Master 1 CHPS

Rapport

Calcul de densité d'état de particule pour la modélisation de nanoparticule à transition de spin (SCO) selon un modèle d'Ising.

Moursal MAHAMOUD ALI
Ange Bertrand Dizo
Ibra Ndiaye
Amina Glenn Stevy

Encadrants :
Dr. Thomas Dufaud
Dr. Devan Sohier

Mai 2019

Table des matières

1 Introduction

Le modèle Ising Ref.[1] donne une description microscopique de la ferromagnétisme qui est causé par l'interaction entre les spins. Un spin est considéré comme une grandeur scalaire pouvant atteindre les valeurs +1 et - 1. Le modèle est simple et statistique. Il montre la transition de phase entre la phase de paramagnétisme à haute température et la phase ferromagnétique à basse température à une température donnée. En fait, la symétrie entre haut et bas est spontanément brisée lorsque la température descend au-dessous de la température critique Ref.[2]. Cependant, le modèle unidimensionnel d'Ising, qui a été résolu avec précision, ne montre aucune transition de phase Ref.[2]. Le modèle d'Ising à deux dimensions a été résolue d'une manière déterministe avec la programmation dynamic Ref.[3]. Malgré de nombreuses tentatives pour résoudre le modèle 3D Ising, on peut dire que ce modèle n'a jamais été résolu exactement. Tous les résultats du modèle tridimensionnel d'Ising ont été utilisés comme approches d'approximation et des méthodes de Monte Carlo. Les méthodes de Monte Carlo ou les méthodes de simulation statistique sont largement utilisées dans différents domaines scientifiques tels que la physique, la chimie, la biologie, la finance informatique. La simulation peut être effectuée en échantillonnant à partir de la fonction de densité de probabilité et en générant des nombres aléatoires de manière uniforme. La simulation du modèle d'Ising sur un grand réseaux augmente le coût de la simulation. Une façon de réduire les coûts de simulation consistent à concevoir des algorithmes plus rapides. Les algorithmes de Swendsen-Wang et Wolff Ref.[4][5] et les méthodes de codage à spin multiple Ref.[6] en sont des exemples. Nous proposons une méthode du modèle Ising qui est *Monté Carlo Métropolis* Ref.[7], ensuite nous parallélisons le modèle avec plusieurs processus (ou threads).

Dans ce rapport, nous présentons un algorithme séquentiel dans un premier temps et dans un second temps la partie parallèle pour calculer la densité d'état d'un plan 2D d'Ising à l'aide de la méthode de *Monté Carlo Métropolis*. Ensuite, nous exécutons l'algorithme sur un ordinateur portable classique et ensuite sur le cluster de la MDLS (Maison de la Simulation) en utilisant C comme langage de programmation et OpenMP : est un modèle de programmation utile dans les systèmes HPC dans lequel des Threads (processus léger) partagent les tâches au sein d'un hotspot : la partie du code qui consomme plus de CPU, ce modèle a été conçu pour des architectures à mémoire partagée. Ainsi, une implémentation parallèle avec la bibliothèque *MPI* est mis en place, *MPI* qui est aussi un modèle de programmation mieux qu'OpenMP dans les systèmes HPC et conçu pour des architectures à mémoire distribuée.

2 Présentation du Modèle

Le modèle d'Ising est un modèle fréquemment utilisé pour tester de nouvelles idées et méthodes en physique statistique. Afin de représenter les atomes qui composent un matériau magnétique, on fixe deux hypothèses :

— chaque atome a un moment magnétique appelée spin. Pour simplifier, les spins n'auront que deux orientations possibles : haut (on dira que le spin est +1) ou bas (-1) de chaque molécule de spin-crossover (SCO) ;
— initialement, ils sont orientés aléatoirement.

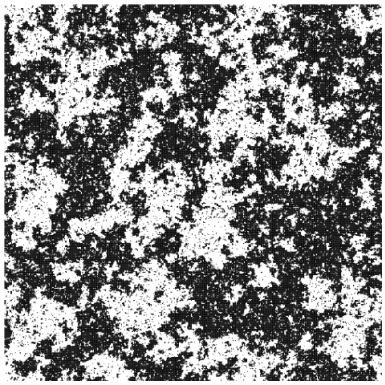


FIGURE 1: Configuration 2D. En noir les spins +1 et en blanc les -1.

Sur la base de chaque état de configuration, le système SCO macroscopique est décrit par les variables suivantes : m, s, et c avec « m » le nombre d'atomes de spin du domaine, « s » le nombre de paires d'atomes adjacents ayant différents spins et « c » le nombre d'atomes se trouvant sur le bord, respectivement pour l'aimantation totale, les corrélations à courte portée et l'aimantation de surface. De ce fait, nous déterminons la densité des macrostats d (m, s, c), en donnant le nombre de configurations microscopiques ayant les mêmes valeurs m, s et c.

L'Hamiltonien associée à ce modèle est indiqué à l'équation (1) :

$$\hat{\mathcal{H}} = \frac{\Delta - K_B T\ lng}{2} \sum_{i=1}^{n} \sigma_i \ - G \sum_{i=1}^{n} \sigma_i < \sigma > \ - J \sum_{<i,j>} \sigma_i \sigma_j \ - L \sum_{k=1}^{M} \quad (1)$$

« Cet hamiltonien permet de simuler le comportement d'un matériau dans un environnement caractérisé par ses interactions non seulement au cœur du matériau, mais également à ses bords. Dans l'expression utilisée pour l'hamiltonien, σ est un opérateur des spins fictifs pouvant prendre les valeurs -1 (état de spin faible) et $+1$ (état de spin élevé), Δ est l'écart entre les états LS et HS , et g = g HS / g LS représente le rapport entre les dégénérescences respectives g HS et g LS des états HS et LS. J et G représentent les paramètres des interactions à court et à long terme, respectivement et L est un paramètre lié à la contribution des interactions entre les molécules situées sur le bord (surface) et l'environnement local. L'ajout de la constante L est nécessaire, car les molécules à la surface ont des propriétés spécifiques. » Selon l'article [3].

Les variables m, s et c sont définies par les équations (2), (3) et (4) :

$$m = \sum_{i=1}^{n} \sigma_i \ (2) \quad s = \sum_{<i,j>} \sigma_i \sigma_j \ (3) \quad c = \sum_{k=1}^{M} \sigma_k^{'} \ (4)$$

l'équation (1) peut s'écrire alors :

$$\hat{\mathcal{H}} = (\frac{\Delta - K_B T \ ln \ g}{2} \ - G < \sigma >) m \ - Js \ - Lc \ (1)$$

Pour chaque m, s et c désigne une configuration donnée de densité d(m,s,c). Ainsi, on procède plusieurs itérations de *monté carlo* pour cumuler plus de configuration et on calcule ainsi la densité des macrostats. Pour commencer on s'est posé des questions, comment accepter ou rejeter une configuration ? Selon quelle condition ? Quelle est la condition d'arrêt de l'algorithme ?

2.1 Méthode de Monte-Carlo

Définition

Les méthodes de Monte-Carlo sont particulièrement utilisées pour calculer des intégrales en dimensions plus grandes que 1 (en particulier, pour calculer des surfaces et des volumes). Elles sont également couramment utilisées en physique des particules où des simulations probabilistes permettent d'estimer la forme d'un signal ou la sensibilité d'un détecteur. La comparaison des données mesurées à ces simulations peuvent permettre de mettre en évidence des caractéristiques inattendues, par exemple de nouvelles particules [8].

2.2 Choix de la méthode

Dans cette large famille de méthode *Monté Carlo*, dans un premier lieux il nous a demandé dans le cadre de ce projet d'explorer la méthode **Entropic sampling** pour approcher à la densité d'état. Certes qu'on a regorgé de documentation à ce sujet, on n'a pas trouvé une condition au-qu'elle on pourrait se baser pour accepter ou rejeter une configuration donnée. De ce fait, on a opter **Monté Carlo Métropolis** pour s'amener à une meilleure approche de la densité d'état, selon la variation de l'énergie, la constante de Boltzmann et la température.

3 Monté Carlo Métropolis

3.1 Principe

Le principe consiste en partant d'un état actuel pseudo-aléatoire et de générer un état d'essai aléatoire qui est « proche » de l'état actuel du système. « Proche » signifie ici que l'état de l'essai doit être presque identique à l'état actuel, à l'exception d'un petit changement aléatoire, effectué généralement sur une seule particule ou spin. Ce qui est notre cas de choisir un ou plusieurs spins (cas parallèle) de façon aléatoire. L'état d'essai d'un système de spin implique généralement un retournement ou une rotation aléatoire d'un spin unique ou plusieurs dans le cas parallèle.

3.2 Démarche Algorithmique

1. Sélectionnez un spin de manière aléatoire et calculez l'énergie d'interaction entre ce spin et ses voisins les plus proches (E).
2. Retournez le spin s (i, j) et calculez à nouveau l'interaction de l'énergie (E ').
3. On fait la différence d'énergie (E') - (E) si cette différence est inférieur ou égale à 0, le spin est accepté, sinon la configuration est accepté selon une probabilité : si un nombre réel aléatoire entre [0,1] est inférieur à $exp\frac{-\Delta E}{K*T}$ où K est la constante de Boltzmann et T une température donnée.
4. Répétez les étapes 1 à 3 jusqu'à ce que nous soyons sûr que chaque rotation a été inversée.
5. Calculez l'énergie totale du réseau pour toutes les itérations (E^i_{total})

   Les étapes ci-dessus forme une itération de Monte Carlo. Nous effectuons assez d'itération (N_{MC} nombre de fois de MC) et finalement en moyenne sur (E^i_{total}) pour obtenir Etotal :

   $E_{total} = \frac{1}{N_{MC}}(\sum E^i_{total})$

   Puisque cette énergie totale E_{total} de toutes les itérations n'influence pas sur la condition de garder une configuration ou pas donc on ne va pas le calculer. Par contre, l'énergie pour chaque itération est le socle de notre algorithme.

3.3 Peudo-Code MC Métropolis

Algorithm 1 MC Métropolis

$Mat[N][N]$; domaine d'étude 2D
Nb_iter_{MC} : Nombre d'itération de MC
for $0 : Nb\_iter_{MC}$ **do**
  $SelectRandoom(Mat, ai, aj)$
  $Energy_{actuel} \leftarrow CalculEnergy(Mat)$
  $Mat[ai][aj] \leftarrow -Mat[ai][aj]$ //retournement de spin
  $Energy_{essai} \leftarrow CalculEnergy(Mat)$
  $\Delta E \leftarrow Energy_{essai} - Energy_{actuel}$
  if $\Delta E \leq 0$ **then**
    $Accept(Mat)$; // on accepte la nouvelle configuration
  else
    $x \leftarrow RANDOOM\_UNIFORM()$; // un nombre réel aléatoire entre [0.0,1.0]
    if $x < e^{\Delta E*\beta}$ **then**
      // avec $\beta = 1/(K_B$*T), K_B constante de Boltzmann et T température
      $Accept(Mat)$;
    else
      $Mat[ai][aj] \leftarrow -Mat[ai][aj]$ // on annule le retournement
    end if
  end if
end for

4 Implémentation séquentiel du modèle

4.1 Principe

En ce qui concerne l'implémentation de ce modèle, dans un premier temps on a implémenté en séquentiel avec le langage de programmation *standard* **C** et par la suite on verra un peu plus loin la parallélisation avec l'API OpenMP.

Pour commencer, on part d'une matrice 2D de taille N*N, chaque case (i,j) de la matrice représente le comportement d'un molécule c'est-à-dire un spin qui a pour valeur ± 1. Et ces valeurs sont générées de façon aléatoire avec la fonction rand() du système, mais pour avoir un bon générateur sa nécessite un peu de technique et nous verrons dans la partie démarche.
Ensuite on tire de façon aléatoire avec une probabilité de $\frac{1}{N}$ de case (i,j) et on calcule son énergie par rapport à ses voisins les plus proches dans la grille avec la fonction ***ith_energy*** qui retourne un type double, ainsi on le multiplie par -1 pour lui faire subir un retournement, après avoir fais cela on recalcule la nouvelle énergie. Alors on effectue la différence d'énergie avant le retournement de la molécule et après, si cette énergie est négative ou nul, on accepte la configuration. Sinon on compare avec un nombre réel x aléatoire uniformément distribuer entre [0.0,1.0], si x $<$ exp(-$\Delta E * \beta$) : avec $\beta = \frac{1}{K*T}$ et T la température est à rentrer en paramètre du programme, alors on accepte la configuration sinon on rejet la configuration et on procède à l'itération suivantes de *MC*. Chaque configuration est composée de 4 variables m,s, c et z ; les trois premiers ont été détaillés précédemment, mais la quatrième z indique la densité associée. Certes, on fait des tirages aléatoire avec des tours de boucle de *MC*, mais une condition d'arrêt est impérative pour optimiser le temps calcule.

4.2 Démarche

1. La fonction **updown** effectue l'initialisation de la matrice 2D avec ± 1 selon une graine aléatoire de $rand()\%2$, si la graine est inférieure à 0.5 elle retourne (+1) sinon (-1).
2. Pour tirer un élément de la matrice aléatoirement, les indices i et j vont varier selon une probabilité de $\frac{1}{N}$ avec N(n1*n2) la taille de la matrice, pour ce faire on utilise la fonction $srand48(5321)$ avec une graine assez grande pour la génération d'une nouvelle séquence de nombres pseudo-aléatoires, qui seront fournis par rand(), ainsi i prend comme valeur $rand()\%N$ et j de même.
3. On calcule l'énergie avec la fonction **ith_energy**, différent cas s'impose selon la position où se trouve l'élément tirer puisque le calcule se fait selon son voisin le plus proches, si l'élément (i,j) est sur le bord il possède soit 2 voisins ou 3 et s'il se trouve sur la surface il en a 4. Soit E_{old} l'énergie calculer avant le retournement et E_{essai} l'énergie après retournement de l'élément.
4. La différence énergétique est $\Delta E = E_{essai} - E_{old}$, si $\Delta E \leq 0$ on stock la configuration : on fait appelle à la fonction **Calcul_conf** permettant de calculer une configuration d(m,s,c) et qui retourne un pointeur contenant cette configuration et on le stock dans un tableau (*). Sinon on prend un nombre pseudo-aléatoire **x** avec la fonction $drand48()$ qui renvoi des valeurs réelles en virgule flottante uniformément distribuées dans l'intervalle [0.0, 1.0], si **x** $<$ exp(-$\Delta E * \beta$) on effectue la même tache (*) sinon on annule le retournement de la case en le ré-multipliant par -1 et on ne stock pas.
   (*) On stock en bloc de 4 à l'aide de la fonction **add_config** toutes ces configurations sont stockées dans un tableau de taille 4 fois le nombre d'itération de *MC*, les variables

m, s et c sont stocker au trois premier bloc et la quatrième bloc indique la densité de cette configuration c'est-à-dire le nombre de fois de tour de MC où on tombe sur la même configuration.
Le stockage d'une configuration s'effectue par comparaison des configurations déjà enregistrer dans le tableau, si elle existe donc on n'injecte pas celle-ci dans le tableau, mais sa valeur de z est mis à jour.
Ici, on prend une plage de température de 290–320 K Ref.[1] pour voir l'impact que ça donne au niveau de la densité.
Pour voir vraiment l'impact de la température il nous a fallut de faire plusieurs testes avec des matrices de petites tailles tout en fixant le nombre d'itération pour chaque matrice et on a constaté que de 275K - 290K la densité devient important : il y a plus de configurations acceptées et de 290K - 320K le système semble être instable : la majorité des configurations sont rejetées. Donc pour le reste de nos calcule on prend la température intermédiaire 290K.

5. On impose une condition permettant d'arrêter l'insertion des nouvelles configurations : le cas où le nombre de répétition de ces configurations sont assez suffisante pour faire une approximation très fine de la densité. Pour ce faire, on se fixe une condition visant à terminer la recherche des nouvelles configurations au cas où la somme de celles-ci est supérieur ou égale 2^N (le nombre maximal de configuration possible), N étant le nombre de *spin* présent dans le réseau.

Algorithm 2 Pseudo – code du critère d'arrêt

Nb_iter : Nombre d'itération de MC
while $\sum zi \leq 2^N$ **do**
  $Monte\_carlo(Nb_iter, Z)$;
end while

4.3 Objectif

- Bien choisir une graine aléatoire pour avoir des résultat significatifs
- Trouver une condition d'acceptation ou de rejet d'une configuration.
- Gérer le stockage des configurations (injection de quatre cases ou incrément d'une case).
- Etudier l'impacte de la température sur la densité.
- Avoir un critère d'arrêt qui nous donne une approximation très fine de la densité.

4.4 Résultats du code séquentiel

Nous allons faire une première comparaison de notre implémentation séquentielle (avec la condition d'arrêt) avec les résultats du calcule déterministe avec la programmation dynamic.
On prend l'exemple d'une grille de $5 * 5$, on visualise les résultats pour le cas déterministe et les miens avec du calcule probabiliste avec 40 000 000 d'itérations de MC.
La durée d'exécution de ce dernier est 9.5196 minutes.
ces résultats révèlent une approximation assez fine de la densité avec une certaine marge d'erreur.

Les figures ci-dessous illustrent la comparaison :

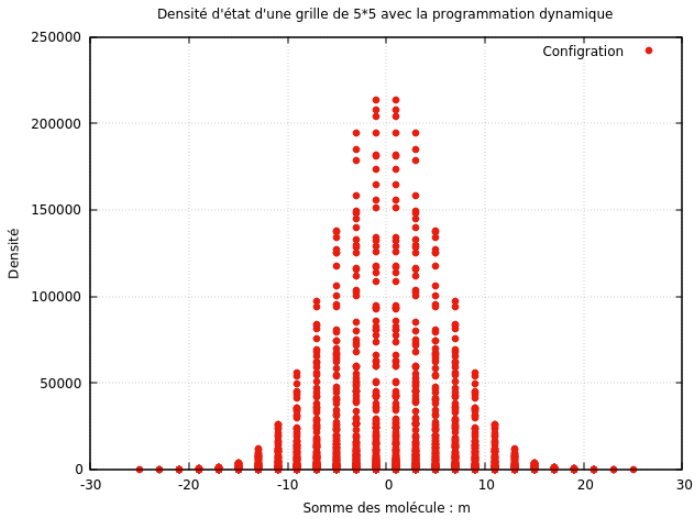


FIGURE 2: Densité Exacte

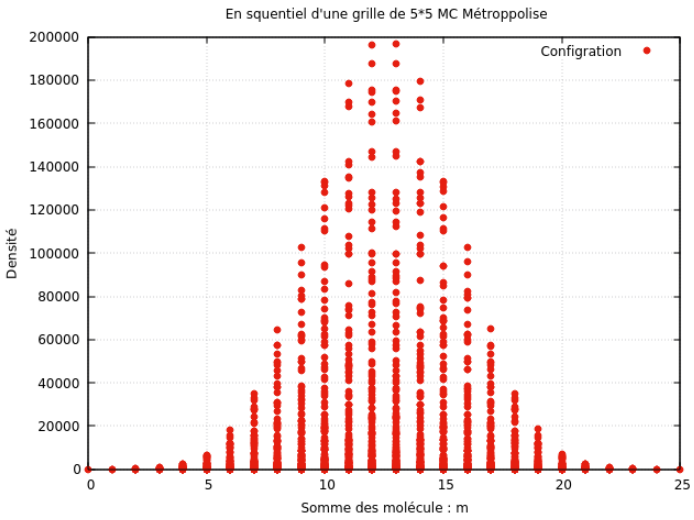


FIGURE 3: Densité Approché en séquentiel

5 Parralélisation du modèle

Avant d'entamer la parallélisation de ce modèle deux critères majore s'impose : repérer la partie du code qui prend plus de calcule et par la suite paralléliser, selon le paradigme du système distribué avec *MPI* (Message Passing Interface) ou partagé avec OpenMP sachant que les deux implémentations sont possibles.
On a cherché le point chaud de notre algorithme c'est-à-dire la partie qui prend plus de temps en exécution dans notre implémentation et on a constaté que la boucle d'itération de *MC* consome beaucoup de *CPU*.

Dans un premier temps nous détaillons l'implémentation avec l'*API OpenMP* qui est assez similaire au code séquentiel, mais en ajoutant des directives *OpenMP* permettant de paralléliser la boucle de Monté Carlo avec un nombre de threads donnée.
Et dans une seconde partie nous allons paralléliser avec la bibliothèque *MPI* en utilisant plusieurs processus dans le but de gagner en performance.

5.1 Open MP

Définition

Open Multi-Processing ou OpenMP est une interface de programmation pour le calcul parallèle sur une architecture à mémoire partagée. Il se présente sous la forme d'un ensemble de directives, d'une bibliothèque logicielle et de variables d'environnement. OpenMP est portable et « dimensionnable ». Il permet de développer rapidement des applications parallèles à petite granularité en restant proche du code séquentiel. La programmation parallèle hybride peut être réalisée par exemple en utilisant à la fois OpenMP et MPI Ref.[8].

5.1.1 Principe

Le principe de OpenMP s'appuie sur la création de processus légers, appelés threads, qui ont accès à une mémoire commune dans laquelle sont stockées des variables communes. Chaque thread dispose également d'une petite mémoire qui contient des variables dites privées et on peut

repartir les tâches au sein d'une boucle selon le nombre de thread disponible. Cette répartition de tâche peut se faire avec différente clause «schedule» : *dynamic*, *static*, *guided* etc.
En se basant sur des chapitres acquis au cours de ce semestre, le choix de ces directives nous paraissait assez intuitif, sachant que notre but est de trouver une bonne manière de repartir les itérations au sein de la boucle de *MC* afin de tirer parti une meilleur scalabilité.

Voici un exemple de directives :

```
#pragma omp for schedule(static,100)
```

Ici, la clause *static* permet de dispatcher une boucle de 100 itérations (*par exemple*) en bloc de 100 par thread, la valeur 100 indique le « *chunks* » : la quantité des tâches dispatcher par thread. Cependant la clause « *dynamic* », les itérations de boucle sont divisées en morceaux de taille et planifiées dynamiquement entre les threads ; quand un thread termine un morceau, il en affecte un autre dynamiquement. Puisque nous manipulons des très grande itération de *MC* on a choisit donc la clause « *dynamic* ».

La figure suivante illustre mieux ce principe de fonctionnement du *OpenMP* :

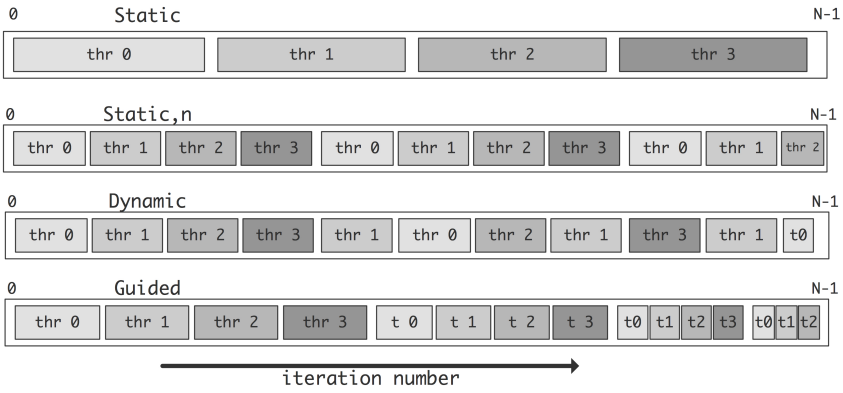


FIGURE 4: Exemple de différents clause omp parallel for (source)

Maintenant que plusieurs threads vont tirer aléatoirement en même temps sur la même grille, alors la graine aléatoire avec la fonction ***srand()*** va dépendre du numéro unique de chaque threads.

5.1.2 Démarche

1. Nous avons ajouté la directive OpenMP ***#pragma omp for schedule(dynamic,1000)*** avant la boucle d'itération de MC, chaque thread traitent au début 1000 itérations et lorsqu'un *thread* termine sa tâche, un autre morceaux lui est infecté dynamiquement.
2. Dans chaque itération de *MC* les mêmes fonctions du code séquentiel sont traités et parmi eux il y a la fonction ***« add_conf »*** et on sait que cette fonction permet d'écrire les configurations dans un tableau, alors puisque les threads traitent celle-ci en parallèle aura t- il la possibilité que plusieurs threads modifient en même temps une case de ce tableau ? La réponse est non, parce que ce cas de figure est quasi impossible en raison de notre graine aléatoire qui dépend du numéro de chaque thread : ***srand(omp_get_thread_num())***.

5.1.3 Objectif

- Dispatcher équitablement et dynamiquement la boucle d'itération de MC en fonction de nombre de threads dans le but d'avoir une scalabilité.
- Choisir techniquement une graine différente pour chaque thread
- Avoir plusieurs threads qui tirent en parallèle sur la même grille

5.1.4 Résultats et Scalabilité

Résultats

En essayant de s'approcher à la densité d'état du calcule exacte, on prend l'exemple d'une grille de $5 * 5$ et on a pu obtenir un nombre d'itération optimum pour une densité porche de celle-ci. Dans cette partie, le critère d'arrêt ne fonctionne pas avec le *multi-thread*, parce que tout simplement la boucle d'itération est divisée en nombre de thread et pour sortir de la boucle il faut arrêter toutes les threads qui exécutent cette boucle, avec plusieurs tentations et de testes on a pu trouver un nombre d'itération optimum qu'on a trouvé à la main, optimum dans le sens où on a une densité assez proche (dans le cas d'une grille de $5 * 5$) à celle du calcule exacte.

Voici deux figures qui illustrent à comparer le calcule exacte et nos calcule.
La figure à droite sont nos calcule avec un nombre d'itération optimum qui est $33 * 10e6$ et une température de 290.

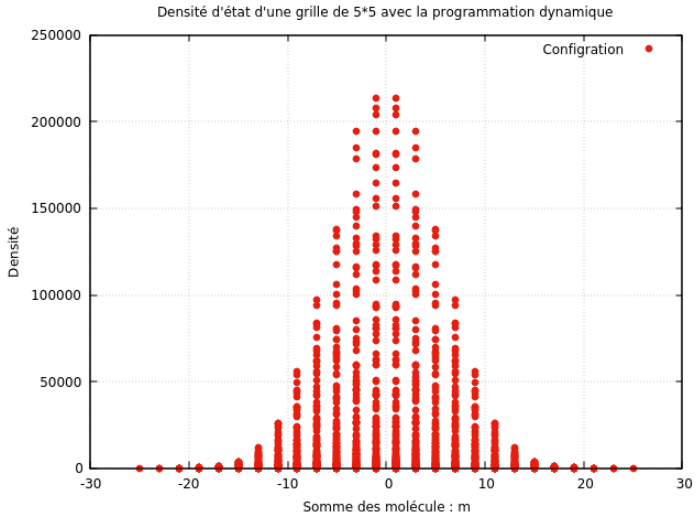


FIGURE 5: Densité Exacte

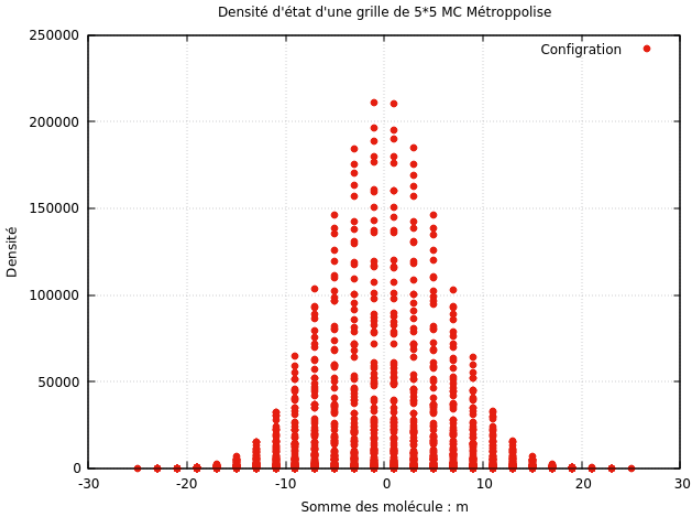


FIGURE 6: Densité Approché avec OpenMP

Scalabilité

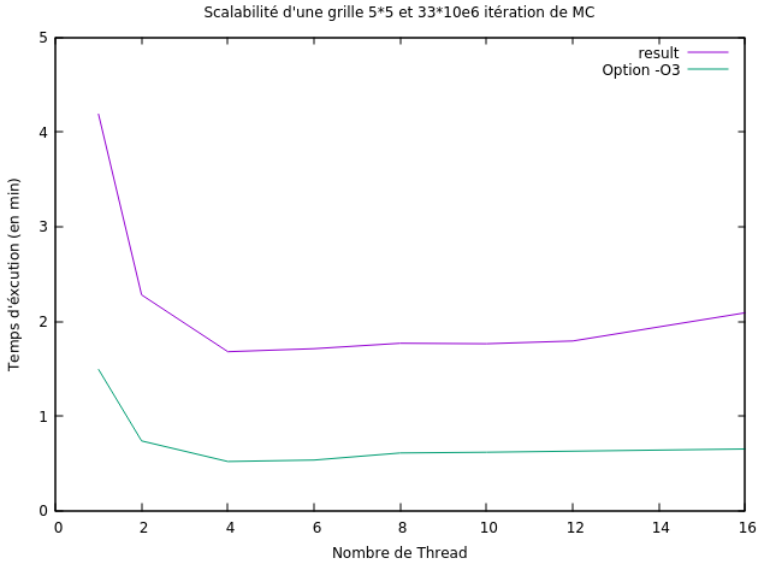
La durée d'exécution d'un seul *threads* fais la moitié du temps d'exécution ceci est dû au fait qu'il n'y a pas une condition d'arrêt, car elle a un impact sur la performance.

Ce tableau représente les durées d'exécution de la figure[4].

Nombre de Threads	**durée d'exécution (en minutes)**
1	4.191836
2	2.279182
4	1.679524
6	1.710597
8	1.767566
10	1.763051
12	1.791693
16	2.089230

D'après nos résultats avec OpenMP on a constaté de *1* ***thread*** à *2* ***thread*** on a un gain de performance de environ 45%, de *2* ***thread*** à *4* ***thread*** environ 10% et au-déla de *4* ***thread*** on a une scalabilité faible.

Voici un graphe qui illustre les résultats :



5.2 MPI

Définition

Message Passing Interface (MPI), est une norme conçue en 1993-94 pour le passage de messages entre ordinateurs distants ou dans un ordinateur multiprocesseur. Elle est devenue de facto un standard de communication pour des nœuds exécutant des programmes parallèles sur des systèmes à mémoire distribuée. Elle définit une bibliothèque de fonctions, utilisable avec les langages C, C++ et Fortran [9].

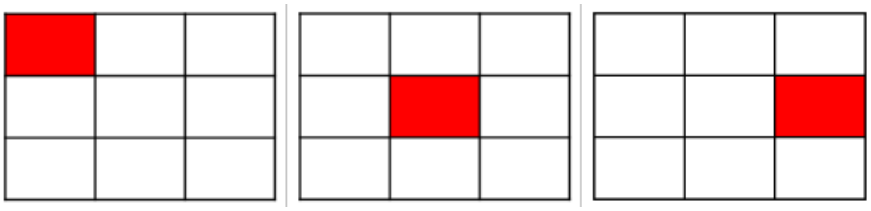
5.2.1 Principe

Dans cette partie l'objectif étant de parcourir toute la maille pour sortir le maximum de configurations avec un nombre de processus donnée, car pour un seul processus cela met plus de temps de parcours une maille surtout s'il s'agit à des matrices de grandes tailles.
On décide alors d'augmenter le nombre de processus de telle sorte que chacun commence le parcours de la maille à des endroits différents.

5.2.2 Démarche

1. Chaque processus répète le comportement de la partie séquentielle : tirage d'une case aléatoire, calcule de la configuration, stockage dans un tableau de configurations, etc).
2. Chaque processus possède son propre tableau de configuration qu'ils envoient au processus zéro qui lui se chargera de les stocker dans son tableau et le stockage se fait avec la même procédure vue précédemment.
3. Chaque tirage est initialisé en fonction du rang (srand(rang)) pour être sûr que les processus commenceront le parcours à des endroits différents.
4. On choisit volontairement que chaque processus initialise aussi une version de la matrice avec une graine différente, afin d'éviter des comportements semblables et ainsi d'avoir plus de configurations.
5. La condition d'arrêt est prise en compte sur cette partie donc on pourra faire une réel comparaison de la scalabilité avec la partie séquentielle.

   voici une illustration de trois processus qui tire sur des zones différentes :



   Les configurations sont mises dans le tableau tab du processus 0 et tab[0] de chaque processus contient la taille du tableau. Ensuite, tous les processus appellent send() ;

```
if(rank != 0){

    MPI_Send(tab, cnt, MPI_INT, 0, tag1, MPI_COMM_WORLD);

            }
```

   Le processus 0 reçoit les tableaux et les stocke dans son tableau tab grâce à la fonction ***fusion(array_0, array_recv)*** ;

   La fonction ***fusion()*** prend en argument deux tableaux et leurs tailles et stocke le tableau reçu dans le tableau du processus 0. Pour cela, pour chaque *case (m,c,s)* du tableau reçu, elle vérifie si le tableau de 0 contient une case identique si oui leurs z sont sommés, sinon cette nouvelle configuration est stockée dans un tableau intermédiaire. Quand toute cette étape étant terminée, le tableau intermédiaire est collé à la suite

```
if(rank == 0 && size>1){

    for(int i=1;i<size;i++){

     MPI_Recv(buf, runs*4, MPI_INT,i, tag1, MPI_COMM_WORLD, &sta);
     ta = buf;
     fusion(tab, ta,&cnt, &ta[0]);


                  }
```

du tableau du processus 0. Cette opération est répétée par le processus 0 pour chaque *recv( )*.

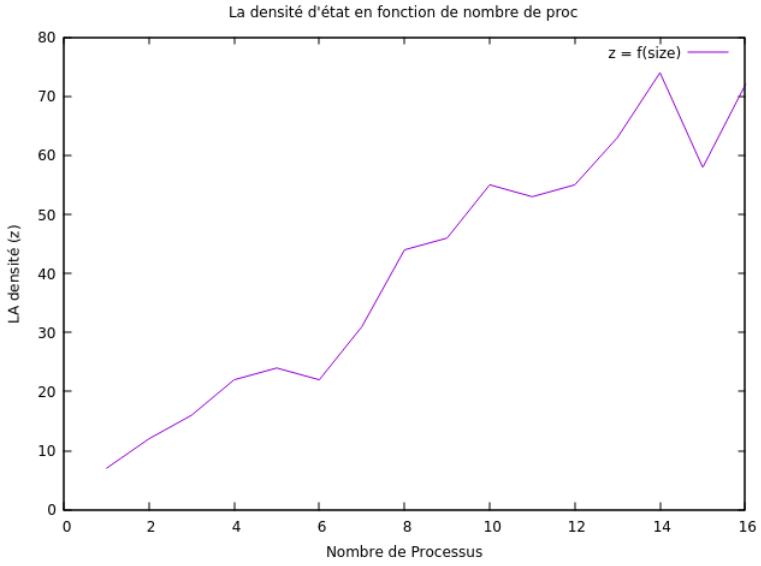
5.2.3 Objectif

- Plusieurs processus tire aléatoirement avec une graine unique par processus et chacun possède son tableau de configuration.
- Chaque processus initialise la matrice avec une graine différente.
- Un seul processus se charge de stocker tout les configurations des autres processus

5.2.4 Résultats et Scalabilité

Résultats

Pour commencer, nous allons voir l'influence du nombre de processus sur la densité. Ainsi on prend une maille de $4 * 4$, avec huit itérations (on prend un petit nombre d'itérations pour voir plus facilement les nouvelles configurations), les exécutions sont aléatoires : deux exécutions consécutives ne donnent pas forcement le même résultat, on fait varier le nombre processus et on compte le nombre de configuration pour voir l'impact sur la densité. On obtient le graphe suivant :



On remarque que l'augmentation du nombre de processus augmente le nombre de configurations. Les décroissances peuvent être dues au tirage aléatoire successif, car deux tirages successifs avec le même nombre de processus ne donne pas forcement le même « z ». Dans cette étude on peut dire que la scalabilité semble forte, car l'augmentation du nombre de processus réduit le

parcours de la maille (nombre d'itérations) par chaque processus, comme le travail est divisé par le nombre de processus.

Scalabilité

La deuxième étude consiste à fixer le nombre de configurations, faire varier le nombre de processus et mesurer le temps mis.
On prend un nombre de configuration égale à 33 554 432, c'est la tâche qu'on a effectuée en séquentiel. On divise cette tâche par le nombre de processus et on analyse comment varie le temps.

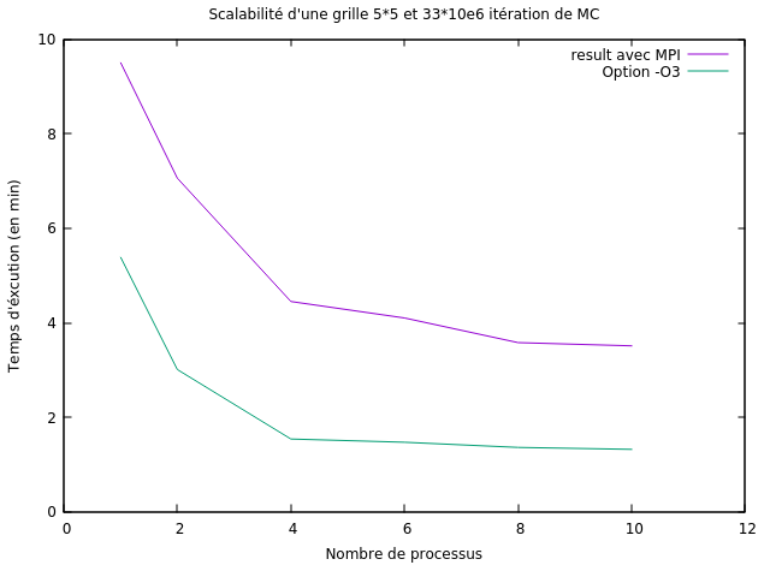
Voici un tableau récapitulatif de la scalabilité par niveau de nombre de processus :

Nombre de Processus	1	2	4	6	8	10
Durée global (en min)	12m51	7m06	4m45	4m10	3m58	3m51
Speed UP (option -O3)	5m39	3m01	1m54	1m47	1m36	1m32
Temps comm (Send - Recv)	-	0.000185	2m30	0.28s	1m48	1m97
(Zmax/Nombre de processus)	33554432	16777216	8388608	5592405	4194304	3355443

Zmax : est le nombre de configuration maximum.

La première case est la case « témoin », un processus sans communication. On remarque que le temps d'exécution diminue quand on augmente le nombre de processus.

Voici la version Graphique :



Sur cette version parallèle (distribuée), on utilise un stockage de mémoire coûteux, chaque processus à ses versions des données et donc une allocation d'un tableau de taille quatre fois le nombre d'itération par processus, ici la complexité en mémoire est assez grande par rapport au code séquentiel ou la partie parallèle avec OpenMP.

La complexité mémoire représente le coût en mémoire de notre algorithme. Sur cette version parallèle (distribuée), chaque processus a la même version des données ce qui donne un coût égal aux nombre de processus fois la Complexité mémoire de la partie séquentielle.

6 Complexité de l'algorithme

Complexité en temps

Notre implémentation consiste à parcourir un tableau de configuration dont la taille dépend du nombre d'itération, cette implémentation à la même ordre de complexité temporelle que l'algorithme de tri rapide avec un comportement en n^2 dans le pire cas, un comportement en $n * log_2(n)$ dans le meilleur cas et également en moyenne (avec n la taile du tableau de configuration). Mais nous avons mesurer la durrée d'exécution d'une densité proche à celle du calcul exacte avec les matrices de taille $5*5$ et $6*6$ on a remarqué que la complexité temporelle est d'odre linéaire pour N < 7 (N taille de la matrice) mais à partir de ceci plus la dimension de la matrice augmente (ex : $10 * 10$) plus on fait plus d'itération pour obtenir une meilleure densité plus la durée d'exécution croit.
De petit taille (de $2 * 2$ à $5 * 5$) la complexité temporelle est d'odre linéaire, de $6 * 6$ à $10 * 10$ elle est d'odre $n * log_2(n)$ et à partir de $10 * 10$ la complexité temporelle est quadratique (n^2) voir exponentiel.

Complexité en espace

l'étude de la complexité en mémoire nous permet de trouver l'ordre de grandeur de l'espace mémoire simultanément utilisé.
Le type de la matrice dans notre programme est un *entier* (***int***) et on sait qu'un entier est codé en *4 octets* donc le stockage de la matrice vaut N fois *4 octets* avec N = n_l * n_c. On a aussi en parabole le stockage du tableau de configuration qui « gonfle » en fonction de nombre de nouvelles configurations, ce tableau est un entier avec une taille 4 fois le nombre d'itération, pourquoi ainsi ? Parce qu'on part du principe qu'il y ait au moins une nouvelle configuration par itération. Ici, le pire de cas est, si on atteint la densité exacte, alors toutes les configurations sont atteintes et il faudra stocker $\frac{2^N}{N}$.
Pour une matrice de 5 par 5, $\frac{2^{25}}{25} * 4 = 5368709.12$ **octets** est nécessaire pour stocker et pour une matrice de 6 par 6 on a besoin d'un stockage de $\frac{2^{36}}{36} * 4 = 7635497415.11$ **octets** voir en **gigaoctets** 7.635497415 **Go**. D'où parmi les raisons pour laquelle on se limite à une matrice de $5 * 5$.

7 Résultat et Comparaison

Exécutant les différents programmes qu'on dispose avec un nombre d'itération fixe, on obtient les résultats optimaux suivant :

Version	Temps obtenu (en minute)
Code d'origine	9.51 (1 processus Seq)
OpenMP	1.67 (4 thread)
MPI	3.51 (10 coeur)

On peut en conclure que l'implémentation *MPI* avec la condition d'arrêt nous a fait gagner un facteur de 3 en performance malgré le gain avec l'implémentation de *OpenMP* qui n'y a pas une condition d'arrêt donc moins robuste.

Lors de la deuxième séance à la maison de la simulation on a eu l'occation de faire tourner nos études parallèles sur le supercalculateur et on a pu obtenir des résultats de performance significative jusqu'à 32 coeur avec une matrice de $10 * 10$ jusqu'à $1 * 10^{11}$ itérations mais les résultats de ce dernier étaient erronés à cause de notre condition d'arrêt qui n'était pas valide ce jour-là.

8 Conclusion

Pour clôturer le projet, on peut dire que l'objectif fixé au début qui était d'implémenter un code séquentiel avec la méthode de *Monté Carlo Metropolis* et par la suite de paralléliser avec deux modèles parallèles différent a été atteint.
Cependant, pour avoir une amélioration du temps d'exécution on approfondit une étude en parallèle, une version MPI et une OpenMP. La version OpenMP étant plus rapide sur les premiers cores, en augmentant le nombre cores la version MPI donne des bons résultats bien que gourmande en mémoire.
Une autre implémentaion *MPI* était envisageable avec ***"MPI Cart"*** et on compte poursuivre ce sujet pour voir le gain derière et d'avoir une introduction globale sur ce sujet.

Références

[1] https ://fr.wikipedia.org/wiki/Modèle_d'Ising.

[2] F. Varret I. Sheto, J. Linares. Monte carlo entropic sampling for the study of metastable states and relaxation paths. *Physical review E*, page 56, 1997.

[3] D. Sohier T. Dufaud P-R. Dahoo S-E. Allal, C. Harlé and J.Linares. Three stable states simulated for 1d spin-crossover nanoparticles using the ising-like model. *European Journal of Inorganic Chemistry*, 2017. http ://dx.doi.org/10.1002/ejic.201700598.

[4] Brown G. Thompson S.H. Rikvold P.A. Deskins, W.R. Kinetic monte carlo simulations of a model for heat-assisted magnetization reversal in ultrathin films. *Phys. Rev. B 84*, 2011.

[5] Kozlowski M. Wrobel J. Wejrzanowski T. Kurzydlowski K.J.-Goyhenex C. Pierron-Bohnes V. Rennhofer M. Malinov S. Kozubski, R. Atomic ordering in nano-layered fept : multiscale monte carlo simulation. *Comput. Mater. Sci. 49(1)*, pages 80–84, 2010.

[6] Parker G.J. Lyberatos, A. Cluster monte carlo methods for the fept hamiltonian. *J. Magn. Magn. Mater. 400*, page 266–270, 2016.

[7] méthode de monte-carlo. https ://fr.wikipedia.org/wiki/.

[8] Openmp. https ://fr.wikipedia.org/wiki/OpenMP.

[9] Message passing interface (mpi). https ://fr.wikipedia.org/wiki/Message_Passing_Interface.