Yade Technical Archive (2019)
Document Link
www.yade-dem.org

Calcul distribué MPI pour la dynamique de systèmes particulaires

Bruno Chareyre[1], Robert Caulk[1], William Chèvremont[2], Thomas Guntz[4], François Kneib[1], Deepak Kunhappan[1], and Jean Pourroy[3]

[1]Université Grenoble Alpes, Laboratoire 3SR, Grenoble, France
[2]Université Grenoble Alpes, Laboratoire Rhéologie et Procédés, Grenoble, France
[3]École normale supérieure Paris-Saclay, Hewlett Packard Enterprise, Paris, France
[4]Inria, Grenoble Area, France

ABSTRACT

L'exécution parallèle de simulations DEM est étudiée, sur la base du module python « mpy » récemment développé pour le logiciel Yade-dem.org. Le module décompose le problème DEM en sous régions assignées à différents process MPI en utilisant la librairie python mpi4py. L'analyse des couts de communication conduit à identifier un communication des workers vers le master qui est exécuté de manière séquentielle et représente une part considérable du temps total pour les plus gros calculs. Une réimplémentation parallèle (Gather) permet de diviser ce temps par 10 dans certain cas. D'autres améliorations de sont apportées en relation avec l'infrastructure de build et le déploiement NIX, et un module de décomposition automatique en sous-domaines est introduit.

Keywords: Yade, MPI, parallelization, GENCI, GRICAD

1 INTRODUCTION DU SUJET

1.1 Context scientific et technique

i Domaine d'application La simulation de matériaux granulaires par la méthode des éléments discrets (DEM) a des applications en recherche (physique, mécanique, matériaux) et en ingéniérie (Génie des procédés, Génie Civil,...). Le script de test proposé pour le hackathon simule le dépôt d'une masse de particules sur un support rigide (fig. 1) sous l'action de la gravité.

ii Code utilisé (est-il connu dans le métier ?) Le sujet concerne le logiciel Yade-DEM, issu d'un projet de développement open source débuté au laboratoire 3SR (Grenoble) en 2004. Le code source est principalement écrit en C++. Les classes C++ sont systématiquement exposées en Python (boost::python), langage par lequel les utilisateurs construisent et contrôlent les simulations. Yade est un logiciel DEM connu internationalement, et l'un des principaux codes open source.

iii Etat de l'art Jusqu'à présent Yade-DEM est uniquement parallélisé en mémoire partagé (OpenMP). D'autres codes DEM (Esys, Liggghts) bénéficient d'un parallélisme en mémoire distribuée par des techniques de décomposition en sous-domaines dont s'inspire dans une certaine mesure l'extension Yade-MPI étudiée dans le hackathon. Une spécificité de Yade-MPI sur le plan technique est que la décomposition en sous-domaine est attaquée par le haut en langage Python plutôt que dans le C++ originel du code source.

iv Difficulté technique identifiée:

Complexité du code source Si l'extension MPI de Yade se présente comme un module Python de seulement 500 lignes, elle utilise de manière intensive l'interfacage C++/Python de Yade. Pour être autonome l'équipe doit donc comprendre non seulement la logique interne du module MPI mais aussi le wrapping Python des classes C++ (boost::python), et dans une certaine mesure l'architecture du code C++ lui même (111 343 lignes de code).

Faible maturité de certaines librairies. L'implémentation de MPI par la librairie python mpi4py n'est que partiellement documentée (la fonction Gather, utilisée plus loin, n'est pas documentée du tout). Contraintes liées au déploiement L'équipe est parvenu à configurer un environnement NIX permettant de compiler et installer Yade (qui implique de nombreuses dépendances) sur le cluster Gricad/Froggy dans la phase préparatoire pré-hackaton. Deux contraintes réduisant l'agilité du développement collaboratif demeurent : * impossibilité d'accéder au dépôt github en écriture depuis le cluster * chaque modification du fichier mpy.py (extension MPI de yade) demande une recompilation complète du paquet NIX yade (15min), à cause principalement du placement obscur des répertoires d'installation NIX.

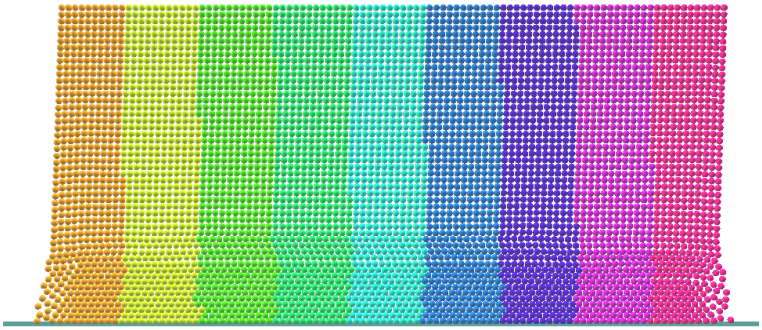

Figure 1. Masse de particules en cours de déposition sur un support rigide. Le code couleur indique les sous domaines calculés chacun par un processus MPI.

1.2 Objectifs

i Conduire à partir du code parallélisé et du script de test fourni des tests de scalabilité forte et faible pour un nombre de particules Np entre 10e4 et 10e7 et un nombre de coeurs Nc compris entre 1 et 100.

ii Produire un bilan détaillé des coûts associés aux différentes communications, en fonction du type de communication et pour les différentes tailles de problème, et proposer sur cette base une explication des résultats de scalabilité.

iii Améliorer l'implémentation initiale afin de réduire le temps d'exécution du script de test. L'amélioration sera quantifiée par la borne supérieure Cmax du nombre de Cundall (Np / wallTime) sur l'ensemble des combinaisons Np/Nc. Cmax sera mesuré avant et après le hackaton pour une itération temporelle standard et pour une itération incluant une redistribution des sous-domaines. Le rapport des Cmax avant-après quantifiera pour les deux types d'itération les progrès réalisés, la moyenne des deux donnera un indicateur global de réussite. L'objectif est d'atteindre au moins 2.

iv (Optionnel) En plus du script de test initial (dépôt de particules sous gravité) proposer d'autres conditions de simulations (ex. plus ou moins dynamique, plus ou moins homogène dans l'espace,...), représentatives des usages typiques de la DEM. Si les performances de l'implémentation sont substantiellement différentes pour un nouveau problème on associera à ce nouveau problème un facteur d'amélioration de Cmax agrégé aux précédents pour calculer l'indicateur global de réussite.

2 METHODOLOGIE APPLIQUEE

2.1 Les outils utilisés

- La version de Yade avec support MPI était fournie en début de Hackathon via la branche git dédiée (https://github.com/bchareyre/yade-mpi), git a donc étét naturellement utilisé pendant le hackathon.
- La compilation est configurée par cmake et exécutée par gcc, via le wrapper de compilateur mpicxx (spécifique MPI).
- L'environnement de compilation a été paramétré dans des expressions Nix.
- La soumission des tâches se fait par le gestionnaire OAR
- L'implémentation parallèle s'appuie sur les librairies mpi4py (python) et OpenMPI (C++)
- l'interfacage boost/python utilise la librairie boost::python.

- La librairie yappi (python) est utilisé pour profiler des fonctions spécifiques du module mpy.py.
- Les tests de profilage sont exécutés en mode 'batch' grâce à l'insertion dans le programme yade-batch originel de directives OAR. Le nouveau mode batch génère des scripts soumis au système de gestion des jobs (OAR).

2.2 Conception du code

i Scripts de test:
Les performances sont évaluées par des analyse paramétriques des temps de calcul en faisant varier de manière systématique - en mode batch - le nombre de sous-domaines et le nombre de particules. On fixe le nombre de particule totale pour les tests de scalabilité forte, ou le nombre de particule par sous-domaine pour les tests de scalabilité faibles. On exécute dans tous les cas 2000 itérations temporelles. Les résultats de chaque exécution sont collectés dans des fichiers textes globaux ainsi que dans des fichiers plus détaillés spécifiques à chaque calcul.

ii Outils d'analyse de performance:
Pour détecter les hot spot nous avons envisagé d'utiliser l'outil de profiling officiel de Linux, perf_event, qui possède son propre back-end dans le noyau lui permettant d'avoir accès à de nombreux compteurs matériels responsable de l'acquisition des événements à mesurer (Fig. 2).

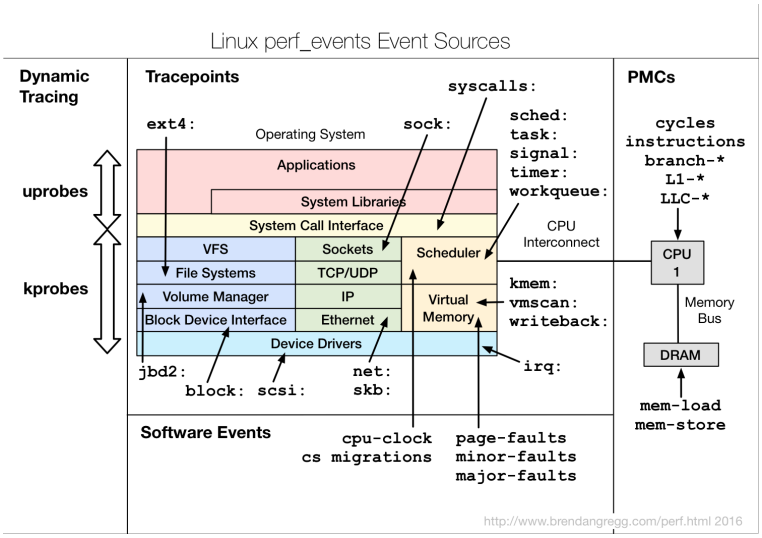

Figure 2. perf_event, l'outil de profiling officiel de Linux.

Malheureusement, le back-end ne permettait pas aux utilisateurs n'ayant pas les droits d'administrateur (root) de pouvoir accéder à ces informations très utiles. La seul informations que nous avons pu récupéré était le nombre d'instructions passé dans chaque fonction, ce qui est suffisant pour le travail de recherche de hot spot. La commande utilisé pour récupérer le profile de l'application est la suivante:

```
$ perf record -F 99 -g  mpiexec -np 4 ./yadempi-daily
```

Elle récupére le profile de l'application sur un serveur de calcul à une fréquence de 99Hz et enregistres les résultats dans un fichier qui peut être ensuite lu avec la commande

```
$ perf report --stdio
```

Pour faciliter la lecture de ce fichier, nous avons utilisé un outil développé par Netflix (https://github.com/brendangregg/FlameGraph). Les profils générés ressemblent au suivant:

Ce graph indique que 100% (la totalité de la largeur) de l'application (2e étage en partant du bas) se déroule dans yadempi-daily. La répartion du temps total entre les différentes fonctions apparaît dans les lignes supérieures. Par exemple ForceContainer::sync prend 17% du temps totale d'exécution. L'analyse détaillée de ce type de graph a montré que certaines communications MPI était particulièrement longues. Ceci a motivé une analyse plus détaillée des temps de communication, présentée dans la section suivante.

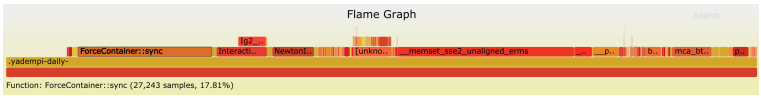

Figure 3. FlameGraph, utilisé pour faciliter la lecture de ce fichier de perf

iii Investigation des coûts liés aux communications

Il a été décidé que les optimisations réalisées pendant le hackathon concernerait exclusivement la parallélisation, c'est à dire que les sous-algorithmes séquentiels (et insécables) existants ont été considérés comme optimaux. En concordance avec l'objectif 2, et en l'absence d'une fonctionnalité de monitoring satisfaisante au sein de la librairie mpi4py, un mécanisme de mesure des temps de communication du module mpy.py a été développé pour chacun des changes de données. Les fonctions comm.send() et comm.bcast() de la librairie mpi4py ont été encapsulées dans des fonctions effectuant des mesures temporelles et ajoutant les rsultats dans un dictionnaire global. Exemple pour la fonction send() ci dessous, et résultats typiques dans le Figure 4:

```
def send(timing_name,*args,**kwargs):
timing_name=str(rank)+"_"+timing_name
ti=time.time()
#NOTE: la communication ci-dessous est bloquante -> timing possible
comm.send(*args,**kwargs)          #NOTE: appel de la fonction mpi4py
if(not timing_name in timings.keys()):
timings[timing_name]=[0,0]
timings[timing_name][0]+=1
timings[timing_name][1]+=time.time()-ti
```

	Call	Count	Tot. Time (s)
Worker 1	1_**mergeScene**_bound	49	1,0079
	1_**mergeScene**_states	49	0,0144
	1_checkcollider	50	0,0111
	1_updateDomainBounds	150	0,0069
	1_isendRecvForces	50	0,0038
	1_splitScene_intersections	50	0,0016
Worker 2	2_**mergeScene**_states	49	1,6921
	2_**mergeScene**_bound	49	0,8812
	2_checkcollider	50	0,0091
	2_updateDomainBounds	150	0,0055
	2_isendRecvForces	50	0,0030
	2_splitScene_intersections	100	0,0015
Worker 3	3_**mergeScene**_states	49	3,2408
	3_**mergeScene**_bound	49	0,8916
	3_checkcollider	50	0,0070
	3_updateDomainBounds	150	0,0056
	3_isendRecvForces	50	0,0032
	3_splitScene_intersections	50	0,0016

Figure 4. temps d'exécution des différentes communications

iv Optimisation ciblée:

Il apparaît dans les que les deux communications de la fonction mergeScene() sont responsables de 99% des temps totaux de communication, les efforts ont donc été concentrés sur ces communications. Nous constatons que le temps de communication croît linéairement avec le nombre de workers car le master reçoit les messages de manière séquentielle dans une boucle. Cela confirme le potentiel lié à l'optimisation de ces communications, en particulier pour un grand nombre de workers. Ci-dessous, les trois optimisations envisagées puis réalisées. Aucun gain de performance n'est attendu de cette modification pour les itérations standard car la fonction mergeScene() (Fig. 5) n'est pas appelée. Elle est utilisée uniquement dans l'étape de redistribution des domaines (aussi appelée split/merge) qui intervient généralement après plusieurs centaine d'itérations standards. Cela n'empêche pas un gain de performance notable, comme on le verra plus loin.

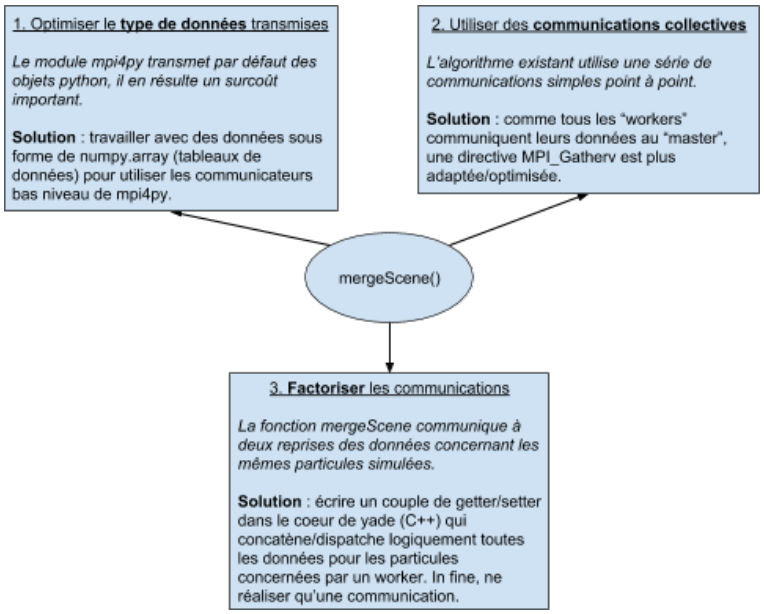

Figure 5. mergeScene() fonction

2.3 Originalité du code et du développment (librairie utilisée)

Une originalité du code développé est son caractère fortement hybride c++/python, qui se retrouve jusque dans l'implémentation MPI. En effet, même si la plupart des messages sont transférés par appels de fonctions du module mpi4py, certaines communications critiques pour la performance sont directement implémentées dans des fonctions c++ (librairie OpenMPI) qui forment le message, le communiquent, et l'interprètent. Ces fonctions sont in fine activées depuis python, sans que les données communiquées ne passent dans le contexte python à aucun moment.

2.4 Gestion des versions

Compte tenu de l'impossible accès en écriture sur github.com à partir du cluster Froggy la gestion de version a été abordée de manière conservative et court-termiste: pour éviter de résoudre les conflits plusieurs variantes du module mpy.py ont été introduites dans le dépôt github, puis modifiées incrémentalement par les différents sous-groupes de travail. L'intégralité des révisions est consultable sur github[1]. Les révisions par les membres de l'équipe sont au nombre de:

- 3 révisions pré-hackathon (préparatifs par Robert Caulk)
- 15 révisions les 6 et 7 décembre (ensemble de l'équipe + 1 bugfix par Bruno Chareyre)
- révisions post-hackathon, développements réalisé pour la plupart les 6-7 décembre mais pas immédiatement déposés (bissection), ainsi que téléversement des résultats de tests [2].

  Un second dépôt git [3] est dédié à la configuration de NIX:

- 5 révisions pendant le hackathon

(1) https://github.com/bchareyre/yade-mpi/commits/master
(2) https://github.com/bchareyre/yade-mpi/commit/3417d8d1c4015a2167b2ae6b700d513be7483e31

(3) https://github.com/williamchevremont/nix-yade

2.5 La répartition des tâches au sein de l'équipe

Brainstorming initial	Compilation/exécution sous NIX	Scripts de scalabilité	Profilage	Optimisations	Domain decomposition
All	William Robert	William Robert	Thomas François Jean	Thomas François Jean	Deepak Robert

3 RESULTATS

3.1 Performance

Les temps d'exécutions mesurés avant et après le hackathon sont présentés dans Figure 6. On présente le temps d'exécution total et la part de ce temps consacré aux étapes split/merge. Les résultats sont aussi donnés sous forme du nombre de Cundall conformément aux objectifs. Ce nombre, mesure classique en DEM, est défini comme le nombre d' itération*particule par seconde. Un programme qui effectue 10 itérations temporelles par seconde pour un problème à 100 000 particules a un nombre de Cundall égal à 106.

Les résultats montrent un gain de performance notable sur le wall time, qui est intégralement expliqué par le temps gagné dans les étapes de redistribution de domaine. L'accélération de ces étapes approche un facteur 10 pour les plus grands problèmes, ce qui conduit à une réduction du wall time par un facteur entre 2 et 4 dans la plupart des cas.

L'objectif n°3 concernant la performance est atteint (ci-dessous), malgré un mode de calcul qui nous semble assez défavorable: le facteur moyen 2.1 est inférieur à l'accélération effective des calculs (c'est dû au fait que l'on fait le rapport des optimums globaux plutôt qu'un rapport sur un problème de taille fixe).

Calcul des indicateurs de succès (objectif n°3)

Facteur d'accélération d'une itération courante:	1 (code inchangé)
Facteur d'accélération d'une étape redistribution (*):	3.2
MOYENNE:	2.1 (OBJECTIF 2)

(*) Optimum post hackathon Cmax=2.05e4 comparé à l'optimum pré-hackathon Cmax=0.64e4 (graph ligne 3, deuxième colonne).

Difficultés rencontrées? L'analyse paramétrique précédente a été limitée par des problèmes de système et de réseau. Premièrement, l'allocation de plus de 128 noeuds a aboutit à des erreurs systèmes imprévisibles, vraisemblablement liées aux communications entre switch du cluster. Des problèmes de mémoire sont également apparu pour les plus grands problèmes (>1.5million sphères). Il est apparu dans certain cas que les messages MPI étaient limités à 2GB.

3.2 Environnement HPC pour développement et tests

Le hackathon a conduit à divers aménagements de la procédure de compilation et de l'environnement NIX dédié. Ces améliorations sont des résultats importants au délà du hackathon dans la mesure où elles faciliteront grandement les travaux ultérieurs. Notamment, une configuration NIX orientée développement (1) a vu le jour et permet désormais la recompilation partielle et l'installation dans le répertoire utilisateur. Ces améliorations ont été documentées sur une page wiki de Gricad (2). D'autres changements concernent la procédure cmake ou le système de soumission de jobs OAR. (https://github.com/bchareyre/yade-mpi/commit/751c157c3b9869c6a05) .
(1) https://github.com/williamchevremont/nix-yade
(2) https://ciment.ujf-grenoble.fr/wiki/index.php/Yade_compilation

3.3 Décomposition en sous domaines

La version préliminaires du module MPI ne prévoyait pas d'allocation automatique des particules aux différents sous-domaines. L'allocation était faite dans le script de test, i.e. potentiellement laissée aux bon soin de l'utilisateur de base de yade-dem. Un module dédié à cette sous-tâche (non listée explicitement dans les objectifs mais paraissant primordiale) a été proposé à l'issu du hackathon. Parmis les algorithmes testés la bissection orthogonale recursive (orthogonal recursive bisection algorithm) est apparu le plus convaincant. Il consiste à couper un ensemble de point par des hyperplans, successivement orthogonaux aux différentes directions de l'espace (comme pour un kd-tree Fig. 7). A chaque niveau du graph le process tri, partitione, puis envoit les informations au processes du niveau suivant. L'implémentation se fait naturellement par des opérations MPI de broadcast et scatter:

```
If process rank <= 2^(level):
        then send : destination rank = process rank+2^(level)
If 2^(level) <= process rank < 2^(level+1):
        then receive : source rank = process rank { 2^(level)
```

L'algorithme est implémenté en python est disponible dans le module tree_decomp.py. Un exemple d'application est donné dans Figure 8.

4 CONCLUSION

i Hétérogénéité de l'équipe : peur d'être une faiblesse mais s'est révélé être une force.

ii Le lourd ticket d'entrée sur la mise en place de l'environnement de compilation et d'exécution sur le serveur Froggy. Cette étape a été très bien prise en main par William qui, grâce à ses connaissances de nix et du serveur, nous a permi de travailler dans de très bonnes conditions.

iii La durée de compilation du projet sur le serveur : le fonctionnement de Froggy fait que nous ne pouvions conserver le résultat du build ce qui entraînait donc une nécessité de builder à nouveau le projet entier sur le serveur (ce qui pouvait prendre entre 5 et 30min selon l'état du serveur). Très difficile de développer sans build incrémental. Heureusement deux solutions nous ont permis de travailler efficacement pour contrer ces problèmes :

- Une grande partie des modifications ont été faites sur le fichier "mpy.py", qui est un fichier python. Pas de problème de compilation. Pour les autres fichiers cpp (ex: Subdomain.hpp/cpp), il fallait attendre la compilation.
- L'utilisation de nos machines personnelles pour utiliser le build incrémental et tester nos modifications plus rapidement. C'était risqué car les comportement de nos machines et de celle du HPC Froggy pouvait différer.

A Matériaux supplementaire

- Video summary
- Poster summary
- YadeMPI GitHub branch

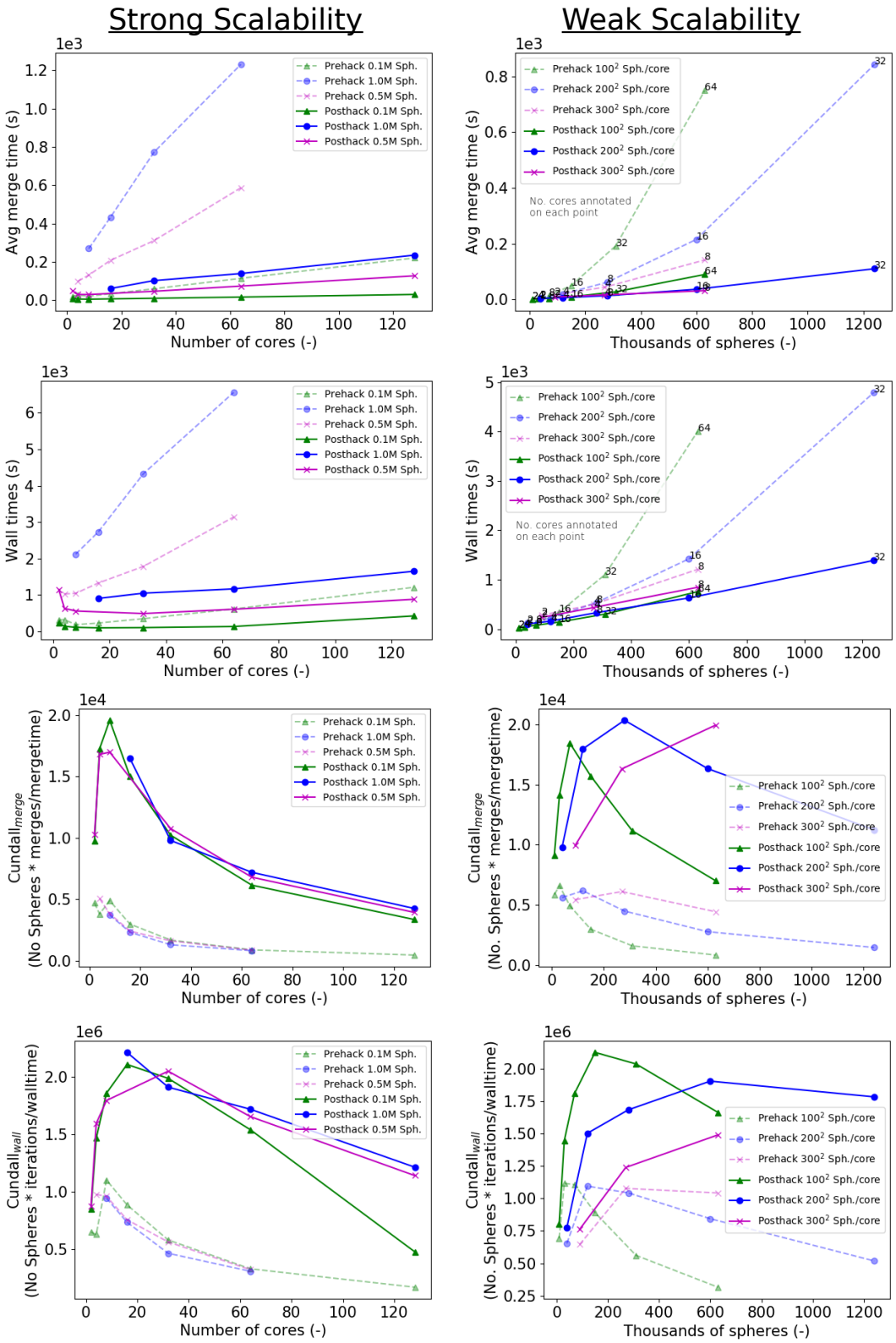

Figure 6. temps d'exécutions mesurés avant et après le hackathon

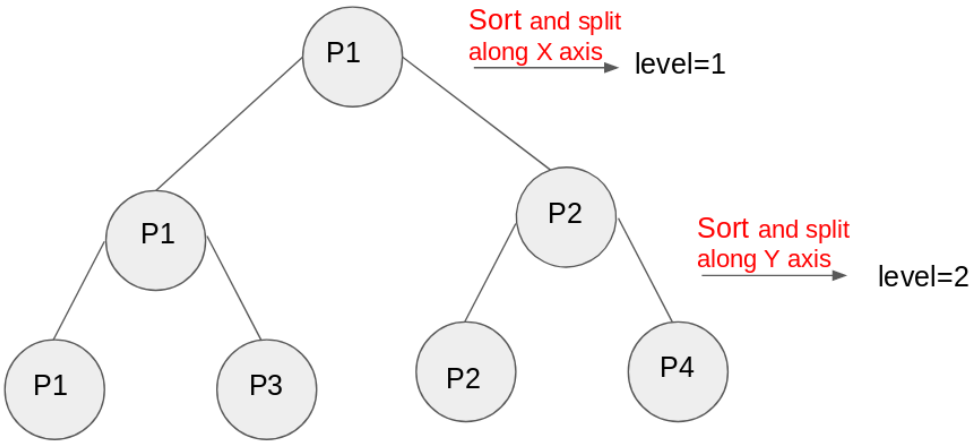

Figure 7. kd-Tree based partitioning

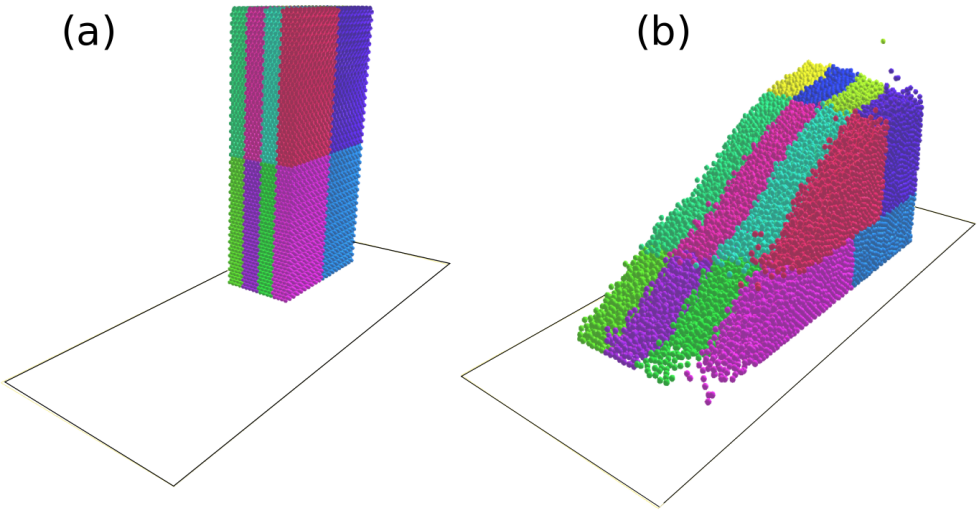

Figure 8. Dynamic load balancing example