file.

(d) On libère la mémoire du contexte défilé.

En comptant les noeuds défilés, on peut déterminer le nombre de noeuds de chaque niveau de l'arbre. Ainsi, on arrête cette procédure lorsque ce nombre est supérieur à un certain seuil, défini arbitrairement à 1000 afin d'occuper tous les threads lors de la résolution des sous-arbres.

3. **Exécution des tâches** : à la fin du parcours en largeur, la file contient les contextes à répartir entre les threads. Ainsi, on boucle sur la file en résolvant chaque contexte avec la fonction `solve` de la version séquentielle, puis en mettant à jour le nombre total de solutions trouvées. **Cette boucle est parallélisée** avec `omp parallel for reduction(+:solutions) schedule(dynamic)`, car chaque contexte peut être résolu indépendemment des autres. La réduction permet d'éviter une portion critique dans la boucle pour incrémenter le nombre de solutions, ce qui améliore la parallélisation. Enfin, on utilise une attribution dynamique des itérations de la boucle (`schedule(dynamic)`) car la durée de résolution des sous-arbres n'est pas identique entre eux.

4. **Résultats** : une fois tous les sous-arbres explorés, la variable `solutions` contient alors le nombre total de solutions. Le programme libère la mémoire occupée par la file avec `free_queue` et par l'instance avec `free_instance` et affiche le nombre de solutions.

Dans cette version, le parcours en largeur peut être assez lent pour certaines instances. Cependant, il permet pour les plus difficiles d'entre elles de répartir au mieux le travail entre les threads, ce qui réduit le temps d'exécution des tâches.

4 Parallélisation avec MPI

Cette partie présente la parallélisation du programme séquentiel avec MPI. Le fichier décrit est le suivant :

`mpi/exact_cover_mpi_bfs.c`

4.1 Chargement de l'instance

Il est préférable de ne pas charger l'instance depuis le système de fichiers sur tous les processus à la fois. En effet, avec beaucoup de processus, ceci pourrait saturer le serveur de fichiers. C'est pourquoi, le chargement de l'instance s'effectue de la manière suivante :

1. **Lecture du fichier** : le processeur principal (de rang 0) charge l'instance depuis le système de fichiers avec la fonction `load_matrix` définie dans la version séquentielle.

2. **Broadcast** : le processeur principal envoie ensuite à tous les autres processeurs l'instance grâce à un *broadcast*. Il utilise le fonction `send_instance` qui appelle plusieurs fois la fonction `MPI_Bcast` pour envoyer les différents attributs de la structure `instance`. Les autres processeurs reçoivent l'instance avec la fonction `recv_instance` qui effectue les mêmes appels à la fonction `MPI_Bcast` en allouant aussi la mémoire pour stocker l'instance.

4.2 Modèle patron-ouvrier

La solution avec MPI présentée implémente un équilibrage dynamique de charge en utilisant un **modèle patron-ouvrier**. Un processeur est désigné comme patron (celui de rang 0) et les autres sont les ouvriers. L'idée est la suivante :

- ➟ Le patron distribue les sous-arbres aux ouvriers un par un.
- ➟ Dès qu'un ouvrier est libre, le patron lui donne un sous-arbre à traiter.

4.2.1 Messages

On définit 5 types de messages échangés entre le patron et un ouvrier :

1. `AVAILABLE` : un ouvrier envoie au patron ce message pour dire qu'il est disponible.
2. `WORK_TODO` : le patron envoie à un ouvrier ce message pour lui donner un sous-arbre à traiter.
3. `WORK_DONE` : un ouvrier envoie au patron ce message pour le prévenir qu'il va recevoir le travail qu'il a effectué.
4. `WORK` : un ouvrier envoie au patron son travail. Cela correspond au nombre de solutions trouvées dans un sous-arbre.
5. `END` : le patron prévient un ouvrier qu'il peut s'arrêter.

Ces tags de messages sont définis dans une énumération.

4.2.2 Procédure

Voici la procédure de ce modèle patron-ouvrier :

(1) **Initialisation** : on initialise MPI, on obtient le nombre de processeurs et le rang du processeur. L'instance est chargée via la procédure décrite précédemment.

(2) **Création des tâches** : le patron démarre la résolution et crée les tâches à distribuer aux ouvriers avec la fonction `solve_bfs_root`. Cette fonction est similaire à `solve_bfs` de la version OpenMP : elle effectue un parcours en largeur de l'arbre en s'arrêtant à un certain niveau de sorte à pouvoir occuper tous les ouvriers. À la fin de l'exécution de la fonction, la file contient les contextes à distribuer aux autres processeurs. Le patron envoie ensuite aux ouvriers la longueur des listes d'options qu'ils vont recevoir avec un *broadcast*.

(3) **Exécution des tâches** : le patron démarre alors la distribution des sous-arbres :

   (a) Les processeurs ouvriers informent le patron qu'ils sont disponibles.

   (b) Le patron reçoit leurs messages et leur envoie une liste d'options à traiter.

   (c) Les ouvriers résolvent le problème pour le sous-arbre demandé avec la fonction `solve`. Quand un ouvrier a fini de traiter un sous-arbre, il prévient le patron qu'il va recevoir son travail et envoie le nombre de solutions trouvées. L'ouvrier repart alors à l'étape (a).

   (d) Le patron reçoit le travail et met à jour le nombre total de solutions trouvées.

   (e) S'il n'y a plus de travail à donner et qu'un ouvrier dit au patron qu'il est disponible, alors le patron lui dit de s'arrêter et l'ouvrier s'arrête.

   (f) Le patron s'arrête lorsqu'il a arrêté tous les processeurs ouvriers. Dans ce cas, il a alors reçu tout le travail demandé.

(4) **Résultats** : le processeur principal (le patron) affiche le nombre total de solutions et le temps d'exécution. La mémoire occupée par la file et l'instance est libérée. Enfin, on appelle `MPI_Finalize` pour terminer le programme.

Grâce à cet équilibrage, tous les processeurs sont occupés en quasi permanence. Le parcours en largeur au début du programme est effectué séquentiellement. Cependant, il permet de créer des tâches relativement équitables et en nombre suffisant. Ainsi, les processeurs sont exploités au mieux lors de l'étape d'exécution des tâches.

5 Parallélisation avec MPI et OpenMP

À partir des parallélisations avec OpenMP et MPI, voici une solution utilisant les deux librairies en même temps. Le fichier décrit est le suivant :

`hybrid/exact_cover_hybrid_bfs.c`

Voici le principe :

- On réutilise le même modèle patron-ouvrier que pour la solution avec MPI.
- Pour chaque ouvrier, on utilise la parallélisation avec OpenMP pour résoudre le problème sur les sous-arbres.

De plus, on note que le parcours en largeur effectué par le processeur principal pour créer les tâches contient la même portion parallèle décrite dans l'étape **2c** de la version avec OpenMP.

6 Checkpointing

Le répertoire `checkpointing` contient le programme de parallélisation avec MPI et celui avec OpenMP et MPI présentés précédemment effectuant en plus du *checkpointing*. Les fichiers décrits sont les suivants :

`checkpointing/exact_cover_mpi_cp.c`
`checkpointing/exact_cover_hybrid_cp.c`

L'idée est de sauvegarder périodiquement des *checkpoints* et de pouvoir repartir du dernier checkpoint.

On définit tout d'abord les variables suivantes :

- `cp_delta` : nombre de secondes entre deux *checkpoints* (défini à 60 secondes)
- `next_cp` : temps du prochain *checkpoint*
- `cp_filename` : nom du fichier de *checkpoint*

Deux fonctions ont été ajoutées pour gérer les *checkpoints* :

- `load_checkpoint` : charge un checkpoint
- `save_checkpoint` : sauvegarde un checkpoint

Un *checkpoint* contient les résultats trouvés jusqu'ici, ainsi que le travail restant. Il est donc composé des éléments suivants :

- Sur la 1ère ligne : le nombre de solutions trouvées.
- Sur la 2ème ligne : le nombre et la longueur des listes d'options restantes.
- Sur les lignes suivantes : les listes d'options restantes.

De plus, pour mieux gérer le travail fait et le travail à faire, des variables supplémentaires sont introduites :

- `task` : numéro de tâche à attribuer
- `task_recv` : numéro de tâche reçue par le processeur principal
- `tasks` : nombre total de tâches à effectuer
- `task_done` : tableau de booléens pour connaître les tâches effectuées
- `tasks_done` : nombre de tâches terminées
- `level` : longueur des listes d'options
- `options` : liste de listes d'options de longueur `level`
- `buffer` : buffer d'envoi des tâches avec numéro de tâche + liste d'options

En effet, les tâches n'étant pas forcément terminées dans l'ordre, le tableau de booléens permet de connaître les tâches réalisées et les tâches non réalisées. C'est aussi pour cela que le numéro de tâche est désormais envoyé avec les options afin de savoir quelle tâche a été complétée.

Les programmes présentés précédemment ont été un peu modifiés pour ajouter le *checkpointing*. Voici les ajouts :

(1) Le nom du fichier de *checkpoint*, contenu dans la variable `cp_filename`, correspond au pattern suivant :

```
in_filename.cp
```

Où `in_filename` correspond au chemin du fichier de l'instance du problème.

Par exemple, si `in_filename` est `../instances/bell13.ec`, alors le *checkpoint* associé est :

```
../instances/bell13.ec.cp
```

(2) La processeur principal recherche l'existence de ce fichier avec la fonction `access`. Deux cas sont possible :

(a) Si un *checkpoint* existe, alors le fichier est lu et les paramètres sont importés avec la fonction `load_checkpoint`. Ce programme importe les données précédemment citées, en créant une liste de listes d'options à distribuer aux autres processeurs (variable `options`).

(b) Sinon, la fonction `solve_bfs_root` est appelée afin de démarrer la résolution du problème à zéro. Cette fonction complète exactement les mêmes variables que la fonction `load_checkpoint`.

(3) Lors de l'envoi d'une tâche, le patron fournit en plus des options le numéro de la tâche. Lorsque l'ouvrier a terminé une tâche, il renvoie alors au patron ce numéro et le nombre de solutions trouvées. Le tableau de booléens peut alors être mis à jour.

(4) Après avoir reçu le travail d'un ouvrier, s'il est temps de sauvegarder un *checkpoint*, le processeur principal appelle la fonction `save_checkpoint`. Afin de gérer les cas où le programme s'arrête pendant l'écriture du *checkpoint*, on écrit le nouveau *checkpoint* dans un fichier temporaire, nommé `in_filename.cp.tmp` où `in_filename` correspond toujours au chemin du fichier de l'instance du problème. Pour connaître les listes d'options à faire, on utilise le tableau de booléens. On peut aussi connaître le nombre de tâches restantes en calculant `tasks - tasks_done`. Enfin, on écrase l'ancien *checkpoint* de manière atomique avec l'appel système `rename`.

(5) À la fin de la résolution, le fichier de *checkpoint* est supprimé.

Avec cette méthode, si le programme plante pendant l'exécution, on peut reprendre le calcul en utilisant exactement la même commande que lors du lancement. En effet, le programme s'occupe de chercher si un *checkpoint* existe et importe les données si c'est le cas.

Si on veut reprendre la résolution depuis le début, il suffit simplement de supprimer le fichier `.cp`.

7 Benchmarks

Cette partie présente les scripts en Python permettant de lancer des benchmarks pour les programmes parallèles présentés précédemment afin de mesurer leur efficacité.

Le dossier `benchmark` contient deux scripts en Python :

- `benchmark_json.py` : lance un benchmark à partir d'un fichier de configuration au format json. En sortie, le programme crée un fichier csv avec les temps d'exécution des programmes.
- `csvgraph.py` : crée des graphes à partir d'un fichier au format csv pour visualiser les résultats d'un benchmark. Ce script utilise la librairie **matplotlib**.

Ainsi, pour tester les parallélisations sur une instance du problème, il suffit de suivre la procédure suivante :

1. On crée un fichier json en spécifiant les options souhaitées [1]. On crée par exemple le fichier nommé `bell13_procs.json` pour tester les parallélisations MPI pur et MPI + OpenMP en faisant varier le nombre de noeuds.
2. On lance le benchmark avec la commande suivante :

   ```
   python3 benchmark_json.py bell13_procs.json
   ```

   Le fichier `bell13_procs.csv` est alors créé [2].
3. On visualise les résultats avec la commande suivante :

   ```
   python3 csvgraph.py bell13_procs.csv
   ```

   Les figures sont sauvegardées dans le fichier `png/bell13_procs.png` [3].

Le script `benchmark_json.py` est utilisable avec Grid5000. On peut alors réserver des noeuds et lancer le benchmark en spécifiant dans le fichier json l'option `"g5k": true`.

8 Résultats

Cette partie présente les résultats obtenus. Tous les tests ont été réalisés sur le cluster *gros* de Nancy. Les programmes ont été compilés avec l'option `-O3`. Voici un tableau récapitulatif du nombre de solutions de chaque instance, et une estimation de la durée de résolution avec le programme séquentiel :

Instance	Solutions	T
`bell12.ec`	4 213 597	1.2 s
`bell13.ec`	27 644 437	8 s
`bell14.ec`	190 899 322	60 s
`matching8.ec`	2 027 025	0.3 s
`matching9.ec`	34 459 425	4.6 s
`matching10.ec`	654 729 075	87 s
`pentomino_6_10.ec`	9 356	$\approx$ 8 s
`pento_plus_tetra_2x4x10.ec`	895 536	$\approx$ 10 min
`pento_plus_tetra_8x10.ec`	13 544 006 752	$\approx$ 47 jours
`pento_plus_tetra_8x8_secondary.ec`	1 207 328	$\approx$ 12 h

Les fichiers de résultats sont disponibles sous deux formats :

- **Tableaux** : dans le répertoire `benchmark/csv` sous forme de fichiers csv
- **Graphes** : dans le répertoire `benchmark/png` sous forme de fichiers png

Les graphes montrent le temps d'exécution et l'accélération obtenue pour les versions testées en fonction du nombre de threads ou de noeuds. Vous trouverez en annexes de ce rapport les graphes obtenus pour les différents benchmarks réalisés.

Le répertoire `benchmark/stdout` contient aussi la sortie standard des benchmarks.

Pour chaque instance, deux benchmarks ont été réalisés :

1. Voir les options en haut du fichier `benchmark_json.py`
2. Chemin du fichier de sortie spécifié dans le fichier de configuration json
3. Fichier de sortie automatiquement dans le dossier `png`

1. Pour les benchmarks `threads`, on lance le programme parallèle OpenMP sur 1 noeud avec x threads.
2. Pour les benchmarks `procs`, on lance :
   (a) Le programme parallèle MPI pur avec un processus par coeur sur x noeuds.
   (b) Le programme parallèle MPI + OpenMP avec un processus par noeud (et un thread par coeur) sur x noeuds.

8.1 Nombres de Bell

Pour les instances `bell12.ec`, `bell13.ec` et `bell14.ec`, on voit qu'au bout de 8 threads avec les versions OpenMP, l'ajout de threads supplémentaires ne permet plus d'accélérer l'exécution. En fait, les programmes parallèles ont été conçus pour résoudre les grosses instances ce qui explique cette majoration. En effet, le parcours en largeur effectué au début n'est pas utile pour ces instances car il s'arrête finalement au niveau des fils de la racine.

8.2 Couplages parfaits du graphe complet

D'après les graphes, on voit que les programmes parallèles avec parcours en largeur sont très bons pour l'instance `matching10.ec` : on a une accélération croissante. On voit ici la différence entre le programme `exact_cover_openmp_bfs.c` et `exact_cover_openmp_tasks.c`. En effet, l'accélération du second programme, qui ne parallélise que les fils de la racine, stagne au bout de 10 threads tandis que celle du programme avec le parcours en largeur continue de croître. En effet, pour ces instances, le parcours en largeur descend plus profondément dans l'arbre (voir les sorties des programmes dans le répertoire `benchmark/stdout`).

Les instances `matching8.ec` et `matching9.ec` sont trop petites pour observer une vraie accélération.

8.3 Polyominos

Les résultats obtenus avec les instances des problèmes de pentaminos et tetraminos sont les plus intéressants car ce sont les instances les plus longues à résoudre.

L'instance `pento_plus_tetra_8x8_secondary.ec` a été résolue en mode interactif avec 3 noeuds et 18 threads par noeud avec le programme parallèle hybride présenté dans le rapport. Le programme a trouvé **1 207 328** solutions en **22 min 52 sec**. En considérant que le programme séquentiel dure 12 heures, on obtient une accélération d'environ **31**.

L'instance `pento_plus_tetra_8x10.ec` a été résolue avec 10 noeuds et 18 threads par noeud par le programme parallèle hybride avec *checkpointing* nommé `exact_cover_hybrid_cp.c`. Le programme a trouvé le nombre total de solutions qui est **13 544 006 752** en **13 h 32 min**. Le calcul a été effectué en deux fois : le programme a été arrêté au bout de 5 minutes et a ensuite repris au dernier *checkpoint* enregistré (voir les sorties dans le répertoire `benchmark/stdout`). Si on considère que le programme séquentiel dure 47 jours, soit 1128 heures, on obtient une accélération d'environ **83**.

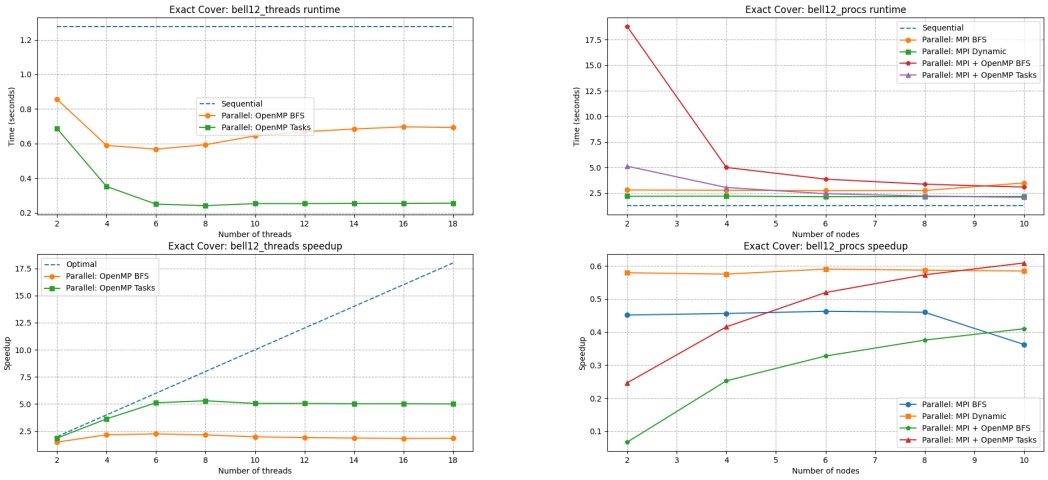
9 Conclusion

Les différentes versions parallèles présentées pour résoudre une instance du problème de couverture exacte sont pertinentes. En effet, on obtient une accélération non négligeable. Cela a permis de trouver le nombre de solutions d'une instance résoluble en plus d'un mois séquentiellement en une demi-journée parallèlement.

Le parcours en largeur est intéressant, car on peut choisir le niveau de parallélisation dans l'arbre de recherche, c'est-à-dire le nombre de tâches à créer. Cela permet d'occuper tous les coeurs à notre disposition. Il est clair que cette partie, globalement séquentielle, ne paraît pas efficace sur de petites instances. Elle est néanmoins très puissante pour les plus grosses instances.

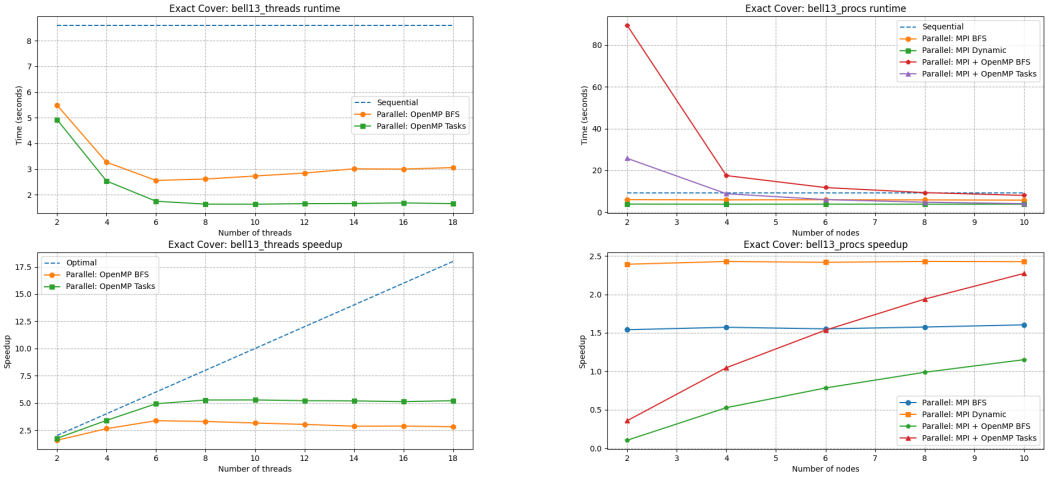
Annexes

Voici les graphes obtenus pour les différents benchmarks réalisés.



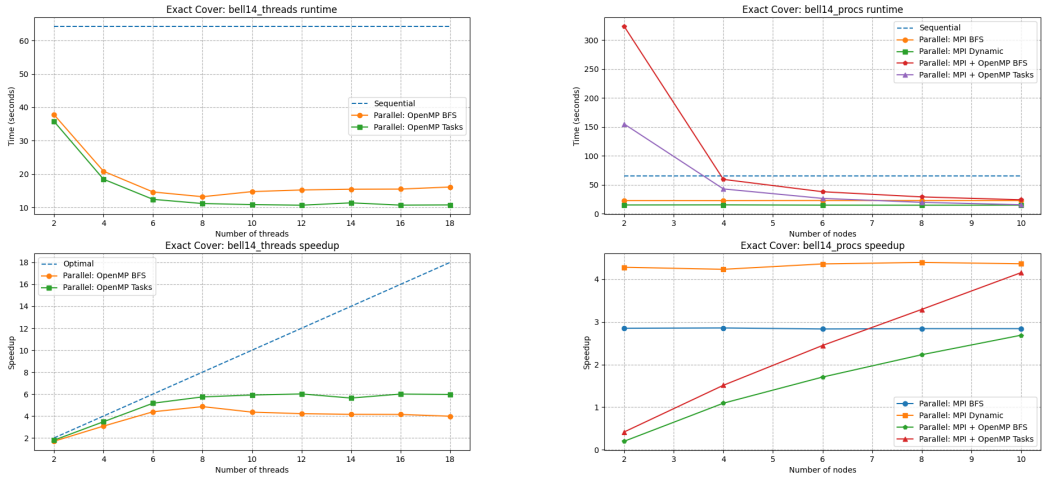
(a) Benchmark `bell12_threads` (b) Benchmark `bell12_procs`

FIGURE 1 – Benchmarks `bell12`



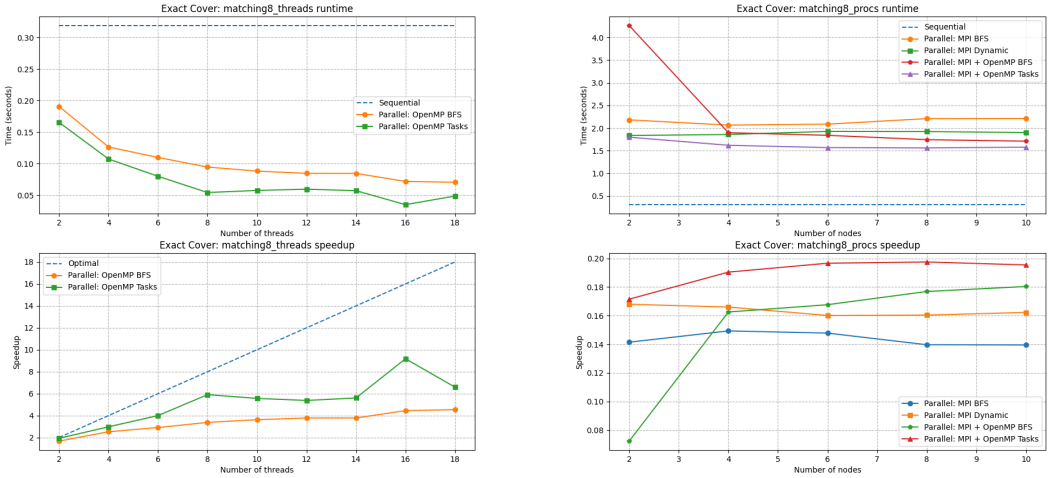
(a) Benchmark `bell13_threads` (b) Benchmark `bell13_procs`

FIGURE 2 – Benchmarks `bell13`



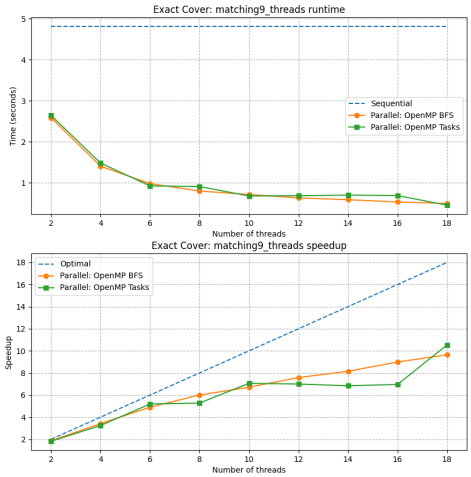
(a) Benchmark `bell14_threads` (b) Benchmark `bell14_procs`

FIGURE 3 – Benchmarks `bell14`

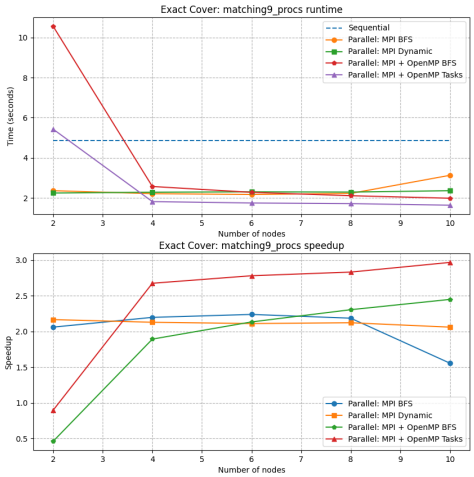


(a) Benchmark `matching8_threads` (b) Benchmark `matching8_procs`

FIGURE 4 – Benchmarks `matching8`

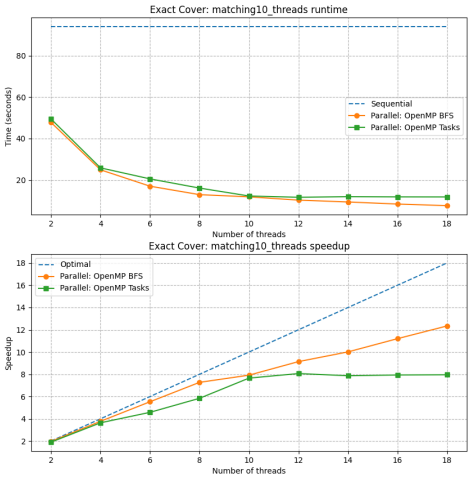


(a) Benchmark `matching9_threads`

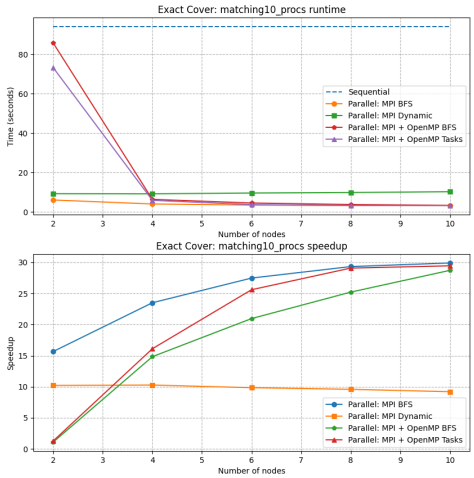


(b) Benchmark `matching9_procs`

FIGURE 5 – Benchmarks `matching9`

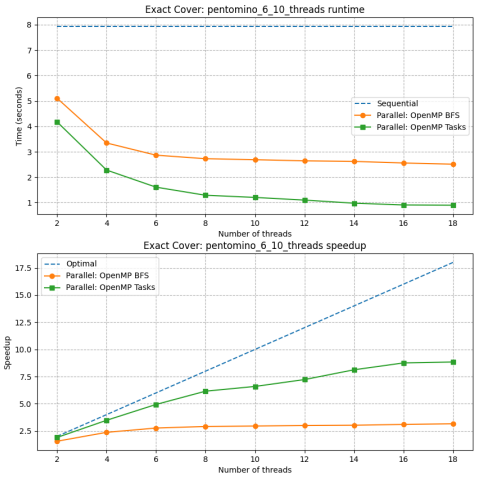


(a) Benchmark `matching10_threads`

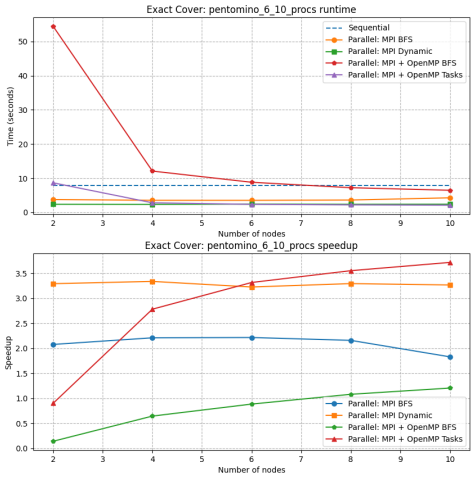


(b) Benchmark `matching10_procs`

FIGURE 6 – Benchmarks `matching10`

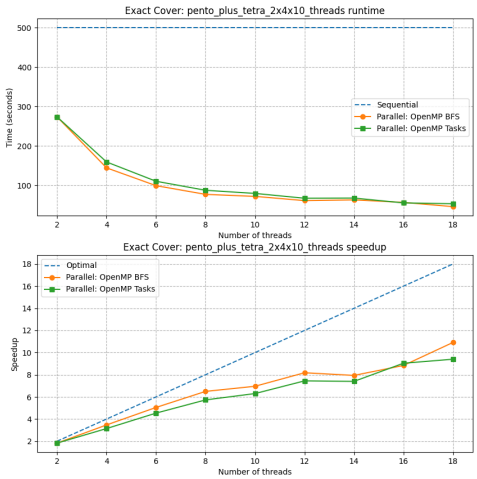


(a) Benchmark `pentomino_6_10_threads`

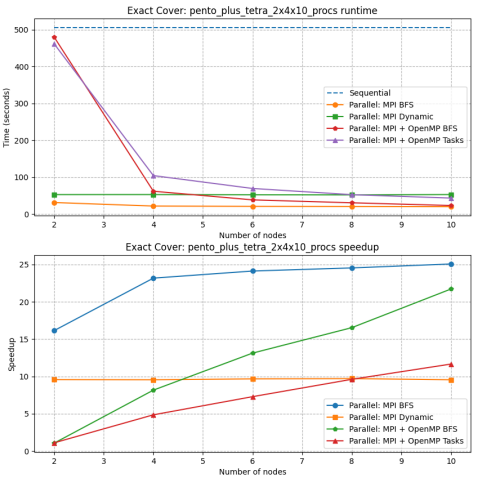


(b) Benchmark `pentomino_6_10_procs`

FIGURE 7 – Benchmarks `pentomino_6_10`



(a) Benchmark `pento_plus_tetra_2x4x10_threads`



(b) Benchmark `pento_plus_tetra_2x4x10_procs`

FIGURE 8 – Benchmarks `pento_plus_tetra_2x4x10_procs`