UNIVERSITÉ DE BORDEAUX : SCIENCES ET TECHNOLOGIES

2015-2016

Programmation Multicoeurs et GPU : Rapport

Projet - Tas de sable abéliens

Table des matières

1 Choix effectués

1.1 Algorithme de référence

Un premier algorithme naïf a été écrit. Il s'agit d'une simple double boucles qui, pour chaque case, fait écrouler le tas de sable sur les quatre cases voisines. Nous utilisons cette algorithme comme référence pour vérifier que les algorithmes optimisés ou parallèles donnent des résultats corrects. Il s'agit de la méthode `compute_naïve`.

1.2 Optimiser l'utilisation du cache

Nous allouons un tableau de taille $DIM * DIM$ et pour nous y faciliter les accès, on alloue un deuxième tableau de taille DIM dont chaque case pointe sur une ligne différente (de taille DIM) du premier tableau.

Dans le cache L1 (32 KB), on peut faire rentrer $32 * 1024/sizeof(unsigned)/DIM$ lignes de la matrice soit 64 lignes pour la matrice de dimension 128 et 16 lignes pour celle de dimension 512. Il est donc naturel de parcourir la matrice ligne par ligne plutôt que via des tuile de 16 par 16 par exemple.

1.3 Algorithmes séquentiels

1.3.1 Division euclidienne

Une première optimisation simple consiste à effectuer des divisions euclidiennes pour faire ébouler un tas de sable d'un seul coup sur les cases voisines. L'avantage est que si un case contient un grand nombre de grains de sable, on va pouvoir la vider en une opération. Seulement, à l'itération suivante, un huitième des grains de sable envoyés chez les voisins reviennent sur la case de départ. Il s'agit de la méthode `compute_eucl`.

1.3.2 Prédiction de branchement

Une autre optimisation consiste à mieux utiliser la prédiction de branchement du processeur. À chaque itération et pour chaque case, on vérifie si la case dépasse une certaine valeur (la taille maximale d'un tas de sable avant éboulement). Or, comme d'une case à l'autre le contenu peut être radicalement différent, on peut peut pas statistiquement prédire si nous allons effectuer un changement (un éboulement) ou non. La prédiction de branchement n'est donc pas efficace ici.

Il faut alors effectuer toujours les mêmes opérations, qu'il y ait un changement ou non.

1.3.3 Diminuer les accès en écriture

Une autre approche que l'éboulement du tas de sable sur les cases voisines consistes à consulter le hauteur du tas de chacun de des voisins d'une case. En fonction de ces quatre hauteurs, on met à jour la valeur de la case, en faisant écrouler les tas de sables voisins sur la case, mais sans modifier le contenu des cases voisines. Avec cette approche, à chaque itération et pour chaque case, on effectue 5 accès lecture et un accès écriture contre 5 accès lecture et écriture pour l'autre approche. Il s'agit de la méthode `compute_eucl_swap`.

L'inconvénient de cette approche est qu'on ne peut pas lire et écrire dans la même matrice sinon les résultats seraient faux. Ainsi, on dispose d'une matrice d'écriture et d'une matrice de

lecture. Pendant qu'on lit la matrice de lecture, on écrit sur une matrice auxiliaire. Ensuite, après une itération, on échange la matrice de lecture et d'écriture.

Nous pouvons avec cette méthode facilement effectuer le même traitement qu'il faille ébouler une case ou non. Ainsi, nous pouvons éviter les prédictions de branchement incorrects.

1.3.4 Autres approches

Nous avons essayé d'autres approches comme la détection des zones stables (`compute_eucl_chunk`), ou l'utilisation d'opérations vectorielles pour tenter de traiter plusieurs cases d'un seul coup (`compute_eucl_vector`).

Pour `compute_eucl_vector` nous avons utilisé les instruction SIMD (SSE2) pour effectuer 4 calcul de case à la fois. Pour cela nous avons rempli un registre de 128 bits avec nos 4 cases (sur une même ligne donc continue en mémoire, virtuelle tout du moins).

Ensuite pour diviser par 4 nous avons déplacer notre registre de 2 bits vers la droite puis nous avons appliqué un masque 128 bits (`11000...11000...110000..11000...`) pour effacer les bits qui se retrouveraient sur la mauvaise case.

Enfin pour obtenir le résultat du modulo, nous appliquons un masque 128 bits 3,3,3,3 sur notre registre de départ.

Le principe de la séparation en chunk à été implémenté. Il s'agit de détecter les zones stables pour ne pas avoir à les refaire tant que les zones autour ne débordent pas sur elle.

1.4 Algorithmes multi-threads

1.4.1 Paralléliser une itération de l'algorithme

Une première solution multi-threads a été réalisée à l'aide d'OpenMP. Il s'agit de la méthode `compute_omp`.

Nous avons déjà vu section 1.2 que parcourir la matrice ligne par ligne est la façon qui permet d'utiliser au mieux le cache.4

Pour la plupart de nos solutions multi-threads, nous répartissons le traitement de chaque ligne de la matrice sur un ensemble de threads. Un thread se voit donc attribuer $chunk = (DIM - 2)/nthreads$ lignes successives et ceux de manière statique (`schedule(static, chunk)`). Dans les cas où le nombre de lignes à traiter par itération n'est pas multiple de $nthreads$, on peut noter que ce nombre est au moins multiple de 2, chaque thread traite donc successivement 2 lignes (`schedule(static, chunk)`).

Les autres politiques d'ordonnancement des threads ne sont pas intéressantes ici car les threads font en théorie tous la même quantité de calcul.

Comme pour `compute_eucl_swap`, lorsqu'on parcourt une case, on ne modifie pas les cases voisines. Cela permet de ne pas modifier les parties de la matrices gérées par d'autres threads.

Chaque thread travaille donc sur une matrice privée de taille $DIM * DIM$ allouée dans la pile (simple décalage du pointeur de pile, donc à priori rapide). Une première itération de l'algorithme est faite, chaque thread modifie sa propre matrice, puis on synchronise tous les threads avec une barrière. Cela nous permet de mettre à jour la matrice de départ avec les nouvelles valeurs suite au débordement de chaque case.

1.4.2 Paralléliser p-itération de l'algorithme

Cette méthode appelé `compute_omp_iter` a un fonctionnement identique à la méthode `compute_omp`. Seulement au lieu de synchroniser à chaque itération, un thread va effectuer le traitement de p-lignes appartenant à son thread voisin sur le morceau de matrice précédant et suivant, puis se synchronise avec les autres après p-itérations asynchrones.

Chaque thread travaille non pas sur une, mais deux matrices privées de taille $DIM * DIM$. Une itération consiste à traiter *chunk* lignes ainsi que les premières lignes des *chunk* voisin. Traiter p-itérations consiste à traiter $DIM * chunk + 2 * \sum_{0<it\leq p} it$ lignes de la matrice (sauf pour le premier et dernier morceau de la matrice).

Pour une itération donnée it, un thread lit dans une de ses deux matrices (la matrice A) puis écrit dans l'autre (la matrice B). À l'itération $it+1$, il échange les deux matrices et lit donc cette fois dans la matrice B puis écrit dans la matrice A. Après p-itérations, chaque thread met à jour le contenu de la matrice partagée par tous les threads.

Au maximum, on alloue $DIM * DIM * sizeof(unsigned) * nthreads + 1$ octets en mémoire.

1.4.3 Rapatriement des données

Une autre approche a été pensée pour éviter le rapatriement des nouvelles valeurs sur la matrice de départ. Celle-ci se base plus sur `compute_eucl_swap`. Chaque thread ne travaille plus sur une matrice privée mais on dispose de deux matrices partagées, une en lecture et une en écriture.

Chaque thread effectue une itération en lisant dans une matrice et en écrivant dans l'autre. Lorsque tous les threads ont terminés, on échange la matrice de lecture et d'écriture.

Il est nécessaire de synchroniser les threads à chaque itération pour être certain que tout le monde s'arrête lorsque toute la matrice est stabilisée. Il s'agit de la méthode `compute_omp_swap`.

1.4.4 Limiter au maximum les synchronisations

Une autre approche a été tentée, avec l'objectif d'utiliser des barrières le moins souvent possible. Il s'agit de la méthode `compute_omp_swap_nowait`. Cette méthode prend le pari de les threads vont évoluer à la même vitesse.

Ainsi, chaque thread tente de stabiliser la région qui lui est assignée sans communication avec les autres threads. Il est important de rappeler comment est réparti le travail. Chaque thread s'occupe de $(DIM - 2)/nthreads$ lignes successives de la matrice. Pour garder des résultats corrects, il faut insérer des dépendances sur les lignes frontières de deux threads. Ces dépendances sont illustrées sur la figure 1.

Pour chaque itération, le traitement de la première ligne d'un morceau de la matrice dépend de la terminaison du traitement de la dernière ligne de l'itération précédente du thread qui s'occupe du morceau précédent (dépendances "avants"). De plus, le traitement de la dernière ligne d'un morceau de matrice dépend de la terminaison du traitement de la première ligne de l'itération précédente du thread qui s'occupe du morceau suivant de la matrice (dépendances "arrières").

Pour implémenter ces dépendances, nous utilisons un tableau de sémaphores de taille le nombre de dépendances soit $2 * (nthreads - 1)$, et initialisées à 0. Chaque thread t (excepté le dernier thread) tente de décrémenter la sémaphore du thread $t + 1$ avant d'écrire la dernière ligne qui lui est attribuée, la ligne frontière entre deux threads. De son côté, le thread $t + 1$ incrémente la sémaphore lorsqu'il a terminé de traiter la première ligne qui lui est attribuée.

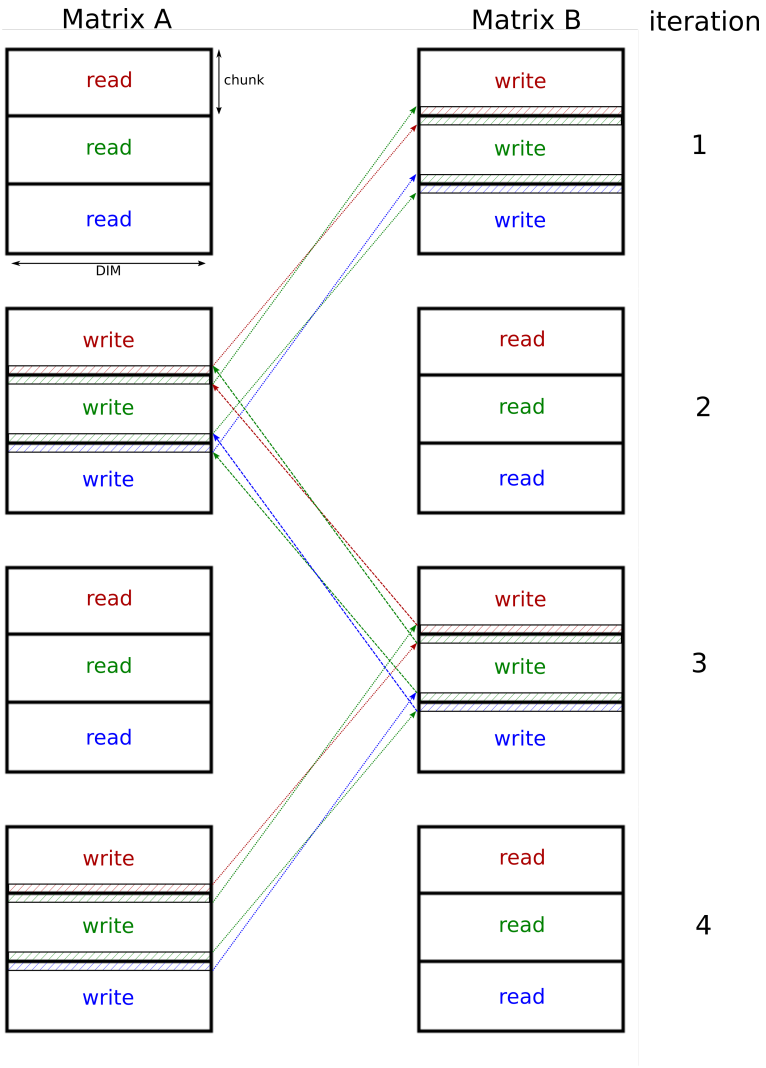

Figure 1 – Méthode `compute_omp_swap_nowait` : Utilisation de sémaphore pour retirer les barrières de synchronisation à chaque étape. L'exemple montre 3 threads qui effectuent 4 itérations de l'algorithme. Chaque thread traite *chunk* lignes de la matrice. Après chaque itération, on échange la matrice de lecture et d'écriture.

On espère en pratique que les threads travaillent à la même vitesse et que la sémaphore contient toujours la valeur 1.

Lorsqu'un thread a stabilisé sa partie de la matrice, cela ne veut pas dire qu'elle va rester stable, jusqu'à la fin. Les éboulement peuvent ne pas encore être parvenu à ses frontières comme cela peut être le cas lorsqu'on initialise une seule case à 100000. Ainsi, chaque thread ayant stabilisé sa matrice s'endort sur une condition (une autre sémaphore).

Le dernier thread à avoir stabilisé sa matrice, plutôt que de s'endormir lui aussi, réveille tous les autres en leur demandant d'effectuer une dernière itération. Cette itération doit permettre de savoir si des nouvelles valeurs ont été placées aux frontières pendant qu'on était endormi. Si c'est le cas, on recommence le processus depuis le départ. Sinon, l'algorithme se termine.

Il est très important pour cette méthode que le nombre de threads soit multiple de $DIM-2$ pour réduire au maximum le nombre de dépendances et donc de prises de sémaphores.

Pour éviter les inter-blocages, il faut aussi veiller à ne pas s'endormir sur la sémaphore du thread voisin si celui-ci a stabilisé son morceau de matrice et qu'il est endormi ou s'apprête à s'endormir.

1.5 Version OpenCL

Pour la version OpenCL, nous avons choisi de diviser le problème en workgroup de 16x16.

À chaque itération, on regarde côté CPU une case de mémoire GPU positionnée à 1 si il y a eu un changement.

Si c'est le cas on inverse (comme pour les versions swap CPU) le tableau de lecture et le tableau d'écriture (uniquement les pointeurs) puis on lance l'itération suivante.

Cela n'est clairement pas efficace puisque le CPU est obligé d'aller lire en mémoire GPU à chaque itération. Pour avoir de meilleur performance il aurait fallu faire p-itération dans le GPU avant de vérifier niveau CPU.

Cela explique en partie les mauvais résultats obtenus (voir section 2.1.4).

2 Résultats obtenus

Tous les tests ont été réalisés sur la machine hôte `worf` de la salle 203 décrite ci-après :
- NVIDIA GF100GL Quadro 4000
- Intel Xeon E7 v2/Xeon E5 v2/Core i7
- Deux noeuds NUMA :
  - cache L3 15 MB
  - 6 coeurs physiques avec hyperthreading :
    - cache L2 256 KB
    - L1d 32 KB
    - L1i 32 KB

On pourra retrouver la sortie de la commande `lstopo` en annexes (figure 9). Nous exécutons chaque méthode 5 fois puis nous faisons une moyenne des 5 mesures de temps. L'unité des mesures de temps est toujours le milliseconde.

2.1 Comparaison des algorithmes séquentiels

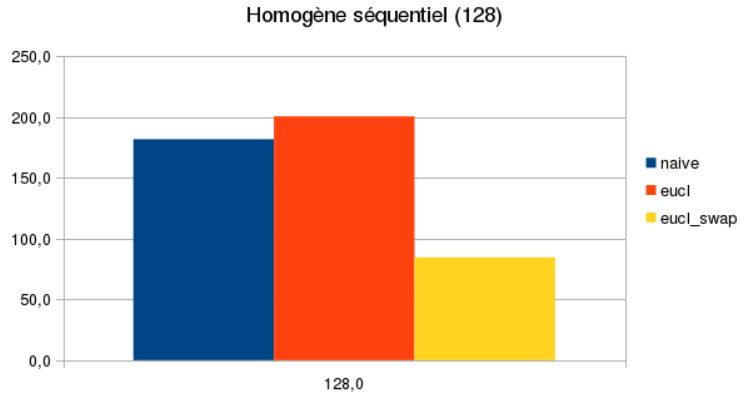

Figure 2

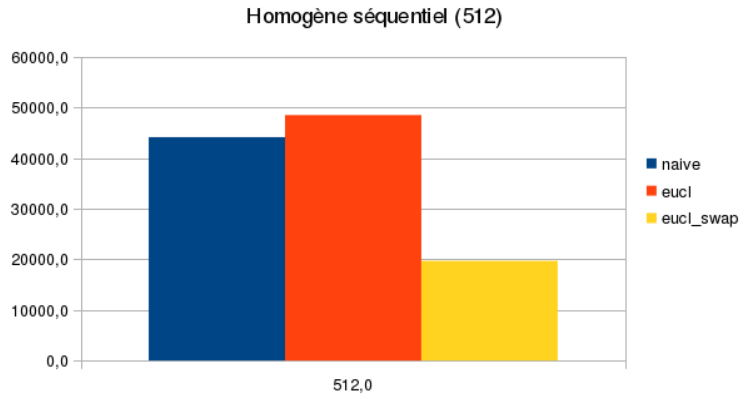

Figure 3

2.1.1 Calcul naïf *versus* division euclidienne

La version qui utilise des divisions euclidiennes `compute_eucl` est 1,1 fois plus lente (figure 2 et 3) que la version naïve `compute_naive` pour les configurations homogènes. Cela parait normal

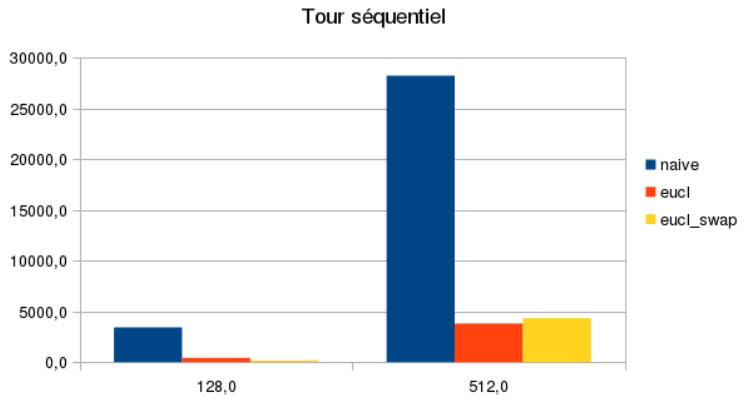

Figure 4

car les valeurs dans les cases à l'initialisation ne dépassent pas 5. Ici, l'utilité de faire une division euclidienne est limité.

En revanche, pour le cas où on place un tas de 100000 grains au centre, la version avec division euclidienne est 7 à 8 fois plus rapide car on réduit le nombre d'itération nécessaire pour arriver à stabilisation (figure 4).

2.1.2 Suppression des mauvaises prédictions de branchement

La suppression de la condition qui vérifie si la valeur d'une case est supérieure à la taille maximale d'un tas de sable améliore les performances d'un facteur 3 environ pour la fonction `compute_eucl_swap`.

Grâce à l'optimisation de la section 2.1.2, notre solution `compute_omp_swap` qui limite les accès en écriture sur les cases voisines est jusqu'à 2,5 fois plus rapide que les autres méthodes séquentielles pour le cas homogène (figure 2 et 3). Sans la suppression de la boucle, cette méthode est plus lente que les autres.

2.1.3 Vectorisation et détection des zones stables

Pour la méthode qui vectorise les calculs, nous n'avons pas obtenus les performances espérées. La gestion des différents registres prend plus de temps que d'effectuer 4 calculs. Cependant peut-être qu'avec des registre de 512 bits (AVX) nous aurions obtenus de meilleurs résultats. En revanche, les calculs sont corrects.

La version qui détecte les zones stable ne donne aussi pas de bons résultats en plus d'avoir quelques bugs.

2.1.4 Version OpenCL

La version OpenCL sur GPU donne aussi des résultats décevant (speed-up de 0,3 pour la matrice de dimension 128 par 128 et 1,3 pour celle de 512 par 512). Cela est en partie du à la détection de la terminaison de l'algorithme.

2.2 Synchronisation toute les p-irérations

Notre méthode `compute_omp_iter` n'est pas entièrement fonctionnelle. Les résultats sont correctes mais aucun gain de temps n'est à relever.

2.3 Algorithmes multi-threads

Pour le calcul du speed-up, nous prenons toujours comme temps de référence le meilleur temps séquentiel pour le cas de figure étudié.

Sur les figures 5, 6, 7, et 8 on peut voir que la méthode `compute_omp` est plus lente que la méthode `compute_omp_swap`. Cela peut s'expliquer par le surplus de temps s'engendre l'étape de rapatriement des données dans la matrice partagée, nécessaire après chaque itération.

Les pics de speed-up correspondent aux exécutions avec un nombres de threads multiples du nombres de lignes à traiter $(DIM - 2)$. Nous pouvons voir l'intérêt de découper la matrice en morceaux égaux pour tous les threads.

Sur les figures 6 et 8 nous pouvons voir que pour une matrice de dimension 512 par 512, le speed-up maximal de 8 est atteint pour un nombre de threads égal à 10. Ce chiffre est rassurant car c'est le nombre maximal de coeur physiques qu'on peut soliciter de la machine `worf` tout en ayant réparti le travail équitablement en morceaux de matrice.

Pour la matrice de taille 128, le speed-up maximal de 3 est atteint pour une exécution avec 6 threads. Nous pouvons nous demander pourquoi 9 threads ne représente pas le nombre de threads idéal, car dans ce cas nous solicitons plus de coeurs physiques tout en répartissant mieux les lignes de la matrices. Le problème est que pour une telle taille de matrice, l'exécution de l'algorithme est très courte : quelques dizaines de millisecondes. Ainsi, avec un nombre de threads trop important, le programme passe plus de temps à synchroniser tous ces threads qu'à calculer.

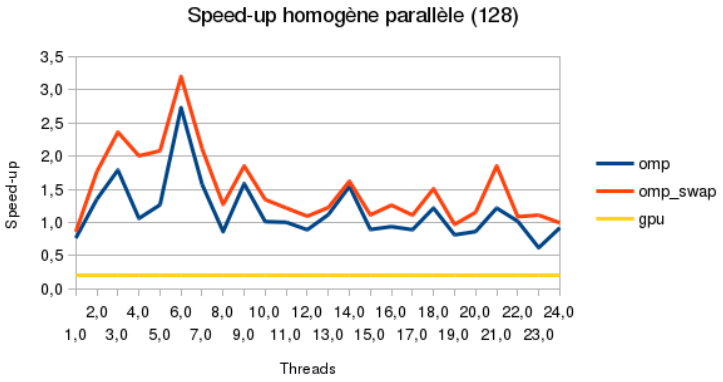

Figure 5

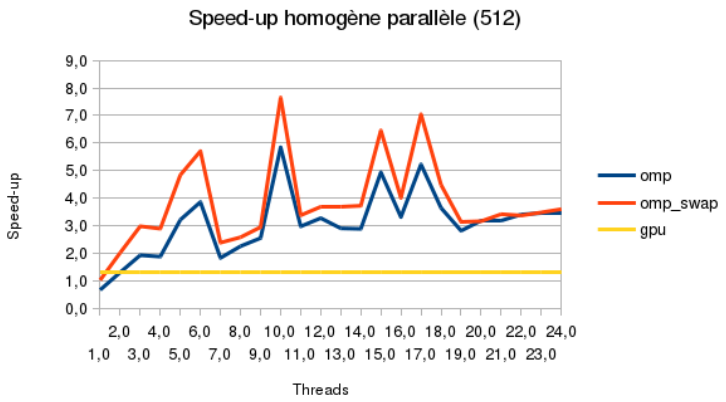

Figure 6

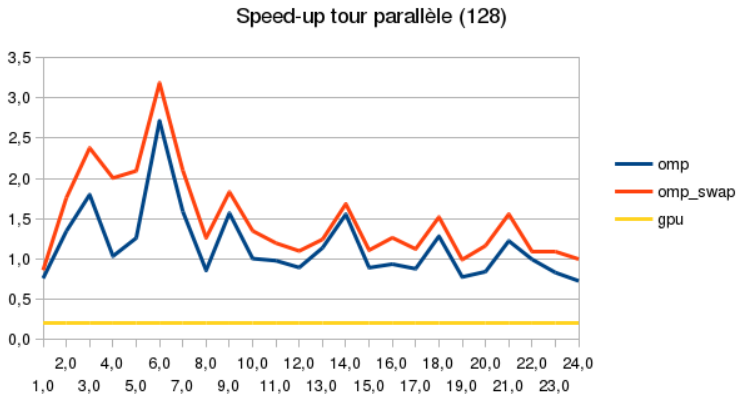

Figure 7

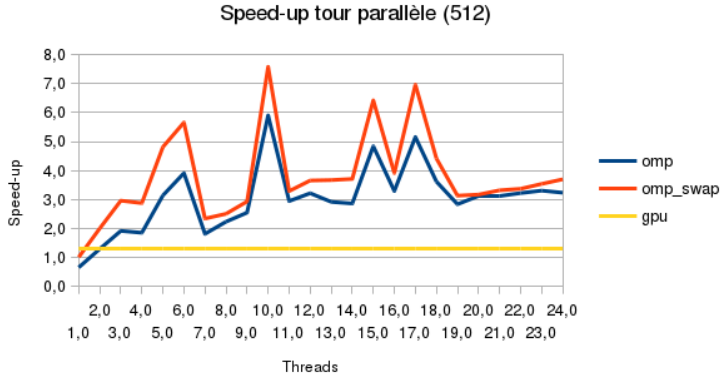

Figure 8

3 Perspectives

3.1 Réduction du champ de recherche

Plutôt que de détecter les zones stables, on pourrait détecter *la* zone instable de la matrice.

Pour réduire le temps que prend une itération, nous pouvons essayer de détecter le plus petit rectangle de la matrice qui contient toutes les cases instables.

Il faut donc garder 4 entiers pour garder en mémoire la colonne minimal et maximal ainsi que la ligne minimal et maximal entre lesquelles se trouvent l'intégralité des cases instables.

Cette méthode devrait permettre de réduire le temps d'exécution du programme pour le cas de la tour de sable.

En revanche, cela n'apporte rien en terme de parallélisation.

3.2 Suppression des barrières

Notre méthode `compute_omp_swap_nowait` n'a pas pu être terminée à temps. Il aurait été intéressant de vérifier dans la pratique si prendre le pari que les threads travaillent à la même vitesse (à une itération près) est gagnant ou pas.

En effet, on ne perd presque pas de temps à appeler la fonction `sem_wait()` lorsque la sémaphore contient la valeur 1, et chaque thread effectue cette vérification que une ou deux fois par itération.

En revanche, il se peut que la probabilité que la sémaphore contienne 0 à l'instant où un thread souhaite traiter une frontière soit suffisamment trop élevée pour que cette méthode soit intéressante (à cause des dépendances "arrières").

Annexes

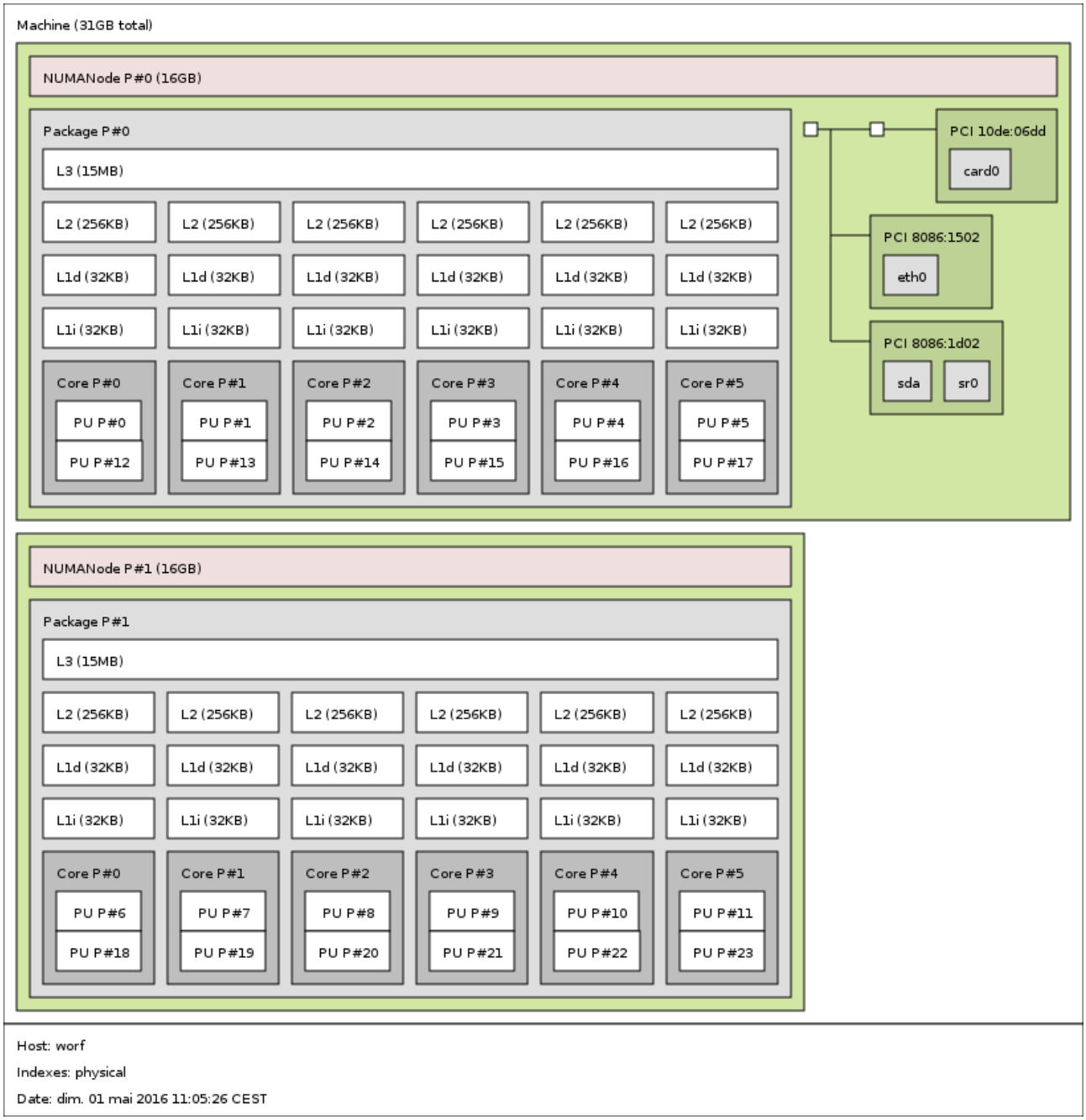

Figure 9 – Configuration de la machine hôte `worf`.