Résolution des systèmes linaires dans un contexte distribué en se basant sur les algorithmes itératifs asynchrones

Fayssal Bendaoud, Zakaria Yahouni, Badr Benmammar, Fayçal Belkaid

Résolution des systèmes linaires dans un contexte distribué en se basant sur les algorithmes itératifs asynchrones

BENDAOUD Fayssal
Laboratoire de Télécommunication Tlemcen (LTT)
Université Abou Bekr Belkaid
Tlemcen, Algérie

YAHOUNI Zakaria
Manufacturing Engineering Laboratory of Tlemcen (MELT)
Université Abou Bekr Belkaid
Tlemcen, Algérie

BENMAMMAR Badr
Laboratoire de Télécommunication Tlemcen (LTT)
Université Abou Bekr Belkaid
Tlemcen, Algérie

BELKAID Fayçal
Manufacturing Engineering Laboratory of Tlemcen (MELT)
Université Abou Bekr Belkaid
Tlemcen, Algérie

Abstract—**Nous nous sommes intéressés dans le cadre de ce papier à étudier quelques méthodes de résolution des systèmes linéaires à grande échelle en utilisant les algorithmes itératifs asynchrones (IACA) pour supporter le parallélisme dans un contexte distribué. L'outil JACE nous a permis d'atteindre nos objectifs, à savoir la mise en œuvre et la comparaison du temps d'exécution entre les trois méthodes analysées : Jacobi, Gradient optimal et Gradient conjugué. Egalement, nous avons amélioré le résultat de la méthode Jacobienne en utilisant une compression hybridée entre CSR et Ellpack-Itpack qui a montrée son intérêt dans le cas des matrices creuses.**

Mots- Clés— **Systèmes linéaires; algorithmes IACA; parallélisme; JACE ; CSR; Ellpack-Itpack.**

I. INTRODUCTION

De nombreuses applications en calcul scientifique, telles que la simulation en mécanique des structures dynamique des fluides, se ramènent à la résolution de problèmes matriciels linéaires de grande taille (généralement dits creux), un tel massif système nécessite un temps de calcul laborieux. La résolution de ces systèmes a toujours joué un grand rôle dans la simulation de nombreux problèmes scientifique et industrielle, un tel calcul présente l'inconvénient de mener à des coûts prohibitifs en temps CPU et en espace méloire, donc l'optimisation de ces systèmes est capitale.

Pour résoudre un système linéaire, nous distinguons les méthodes directes et les méthodes itératives. Les méthodes directes donnent un résultat exact du problème modulo les erreurs d'arrondi. Les secondes convergent vers la solution par approximations successives de la solution. Les méthodes itératives sont généralement facilement programmables et parallélisable. En effet, il faut distribuer le travail effectué dans une itération aux processeurs, et à la fin de chaque itération les processeurs s'échangent les données dont ils ont besoin et déterminent si le critère de convergence est atteint. De ce fait, nous distinguons plusieurs types d'algorithmes itératifs dans le contexte parallèle : ISCS, ISCA et IACA.

Puisque les méthodes sélectionnées influent sur le comportement du système, la stratégie optimale utilisée pour résoudre ce type de problème devient un sujet important. Le but de cet article qui étend la recherche menée dans [1] et [7] est de comparer entre trois méthodes de résolution (Jacobi, Gradient optimal et Gradient conjugué) sur l'environnement JACE proposé dans [1] afin d'optimiser le temps d'exécution au maximum.

Dans ce papier, nous commençons par donner un aperçu sur les algorithmes itératifs ainsi que sur les systèmes linéaires. En se basant sur les travaux [1, 4, 7, 8] qui ont montrés l'intérêt d'utiliser de l'asynchronisme dans le cadre d'un calcul parallèle de très grand taille, nous détaillons, par la suite notre contribution qui consiste à utiliser Jace pour comparer le temps d'exécution de trois algorithmes différents de résolution de systèmes linéaires (Jacobi, Gradient optimal et Gradient conjugué). Ainsi nous avons montré l'intérêt de compression des systèmes linéaires représentés par des matrices creuses en utilisant les deux méthodes de compression CSR et Ellpack-Itpack [9] afin d'optimiser le temps d'exécution au maximum.

Le reste du papier est organisé comme suit. Dans la section 2, on commence par une revue de la littérature sur les problèmes des algorithmes itératifs asynchrones. La section 3 et 4 décrit les algorithmes itératifs et les types existant pour ces algorithmes ainsi les trois méthodes de résolution des systèmes linéaires en utilisant ces algorithmes itératifs . La section 5 décrit l'environnement que nous allons utilisés pour faire nos expérience, pour arriver en fin à présenter nos résultats et les améliorer en utilisant des méthodes de compression afin d'améliorer le temps de calcul. Enfin, la dernière section conclut le travail.

II. REVUE DE LA LITTÉRATURE

Avec l'émergence des services et applications scientifiques gourmandes en ressources de calcul, il est devenu indispensable de faire des recherches visant à évoluer les performances des algorithmes itératifs ainsi que sur les systèmes linéaires en se basant sur les travaux [1, 4, 7, 8] qui ont montrés l'intérêt d'utiliser de l'asynchronisme dans le cadre d'un calcul parallèle de très grand taille,
Le but de ce travail est d'étendre les travaux proposé dans [1] et [7] et consiste à faire une comparaison entre plusieurs méthodes pour résoudre un système linéaire dans le cadre IACA et en utilisant JACE, puis nous avons visés un cas plus réel où la matrice est creuse, dans ce cadre nous avons montré l'intérêt de la compression de la matrice.

III. ALGORITHMES ITÉRATIFS

Avant d'entamer le principe des algorithmes itératifs parallèles, il est indispensable d'aborder le principe des algorithmes itératifs séquentiels.

A. *Algorithmes itératifs séquentiels*

Nous nous intéressons à un problème de type :
$x^{k+1} = f(x^k)$ avec x^0 donné et $k = 0,1$ ....

L'algorithme itératif séquentiel s'écrit comme suit :

For k = 1 **to** … **do**
$x^{k+1} = f(x^k)$ $x^{k+1} = f(x^k)$
End for

Les x^k sont des vecteurs de dimension n et f est une fonction de R^n dans R^n.

Pour mettre en œuvre un algorithme itératif séquentiel, il faut définir un seuil d'arrêt afin de terminer l'algorithme dés lors que celui-ci passe sous le seuil. Le test d'arrêt le plus utilisé est :

$$\max_{i \in 1..n} |x^{k+1} - x^k| < \varepsilon$$

Dans les algorithmes itératifs parallèles, les données et les routines de calcul sont réparties sur plusieurs processeurs. C'est pour cette raison que ces algorithmes sont divisés en deux grandes catégories : algorithmes synchrones et asynchrones.

B. *Algorithmes itératifs parallèles synchrones*

Le model des algorithmes itératifs synchrones est comme suit :
$(x_1^0, x_2^0, \dots, x_m^0)$ est fixé
For k=0 to … **do**
For m=0 to … **do**
$x_i^{k+1} = F_i(x_1^k, x_2^k, \dots, x_m^k)$
End for
End for

A chaque itération, le processeur chargé de calculer le bloc de composantes i connaît les valeurs de toutes les composantes impliquées dans le calcul de F_i. Il calcule les nouvelles composantes x^{k+1} et les envoie à tous les processeurs qui l'ont besoin pour calculer l'itération suivante.

Selon le mode de communication employé, on peut distinguer deux types d'algorithmes : ISCS pour Itérations Synchrones et Communications Synchrones, ISCA pour Itérations Synchrones et Communications Asynchrones.

La forme la plus simple consiste à utiliser des communications synchrones. Dans cette variante les communications s'effectuent après la fin de l'étape de calcul. La figure 1 schématise le principe des ISCSs.

Le déroulement d'un algorithme ISCS, représenté par la figure 1, montre qu'il y a des périodes d'inactivité dues aux échanges de données et aux calculs liés à la détection de convergence.

La seconde variante consiste à utiliser des communications asynchrones, ils sont un peu plus rapides car les communications tendent à êtres masquées par les calculs. Mais les périodes d'inactivité ne sont pas totalement supprimées pour autant (figure1).

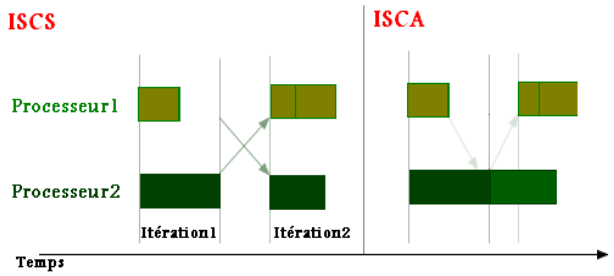

Figure 1. Algorithme ISCS VS ISCA

C. *Algorithmes itératifs parallèles asynchrones*

Un algorithme synchrone est un cas particulier d'un algorithme asynchrone (dont les itérations sont synchrones). En effet, en désynchronisant les itérations, les communications sont forcement asynchrones. Afin de rester cohérent avec la dénomination précédente (asynchrone), ces algorithmes sont nommés IACA pour Itérations Asynchrones et Communications Asynchrones [2].

L'algorithme général des IACAs est comme suit :

For k = 0 to ….**do**
For m = 0 to … **do**
If (i (k)) **then**
$x_i^{k+1} = F_i(x_1^k, x_2^k, \dots, x_m^k)$
Else
$x_i^{k+1} = x_i^k$
End for
End for

A chaque itération, certains processeurs (ceux qui appartient à (K)) calculent une nouvelle itération, les autres nœuds reprennent le résultat de l'itération précédente. Lorsqu'un processeur calcule une nouvelle itération, il prend les dernières données dont il dispose.

Ces algorithmes sont extrêmement performants dans le cas où les données à traiter sont de taille assez conséquente et que les machines sont très hétérogènes.

Identify applicable sponsor/s here. If no sponsors, delete this text box. (sponsors)

La figure 2 permet de mieux expliquer le principe des IACAs.

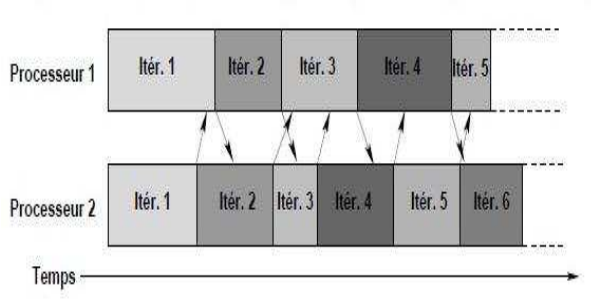

Figure 2. Algorithme IACA

IV. LES METHODES DE RESOLUTION DES SYSTEMES LINEAIRES

A. *Méthodes de relaxation*

Nous nous intéressons à résoudre le système A*x = b avec A matrice définie positive symétrique, x et b deux vecteurs$(A\epsilon R^{n,n}\ x, b\ \epsilon R^{n})$.

Les méthodes de relaxation sont fondées sur le principe d'écrire la matrice A sous la forme A=M-N. Avec M, N deux matrices, donc nous passons de la résolution de A*x = b à la résolution de M*x = N*x + b.

Pour mettre en œuvre la méthode de relaxation de manière efficace, il faut choisir M de tel sorte quel soit facilement inversible.

1) *Méthode de Jacobi*

Elle consiste à prendre M=D et N=-(U+L) avec D une matrice diagonale stricte à partir de la matrice A, U est une matrice triangulaire supérieure stricte, L est une matrice triangulaire inférieure stricte.

Donc la méthode de Jacobi consiste à résoudre la formule récurrente suivante :

$$x_i^{k+1} = \frac{1}{A_{ii}}\left(b_i - \sum_{j\neq i} a_{ij}x_j^k\right)$$

Avec x^0 donné.

Pour la convergence nous utiliserons le théorème suivant :

Théorème : si la matrice A est diagonale strictement dominante i.e. :

$$\forall\, i\ |A_{ii}| > \sum_{i\neq j} |A_{ij}|$$

Alors la méthode de Jacobi converge.

Donc dans le choix de la matrice A, il faut bien vérifier la propriété précédente.

2) *Méthode de Gradient*

Soit la fonction $J(x) = \frac{1}{2}\, x^T Ax - b^T x$ avec A une matrice définie positive symétrique, b et x deux vecteurs.

L'idée des méthodes de gradient est de passer d'un problème d'optimisation de la fonction $J(x)$ à la résolution d'un système linéaire $A * x = b$.

En effet pour minimiser $A * x = b$ il faut que le gradient $\nabla J(x) = 0$, telque le gradient $\nabla J(x) = Ax - b$.

a) Méthode de Gradient avec pas optimal

Elle consiste à itérer les étapes suivante jusqu'à ce que la convergence soit atteinte :

- Choisir un vecteur initial x^0 et pour k>0.
- Calculer le gradient de J : $\nabla J(x^k) = Ax^k - b = -R$.
- Calculer $x^{k+1} = x^k + \alpha R$ avec $\alpha^k = \frac{(R^k, R^k)}{(AR^k, R^k)}$.

b) Méthode de Gradient conjuqué

Voici l'algorithme de Gradient conjugué :

- Choisir $x^0 \epsilon R^N$, poser $R^0 = b - Ax^0$ et $P^0 = R^0$
- Choisir une tolérance T.
- Poser k = 0.

While ($\|R^k\|$>T) **do**

- Calculer le vecteur $Z^k = AP^k$.
- $s^k = (R^k, R^k)/(Z^k, P^k)$.
- $x^{k+1} = x^k + s^k P^k$.
- $R^{k+1} = R^k - s^k Z^k$.
- $t^k = (R^{k+1}, R^{k+1})/(R^k, R^k)$.
- $P^{k+1} = R^{k+1} + t^k P^k$.
- k=k+1.

End while

V. JACE (JAVA ASYNCHRONOUS COMPUTATION ENVIRONMENT)

JACE [1] est un environnement de programmation distribué permettant le calcul itératif synchrone et asynchrone, il permet de relier un ensemble de machines pour construire une machine virtuelle parallèle. Il est écrit entièrement en java donc il bénéficie des caractéristiques de ce langage notamment : il est multithreadé et portable sur toutes les plates-formes. JACE se caractérise aussi par:

- Communication via RMI (Remote Method Invocation).
- Gestion automatique de la convergence.
- Gestion transparente des mécanismes liés aux itérations asynchrones.
- Tolérance aux pertes de messages : la volatilité du système en termes de communication est gérée automatiquement par le noyau JACE via le mécanisme des files d'attente.

En plus, les environnements comme MPI (Message Passing Interface) et PVM (Parallel Virtual Machine) [6] sont monothreadés c.à.d ne supportent pas l'utilisation des processus légers, donc il est impossible de concevoir un algorithme IACA efficace. Pour les environnements multithreadés comme PM2 (Parallel Multithreaded Machine), le calcul évolue en parallèle avec la communication dans des threads différents. Le problème est que c'est au programmeur de gérer lui même les threads de communication et de calcul, la synchronisation pour l'accès au ressources partagées ..., ce qui n'est pas évident car il faut faire très attention aux situations d'interblocages le problème classique de la programmation multithreadée. En plus, les threads ne sont pas gérés de la même façon sur les différents systèmes d'exploitation.

Par contre JACE : il est portable car il est écrit entièrement en Java, gère de façon transparente les mécanismes liés à l'asynchronisme : écrasement des messages, détection de la convergence ..., c'est pour ça qu'il est le plus adéquat pour la mise en œuvre des algorithmes IACA.

VI. CONTRIBUTIONS

A. *Cas d'une matrice dense*

Pour mettre en œuvre les algorithmes itératifs asynchrones dans un système distribué avec JACE, nous avons commencé par créer un cluster de huit machines, 2 cœurs chacune (figure3) en installant JACE sur chaque machine. Par la suite, nous avons implémenté les trois algorithmes à savoir Jacobi, Gradient optimal (GOP) et Gradient conjugué (GC) sur ce cluster.

Figure 3. Cluster de 16 cœurs

La figure 4 montre le temps de calcul obtenu en fonction du nombre de cœurs impliqués dans le calcul réparti. Nous avons utilisé une matrice de taille 8000x8000 pour les trois méthodes implémentées. Ici, nous avons ignoré le temps de communication entre machines et nous avons calculé uniquement le temps de traitement nécessaire pour avoir la convergence pour les trois méthodes.

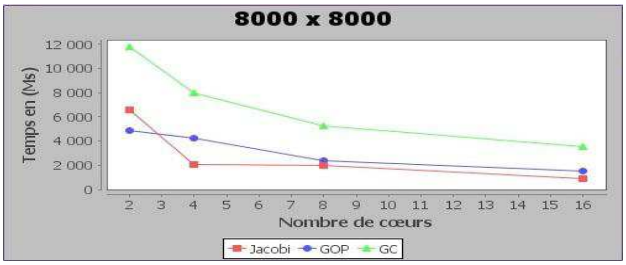

Figure 4. Comparaison entre Jacobi, Gradient optimal et Gradient conjugué

Nous remarquons que : Plus le nombre de machines ou le nombre de cœurs augmente, plus le temps d'exécution des trois méthodes diminue.

- La méthode de Jacobi en rouge est la plus performante en temps d'exécution, cela peut être expliqué au fait qu'une itération d'un algorithme de Gradient est plus coûteuse par rapport à une itération de Jacobi.

Nous nous sommes intéressés par la suite à une comparaison entre le temps de calcul et le temps de communication pour la méthode de Gradient avec pas optimal (GOP).

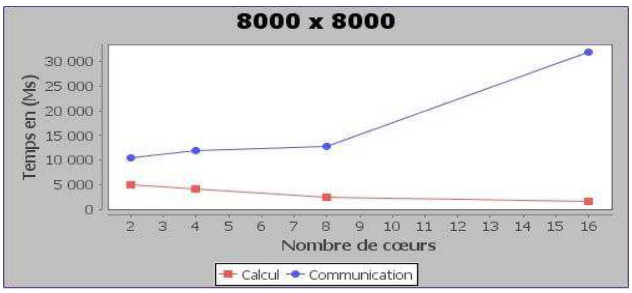

Figure 5. Comparaison entre le temps de calcul et le temps de communication (GOP)

D'après la figure 5, le temps de calcul diminue avec plus de processeurs participants en raison de moins de données par calcul. Évidemment, davantage de processeurs impliqués causeront des communications plus intensives qui entraîneront plus de temps de communication, ce qui est logique et compatible avec les notions classiques de complexité. En effet, si nous posons N le nombre de machines et M la taille de la matrice. Pour le calcul, la complexité dépend de M/N, car la matrice est divisée en bandes horizontales suivant le nombre de machines. Pour les communications, chaque machine doit communiquer son résultat aux (N-1) autres machines qui donne au final une complexité de O (n^2).

Résultat : Il faut faire des compromis entre le calcul et la communication, c.à.d. si nous voulons gagner en temps de calcul, nous devrions perdre en temps de communication pour des matrices de taille normale. Cela permet de dire que le temps de communication, dans ce cas, domine le temps de calcul. Mais dés que la taille de la matrice devient trop importante, le temps de calcul devient le dominant. Toutefois, faire des calculs sur ce type de matrice et stocker cette quantité d'éléments est très couteux en termes de mémoire et nécessite énormément de ressources matérielles.

B. Cas d'une matrice creuse

Pour être considérée comme creuse [11], une matrice doit contenir une certaine proportion d'éléments nuls. Les auteurs dans [5] ont définit différentes catégories de matrices creuses :

- **Catégorie 1 :** les matrices pas creuses avec au plus 50% de « 0.0 ».
- **Catégorie 2 :** les matrices peu creuses qui contiennent entre 50% et 70% de « 0.0 ».
- **Catégorie 3 :** les matrices moyennement creuses qui contiennent entre 70% et 80% de « 0.0 ».
- **Catégorie 4 :** les matrices franchement creuses qui contiennent entre 80% et 90% de « 0.0 ».
- **Catégorie 5 :** les matrices totalement creuses avec au moins 90% de « 0.0 ».

Dans ce qui suit, nous avons implémenté l'algorithme de Jacobi en choisissant une matrice franchement creuses avec 80% de 0.0. Nous avons obtenu les résultats indiqués dans le tableau suivant :

TABLE I. JACOBI AVEC MATRICE FRANCHEMENT CREUSES(NOMBRE DE COEUR /TAILLE N)

	80 (taille matrice)	**800**	**2000**	**4000**	**8000**
2 (machines)	3 (ms)	147	748	3222	4850
4	2	68	492	1066	2529
8	3	37	268	734	1479
16	1	22	174	636	1270

C. Structures compréssées des matrices creuses en mémoire

Si nous voulons stocker et manipuler efficacement des matrices creuses, il faut prendre en compte le creux dans l'algorithme. En fait, il n'est pas nécessaire de stocker des éléments nuls ou d'exécuter des calculs les concernant, c'est pour ça qu'il est nécessaire d'utiliser des algorithmes et des formats qui prennent en compte la structure peu dense de la matrice [7], ces algorithmes permettent de gagner en espace mémoire et en temps de calculs en permettant de :

- Ne pas enregistrer les éléments nuls.
- Retrouver aisément un élément en connaissant ses cordonnées.
- Exécuter facilement les opérations courantes.

1) Formats de représentation des matrices creuses

a) Format CSR

CSR [3, 4, 11] est l'acronyme de Compress Sparse Row. C'est le format le plus économe en terme d'espace mémoire. En fait, une matrice creuse est remplacée par trois tableaux unidimensionnels. Si nous note n le nombre d'éléments non nuls de la matrice et m le nombre de lignes de M :

- Un premier tableau A de taille n, contenant les éléments non nuls de la matrice rangés de gauche à droite.
- Un deuxième tableau JA, de taille identique à A, il contient le numéro de colonne de chaque élément non nul de A.
- Le troisième tableau IA, de taille m+1, l'élément i de IA contient le rang dans le tableau A du 1er élément de la ligne i de la matrice M, le dernier élément IA (m+1) contient n+1 où n est la taille de A.

Exemple: soit la matrice M

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 & 2 \\ 3 & 4 & 0 & 5 & 0 \\ 6 & 0 & 7 & 8 & 9 \\ 0 & 0 & 10 & 11 & 0 \\ 0 & 0 & 0 & 0 & 12 \end{bmatrix}$$

La compression au format CSR donne les tableaux suivants :

A = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}.

JA = {1, 5, 1, 2, 4, 1, 3, 4, 5, 3, 4, 5}.

IA = {1, 3, 6, 10, 12, 13}.

L'accès à l'élément M (l, c) de la matrice M se fait par :

- Recherche des indices de début et de fin de la ligne n°1

$$\begin{cases} \text{début} = \text{IA}(l) \\ \text{fin} = \text{IA}(l+1) - 1 \end{cases}$$

- Recherche de la valeur c dans **JA** entre **JA** (début) et **JA** (fin) (**recherche obtenue en complexité logarithmique**).
- Si **c** est trouvé à la position **JA(i)**, alors l'élément **M(l, c) = A(i)**, sinon **M (l, c) = 0**.

b) Format Ellpack-Itpack

Le format CSR présente quelques défauts (l'accès est un peu lent) au contraire pour Ellpack-Itpack [3, 4, 11]. Ce format est basé sur l'utilisation de deux tableaux à deux dimensions (1 : n, 1 : m) ou n est le nombre de lignes de la matrice d'origine et m est le nombre maximum d'éléments non nuls de cette matrice ligne par ligne.

- Le vecteur A, contient les éléments non nuls de la ligne correspondante dans la matrice.
- Le vecteur JA, contient le numéro de la colonne des éléments non nuls de A, zéro pour les éléments nuls.

Soit la matrice M définie précédemment :

Le format Ellpack-Itpack donne les deux matrices suivantes :

$$A = \begin{bmatrix} 1 & 2 & 0 & 0 \\ 3 & 4 & 5 & 0 \\ 6 & 7 & 8 & 9 \\ 10 & 11 & 0 & 0 \\ 12 & 0 & 0 & 0 \end{bmatrix} \quad \text{et} \quad JA = \begin{bmatrix} 1 & 5 & 0 & 0 \\ 1 & 2 & 4 & 0 \\ 1 & 3 & 4 & 5 \\ 3 & 4 & 0 & 0 \\ 5 & 0 & 0 & 0 \end{bmatrix}$$

L'accès à l'élément M (l, c) de la matrice M devient plus rapide que dans le format précédent, il se fera comme suit :

- Recherche de la valeur de c dans JA(l) (**recherche obtenue en complexité logarithmique**).
- Si c est trouvé à la position JA (l, i), M (l, c) = A(l, i), sinon M (l, c) = 0.

D. Parallélisation avec Ellpack-Itpack et CSR

Nous proposons de faire une hybridation entre Ellpack-Itpack et CSR pour arriver à une compression bien optimisée. Nous pourrons profiter de la facilité de découpage d'une matrice et la rapidité d'accès aux données de cette dernière

d'une part, et d'autre part de l'optimisation de stockage d'informations, ce qui est une spécificité pour la méthode CSR pour avoir un gain substantiel en espace mémoire.

La matrice est générée au format Ellpack-Itpack et remplie par un processus d'initialisation. Ce dernier est chargé de la distribution des données sur les différents processeurs, cela pourra se faire en découpant les tableaux A et JA du format Ellpack-Itpack en bandes horizontales, où chaque ligne sera allouée à un processeur, alors que chaque colonne sera distribuée sur l'ensemble des processeurs.

Après, chaque bande sera convertie au format CSR à l'instant de l'envoi de données aux processus de calculs où le volume de données à transmettre étant limité [9].

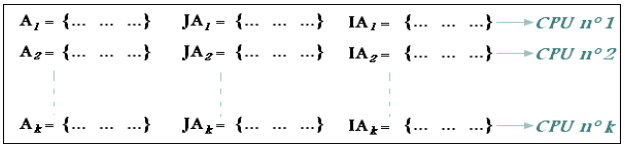

Figure 6. Répartition de la matrice

Algorithme de conversion du Ellpack-Itpack à CSR

Pos = 0 et IA_CSR(0) = 0
Pour i= 0, 1, 2…n faire
  Pour j = 0, 1, 2…n faire
    Si A_Ell(i)(j) ≠ 0 faire
      A_CSR(pos) = A_Ell(i)(j)
      JA_CSR(pos) = JA_Ell(i)(j)
      Pos = pos + 1
    Fin si
  Fin pour
IA_CSR(i+1) = pos
Fin pour

5.4 Résultat obtenu avec Ellpack-Itpack et CSR

Nous avons implémenté la compression Ellpack-Itpack et CSR avec la méthode de Jacobi et avec une matrice franchement creuses (80% de 0.0) de taille 8000x8000.

Le tableau ci dessous montre la comparaison entre le temps de calcul avec et sans compression.

TABLE II. GAINS OBTENUS AVEC ELLPACK-ITPACK ET CSR

	2	**4**	**8**	**16**
Jacobi_sans_compr	4850	2529	1479	1270
Jacobi_avec_compr	15 (ms)	11	10	10

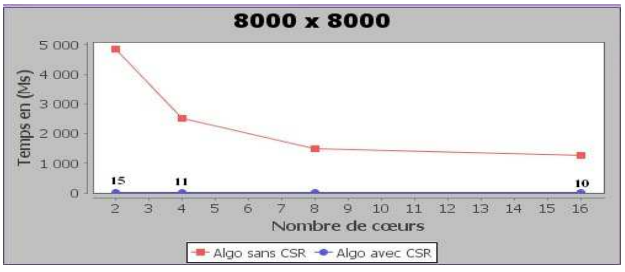

Figure 7. Performance de la compression CSR

Le tableau 2 et la figure 7 montrent bien qu'il y a une très grande différence en temps de calcul. C'est tout à fait logique, car l'algorithme utilisant la compression élimine les éléments inutiles (les zéros) ce qui implique un gain d'espace mémoire et de temps de calcul, ce qui nous permet de dire que même si l'algorithme standard de Jacobi est performent par rapport aux autres algorithmes, lorsqu'il s'agit d'une matrice creuse la compression de cette dernière devient une nécessité.

VII. CONCLUSION

Dans ce papier, nous avons implémenté trois méthodes de résolution de systèmes linéaires avec l'environnement JACE et en utilisant un cluster de 16 cœurs, le résultat obtenu montre clairement l'intérêt des IACAs pour ce type de méthodes notamment pour la méthode Jacobi qui était la plus satisfaisante. Nous avons également montré l'intérêt de la compression des matrices creuse en utilisant les formats CSR et Ellpack-Itpack qui a montré son intérêt dans le cas des matrices creuses afin de limiter le volume des données à transmettre.

REFERENCES

[1] Kamel Mazouzi, "JACE: un environnement d'exécution distribué pour le calcul itératif asynchrone". Thèse de Doctorat. Université de Franche-Comté, 2005.

[2] Frédéric Robin, "Etude d'architectures VLSI numériques parallèles et asynchrones pour la mise en oeuvre de nouveaux algorithmes d'analyse et rendu d'images". Thèse de Doctorat. Ecole supérieure des télécommunications, *PARIS*, 1997.

[3] Nahid Emad, Olfa Hamdi-Larbi et Zaher Mahjoub, "Vers la détection du meilleur format de compression pour matrice creuse dans un environnement parallèle". Rapport de l'université de Versailles Saint-Quentin-en-Yvelines, 2006.

[4] Haiwu HE, "Analyse avancées de la méthode hybride GMRES /LS-ARNOLDI asynchrone parallèle et distribuée pour les grilles de calcul et les supercalculateurs". Thèse de Doctorat. Université de LILLE, 2005.

[5] Alain David et al, "Module LI213 : Types et Structures de Données", Université Pierre et Marie Curie, année 2008-2009.

[6] Pétur Ingi Egilsson "John the Ripper on a Ubuntu 10.04 MPI Cluster", 17. June 2010.

[7] Mourad Hakem. "Asynchronous iterative algorithms & Reliability", University of Franche Comté – France, Juin 11, 2009.

[8] Raphaël Couturier "Algorithmes itératifs asynchrones sur grappes distantes et équilibrage de charge sur topologies dynamiques", Rapport HDR, Université de Franche Comté, 2005.

[9] Richard Wilson Vuduc "Automatic Performance Tuning of Sparse Matrix Kernels", Université de CALIFORNIA, BERKELEY, 2003.

[10] Nahid Emad et al. " Vers la détection du meilleur format de compression pour matrice creuse dans un environnement parallèle", Université de Versailles Saint-Quentin-en-Yvelines, année 2006.

[11] NVIDEA "CUDA Toolkit 4.1 CUSPARSE Library" January 2012.