TDP 6 - Implémentation parallèle du jeu de la vie

Etienne THIERY - Youssef SEDRATI

30 janvier 2017

Dans ce projet nous développons dans un premier temps une implémentation séquentielle efficace du jeu de la vie. Puis nous parallélisons celle-ci en mémoire partagée, d'abord avec openMP, puis plus finement à l'aide des threads POSIX. Enfin, nous distribuons l'implémentation avec MPI.

1 Notice

1.1 Compilation

Le projet dépend de GCC, Cmake, Intel MKL, openMP et MPI. Sur Plafrim, les modules correspondant sont :

```
slurm/14.11.11
compiler/gcc/5.1.0
build/cmake/3.7.0
intel/mkl/64/11.2/2016.0.0
slurm/14.11.11
```

L'ensemble du projet peut être compilé via les commandes :

```
cmake .
make
```

1.2 Exécution

L'exécutable principal `main` prend 3 arguments obligatoires, dans l'ordre suivant :

- Le nom de l'implémentation à utiliser parmi les 6 que nous avons développées et qui sont présentées plus bas : `SEQ`, `OMP`, `PTHREADS`, `MPI_SEQ_SYNC`, `MPI_SEQ_ASYNC` et `MPI_OMP_SYNC`
- Le nombre d'itérations du jeu de la vie à calculer.
- La taille du damier à utiliser. Celle-ci doit être un multiple du nombre de threads multiplié par le nombre d'instances MPI.

L'exécutable lit également 2 variables d'environnement :

- `NUM_THREADS` qui permet de définir le nombre threads utilisés par chaque processus MPI. La valeur par défaut est 1.
- `BLOCK_SIZE` qui permet de définir la taille de bloc à utiliser pour les optimisations liées au cache présentées en section 2.3. La valeur par défaut est 4096.

Exemple pour 100 itérations sur un damier de 8192×8192, en utilisant l'implémentation `MPI_OMP_SYNC` sur 4 processus MPI utilisant 20 threads chacun, et des blocs de taille 2048 :

```
NUM_THREADS=20 BLOCK_SIZE=2048 mpiexec -n 4 ./main MPI_OMP_SYNC 100 8192
```

1.3 Tests de régression

Nous fournissons également un exécutable permettant de comparer les résultats de toutes nos implémentations pour des paramètres donnés. Il s'utilise de la même façon que l'exécutable principal, sans spécifier d'implémentation spécifique.

Exemple :

```
NUM_THREADS=2048 mpiexec -n ./tests 100 512
```

1.4 Benchmarks

L'ensemble des expérimentations et benchmarks présentés plus bas peuvent être reproduis sur plafrim à l'aide de script slurm fournis. Avec `path` le chemin vers le dossier du projet :

```
   sbatch -p mistral --workdir path path/jobs/cacheOptimization.sh
sbatch -p mistral --workdir path path/jobs/benchSEQ.sh
sbatch -p mistral --workdir path path/jobs/benchOMP.sh
sbatch -p mistral --workdir path path/jobs/benchPTHREADS.sh
sbatch -p mistral --workdir path path/jobs/benchMPI_SEQ_ASYNC.sh
sbatch -p mistral --workdir path path/jobs/benchMPI_SEQ_SYNC.sh
sbatch -p mistral --workdir path path/jobs/benchMPI_OMP_SYNC.sh
```

Une fois les résultats obtenus, les figures de ce rapport peuvent être reproduites à l'aide de script gnuplot fournis également.

```
   gnuplot path/plots/cacheOptimization.p

   bash path/jobs/mergeBenches.sh
gnuplot path/plots/speedupOMP.p
gnuplot path/plots/speedupPTHREADS.p
gnuplot path/plots/speedupMPI_SEQ_ASYNC.p
gnuplot path/plots/speedupMPI_SEQ_SYNC.p
gnuplot path/plots/speedupMPI_OMP_SYNC.p
```

2 SEQ : version séquentielle efficace

Avons de travailler sur des implémentations parallèles, nous avons pris le temps de développer une implémentation séquentielle la plus efficace possible (voir `runSeq.c|h`).

L'essentiel de ce travail a consisté à définir une structure `Board` représentant un damier du jeu de la vie, et des primitives efficaces pour compter le nombre de voisins d'une cellule, et pour mettre à jour l'état d'une cellule en fonction de son nombre de voisins.

2.1 Comptage des voisins : accès mémoires séquentiels

Une première optimisation que nous avons implémentée consiste à s'assurer que les accès mémoires lors du comptage des voisins sont séquentiels autant que possible.

Une méthode de comptage naïve consiste à itérer sur chaque cellule, à lire l'état des 8 cellules adjacentes, puis à écrire le nombre de voisin. Les 8 accès en lecture sur les états des cellules adjacentes ne sont pas séquentiels.

Une méthode plus efficace consiste à travailler par colonne (ou en ligne selon le stockage mémoire). Compter les voisins des cellules de la colonne c_i consiste alors à sommer les valeurs de 8 colonnes :

— c_{i-1} décalé d'un élément vers le haut,
— c_{i-1},
— c_{i-1} décalé d'un élément vers le bas,
— c_i décalé d'un élément vers le haut,
— c_i décalé d'un élément vers le bas
— c_{i+1} décalé d'un élément vers le haut,
— c_{i+1},
— c_{i+1} décalé d'un élément vers le bas,

Les accès mémoire sont alors largement séquentiels.

Voir la fonction `count_neighbours_col` de `Board.c` pour l'implémentation exacte.

2.2 Comptage des voisins : vectorisation

Ces sommes sur des vecteurs d'éléments contigus en mémoire sont extrêmement propices à la vectorisation. Plutôt que de faire cela manuellement, nous avons fait appel à l'implémentation MKL du BLAS `axpy`, qui est non threadée mais vectorisée.

Voir la fonction `count_neighbours_col` de `Board.c` pour l'implémentation exacte.

2.3 Comptage des voisins : optimisation cache L1

Cette méthode est également propice à une optimisation de la localité temporelle des accès.

En effet quand on ajoute la colonne c_i puis cette même colonne décalée d'un élément vers le haut, et vers le bas, on fait 3 fois les mêmes accès en lecture (à quelques éléments prêts). Si la colonne tient en cache L1 (en plus de la colonne résultat), les deuxième et troisième accès sont alors largement accélérés.
D'autre part, une fois que l'on a compté le nombre de voisins des cellules de la colonne c_i, on passe à la colonne c_{i+1}, et on relit à nouveaux les mêmes éléments des colonnes c_i et c_{i+1}. Pour que ces éléments soient dans le cache à ce moment la, il faut que celui-ci puisse contenir 2 colonnes entière (en plus de la colonne résultat).

Pour être dans ces 2 cas, on peut utiliser la méthode bien connue consistant à découper les colonnes, traiter d'abord les premiers morceaux de toutes les colonnes, puis les seconds morceaux, etc. Cela revient à itérer sur les colonnes de blocs lignes, plutôt que sur des colonnes entières.
La hauteur des blocs lignes doit être choisie intelligemment en fonction de la taille du cache. On voit que des blocs plus petits assurent que les éléments lus sont la plupart du temps déjà dans le cache (bonne localité temporelle). Cependant, ils diminuent les bénéfices de la vectorisation, qui est elle est plus efficace sur des gros vecteurs.

Il faut donc trouver le compromis optimal. Pour cela, nous avons simplement mesuré le speed-up obtenu grâce à cette optimisation pour différentes tailles de bloc.

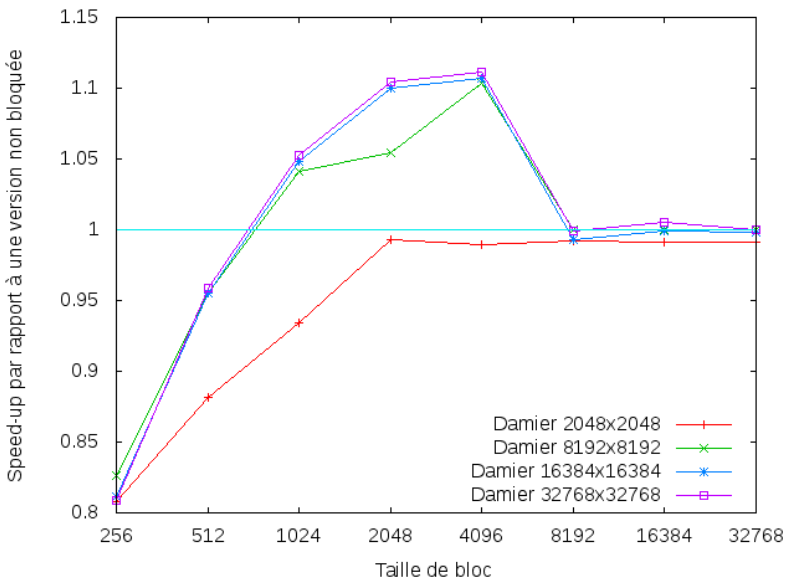

FIGURE 1 – Impact de la taille des blocs sur le temps d'exécution de 100 itérations

On peut voir que pour un "petit" damier (2048×2048), utiliser des blocs trop petits dégrade les performances à cause du sur-coût lié à la vectorisation sur de trop petits vecteurs. Passé une certaine taille, les performances reviennent à la normale, mais ne sont pas améliorées. C'est tout à fait logique : les caches L1 des noeuds Mistral font 32ko, et peuvent donc contenir 8 colonnes

entières de 2048 flottants, la réutilisation est donc déjà optimale.

En revanche, pour un damier plus conséquent (8192×8192 ou plus) on distingue clairement 3 cas de figure :
— Performances de la vectorisation dégradées pour des blocs trop petits.
— Speed-up pour une compromis sur la taille des blocs permettant de tirer bénéfice de la vectorisation et d'effets de cache.
— Performances sous optimales (speedup égal à 1) pour des blocs trop gros pour bénéficier d'effets de cache.

La taille de bloc optimale pour un cache de 32ko semble être 4096. Un cache de 32ko peut contenir 2 vecteurs de 4096 flottants, donc un seul vecteur en plus du vecteur résultat. Il semble donc qu'avec la taille optimale on ne bénéficie que du premier effet de cache présenté plus haut.

Voir la fonction `count_neighbours_block` de `Board.c` pour l'implémentation exacte.

2.4 Mise à jour de l'état des cellules

Bien que les prédicteurs de branchement soient aujourd'hui assez efficaces, les conditionnelles restent mauvaises pour les performances.

La méthode naïve de mise à jour de l'état d'une cellule consiste à formuler des conditionnelles sur l'état précédent et le nombre de voisins. Une simple astuce permet de se débarrasser de celles-ci, et d'améliorer les performances tout en permettant de changer les règles du jeu très facilement.

Une cellule peut être morte ou vivante, et son nombre de voisin est compris entre 0 et 8. Il n'y a donc que $2 * 9 = 18$ cas de figures à prendre en compte, avec chacun un état résultant déterministe. On peut ainsi représenter les règles du jeu par un tableau à deux dimensions de 2×9 booléens.
On peut alors déterminer l'état suivant de chaque cellule en lisant dans la bonne case du tableau. Cela améliore les performances, très légèrement car le comptage des voisins est l'opération la plus coûteuse.

3 OMP : version en mémoire partagée avec synchronisation lourde

Nous avons ensuite développé une version parallèle à l'aide de directives OpenMP (voir `runOMP.c|h`).

La quantité de travail à effectuer par chaque thread étant uniforme, nous avons utilisé une politique d'ordonnancement statique (`schedule(static)`). Des essais préliminaires ont confirmé expérimentalement que c'était la politique qui donnait les meilleurs résultats.

En parallélisant d'abord une implémentation séquentielle naïve, on a pu voir qu'il était légèrement plus efficace de paralléliser uniquement sur une des deux dimensions (ici les colonnes, puisqu'on est en stockage par colonnes) que sur les deux. La différence n'était pas flagrante, mais cela peut se comprendre : en parallélisant sur deux dimensions, le sous domaine d'un processeur peut avoir une forme moins régulière, sur lequel les accès en mémoire sont moins réguliers.

On a ensuite parallélisé notre version séquentielle efficace. Ici le choix de la dimension à paralléliser était simple : la taille de bloc optimale de 4096 déterminée plus haut étant relativement importante, la boucle sur les blocs lignes a très peu d'itérations, et ne permet pas un équilibrage de la charge satisfaisant. On parallélise donc la boucle sur les colonnes à l'intérieur de chaque bloc ligne. Comme la politique statique divise les itérations en n groupes d'itérations consécutives de taille égales où n est le nombre de processus, cela revient donc à attribuer à chaque chaque processus sur un bloc-colonne sur lequel il travaille par bloc ligne.

La figure suivante montre le speed-up obtenu par cette implémentation pour 4 tailles de damier différentes.

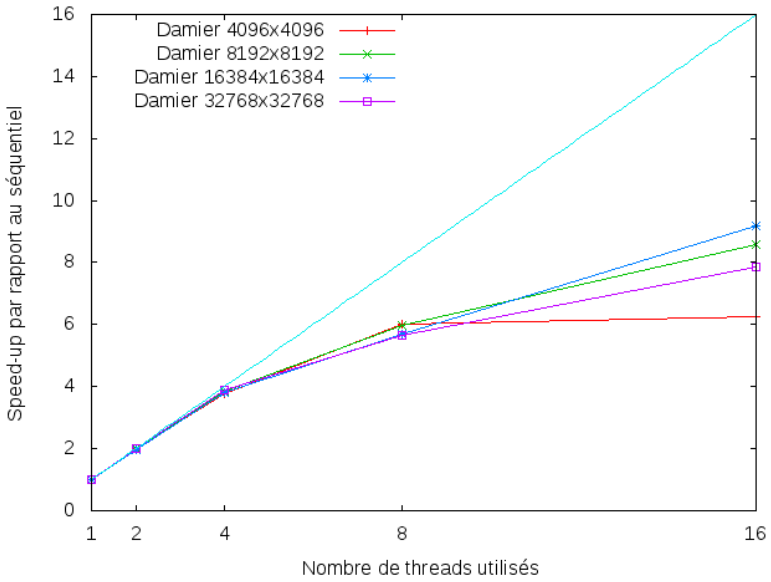

FIGURE 2 – Courbes de speed-up de l'implémentation OMP (sur 100 itérations)

Les courbes ont une allure normale : le speedup croît avec la taille du problème, et l'efficacité décroit lorsque le nombre de threads utilisés augmente. Jusqu'à 8 processeur, les résultats sont satisfaisant, avec une efficacité d'environ 0.75 sur toutes les tailles.

Le premier rentrant en jeu ici est la classique amélioration des performances par l'augmentation de la taille du problème. Elle vient du fait que la parallélisation induit un surcoût initial lié à la création des threads, et le partage du travail entre ceux-ci. La politique statique a pour avantage d'être très facile à mettre en oeuvre, et d'avoir un coût constant vis à vis du nombre de processus. De plus, ici, les seules synchronisation causées par les directives openMP sont globales (barrière à la fin de l'omp for) et ne devraient pas poser de problèmes de performance car la charge de travail entre les processus est très homogène, et que le programme s'exécute sur une machine sur laquelle très peu d'autres processus s'exécutent.

On note tout de même une irrégularité : Pour 16 threads, le speed-up croît avec la taille du problème jusqu'à 16k, mais chute pour 32k.

Notons que les 16 threads sont répartis dans 2 socket de 10, que chaque socket a 64Go de RAM et 25Mo de cache L3.
Un damier de taille 16k occupe 2 Go et un damier de taille 32k occupe 8 Go. Les deux sont bien au delà de la taille du cache L3 donc il ne devrait pas y avoir de différence en terme d'efficacité des caches. Les deux sont bien en dessous de la capacité de la mémoire centrale donc ce n'est pas un problème de swap non plus. Une inspection rapide à l'aide de l'outil `perf` montre que le taux de cache miss sur le cache L3 est le même dans les 2 cas (50%).

L'explication réside surement plutôt dans le caractère NUMA de la machine : l'ensemble des données est stockée dans un des deux bancs mémoire, donc les temps d'accès de la moitié des processeurs sont meilleurs que ceux de l'autre moitié. Les premiers attendent donc systématiquement les seconds à chaque barrière OMP, et d'autant plus longtemps que la taille du problème est grande. Il semble que ce déséquilibre devienne prohibitif pour un damier de 32k. Utiliser une politique d'ordonnancement dynamique ne résout pas vraiment le problème, car celles-ci sont tellement plus couteûse qu'il n'est pas rentable d'équilibrer la charge... Une solution, que nous n'avons pas eu le temps de mettre en place serait donc d'utiliser des binds `hwloc` pour s'assurer que le damier est réparti sur les mémoires des différentes noeuds NUMA, et que chaque thread s'exécute sur un

processeur du noeud où se trouve les données sur lesquelles il travaille.

4 PTHREADS : version mémoire partagée avec synchronisation fine

Les directives OpenMP ne nous permettent qu'une synchronisation assez grossière : chaque threads attend que tous les autres threads aient fini l'itération avant de passer à la suivante.

En utilisant les Pthreads, on peut procéder plus finement. En réalité, un thread n'a besoin de se synchroniser qu'avec les threads en charge des blocs colonne voisins, pour respecter les 2 contraintes suivantes :

— Un thread ne peut écrire sur une de ses frontières pour l'itération i qu'une fois que le voisin correspondant a lu les valeurs de l'itération $i-1$.
— Un thread ne peut lire les valeurs de la frontière d'un voisin pour l'itération i qu'une fois que le voisin a écrit les valeurs valeurs de l'itération $i-1$.

Nous avons ainsi utilisé pour chaque thread 2 sémaphores (un pour indiquer une lecture, l'autre pour indiquer une écriture) par frontière (gauche et droite). De plus on sépare le calcul effectué par chaque thread en 2 ensemble, frontière / le reste, de façon à retarder le plus possible les attentes sur un sémaphore :

— Le thread lit les frontières de ses voisins, puis le signal via les sémaphores.
— Le thread traite le reste de son sous domaine, pour laisser aux autres threads le temps de lire ses frontières.
— Le thread écrit sur ses frontières, et le signale via les sémaphores.

Cela permet d'avoir une certaine désynchronisation à travers le domaine et d'estomper les effets d'un éventuel déséquilibre de charge. On obtient ainsi les courbes de speed-up suivante, sur lesquels on peut faire exactement les mêmes commentaires que pour la version OMP, mais qui sont globalement légèrement meilleures sur les grandes tailles : speedup de 10 au lieu de 9 sur un damier de 16k, et speedup de 9.5 au lieu de 8 sur un damier de 32k.

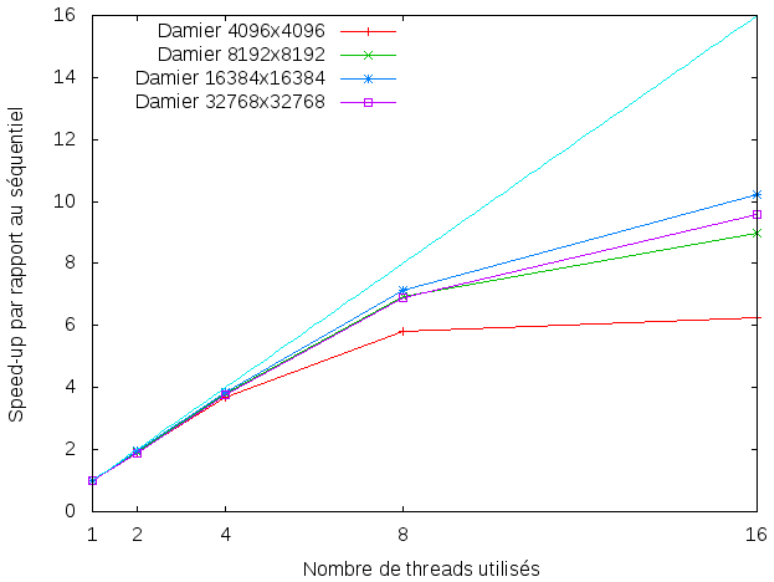

FIGURE 3 – Courbes de speed-up de l'implémentation PTHREADS (sur 100 itérations)

Voir `runPthreads.c|h` pour l'implémentation exacte.

5 MPI_SEQ : version mémoire distribuée

Nous avons ensuite développé 2 versions utilisant MPI, et séquentielles pour chaque processus MPI.

La première version utilise des communications synchrones. C'est ainsi une simple réécriture de la version OMP en remplaçant les copies dans les cellules fantômes par des `sendRecv` MPI.

La figure suivante montre qu'avec un thread par noeud on obtient de très bon speedup sur des damiers de taille 16k ou moins, mais qu'encore une fois on ne passe pas à l'échelle de façon satisfaisante pour un damier 32k. Cette observation met à mal notre hypothèse selon laquelle ce phénomène est du au caractère NUMA des machines puisque dans ce cas les temps d'accès sont uniformes...

Cette expérience apporte toutefois un éclaircissement : le speedup est inférieur pour un damier 32k que pour un damier 16k même lorsque l'on utilise peu de threads. Ce n'est donc pas qu'un problème lié à la synchronisation et à la répartition de la charge sur de nombreux threads.

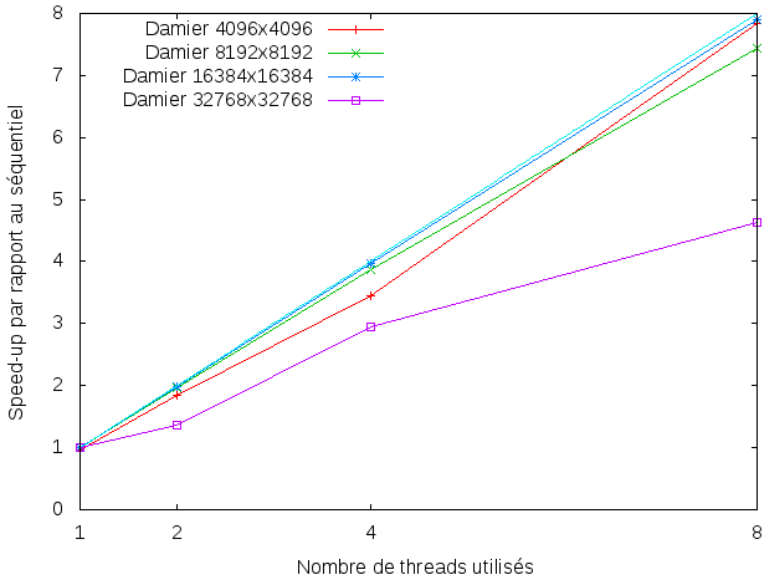

FIGURE 4 – Courbes de speed-up de l'implémentation MPI_SEQ_SYNC (sur 100 itérations, un thread par noeud MPI)

La seconde implémentation utilise des communications persistantes (donc asynchrones). Chaque thread traite ses frontières en priorité, puis envoie le résultat à ses voisins pendant le traitement du reste de son sous domaine. La figure suivante montre que le speedup sous optimal pour un petit nombre de threads était du au surcoût induit par les communications. L'efficacité de cette implémentation en utilisant 8 processus sur un damier de 32k reste cependant bien basse, pour une raison que nous n'avons pas déterminée.

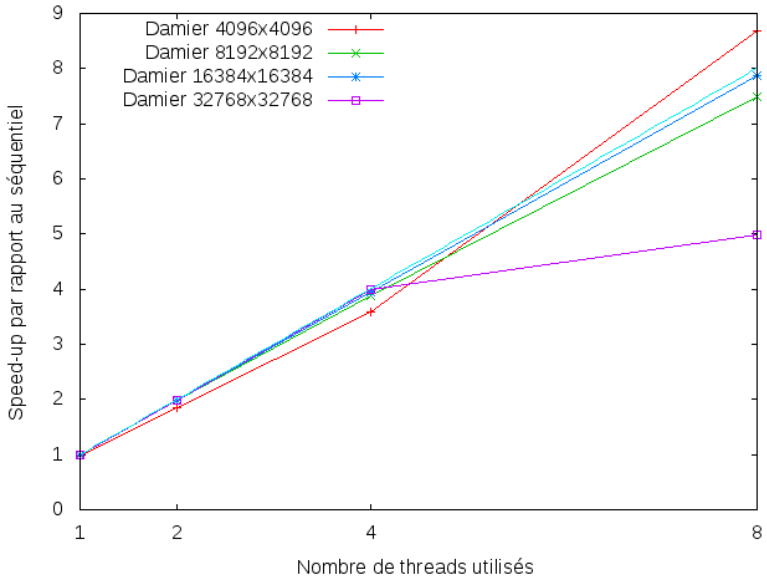

FIGURE 5 – Courbes de speed-up de l'implémentation MPI_SEQ_ASYNC (sur 100 itérations)

Voir `runMPISeq.c|h` pour l'implémentation exacte.

6 MPI_OMP_SYNC : version mémoire distribuée + mémoire partagée

Nous avons fini par combiner les implémentations OMP_SYNC et MPI_SEQ_SYNC pour obtenir une version distribuée et parallélisée en mémoire partagée sur chaque node.

La figure suivante montre les résultats obtenus en utilisant entre 1 et 8 nodes, avec seulement 8 threads par node (puisqu'on a constaté auparavant que cela donne de meilleurs performances qu'avec 16). Nous n'avons pas de commentaires supplémentaires à apporter, on constate toujours des difficultés sur un damier de 16k.

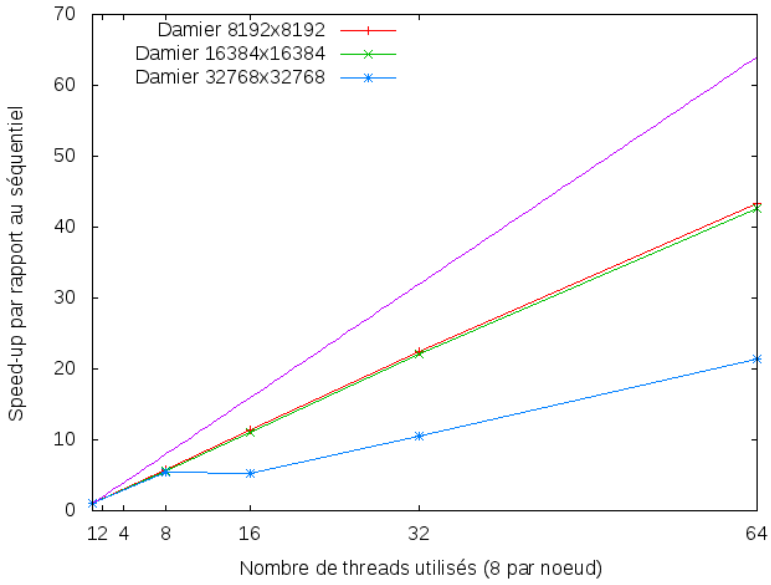

FIGURE 6 – Courbes de speed-up de l'implémentation MPI_OMP_SYNC (sur 100 itérations)

Voir `runMPIOMP.c|h` pour l'implémentation exacte.

7 Comparatif final

Pour finir, nous avons regroupé quelques mesures pour les différentes implémentations :
— le temps d'exécution, pour voir le gain de temps obtenu.
— la performance en Gflops/s des différentes implémentations
— une estimation du débit des accès en mémoire principale des différentes implémentations.

L'estimation du nombre d'opérations flottantes par itération est facile : $9 \times n^2$
— 7 sommes par cellule pour le compte des voisins
— 2 opérations par cellule pour indexer le tableau des états suivants.

L'estimation du nombre d'accès mémoire par itération est plus délicate, car elle dépend de l'optimisme vis à vis des performances du cache. On estime $6 \times n^2$:
— Seulement 3 lecture par cellule pour le comptage des voisins (mise en cache de chacune des 3 colonnes)
— Seulement 1 écriture par cellule pour le comptage des voisins (mise en cache de la colonne du nombre de voisins)
— 1 lecture (de l'ancien état) et 1 écriture (du nouvel état) par cellule pour la mise à jour des états.

Avec ces chiffres, voici un comparatif des performances des différentes versions (pour différents nombre de threads donné entre parenthèses pour certaines) :

	durée (s)	Gflops/s par thread	vitesse d'accès mémoire par node (Go/s)
SEQ	94.75	2.37	6.33
OMP (8 th)	16.61	1.69	36.12
OMP (16 th)	10.30	1.36	58.25
PTHREADS (8 th)	13.26	2.12	45.24
PTHREADS (16 th)	9.27	1.52	64.72
MPI_SEQ_SYNC (4 th)	23.90	2.35	6.28
MPI_SEQ_SYNC (8 th)	12.00	2.34	6.25
MPI_SEQ_ASYNC(4 th)	24.00	2.34	6.25
MPI_SEQ_ASYNC(8 th)	12.04	2.34	6.22
MPI_OMP_SYNC(64 th)	2.22	1.58	33.78

FIGURE 7 – Comparatif des performances pour 100 itérations sur un damier de 16k, soit 600 Go d'accès mémoires, et 225 Gflops

	durée (s)	Gflops/s par thread	vitesse d'accès mémoire par node (Go/s)
SEQ	383.0	2.35	6.27
OMP (8 th)	67.5	1.67	35.56
OMP (16 th)	48.6	1.16	49.38
PTHREADS (8 th)	55.7	2.02	43.09
PTHREADS (16 th)	40.0	1.41	60.00
MPI_SEQ_SYNC (4 th)	130.1	1.73	4.61
MPI_SEQ_SYNC (8 th)	82.7	1.36	3.63
MPI_SEQ_ASYNC(4 th)	96.1	2.34	6.24
MPI_SEQ_ASYNC(8 th)	77.0	1.46	3.90
MPI_OMP_SYNC(64 th)	17.9	0.79	16.76

FIGURE 8 – Comparatif des performances pour 100 itérations sur un damier de 32k, soit 2.4 To d'accès mémoires, et 900 Gflops

On peut retrouver dans ces chiffres une bonne partie des observations des sections précédentes de ce rapport :

- Lorsqu'on utilise 16 threads pour les versions OMP ou PTHREADS, il semble que cela soit la vitesse d'accès mémoire qui limite les performances lorsque l'on passe d'une taille 16k à une taille 32k. En effet, celle-ci ne double pas avec le nombre de threads et les Gflops/s par thread eux, chutent.
- Sur les versions MPI_SEQ, le facteur limitant semble différent. Alors que sur un damier de 16k, la vitesse d'accès mémoire sur chaque node est la même pour 4 ou 8 nodes, sur un damier de 32k la vitesse d'accès mémoire pour 8 nodes est moindre que pour 4 nodes. L'augmentation de taille induit donc un nouveau facteur limitant, et ce n'est visiblement plus le débit mémoire qui est limitant. Ce facteur est probablement lié aux communications.
- Sur la version MPI_OMP, on peut penser que ces 2 problèmes se cumulent.