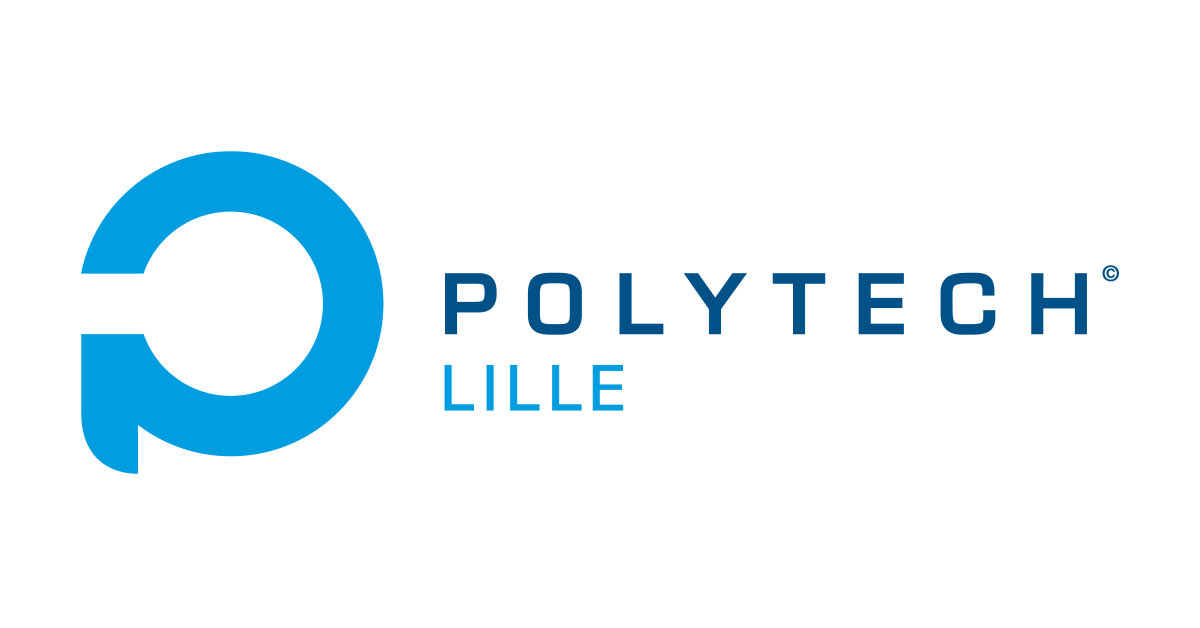
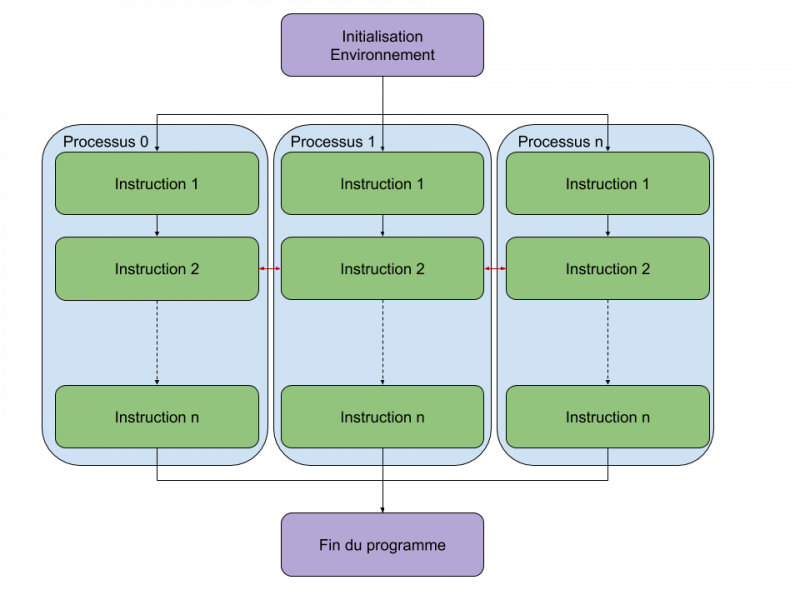
**RAPPORT DE TRAVAUX SUR L’UTILISATION DE MPI SUR LA MÉTHODE DE LA PUISSANCE**



Thomas Firmin - Yann Cauchepin

IS5 - Polytech Lille - 30/11/2020

**Introduction**

Ce travail pratique nous permet d’étudier d'implémenter la méthode de la puissance de façon parallèle avec la technologie MPI. On pourra ainsi comparer les performances de l'algorithme selon une implémentation séquentielle et parallèle MPI (multi-processeur).

Nous disposons pour cela de deux codes en langage C :

* *sequential.c* : création d’une matrice de taille n x n et implémentation de la méthode de la puissance de façon séquentiel.
* *checker.c* : code source ayant pour but de vérifier les résultats.

**I - Stratégie d’implémentation parallèle**

On suppose que le nombre de processeurs *p* disponibles divise la taille *n* d’une matrice carré *A* de taille *[n,n]*.

Pour rappel, la méthode de la puissance permet de résoudre un système: *y = Ax*, en effectuant séquentiellement un produit matrice vecteur *Ax* , une normalisation du vecteur résultat *y.* On réitère ce principe jusqu’à la convergence du vecteur propre. La convergence est déterminée par l’erreur entre le vecteur propre à l’itération *n-1*  avec celui de l’itération *n*.

On suggère une implémentation statique où un processus P0 répartit les données de calculs sur les autres processus Pk et récupère les données résultats. Contrairement au TP précédent, la meilleure stratégie à adopter ici est celle de la parallélisation des données, une implémentation statique. En effet, dans le calcul des fractales de Mandelbrot chaque calcul de pixel avait son propre temps de convergence. Or, ici la convergence est globale, on n’étudie que la convergence du vecteur propre. La charge de travail ligne par ligne est donc identique. Ainsi, il faut répartir les lignes équitablement entre les processeurs, afin d’équilibrer la charge de travail. Si tous les processeurs ont la même quantité de données à traiter alors ils auront la même charge de travail, c’est-à-dire le même nombre de calculs à effectuer. A l’exception du processus principal P0, qui aura la charge de synchroniser les données et de réunir les différents résultats.

Une telle répartition permet aussi de traiter des données plus volumineuses, par rapport à l’utilisation d’un seul processus. En effet, le processus principal n’a plus besoin de stocker toute la matrice, mais seulement la partie de la matrice qu’il doit traiter.

Il y a globalement trois stratégies pour répartir les données de la matrice dans la méthode de la puissance :

* En blocs de sous-matrices. Chaque processeur récupère en début d’implémentation une sous-matrice *[i,j]* de la matrice *A*. A chaque itération il doit récupérer le sous-vecteur *x* de taille *j* correspondant pour déterminer le sous-vecteur résultat *y* de taille *i* correspondant. On choisit d’écarter cette implémentation pour des raisons de simplicité. Elle peut toutefois être intéressante si la taille totale des sous-matrices *[i,j]* est inférieur à la taille d’une ligne/colonne. Cela suppose que l’on dispose de plus de processeurs qu’il n’y de lignes/colonnes de la matrice. Ainsi les données seraient équitablement réparties.
* En bloc de colonnes. Nous étudierons cette stratégie plus tard dans le travail pratique.
* En bloc de lignes. Chaque processeur récupère en début d’implémentation un bloc de ligne de largeur *i* de la matrice *A*. A chaque itération il doit récupérer le nouveau vecteur *x* pour déterminer le sous-vecteur résultat *y* de taille *i* correspondant. On privilégiera cette répartition des données dans un premier temps, puisqu' il s’agit de la méthode la plus simple à implémenter.

A noter que les processus conservent en mémoire le vecteur résultat de l’itération précédente pour pouvoir déterminer l’écart des composantes du vecteur d’une itération à l’autre.

**› Pseudo-code :** Implémentation parallèle MPI statique

Méthode de la puissance Implémentation parallèle MPI

*//Initialisation*

Matrice A[n,n]

Vecteur x[n], y[n]

*//p divise n*

*//k est le nombre de processeurs*

*//Sous-entendu les communications entre p0 et pk n’ont pas lieu si k=0*

**Pour p0 :** t ← n/p *//où t renseigne la largeur d’un bloc de ligne*

**envoie** bloc de ligne A[ik à ik+t] à tous les processeurs pk

**Chaque pk :** **reçoit** bloc de ligne A[ik à ik+t] de p0

*//Boucle de calcul*

**Pour p0 :** **envoie** vecteur initial à tous les processeurs pk

Tant que erreur > seuil **:** *//seuil à définir*

**y** ← **A.x** // Partie la plus coûteuse

//-- **CALCUL PARALLÈLE T1**

**Chaque pk : reçoit** signal de p0 *//Chronologie T1 à T4*

**reçoit** vecteur de p0 et le stocke dans x

calcule y[ik à ik+t]

**norm(y)** // Calcule la norme du vecteur résultat

**Chaque pk :** calcule une partie de la norm : y[ik à ik+t]²

**envoie** y[ik à ik+t]² à p0

//Synchronisation -- **CALCUL SÉQUENTIEL T2**

**Pour p0 :** **reçoit** y[ik à ik+t]² de tous les processeurs pk

calcule (au fur et à mesure des réceptions) la somme des y[ik à ik+t]² pour avoir y²

calcule norm = racine\_carré(y²)

**envoie** norm à tous les processeurs pk

//-- **CALCUL PARALLÈLE T3**

**Chaque pk :** **reçoit** norm de p0

calcule y[ik à ik+t] = y[ik à ik+t] / norm

**erreur** ← **norme2(x - y)**

**Chaque pk :** calcule une partie de l’erreur :

(x[ik à ik+t]-y[ik à ik+t])²

**envoie** (x[ik à ik+t]-y[ik à ik+t])² à p0

**envoie** y[ik à ik+t] à p0 //Récupération de y final

//Synchronisation -- **CALCUL SÉQUENTIEL T4**

**Pour p0 :** **reçoit** (x[ik à ik+t]-y[ik à ik+t])² et y[ik à ik+t]

de tous les processeurs pk

calcule (au fur et à mesure des réceptions) la somme des (x[ik à ik+t]-y[ik à ik+t])²

pour avoir (x[i]-y[i])²

calcule erreur = racine\_carré((x[i]-y[i])²)

**SI** erreur > seuil**ALORS**

**x** ← **y**

**Pour p0 :** **envoie** signal à tous les processeurs pk

**envoie** x à tous les processeurs pk

**SINON**

**Fin tant que**

**Pour p0 :** **modalités affichage :** *nombre d’itérations, temps de calcul*

**II - Analyse de l’implémentation parallèle MPI**

**› Communications**

Dans l’implémentation statique, le but est de paralléliser au mieux les données et de réduire au maximum le nombre de communication entre les processus. Ainsi, on peut distinguer quelques communications nécessaires:

* Répartition des données :

Au départ, nous pouvons adopter plusieurs stratégies. Dans ce travail pratique, la matrice est générée aléatoirement, elle n’est pas stockée en mémoire ROM.

On peut donc simuler deux situations:

* Si chaque bloc de la matrice est généré par le processus P0, puis envoyé aux autres processus. Alors on simule une mémoire non partagée entre les processeurs.
* Si chaque bloc est généré par chacun des processus, alors on simule d’une certaine façon le cas d’une mémoire partagée. Ce n'est pas exactement le cas ici, puisqu’il n’y a pas de communication, chaque bloc de la matrice est généré séquentiellement et non récupéré.

On peut supposer d’ores et déjà que la première implémentation sera moins efficace que la seconde puisque le processus principal devra envoyer des données volumineuses aux autres processus par des communications MPI.

Concernant le vecteur initial *x*. Puisque chaque élément est initialisé à *1/n*, chaque processus peut le calculer lui-même, ce qui permet d’éviter des communications entre le processus principal et les autres.

* Envoie et réception de la norme :

La somme des est parallélisable, mais pas la racine carré.  
Ici il s’agit d’un premier point de synchronisation entre tous les processus. Une fois le calcul matrice-vecteur effectué, tous les processus doivent envoyer la partie de la norme qu’ils ont calculée vers le processus principal. Le processus principal doit effectuer la somme de ces “parties de norme”, effectuer la racine carrée de la somme, et envoyer la norme à tous les processus pour que ceux-ci puissent normaliser la partie du vecteur *y* dont ils ont la charge.

* Envoie et réception des sous-parties du vecteur résultat :

Une fois que tous les processus ont normé leur sous-vecteur *y* qu’ils ont calculé précédemment, ceux-ci doivent les envoyer au processus principal. Ainsi le processus principal doit recevoir et reconstruire le vecteur *y*, à partir des "parties" envoyées par les autres processus et de la “partie” qu’il a lui-même calculé.

* Envoie et réception de l’erreur :

La somme des est parallélisable, mais pas la racine carré.

Il s’agit ici encore d’un point de synchronisation entre les processus où tous les processus calculent leur “partie” de l’erreur et l’envoie au processus principal. Celui-ci reçoit ces "parties", les additionne et effectue la racine carrée de la somme.

* Envoie et réception de l’ordre d’arrêt :

Une fois leur partie de l’erreur envoyée, tous les processus sont en attente d’un ordre du processus principal. Une fois que la condition d’arrêt a été déterminée par le processus principal, celui-ci l’envoie à tous les autres processus. On distingue donc 2 cas:

* La condition d’arrêt n’est pas vérifiée:

Le processus principal envoie l’ordre aux autres processus d’effectuer une itération de plus.

Puis le processus principal envoie le vecteur *x*, reconstruit en récupérant les sous-vecteurs résultats *y* des processeurs.

* La condition d’arrêt est vérifiée:

Le processus principal envoie l’ordre d’arrêt aux autres processus.

*NB: taille\_bloc = n/nombre\_processus*

| Etape | Fonctions | Justification | Taille du message |
| --- | --- | --- | --- |
| Envoie des données  *(par le processus principal)* | MPI\_Ssend | On utilise une émission bloquante synchrone, notamment à cause de la taille des données, et de l’implémentation. En effet, chaque bloc est calculé par le processus principal. Il réécrit en mémoire sur le bloc précédent. Ainsi on ne stocke pas entièrement la matrice mais seulement par itération d’un bloc. Puisque les messages peuvent être très longs, on privilégie donc Ssend qui ne passe pas par un tampon dont la taille est limitée. | Blocs de matrice :  taille\_bloc\*n  Vecteur *x* :  n |
| Réception des données  *(par les autres processus)* | MPI\_Recv | La réception doit être bloquante puisque l’on ne peut pas commencer la multiplication sous-matrice/vecteur avant de recevoir l’ensemble des données. | Blocs de matrice :  taille\_bloc\*n  Vecteur *x* :  n |
| Envoie des parties de la norme et erreur *(par les autres processus)* | MPI\_Send | On utilise une émission bloquante standard, puisque le processus n’a pas d’autres calculs à faire, indépendamment des autres processus. | 1 (\*taille d’un flottant) |
| Réception des parties de la norme et erreur  *(par le processus principal)* | MPI\_Recv | On utilise une réception standard bloquante, puisque le processus doit recevoir toute les “parties” de norme des autres processus pour pouvoir calculer la norme.  Il n’a de plus pas d’autres calculs indépendants à faire. | 1 (\*taille d’un flottant) |
| Envoie de la norme  *(par le processus principal)* | MPI\_Isend | On utilise une émission non bloquante standard, puisque le processus principal peut entamer la normalisation de sa partie du vecteur propre, indépendamment de l'envoi de la norme aux autres processus. | 1 (\*taille d’un flottant ou entier) |
| Réception de la norme, condition d’arrêt *(par les autres processus* | MPI\_Recv | On utilise une réception bloquante car les autres processus ont besoin de recevoir la norme, pour pouvoir normaliser leur partie du vecteur. Ils n’ont pas de calculs indépendants à faire en attendant.  De même pour la condition d’arrêt, ils doivent attendre l’ordre d’effectuer ou non une nouvelle itération. | 1 (1\*taille d’un entier) |
| Envoie des sous-parties du vecteur *y*  *(par les autres processus)* | MPI\_Send | On utilise une émission bloquante standard, on suppose que la taille du vecteur est assez faible pour être stockée dans le tampon. Les autres processus n’ont pas de calculs intermédiaires indépendants à faire durant l’envoi. Donc émission bloquante. | Vecteur *y* :  taille\_bloc |
| Réception des sous-parties du vecteur X  *(par le processus principal)* | MPI\_Recv | On utilise une émission bloquante standard. En effet, le processus principal doit attendre de recevoir tout le vecteur *y*, pour ensuite le renvoyer à tous les processus. | Vecteur *y* :  taille\_bloc |

A noter que la première version de l’implémentation pourrait être optimisée au niveau de l’envoie du vecteur *x* pour une nouvelle itération. En effet, chaque processus contient déjà un sous-ensemble du vecteur *x* avec le sous-vecteur *y* qu’il possède. On pourrait donc envoyer aux processus les parties du vecteur de *x* qu’il leur manque pour entamer une prochaine itération.

De même, on pourrait traiter le cas où le nombre de lignes *n* n’est pas divisible par le nombre de processus *p* disponibles, auquel cas les processus aurait un nombre équivalent de lignes plus ou moins une ligne. Il s’agit plus d’une amélioration d’ordre pratique qui n’est pas dans le sujet de l’analyse ici.

**III - Comparaison des implémentations séquentiel et parallèle MPI**

**› Performances/Accélération**

On se propose de comparer les performances en termes d’accélération et d’efficacité les deux implémentations séquentielles et parallèle de la méthode de la puissance : on décide de comparer leur exécution sur une matrice de taille *n* variable et avec un nombre de nœuds de calculs *p* variable, vérifiant bien une répartition identique du nombre de lignes entre les nœuds (*p* divise *n*).

A noter que le seuil de convergence du vecteur propre est fixé pour toutes les implémentations à 1-9.

On évalue bien le temps d'exécution de la méthode de la puissance, et non l’envoie des données (matrice + vecteur x) au départ. Ce temps est de plus ou moins 12sec, selon le nombre de messages et la taille de ceux-ci envoyer.

*\*Temps d'exécution en secondes, Accélération dessous (rapport au temps séquentiel)*

| Taille *n* de la matrice | Implémentation séquentielle | Implémentation parallèle basique | | |
| --- | --- | --- | --- | --- |
|
| *p* = 2 | *p* = 4 | *p* = 8 |
| 4096  (252 itérations) | 9.9 | 5.6  Acc : 1.8 | 3.1  Acc : 3.2 | 2.1  Acc : 4.7 |
| 8192  (322 itérations) | 50.2 | 27.5  Acc : 1.8 | 14.8  Acc : 3.4 | 9.3  Acc : 5.4 |
| 12288  (246 itérations) | 86.8 | 46.7  Acc : 1.9 | 24.4  Acc : 3.6 | 16.5  Acc : 5.2 |
| 16384  (326 itérations) | 204.8 | 111.9  Acc : 1.8 | 57.2  Acc : 3.6 | 31.5  Acc : 6.5 |

La conception de la matrice reste identique pour une *n* donnée : la valeur de *n* est déterministe dans nos implémentations, ce qui permet d’effectuer le même nombre d’itérations et de comparer sans biais les temps d'exécution entre les implémentations.

Les variations entre les temps d'exécution de la méthode de la puissance selon différentes tailles *n* pour une même implémentation n’ont pas vraiment de sens à être comparé : le temps d'exécution d’une itération est semblable. Ce qui nous intéresse est l’accélération obtenue entre l’implémentation parallèle par rapport à celle séquentielle pour un même nombre de lignes, selon un nombre différent de nombre de nœuds.

On remarque selon différentes tailles de la matrice, que l’augmentation du nombre de nœuds disponibles permet de réduire significativement les temps de calculs. Cependant le fait de doubler le nombre de nœuds, ne double pas la vitesse d’exécution. Si l’on étudie l’accélération, on remarque très bien qu’entre 2 et 4 nœuds, cette accélération est doublée par 2. Cependant entre 4 et 8 noeud, l’accélération n’est pas doublée, (\*1.80, pour n=16384). Cela est certainement dû à l’augmentation du nombre de communications entre les processus. On peut donc supposer que pour un nombre de nœuds infini, la limite du temps de calcul sera le temps de communication entre les processus, en théorie.

**› Limites**

On s’intéresse maintenant à l'exécution de la méthode de la puissance afin de comparer les limites de l’implémentation séquentielle par rapport à celle parallèle. Par défaut, nous utiliserons 8 nœuds dans l’implémentation parallèle.

| Taille *n* de la matrice | Implémentation séquentielle | Implémentation parallèle  avec 8noeuds |
| --- | --- | --- |
|
| 30000 | 389 itérations  815.1 s  MFlops : 819.4 | Allocation ... |
| 51025 | Allocation ... | Allocation ... |
| 51026 | Impossible d’allouer la matrice | Allocation ... |
| 408 208 | Impossible d’allouer la matrice | Impossible d’allouer la matrice |

On peut observer qu’à partir d’une valeur de *n* supérieur à 51026, l’algorithme séquentiel ne peut pas allouer toute la matrice. Cette allocation est toutefois réalisable avec l’implémentation parallèle car l’allocation s’effectue de manière itérative avec un seul bloc : l’allocation devient impossible lorsque la taille du bloc dépasse 51026 lignes (c’est-à-dire à partir de 408208 lignes pour 8 nœuds).

Cependant, même si l’implémentation parallèle peut allouer une matrice de taille plus grande, il faut vérifier que la capacité du buffer a un espace mémoire suffisant pour faire transiter le nombre important de lignes. Si ce n’est pas le cas, il faudra donc passer par des communications synchrones, n’utilisant pas de buffer.

**IV - Critique sur des stratégies alternatives de parallélisation**

On se propose de répartir les données de calcul en bloc de colonne.

Chaque processeur Pk (dont processus principal P0) récupère en début d’implémentation un bloc de colonnes de taille *t* de la matrice *A* et un sous-vecteur *x* de taille *t*. A chaque itération, il calcule le vecteur résultat intermédiaire *yk* puis récupère un sous-vecteur de *y* de taille *t* pour le normaliser et participer au calcul de l’erreur quadratique. Seul le sous-vecteur *y* de l’itération précédente est conservé en mémoire pour pouvoir calculer l’écart entre deux itérations. Le vecteur résultat final *y* est obtenu en faisant la somme des *yk*.

En comparant avec les phases de calcul de la répartition par ligne, la méthode de la puissance serait alors traité avec :

* Parallèle T1’ : Vecteur résultat intermédiaire, Norme intermédiaire
* Le processeur P0 ne communique plus qu’un sous vecteur *x* de taille *t* pour le produit matrice vecteur *Ax*.
* Le temps de calcul du produit matrice vecteur est similaire à celui de la répartition par bloc de lignes. La différence principale est l’accès mémoire.
* Tous les processus Pk calculent aussi la valeur au carré des composants du vecteur *yk* de taille *n* obtenu.
* Tous les processus Pk communiquent au processus P0 la norme intermédiaire du vecteur résultat *yk.*
* Tous les processus Pk communiquent au processus P0 le vecteur *yk* de taille *n* qu’ils ont calculé.
* Séquentiel T2’ : Vecteur résultat final, Norme totale du vecteur résultat non normée
* Le processus P0 reçoit les *k* vecteur *yk* de taille *n*. Il effectue un **Reduce with add (*MPI\_Reduce)*** pour obtenir le vecteur résultat *y*.
* Le calcul de la norme totale du vecteur *y* par P0 est inchangé en nombre de termes. Il s’agit de faire la somme des normes intermédiaires des *yk* comme dans le cas de la répartition par bloc de lignes.
* Le processus P0 envoie répartit un sous-vecteur de *y* de taille *t* à tous les processus Pk.
* Le processus P0 envoie la norme calculée à tous les processus Pk.
* Parallèle T3’ : Calcul vecteur normé, Calcul écart au carré
* Les processus Pk reçoivent un sous-vecteur *y* de taille *t* calculé par P0.
* Les processus Pk reçoivent la norme calculée par P0.
* Le calcul des sous-vecteurs normés de *y* ainsi que le calcul de l’écart avec le sous-vecteur précédent est effectué comme dans la répartition en bloc de lignes
* Séquentiel T4’ : Calcul de l'erreur quadratique total
* Les processus Pk envoient l’écart au carré de chaque composant du vecteur *y* qu’ils ont calculé à P0.
* Le processus P0 calcule la somme des écarts à la racine carrée pour obtenir l’erreur réelle.
* Le test sur l’erreur reste inchangé.

La plus grande différence avec une répartition par bloc de lignes est le calcul matrice vecteur. Le reste des séquences de calculs, tel que la normalisation du vecteur et le calcul de l’écart des composants du vecteur, reprend la même parallélisation.

Il semble plus rapide d’effectuer le calcul matrice vecteur avec la répartition en bloc de colonnes et en exécutant/récupérant, les sous-vecteurs avec un **Reduce with add**. Toutefois, des problèmes d’imprécision due à la somme de flottants peuvent impacter le résultat final.

**Conclusion**

Ce travail pratique nous a permis, une nouvelle fois, d’observer l’intérêt d’utiliser une implémentation parallèle pour l'exécution d’un produit matrice vecteur, ici s’agissant de la méthode de la puissance. Nous avons appliqué à nouveau la technologie *MPI* et observé les intérêts et inconvénients des différents types de communications (notamment celles utilisant un buffer) . Ce travail nous a également permis de réfléchir à la parallélisation de la méthode de la puissance et aux différentes stratégies de répartition des charges. Il pourrait être intéressant pour un prochain travail de travailler et d’observer les performances/limites d’une implémentation multi-processeurs et multi-thread et en utilisant une répartition en bloc de colonnes.

**› Question facultative**

On peut utiliser les routines BLAS dans notre algorithme.

Pour la multiplication de matrice/vecteur dans notre algorithme, on peut utiliser la routine: **dgemv**

Pour le calcul de la norme euclidienne, on peut cette fois-ci utiliser **dnrm2**. Toutefois on perd ici la parallélisation de la somme des carrés des éléments du vecteur.

Pour effectuer les différents calculs vecteur-scalaire (notamment pour la normalisation) on peut utiliser **dscal**

Une implémentation avec BLAS est disponible. Mais nous n’avons pas pu comparer les performances avec les autres implémentations. Le nombre d’itérations nécessaires à la convergence est drastiquement différent, et nous n’avons pas réussi à trouver d’où cela venait. Cependant une version de sequentiel.c avec BLAS, fonctionne et donne de bon résultats. :(

Source:

<http://www.netlib.org/lapack/explore-html/index.html>