INP TOULOUSE ENSEEIHT

Master 2 Informatique et Mathématiques Appliquées
Année universitaire 2014/2015

Rapport de stage de fin d'études

Période du 16/03/2015 au 15/09/2015

Projet R++, the next step

Passage à l'échelle de la régression linéaire aux données de grande dimension: Big Data

Elaboré par:
Zakariae Jorti

Encadré par:
Serge Gratton, tuteur et responsable du Master
Christophe Genolini, chef de projet
Xavier Vasseur, chercheur de l'équipe ALGO

Plan

I. Description du laboratoire d'accueil et des équipes de recherches

(a) UMR 1027 INSERM, Université de Toulouse III

(b) Équipe Parallel Algorithms, CERFACS, Toulouse

II. Contexte, positionnement et objectif

III. Programme scientifique et technique

1. Principe de la Régression en statistique
2. La régression vue en algèbre linéaire numérique
    a. Passage au problème des moindres carrés
    b. Introduction de quelques méthodes de factorisation (Cholesky, QR)
    c. Recours aux algorithmes par bloc
3. Parallélisation :
    a. Distribution de données
    b. Implémentation parallèle des algorithmes par bloc
4. Haute performance et données de grande dimension :
    a. Introduction de ScaLAPACK
    b. Les méthodes de factorisation out-of-core et leurs avantages
    c. Aspects techniques out-of-core

IV. Applications et résultats expérimentaux

1. Type de matrices étudiées

2. Tests de régression linéaire sur Matlab & R
3. Régression linéaire en in-core avec ScaLAPACK
4. Optimisation de la distribution de données in-core sur ScaLAPACK
5. Régression linéaire en out-of-core avec des prototypes de ScaLAPACK
6. Difficultés & obstacles

V. Conclusion & perspectives

1. Récapitulatif du travail effectué
2. Perspectives futures
3. Remerciements
4. Conclusion générale

I. Description du laboratoire d'accueil et des équipes de recherche

(a) UMR 1027 INSERM, Université de Toulouse III

L'Institut national de la santé et de la recherche médicale (INSERM) est une institution française de recherche biomédicale et de santé publique.

Créé en 1964, l'INSERM est un établissement public à vocation scientifique et technique. Le suivi et l'expertise scientifique font ainsi partie intégrante des missions de l'INSERM.
L'institut regroupe 339 unités de recherches, dirigées par 6500 membres de personnel permanent. 80% des unités de recherche de l'INSERM sont intégrées dans les centres hospitaliers universitaires.

L'Inserm joue un rôle clé dans la création de l'Espace européen de la recherche et s'engage dans des partenariats étroits avec des équipes et laboratoires de recherche à l'étranger afin de mieux se faire connaître au niveau européen et international.

Tout au long de mon stage avec l'INSERM, j'ai été encadré par M. Christophe Genolini chercheur au sein de l'unité mixte de recherche UMR 1027.
Les travaux de l'UMR1027 "Épidémiologie et analyses en santé publique : risques, maladies chroniques et handicaps" ont pour objectif d'accroître les connaissances sur les déterminants physiopathologiques et sociaux, les modes de prise en charge et les conséquences des pathologies chroniques.

(b) L'équipe Parallel Algorithms, CERFACS, Toulouse

Le CERFACS (Centre Européen de Recherche et de Formation Avancée en Calcul Scientifique) est un organisme de recherche qui a pour objectif de développer des méthodes avancées pour la simulation numérique et la résolution algorithmique de problèmes scientifiques et technologiques de grande envergure qui revêtent de l'intérêt pour la recherche, mais aussi pour l'industrie, et qui requièrent l'utilisation du matériel informatique le plus puissant disponible.

Le CERFACS accueille des équipes interdisciplinaires, pour la recherche et la formation avancée qui sont composées de: physiciens, mathématiciens appliqués, numériciens, et ingénieurs logiciel.

A peu près 150 personnes travaillent au CERFACS, incluant plus de 130 chercheurs et ingénieurs, en provenance de 10 pays différents. Ils travaillent sur des projets spécifiques s'articulant autours de neuf domaines principaux de recherche: algorithmie parallèle, couplage de codes, aérodynamique, turbines à gaz, combustion, climat, impact environnemental, assimilation de données, acoustique.

Durant mon stage, j'ai intégré l'équipe ALGO (algorithmes parallèles) dont les principaux objectifs sont:

- L'étude et le développement de méthodes numériques et logiciels pour un usage optimal (performance et fiabilité) des ordinateurs scalaires, vectoriels, et parallèles dans le calcul scientifique.

- La promotion du calcul haute performance par le biais d'une formation avancée et des activités de conseil.

Au sein de cette équipe, j'étais encadré par M. Serge Gratton professeur à l'ENSEEIHT, chercheur à l'IRIT et au CERFACS, et par M. Xavier Vasseur, chercheur au CERFACS.

II. Contexte, positionnement et objectif de la proposition

Les dernières années ont vu une profonde modification du paysage statistique. D'un côté, nombre de problèmes insolubles en un temps raisonnable sont résolus par des méthodes utilisant l'aléatoire et donnant un résultat approximatif (comme par exemple les algorithmes de classification [19], ou la construction d'un modèle de régression). La qualité des résultats est alors fortement liée au nombre de tentatives pour trouver la meilleure solution [20,21]. Le coût en temps devient un facteur limitant. D'un autre côté, les méthodes d'acquisition des données ont bouleversé tous les champs disciplinaires: Des puces enregistrant automatiquement et de manière quasi continue des données individuelles aux enquêtes épidémiologiques effectuées par Internet [22] en passant par les données issue de la bourse ou du Web, il n'est plus rare d'avoir à analyser des bases de données comportant des millions d'individus ou de variables. La taille des données devient ainsi un problème critique.

Ces problématiques sont actuellement un sujet brûlant dans la communauté informatique. Les divers «déluges de données» et autres applications basées sur l'exploitation massive de données de grandes tailles ont d'ores et déjà donné lieu à de nombreux travaux touchant aux aspects parallèles, entrées/sorties et mémoire des ressources nécessaires pour les résoudre [23,24,25].

R++, the next step est un projet de développement d'une nouvelle implémentation du langage et environnement d'analyse statistique. Il est **compilable**, intègre de la **gestion du parallélisme** et permettant l'exploitation des **bases de données grande dimension**. R++ est également un **langage objet**. Par analogie avec C++ et C, il s'appelle R++. A l'origine de R++, il y a R qui est un langage riche, en pleine expansion, présentant de nombreux avantages [*https://www.r-project.org/].*
Il est donc naturel de s'interroger sur la pertinence d'une nouvelle implémentation. Examiner les forces mais aussi les limitations de R met clairement en lumière l'intérêt d'un tel projet. D'une part, R est gratuit et en open source, il permet une rapide intégration de nouvelles méthodes et est soutenu par une large communauté. D'autre part, on sait qu'il présente quelques limites, notamment la difficulté à gérer les données de grande dimension et la non gestion intrinsèque du parallélisme. Le projet R++ auquel je contribue dans le cadre de mon stage de fin d'études, aux côtés d'autres chercheurs et spécialistes, s'inscrit dans la lignée des projets pour répondre aux problématiques décrites précédemment, et a donc pour vocation de dépasser les limites de R tout en conservant ses forces. A terme, le but est d'améliorer les performances mais aussi de faciliter la vie de l'utilisateur.

Parmi les différents fonctionnalités d'analyse statistique, nous nous sommes plutôt penchés au cours de ce stage, vers les problèmes de régression qui revêtent une

importance capitale dans le domaine des statistiques.
En effet, la complexité croissante des procédés et les avancées majeures dans les systèmes de mesure ont entraîné une prolifération des données composées de grands nombres de variables fortement corrélées, et souvent l'intérêt du chercheur est de décrire, résumer ou découvrir des relations structurelles entre les variables mesurées, dans la plupart des applications, l'objectif est de relier un sous-ensemble de variables, appelé l'espace réponse, à l'ensemble du reste des variables qu'on appelle l'espace prédicteur. Par exemple, un chercheur pourrait être tenté de relier un ensemble de variables de qualité mesurés sur un produit fini à un ensemble de variables de procédés mesurées tout au long du procédé de production. Les techniques de régression se présentent alors comme les méthodes de choix dans de domaine de l'analyse statistique pour traiter de tels sujets. En particulier, l’une des méthodes statistiques les plus utilisées dans les sciences appliquées est la méthode de régression linéaire. Elle permet d’évaluer que la hausse d’une variable (par exemple, l’aide au développement) est associée à un effet plus ou moins important à la hausse ou à la baisse sur une autre variable (par exemple, la croissance économique).

Le passage en algèbre linéaire numérique se fait via la formulation d'un problème de moindres carrés comme suit:
Soit a_1 ... a_n l'ensemble des variables prédictrices. Par soucis de simplicité, on suppose qu'on dispose d'une seule variable à expliquer b. On note alors pour un j fixé dans [1,n] et i variant dans [1,m], b_i et a_{ij} sont respectivement les i-ièmes observations des variables b et de a_j . En stockant ces données sous forme de matrice $A = (a_{ij})$ et de vecteur colonne $b = (b_i)$, on peut formuler le problème de moindres carrés équivalent comme étant: retrouver le vecteur x de coefficients (x_j) qui minimise la somme des carrés d'erreurs $(b_i - \sum a_{ij} x_j)^2$.

Ce genre de problèmes très fréquents en statistiques est très bien maîtrisé dans le domaine de l'algèbre linéaire numérique, et pour faciliter sa résolution il convient de factoriser la matrice A sous une forme particulière aux propriétés bien connues (QR, LU, LL^T, etc...). Des routines de factorisation-résolution ont déjà été implémentées dans la bibliothèque LAPACK(Fortran) dans un premier temps, avant d'être ré-implémentées pour les ordinateurs parallèles à mémoire distribuée dans la bibliothèque ScaLAPACK(Fortran).
Ces outils se présentent actuellement comme les plus performants en terme de taille de problèmes traités, précision et temps d'exécution (cf. Chapitre IV), voilà pourquoi il est très avantageux de les incorporer dans R. Toutefois, leur utilisation s'avère un peu compliquée pour quelqu'un qui ne s'y connaît pas forcément en algèbre linéaire numérique ou en Fortran. Par conséquent, nous mettons au point un programme qui traite des problèmes de régression de grande dimension, mais qui reste simple

d'utilisation pour profiter à tout type d'utilisateurs: Ainsi, pourront l'exploiter des médecins, des chercheurs, des statisticiens, des datascientists, etc...
Vu sous cet angle, R++ apparaît donc comme un projet d'adaptation des méthodes mathématiques d'algèbre linéaire numérique, et informatiques de parallélisation au contexte statistique.

III. Programme scientifique et technique

Principe de la Régression en statistique

Une situation courante en épidémiologie est d'avoir à sa disposition deux ensembles de données de taille n, $\{y_1,y_2,...,y_n\}$ et $\{x_1,x_2,...,x_n\}$, obtenus expérimentalement ou mesurés sur une population. Le problème de la régression consiste à rechercher une relation pouvant éventuellement exister entre les x et les y, par exemple de la forme y=f(x). Lorsque la relation recherchée est affine, c'est-à-dire de la forme y=ax+b, on parle de régression linéaire. Mais même si une telle relation est effectivement présente, les données mesurées ne vérifient pas en général cette relation exactement. Pour tenir compte dans le modèle mathématique des erreurs observées, on considère les données$\{y_1,y_2,...,y_n\}$ comme autant de réalisations d'une variable aléatoire Y et parfois aussi les données$\{x_1,x_2,...,x_n\}$comme autant de réalisations d'une variable aléatoire X. On dit que la variable Y est la variable dépendante ou variable expliquée et que la variable X est la variable explicative.

Les données$\{(x_i,y_i),i=1,...,n\}$ peuvent être représentées par un nuage de n points dans le plan (x, y), le diagramme de dispersion. Le centre de gravité de ce nuage peut se calculer facilement : il s'agit du point de coordonnées $(x,y)=(\Sigma x_i / n, \Sigma y_i / n)$. Rechercher une relation affine entre les variables X et Y revient à rechercher une droite qui s'ajuste le mieux possible à ce nuage de points. Parmi toutes les droites possibles, on retient celle qui jouit d'une propriété remarquable : c'est celle qui rend minimale la somme des carrés des écarts des valeurs observées y_i à la droite $\hat{y}_i=ax_i+b$. Si ε_i représente cet écart, appelé aussi résidu, le principe des moindres carrés ordinaires (MCO) consiste à choisir les valeurs de a et de b qui minimisent $E=\sum_{i=0}^{n} \varepsilon_i{}^2=\sum_{i=0}^{n} (y_i-(ax_i+b))^2$

Un calcul montre que ces valeurs, notées â et $\hat{b}$, sont´égales à

$$\hat{a}=\frac{\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})}{\sum_{i=1}^{n}(x_i-\bar{x})^2} \qquad et \qquad \bar{b}=\bar{y}-\hat{a}\,\bar{x}$$

On exprime souvent â au moyen de la variance de X, s_x^2, et de la covariance des variables aléatoires X et Y, cov_{xy}:

$$\hat{a}=cov_{xy}/s_x^2 \;\;,\;\; avec \;\; s_x^2=\frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})^2 \quad et \quad cov_{xy}=\frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})$$

La régression vue en algèbre linéaire numérique

a. Passage au problème des moindres carrés

Dans le cas général où notre variable quantitative Y dite à expliquer est mise en relation avec p variables quantitatives $X^1,...,X^p$ dites explicatives. Les données sont supposées provenir de l'observation d'un échantillon statistique de taille n (n > p+ 1) de $\mathbb{R}^{(1+p)}$: $(x_i^1,\dots,x_i^j,\dots,x_i^p, y_i) \quad i=1,\dots,n$

L'écriture du modèle linéaire dans cette situation conduit à supposer que l'espérance de Y appartient au sous-espace de $\mathbb{R}^n$ engendré par { $1,X^1,...,X^p$ } où 1 désigne le vecteur de $\mathbb{R}^n$ constitué de "1" . C'est-à-dire que les (p+ 1) variables aléatoires vérifient : $y_i=\beta_0+\beta_1 x_i^1+\beta_2 x_i^2+\dots+\beta_p x_i^p+u_i \quad i=1,2,\dots,n$ avec les hypothèses suivantes [28] pour les modèles statistiques linéaires:

- Les u_i sont des termes d'erreur, d'une variable U, non observés, indépendants et identiquement distribués ; $E(u_i) = 0$; $Var(U) = \sigma_u^2 I$. (Gauss-Markoff)

- Les termes x^j sont supposés déterministes (facteurs contrôlés) ou bien l'erreur U est indépendante de la distribution conjointe de $X^1,...,X^p$:
  $E[Y|X^1,\dots,X^p]=\beta_0+\beta_1 X^1+\beta_2 X^2+\dots+\beta_p X^p \quad et \quad Var[Y|X^1,\dots,X^p]=\sigma_u^2$.

- Les paramètres inconnus $\beta_0,...,\beta_p$ sont supposés constants.

- Les données sont rangées dans une matrice X (n × (p+ 1)) de terme général x^j_i , dont la première colonne contient le vecteur 1($x^j_0 = 1$), et dans un vecteur Y de terme général y_i. En notant les vecteurs $u = [u_1 \dots u_n]^T$ et $\beta = [\beta_0\ \beta_1 \dots \beta_p]^T$, le modèle s'écrit matriciellement : $y = X\beta + u$.

Conditionnellement à la connaissance des valeurs des X^j, les paramètres inconnus du modèle : le vecteur β et σ_u^2 (paramètre de nuisance), sont estimés par minimisation du critère des moindres carrés (M.C.) . L'expression à minimiser sur $\beta\in\mathbb{R}^{1+p}$ s'écrit:

$$\begin{aligned}\sum_{i=1}^{n}(y_i-\beta_0-\beta_1 x_i^1-\beta_2 x_i^2-\dots-\beta_p x_i^p)^2 &= \|y-X\beta\|^2\\ &= (y-X\beta)^T(y-X\beta)\\ &= y^T y-2\beta^T X^T y+\beta^T X^T X\beta\end{aligned}$$

b. *Introduction de quelques méthodes de factorisation (Cholesky, QR)*

Équations normales

Par dérivation matricielle de la dernière équation on obtient les “équations normales”:

$X^Ty - X^TX\beta = 0$

dont la solution correspond bien à un minimum car la matrice hessienne $2X^TX$ est semi définie-positive.

Nous faisons l’hypothèse supplémentaire [28] que la matrice X^TX est inversible, c’est-à-dire que la matrice X est de rang (p+1) et donc qu’il n’existe pas de colinéarité entre ses colonnes. En pratique, si cette hypothèse n’est pas vérifiée, il suffit de supprimer des colonnes de X et donc des variables du modèle.

Alors, l’estimation des paramètres β_j est donnée par :

$b = (X^TX)^{-1} X^Ty$

et les valeurs ajustées (ou estimées, prédites) de y ont pour expression :

$$\hat{y} = Xb = X (X^TX)^{-1} X^Ty = Hy$$

où $H = X (X^TX)^{-1} X^T$ est appelée “hat matrix”.

Géométriquement, c’est la matrice de projection orthogonale dans $\mathbb{R}^n$ sur le sous-espace Vect(X) engendré par les vecteurs colonnes de X.

On note : $e = y - \hat{y} = y - Xb = (I - H) y$ le vecteur des résidus ; c’est la projection de y sur le sous-espace orthogonal de Vect(X) dans $\mathbb{R}^n$.

Résolution des équations normales à l'aide de la factorisation de Cholesky

Une méthode standard pour résoudre les équations normales $X^T X \beta = X^T y$ est de procéder à une factorisation de Cholesky [29], qui consiste à construire une matrice R triangulaire supérieure, vérifiant $X^T X = R^T R$. Le problème devient donc: $R^T R \beta = X^T y$.

L'algorithme est le suivant:

Moindres carrés via les équations normales et la factorisation de Cholesky
1. Former la matrice $X^T X$ et le vecteur $X^T y$
2. Calculer la factorisation de Cholesky $X^TX = R^T R$
3. Résoudre le système triangulaire-inférieur $R^Tw = X^Ty$ pour w.
4. Résoudre le système triangulaire-supérieur $R\beta = w$ pour β.

Les étapes qui requièrent le plus de travail pour ce calcul sont les deux premières.Grâce à la symétrie, le calcul de X^TX ne nécessite que $m \times n^2$ flops, la moitié du coût de calcul si X^T et X étaient des matrices arbitraires de mêmes dimensions. La factorisation de Cholesky, qui exploite aussi la symétrie, nécessite $n^3 / 3$ flops. En fin de compte, la résolution du problèmes aux moindres carrés par les équations normales et la factorisation de Cholesky entraîne une charge totale de travail de : $m \times n^2 + n^3 / 3$ flops.

<u>*N.B. :*</u> flop <=> opération en virgule flottante comme l'addition et la multiplication.

Résolution du problème des moindres carrés à l'aide de la factorisation QR

i) Triangularisation de Householder

La méthode de Householder consiste à multiplier une matrice A à gauche par une succession de matrices élémentaires unitaires Q_k , de sorte que la matrice résultante $\underbrace{Q_n \cdots Q_2 Q_1}_{Q^T} A = R$ est triangulaire supérieure. Le produit $Q = Q_1^T Q_2^T \ldots Q_n^T$ est aussi unitaire, et donc A = QR est une factorisation QR complète de A [31].

On doit cette méthode à Alston Householder qui en a initialement proposé l'idée en 1958. Un moyen ingénieux de former la matrice Q_k telle que $Q_n \ldots Q_2 Q_1 A$ soit triangulaire supérieure est de choisir Q_k pour introduire des zéros en dessous de la diagonale à la k-ième colonne tout en préservant tous les zéros introduits précédemment. Par exemple, dans le cas 5×3, trois opérations Q_k sont appliquées comme suit. Dans ces matrices, le symbole $\ast$ représente une entrée qui n'est pas nécessairement nulle, et le même caractère en gras, indique une entrée qui vient d'être changée. Les entrées vides sont nulles.

$$\underset{A}{\begin{bmatrix} \ast & \ast & \ast \\ \ast & \ast & \ast \\ \ast & \ast & \ast \\ \ast & \ast & \ast \\ \ast & \ast & \ast \end{bmatrix}} \overset{Q_1}{\rightarrow\rightarrow} \underset{Q_1 A}{\begin{bmatrix} \boldsymbol{\ast} & \boldsymbol{\ast} & \boldsymbol{\ast} \\ \mathbf{0} & \boldsymbol{\ast} & \boldsymbol{\ast} \\ \mathbf{0} & \boldsymbol{\ast} & \boldsymbol{\ast} \\ \mathbf{0} & \boldsymbol{\ast} & \boldsymbol{\ast} \\ \mathbf{0} & \boldsymbol{\ast} & \boldsymbol{\ast} \end{bmatrix}} \overset{Q_2}{\rightarrow\rightarrow} \underset{Q_2 Q_1 A}{\begin{bmatrix} \ast & \ast & \ast \\ & \boldsymbol{\ast} & \boldsymbol{\ast} \\ & \mathbf{0} & \boldsymbol{\ast} \\ & \mathbf{0} & \boldsymbol{\ast} \\ & \mathbf{0} & \boldsymbol{\ast} \end{bmatrix}} \overset{Q_3}{\rightarrow\rightarrow} \underset{Q_3 Q_2 Q_1 A}{\begin{bmatrix} \ast & \ast & \ast \\ & \ast & \ast \\ & & \boldsymbol{\ast} \\ & & \mathbf{0} \\ & & \mathbf{0} \end{bmatrix}}$$

D'abord, Q_1 opère sur les lignes 1,...,5, en introduisant des zéros aux positions (2,1), (3,1), (4,1), et (5,1). Ensuite, Q2 opère sur les lignes 2,...,5, en introduisant des zéros aux positions (3,2), (4,2), et (5,2) mais sans affecter les zéros déjà introduits par Q1. Finalement, Q3 opère sur les lignes 3,...,5, en introduisant des zéros aux positions (4,3) et (5,3) sans affecter les zéros introduits précédemment.

En règle générale, Q_k opère sur les lignes k,...,m. Au début de l'étape k, il y a un bloc de zéros dans les k-1 premières colonnes de ces lignes. Appliquer Q_k permet de former des combinaisons linéaires de ces lignes, et les combinaisons linéaires des entrées nulles restent nulles. Après n étapes, toutes les entrées en dessous de la diagonale ont été éliminées et $Q_n \ldots Q_2 Q_1 A = R$ est triangulaire supérieure.

ii) Réflecteurs de Householder

Comment construire les matrices Q_k pour introduire des zéros comme indiqué précédemment? La démarche standard est comme suit. À la k-ième étape, Q_k est choisie comme étant une matrice unitaire de la forme $Q_k = \begin{bmatrix} I & 0 \\ 0 & F \end{bmatrix}$, où I est la matrice identité de taille (k-1)×(k-1) et F est une matrice unitaire de taille (m+1-k)×(m+1-k). La multiplication par F doit introduire des zéros à la k-ième colonne. L'algorithme de Householder opte pour une matrice particulière F appelée réflecteur de Householder.

Supposons qu'au début de l'étape k, les entrées k,...,m de la k-ième colonne sont données

par le vecteur x. Pour introduire les zéros appropriés à la k-ième colonne, le réflecteur de Householder F devrait affecter la zone suivante:

$$x = \begin{bmatrix} * \\ * \\ * \\ \vdots \\ * \end{bmatrix} \xrightarrow{F} Fx = \begin{bmatrix} \|x\| \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \|x\| e_1$$

La formule exacte du réflecteur F est donnée par: $F = I - (2 / v^T v)\, vv^T$

où $v = \text{signe}(x_1) \|x\| e_1 + x$; x_1 étant le premier élément de x et signe la fonction de $\{0,1,-1\}^{\mathbb{R}}$: $y \rightarrow \begin{Bmatrix} signe(y)=1 & si & y>0 \\ signe(y)=0 & si & y=0 \\ signe(y)=-1 & si & y<0 \end{Bmatrix}$

Notons que F est une matrice unitaire de rang plein.

On en déduit donc l'algorithme de factorisation d'une matrice par triangularisation de Householder [31] :

Factorisation QR par triangularisation de Householder

Pour k = 1 jusqu'à n

- $x = A_{k:m,k}$
- $v_k = \text{signe}(x)\|x\|_2 e_1 + x$
- $v_k = v_k / \|v_k\|_2$
- $A_{k:m,k:n} = A_{k:m,k:n} - 2v_k(v_k^T A_{k:m,k:n})$

En reprenant la même notation, on a construit une matrice $Q' = Q_n \ldots Q_2 Q_1$ telle que:
$Q' * A = R$. En plus, chaque matrice F vérifiant $F^T F = I$, on a Q_k orthogonale pour tout k.
Et par la suite, on établit que le produit $Q_n \ldots Q_2 Q_1$ est aussi une matrice orthogonale.
Par conséquent, on pourra noter $Q = (Q')^{-1}$ et écrire: $A = Q*R$ où Q orthogonale et R triangulaire supérieure.

iii) Résolution du problème des moindres carrés

La méthode classique moderne pour résoudre un problème aux moindres carrés, connue dès les années soixante, est basé sur la factorisation QR qui est formulée comme suit:
Sous la même hypothèse faite précédemment, (c-à-d X de rang plein rang(X) = n $\leqslant$ m), il existe une matrice orthogonale Q de $\mathbb{R}^{m\times m}$ et une matrice R triangulaire supérieure de $\mathbb{R}^{m\times n}$ telles que: $X = Q\,R = Q*\begin{bmatrix} R_1 \\ 0 \end{bmatrix}$ et $R_1 \in \mathbb{R}^{n\times n}$.

La matrice Q est formée à partir du procédé de triangularisation de Householder.
En posant $Q^T y = \begin{bmatrix} c \\ d \end{bmatrix}$ et $c \in \mathbb{R}^n$; $d \in \mathbb{R}^{m-n}$ on a le critère des moindres carrés à minimiser pour β qui devient:

$$\|X\beta - y\|_2^2 = \|Q^T X\beta - Q^T y\|_2^2 \qquad (car\ Q^T\ orthogonale)$$
$$= \left\| \begin{bmatrix} R_1 \\ 0 \end{bmatrix} * \beta - \begin{bmatrix} c \\ d \end{bmatrix} \right\|_2^2$$
$$= \|R_1\beta - c\|_2^2 + \|d\|_2^2$$

d ne dépendant pas de β, il en découle que la solution au problème des moindres carrés correspond à celle du système triangulaire supérieur $R_1\beta = c$.

Remarque:
Force est de noter ici qu'en utilisant la décomposition précédente, et en remplaçant dans les équations normales, on aboutit sur la relation $R_1^T R_1\beta = [R_1^T\ 0]\ Q^T y$, ce qui est équivalent à l'équation $R_1\beta = c$ puisque R_1 est une matrice carrée de rang plein donc inversible.

L'algorithme suivant résume les étapes de résolution :

Moindres carrés via la factorisation QR
1. Calculer la factorisation QR de A 2. Calculer les n premières composantes du vecteur $Q^T y$: c 3. Résoudre le système triangulaire supérieur $R_1\beta = c$ pour β.

Dans cet algorithme, c'est l'étape de factorisation qui domine en terme de charge de travail pour cet algorithme.
Le nombre total d'opérations est estimé à : $2n^2 \times (3m - n)/3$ flops.

c. Recours aux algorithmes par bloc

En regardant les algorithmes décrits plus haut de plus près, il nous paraît clair que certains processus de factorisation pourraient être considérablement améliorés puisque dans les cas les plus simples par exemple, les mêmes opérations en virgule flottante se feraient en même temps, en respectant l'ordre d'exécution.
En effet, poser les formules de factorisation sous une forme par blocs nous amène à réexaminer les différents algorithmes de factorisation, afin de les transformer en des algorithmes par blocs: des algorithmes qui au lieu d'opérer sur les éléments un par un de la matrice de départ, opéreront sur les blocs ou sous-matrices de celle-ci.

i) Factorisation de Cholesky

Pour illustrer ce point, considérons dans un premier temps l'algorithme de factorisation de Cholesky, qui factorise une matrice symétrique définie positive en $A = U^T U$ où U doit être triangulaire supérieure:

$$U=\begin{pmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ & u_{22} & \dots & u_{2n} \\ & & \ddots & \vdots \\ & & & u_{nn} \end{pmatrix}$$ où n est la taille de la matrice A.

D'après la version standard de l'algorithme, la matrice U est calculée de la manière suivante:

$$(1) \qquad u_{ii}=(a_{ii}-\sum_{k=1}^{i-1} u_{ki}{}^2)^{1/2} \qquad i=1,2,\dots,n$$

$$(2) \quad u_{ij}=(a_{ij}-\sum_{k=1}^{i-1} u_{ki} u_{kj})/u_{ii} \qquad j=1+i,2+i,\dots,n$$

Passons maintenant à la version par bloc de la factorisation de Cholesky:
On écrit d'abord la formule de factorisation recherchée sous une forme par bloc [29] :

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} U_{11}^T & \\ U_{12}^T & U_{22}^T \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ & U_{22} \end{pmatrix}$$

Si on fixe la taille du bloc à d, alors A_{11} et U_{11} sont de taille d×d, A_{12} et U_{12} de taille d×(n-d), et A_{22} et U_{22} de taille (n-d)×(n-d). Parmi ces sous-matrices, A_{11}, A_{22} sont carrées et A_{12} et U_{12} sont rectangulaires, tandis que U_{11}, U_{22} sont triangulaires supérieures. On déduit de la formule de factorisation par bloc les relations suivantes qui vont servir à retrouver les sous-matrices de U qu'on recherche, à savoir U_{11}, U_{12}, U_{22}:

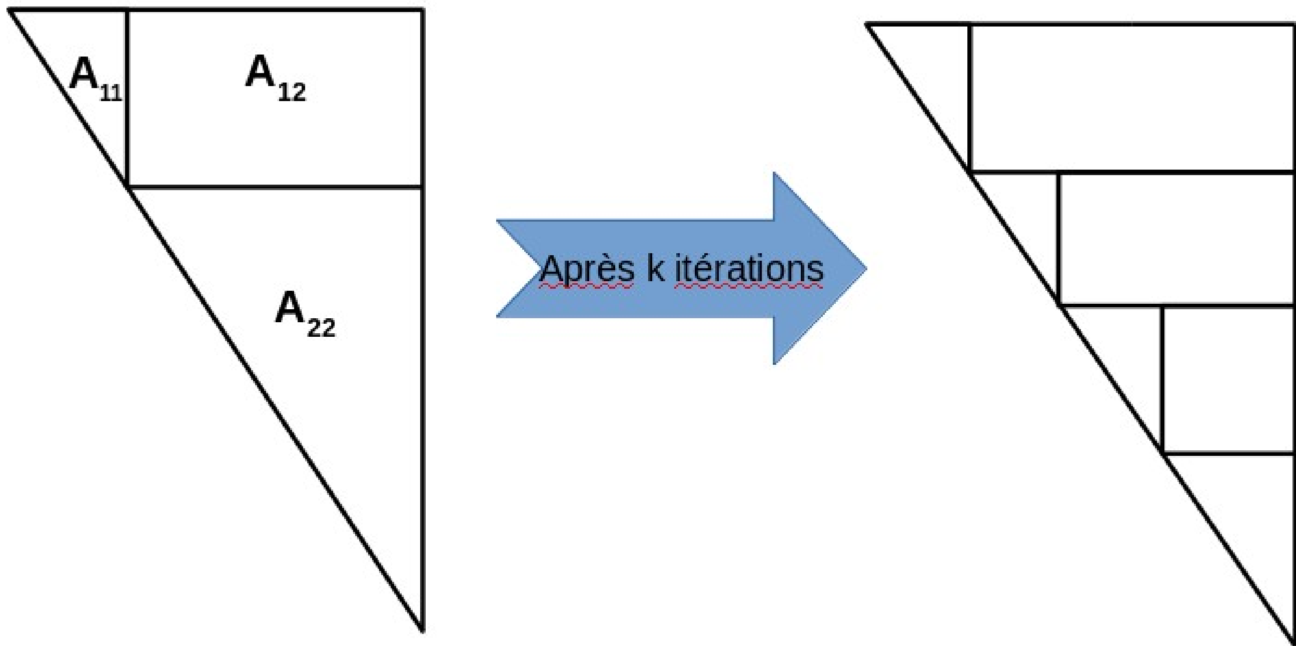
$$(3) \quad A_{11} = U_{11}^T U_{11}$$
$$(4) \quad A_{12} = U_{11}^T U_{12}$$
$$(5) \quad A_{22} = U_{12}^T U_{12} + U_{22}^T U_{22}$$

L'équation (3) est la factorisation de Cholesky de la sous-matrice A_{11}.
Elle peut être calculée directement avec les formules (1) et (2) d'une factorisation Cholesky standard.

Si on ajuste convenablement d la valeur de la taille du bloc, i.e. la taille de A_{11}, il est probable que ce calcul-là soit effectué au niveau du cache.
Une fois U_{11} calculée, U_{12} est retrouvée via la formule: $U_{12} = (U_{11}{}^T)^{-1} A_{12}$.
Il ne reste maintenant plus que U_{22} à calculer. Pour cela, on considère A_{22}' sur laquelle on va ré-appliquer le même procédé en la substituant à A : $A_{22}' = A_{22} - U_{12}{}^T U_{12}$ (6)
L'équation (5) devient: $A_{22}' = U_{22}{}^T U_{22}$ (7)
L'équation (6) indique une modification ou plutôt mise à jour de A_{22} par U_{12}.
La nouvelle sous-matrice A_{22} modifiée est appelée A_{22}'.
Généralement, elle est encore de grande taille et il faut ré-appliquer le même procédé par bloc dessus jusqu'à retrouver une sous-matrice A_{22}' de taille inférieure ou égale à d.

A ce moment-là, on peut se servir des formules standard (1) et (2) pour calculer la factorisation de Cholesky de la dernière sous-matrice A_{22}' .

La figure ci-dessous illustre le procédé par bloc pour une factorisation de Cholesky:



En résumé, l'algorithme par bloc pour une factorisation de Cholesky peut se présenter comme suit:

Factorisation de Cholesky par bloc
1. Partitionnement de la matrice A en blocs de sous-matrices A_{11} , A_{12} , A_{22} avec une taille de bloc d. 2. Factorisation de Cholesky de la sous-matrice $A_{11} = U_{11}^T U_{11}$, rendant U_{11}. 3. Calcul de l'inverse de U_{11}^T : $(U_{11}^T)^{-1}$. 4. Calcul de U_{12}: $U_{12} = (U_{11}^T)^{-1} A_{12}$. 5. Mise à jour de A_{22}: $A_{22}' = A_{22} - U_{12}^T U_{12}$. 6. Si la taille de A_{22}' est inférieure ou égale à d, factorisation de Cholesky de A_{22}' : $A_{22}' = U_{22}^T U_{22}$, rendant U_{22}. Sinon, reprise de l'étape 1· avec A_{22}' à la place de A.

ii) Factorisation QR

De même que la factorisation de Cholesky par bloc, la version par bloc de la factorisation QR comporte elle aussi plein d'opérations matrice-matrice [15,16].

En effet, étant donné une matrice A de taille M×N, on cherche la factorisation A = QR, où Q est une matrice M×M orthogonale, et R est une matrice M×N triangulaire

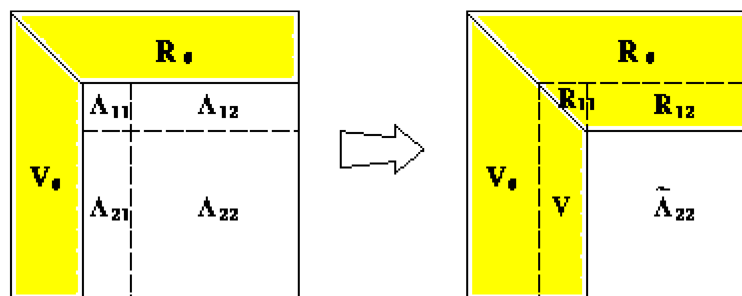
supérieure. A la k-ième étape du calcul, on partitionne la factorisation [30] à la sous-matrice m×n de $A^{(k)}$ comme suit: $A^{(k)} = (A_1 \quad A_2) = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = Q \cdot \begin{pmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{pmatrix}$

où le bloc A_{11} est $n_b \times n_b$, A_{12} est $n_b \times (n-n_b)$, A_{21} est $(m-n_b) \times n_b$, et A_{22} est $(m-n_b) \times (n-n_b)$.
A_1 est une matrice $m \times n_b$, contenant les n_b premières colonnes de la matrice $A^{(k)}$, et A_2 est une matrice $m \times (n-n_b)$, contenant les $(n-n_b)$ dernières colonnes de la matrice $A^{(k)}$.
R_{11} est une matrice triangulaire supérieure de taille $n_b \times n_b$.

Une factorisation QR est réalisée sur le premier panneau $m \times n_b$ de $A^{(k)}$ (i.e. A_1).
En pratique, Q est calculée en appliquant une série de transformations de Householder à A_1 de la forme $H_i = I - \tau_i v_i v_i^T$ où $i \in \{1,...,n_b\}$.
Le vecteur v_i est de taille m avec des valeurs nulles pour les (i-1) premières composantes et 1 pour le i-ème élément, $\tau_i = 2/(v_i^T v_i)$.
Durant la factorisation QR, le vecteur v_i écrase les éléments de A sous la diagonale, et τ_i est stocké dans un vecteur.
Par ailleurs, il est possible de démontrer que $Q = H_1 H_2 \ldots H_{nb} = I - V T V^T$, où T est une matrice $n_b \times n_b$ triangulaire supérieure et la i-ème colonne de V est égale à v_i.

L'équation par bloc donne: $\tilde{A}_2 = \begin{pmatrix} \tilde{A}_{12} \\ \tilde{A}_{22} \end{pmatrix} \leftarrow \begin{pmatrix} R_{12} \\ R_{22} \end{pmatrix} = Q^T A_2 = (I - VT^T V^T) A_2$

La figure suivante illustre le procédé de factorisation par bloc à l'étape k:



Pendant le calcul, la séquence des vecteurs de Householder V est calculée, la rangée de lignes R_{11} et R_{12}, et la matrice restante A_{22} .sont mises à jour. Par la suite, la factorisation se poursuit récursivement en appliquant les étapes mentionnées ce-dessus à la matrice $\tilde{A}_{22}$ de taille $(m-n_b) \times (n-n_b)$.

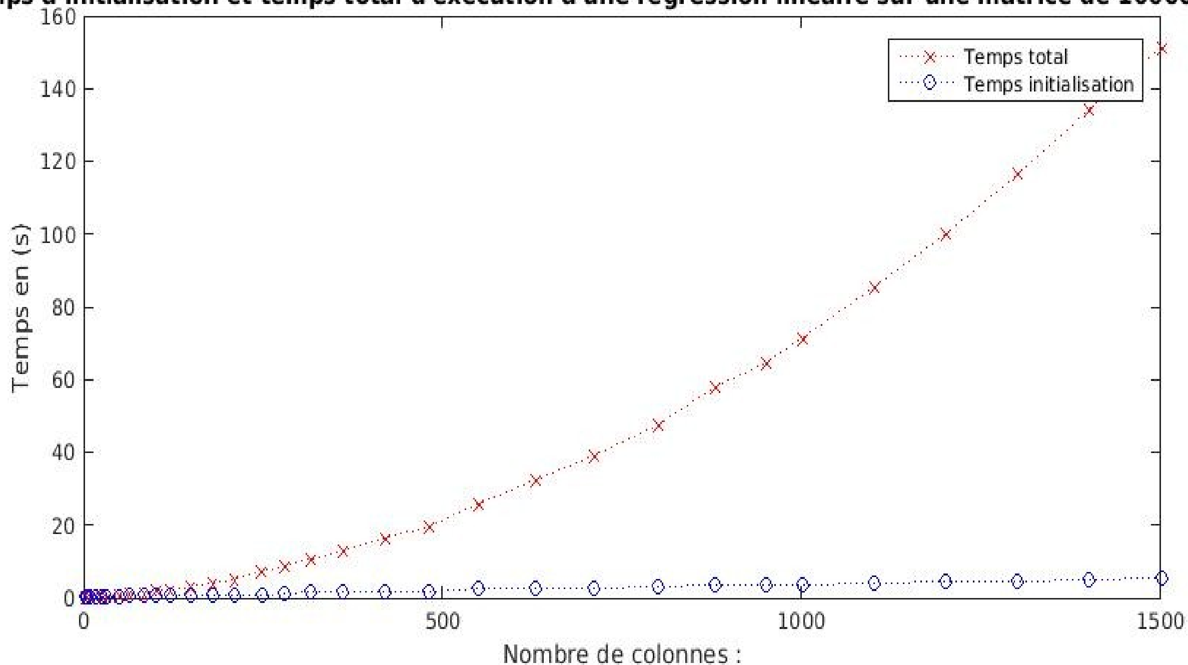
iii) Choix de la méthode QR:

Par soucis de stabilité numérique, on a préféré retenir la méthode QR pour la résolution du problèmes des moindres carrés, parce qu'elle est plus robuste en termes de précision et de stabilité inverse (les hypothèses requises pour appliquer QR sont moins contraignantes que celles pour résoudre les équations normales) [26,27].

Parallélisation :

Une fois le principe des algorithmes par bloc maîtrisé, on serait tenté de passer aux solveurs parallèles pour réduire le temps de calcul sur les grands problèmes. Il est à noter que ce gain en temps d'exécution ne sera réalisable que si la taille du problème à traiter dépasse un certain seuil, au dessous duquel il est conseillé de se contenter d'une résolution séquentielle. Cela s'explique par le fait que paralléliser un code nécessite d'abord un partage des tâches entre processus. Et qui dit partage des tâches, dit forcément partage des données. De plus, il faut aussi prendre en compte le coût des communications inter-processus.

La figure ci-dessous montre par exemple le temps total d'exécution en parallèle d'une régression linéaire et le temps d'initialisation/répartition d'une matrice aléatoire à nombre de lignes fixe, et nombre de colonnes variable, sur la grille de processus.



On remarque bien que plus la taille de la matrice augmente, plus le temps d'initialisation devient négligeable par rapport au temps total d'exécution.

Ainsi, il apparaît que faire tourner plusieurs processus en parallèle permet effectivement d'optimiser les performances de calcul, mais il faut d'abord maîtriser tous les aspects inhérents à ce procédé, comme la distribution de données ou la communication inter-processus.

a. Distribution de données

La façon dont les données sont réparties sur une hiérarchie mémoire d'un ordinateur a un impact majeur sur l'équilibrage de charge et des communications d'un algorithme concurrent. La distribution bloc-cyclique de données permet de s'affranchir des problèmes inhérents à la répartition des charges et au mouvement de données. Les algorithmes par bloc introduits précédemment, sont utilisés pour maximiser la performance locale au niveau des nœuds.

Puisque la décomposition de données détermine en grande partie la performance et la scalabilité d'un algorithme concurrent, de nombreux travaux de recherche [2,3,4,5] ont été consacrés aux différentes décompositions de données [6,7,8]. En particulier, la distribution bloc-cyclique de dimension 2 [9] a été proposée comme une possible décomposition basique d'usage général pour les bibliothèques d'algèbre linéaire dense parallèle [10,11,12,13] comme ScaLAPACK (cf III.4.a-).

La distribution bloc-cyclique est avantageuse de par sa scalibilité [14], son équilibrage de charges, et ses propriétés de communication [11]. Le calcul réparti par bloc s'opérera alors dans un ordre consécutif comme dans le cas d'un algorithme séquentiel conventionnel. Cette propriété essentielle de la distribution bloc-cyclique des données explique pourquoi le design de ScaLAPACK a été capable de réutiliser l'expertise numérique et logicielle de la bibliothèque séquentielle LAPACK.

Dans le cadre de l'algèbre linéaire, cette distribution particulière est sollicitée car elle offre une méthodologie simple et générale pour distribuer une matrice divisée par blocs sur des ordinateurs parallèles à mémoire distribuée.

Supposons qu'on ait M objets de données indexés par $m \in \{0,1,...,M-1\}$ et qu'on désire les répartir sur P processus. La distribution bloc-cyclique consiste alors à construire un mapping de l'indice global m au triplet d'indices (p,b,i) où:

- p désigne le processus auquel l'objet de données est assigné,
- b désigne le numéro de bloc dans le processus p qui contient l'objet de données,
- i indique l'emplacement local de l'objet de données dans le bloc b.

Par suite, si le nombre d'objets de données par bloc est égal à m_b et le nombre de

processus est P, la distribution bloc-cyclique de données peut s'écrire comme suit:

$$m \rightarrow \langle s \bmod P,\ \lfloor \frac{s}{P} \rfloor,\ m \bmod m_b \rangle \qquad où \qquad s = \lfloor m/m_b \rfloor$$

La distribution d'une matrice de M × N éléments peut être vue comme un produit tensoriel de deux mappings: Un qui distribue les lignes de la matrice sur P processus, et un autre qui distribue les colonnes sur Q processus.C'est-à-dire que l'élément d'indices globaux (m,n) ∈ {0,1,...,M-1} × {0,1,...,N-1} est mappé à :

$$(m,n) \rightarrow \langle\ (p,q),\ (b,d),\ (i,j)\ \rangle$$

Ce mapping nous dit que l'élément d'indice global (m,n) dans la matrice est mappé au processus (p,q) où il est stocké dans le bloc situé à l'emplacement (b,d) d'un tableau de blocs. Enfin, à l'intérieur de ce bloc-là, l'élément est stocké à l'emplacement (i,j).
La distribution bloc-cyclique de dimension 2 est ainsi paramétrée par les 4 nombres suivants: P, Q, m_b, n_b, où P × Q est la grille de processus et $m_b \times n_b$ la taille des blocs.
Les blocs distants les uns des autres par un certain pas fixe dans les directions des lignes et des colonnes sont assignés au même processus.

↳ Considérons l'exemple d'application suivant:
Nous disposons d'une matrice de M = 9 lignes et N = 9 colonnes, qu'on souhaite répartir sur une grille de processus (P = 2) × (Q = 3), en prenant des blocs carrés de taille (mb = 2) × (nb = 2). La matrice est d'abord décomposée en blocs mb × nb en commençant par le coin supérieur gauche:

a_{11} a_{12}	a_{13} a_{14}	a_{15} a_{16}	a_{17} a_{18}	a_{19}
a_{21} a_{22}	a_{23} a_{24}	a_{25} a_{26}	a_{27} a_{28}	a_{29}
a_{31} a_{32}	a_{33} a_{34}	a_{35} a_{36}	a_{37} a_{38}	a_{39}
a_{41} a_{42}	a_{43} a_{44}	a_{45} a_{46}	a_{47} a_{48}	a_{49}
a_{51} a_{52}	a_{53} a_{54}	a_{55} a_{56}	a_{57} a_{58}	a_{59}
a_{61} a_{62}	a_{63} a_{64}	a_{65} a_{66}	a_{67} a_{68}	a_{69}
a_{71} a_{72}	a_{73} a_{74}	a_{75} a_{76}	a_{77} a_{78}	a_{79}
a_{81} a_{82}	a_{83} a_{84}	a_{85} a_{86}	a_{87} a_{88}	a_{89}
a_{91} a_{92}	a_{93} a_{94}	a_{95} a_{96}	a_{97} a_{98}	a_{99}

9 x 9 matrix partitioned in 2 x 2 blocks

Ces blocs sont ensuite distribués uniformément sur la grille de processus. Ainsi, chaque processus possède une collection de blocs, qui sont localement stockés de manière contiguë dans un tableau à deux-dimensions:

	0	0	1	1	2
0	a_{11} a_{12}	a_{17} a_{18}	a_{13} a_{14}	a_{19}	a_{15} a_{16}
0	a_{21} a_{22}	a_{27} a_{28}	a_{23} a_{24}	a_{29}	a_{25} a_{26}
0	a_{51} a_{52}	a_{57} a_{58}	a_{53} a_{54}	a_{59}	a_{55} a_{56}
0	a_{61} a_{62}	a_{67} a_{68}	a_{63} a_{64}	a_{69}	a_{65} a_{66}
0	a_{91} a_{92}	a_{97} a_{98}	a_{93} a_{94}	a_{99}	a_{95} a_{96}
1	a_{31} a_{32}	a_{37} a_{38}	a_{33} a_{34}	a_{39}	a_{35} a_{36}
1	a_{41} a_{42}	a_{47} a_{48}	a_{43} a_{44}	a_{49}	a_{45} a_{46}
1	a_{71} a_{72}	a_{77} a_{78}	a_{73} a_{74}	a_{79}	a_{75} a_{76}
1	a_{81} a_{82}	a_{87} a_{88}	a_{83} a_{84}	a_{89}	a_{85} a_{86}

2 x 3 process grid point of view

Remarque:

La distribution bloc-cyclique de données peut reproduire la plupart des distributions utilisées dans les calculs d'algèbre linéaire.

Par exemple, la distribution simple par bloc est un cas spécial de la distribution bloc-cyclique dans lequel la taille de bloc est donnée par $m_b = \lceil M/P \rceil$; $n_b = \lceil N/Q \rceil$.

C'est-à-dire : $(m,n) \rightarrow \langle \ (\lfloor \frac{m}{m_b} \rfloor, \lfloor \frac{n}{n_b} \rfloor),\ (0,0),\ (m \bmod m_b, n \bmod n_b) \ \rangle$

De même, une décomposition purement répartie est un autre cas spécial dans lequel on considère des blocs unitaires: mb = nb = 1,

$$(m,n) \rightarrow \langle \ (m \bmod P, n \bmod Q), \ (\left\lfloor \frac{m}{P} \right\rfloor, \left\lfloor \frac{n}{Q} \right\rfloor), \ (0,0) \ \rangle$$

b. *Implémentation parallèle des algorithmes par bloc*

Dans cette partie, nous focaliserons notre attention sur la factorisation QR et détaillerons l'implémentation parallèle de la version par bloc de cet algorithme, notamment en faisant le point sur les multiples stades de calcul en précisant les fonctions/routines appelées de LAPACK, à la fois en mode séquentiel, et ScaLAPACK mode parallèle.

À propos:

LAPACK (pour **Linear Algebra PACKage**) est une bibliothèque logicielle écrite en Fortran, dédiée comme son nom l'indique à l'algèbre linéaire numérique. Cette bibliothèque fournit notamment des fonctions pour la résolution de systèmes d'équations linéaires, le calcul de valeurs propres et les décompositions de matrices.

Factorisation QR en séquentiel:
Les calculs effectués lors des étapes décrites en 2. c- ii) de la routine LAPACK, DGEQRF, comporte les opérations suivantes:

- DGEQR2: Calcule la factorisation QR de la sous-matrice A_1 formée par les n_b premières colonnes de $A^{(k)}$
  - [Répéter n_b fois ($i=1,...,n_b$)]
    - DLARFG: Génère le réflecteur élémentaire v_i et τ_i
    - DLARF: Met à jour la matrice restante $\tilde{A}_{22} \leftarrow H_i^T A_1 = (I - \tau_i v_i v_i^T) A_1$
- DLARFT: Calcule le facteur triangulaire T du réflecteur de bloc Q.
- DLARFB: Applique Q^T au reste de la matrice à gauche.

  $$\tilde{A}_2 \leftarrow \begin{pmatrix} R_{12} \\ R_{22} \end{pmatrix} = Q^T A_2 = (I - V T^T V^T) A_2$$

  - DGEMM: $W \leftarrow V^T A_2$
  - DTRMM: $W \leftarrow T^T W$
  - DGEMM: $\tilde{A}_2 \leftarrow \begin{pmatrix} R_{12} \\ R_{22} \end{pmatrix} = A_2 - VW$

Factorisation QR en parallèle:
Les étapes correspondantes en mode parallèle, de la routine PDGEQRF de ScaLAPACK (cf. III.4.a-), sont comme suit:

- PDGEQR2: La colonne courante de processus réalise une factorisation QR sur la sous-matrice A_1 formée par les n_b premières colonnes de $A^{(k)}$

- [Répéter n_b fois ($i=1,...,n_b$)]
  - PDLARFG: Génère le réflecteur élémentaire v_i et τ_i
  - PDLARF: Met à jour la matrice restante
- PDLARFT: La colonne courante de processus, qui a une séquence des vecteurs de Householder V, calcule T seulement dans la ligne courante de processus.
- PDLARFB: Applique Q^T au reste de la matrice (à gauche)
  - PDGEMM: V est diffusée horizontalement à travers toutes les colonnes de processus. La transposée de V est multipliée localement par A_2, ensuite les produits sont ajoutés à la ligne courante de processus ($W \leftarrow V^T A_2$).
  - PDTRMM: T est diffusée horizontalement dans la ligne courante de processus à toutes les colonnes de processus et est multipliée par la somme ($W \leftarrow T^T W$).
  - PDGEMM: W est diffusée verticalement à toutes les lignes de processus. Désormais, les processus ont leurs propres portions de V et W, et mettent à jour les portions locales de la matrice A_2 ($\tilde{A}_2 \leftarrow \begin{pmatrix} \tilde{R}_{12} \\ \tilde{R}_{22} \end{pmatrix} = A_2 - VW$)

A priori, on peut constater que les étapes de calcul sont à peu près les mêmes entre les routines séquentielles de LAPACK et les routines parallèles implémentées sous ScaLAPACK(cf. III.4.a-). Cependant, la configuration du mode de calcul parallèle est un peu plus délicate à mettre en place.

Pour garantir un fonctionnement correct de ces routines, comme toute autre routine ScaLAPACK [32], il faut suivre les 4 étapes de base suivantes:

1. Initialiser la grille de processus
2. Distribuer la matrice sur la grille de processus
3. Appeler la routine désirée
4. Libérer la grille de processus

Initialiser la grille de processus:

Cette opération est réalisée par un appel à la routine SL_INIT qui initialise une grille P×Q de processus en utilisant un ordonnancement en lignes des processus, et on obtient par un contexte système par défaut. Ensuite, l'utilisateur peut demander à la grille d'identifier les coordonnées de chaque processus via un appel à la routine BLACS_GRIDINFO.

A propos des contextes ScaLAPACK:

Dans ScaLAPACK, chaque grille de processus est jointe à un contexte. De même, un contexte est associé à chaque matrice globale dans ScaLAPACK. L'utilisation de

contextes permet d'avoir des «univers» séparés pour les passages de messages. Autrement dit, des processus d'une même grille peuvent communiquer en toute sécurité même si d'autres grilles de processus sont en train de communiquer.

Distribuer la matrice sur la grille de processus:

Toutes les matrices globales doivent être distribuées sur la grille de processus avant tout appel à une routine de calcul ScaLAPACK, et il incombe à l'utilisateur d'effectuer cette distribution de données.

A chaque matrice globale à répartir sur une grille de processus doit être attribué un descripteur de tableau qui est initialisé via un appel à la routine DESCINIT.

Ce descripteur est indispensable à la phase de répartition de la matrice puisqu'il stocke les informations requises pour établir un mapping entre chaque coefficient d'une matrice globale d'un côté, et de l'autre, son emplacement mémoire et le processus correspondant qui le gère.

Pour simplifier, on peut assimiler le descripteur à un vecteur d'entiers qui représentent les attributs de la matrice globale (type, contexte système, nombre de lignes, nombre de colonnes, nombre de lignes par bloc, nombre de colonnes par bloc, coordonnées du processus qui gère la première ligne de la matrice, coordonnées du processus qui gère la première colonne de la matrice, dimension maximale des matrices locales au niveau de chaque processus, etc...).

Il faut souligner tout de même que le contenu du descripteur varie en fonction de la forme de la matrice (matrice tridiagonale ou matrice bande), du type de matrice (dense ou creuse) , ou encore du mode de stockage de données (in-core ou out-of-core) et peut contenir des attributs supplémentaires (autres que ceux mentionnés précédemment) notamment dans le cas out-of-core (cf. III.4.c-).

Appeler la routine désirée:

Toute routine ScaLAPACK suppose que les données ont été distribuées avant l'appel à celle-ci. Beaucoup de routines ScaLAPACK nécessite un ou plusieurs tableaux de travail et la quantité d'espace mémoire de travail à passer comme arguments pour y stocker les valeurs temporaires lors du calcul par exemple. Il est conseillé dans ce cas, de toujours procéder à deux appels de la routine: un premier appel avec une valeur de -1 pour la quantité d'espace mémoire de travail, ainsi la valeur correcte sera automatiquement calculée par la routine et retournée. C'est cette valeur qu'il faut alors saisir en entrée pour le deuxième appel.

Libérer la grille de processus

Après que le calcul soit achevé sur une grille de processus, il est recommandé de libérer ces processus via un appel à BLACS_GRIDEXIT. Une fois toutes les opérations de calcul terminées et avant de sortir du programme principal, il ne faut pas oublier de faire appel à BLACS_EXIT.

Haute performance et données de grande dimension :

a. Introduction de ScaLAPACK

ScaLAPACK est une bibliothèque de routines haute-performance d'algèbre linéaire pour les machines parallèles à mémoires distribuée. ScaLAPACK résout des systèmes linéaires denses, ou creux à matrice diagonale par blocs, des problèmes aux moindres carrés, des problèmes de valeurs propres ou de valeurs singulières. Cette bibliothèque est principalement codée en Fortran (à l'exception de quelques routines qui sont codée en C). Le nom de la bibliothèque ScaLAPACK est un acronyme pour Scalable Linear Algebra PACKage, ou Scalable LAPACK [www.netlib.org/scalapack].

Les idées clés intégrées à ScaLAPACK incluent l'utilisation de:

(1) **une distribution bloc-cyclique des données** pour les matrices denses et une distribution des données par bloc pour des matrices à bandes, paramétrable à l'exécution;

(2) **des algorithmes par bloc** pour assurer une réutilisation maximale de données;

(3) **des composants modulaires de bas niveau bien conçus** pour qui simplifient la tâche de parallélisation des routines de haut niveau en rendant leur code source très proche de celui du cas séquentiel.

Les objectifs du projet ScaLAPACK sont les mêmes que ceux de LAPACK, à savoir:

✔ Efficacité (faire exécuter aussi rapidement que possible),

✔ Scalabilité (à mesure que la taille des problèmes et le nombre de processeurs croissent),

✔ Fiabilité (notamment les bornes d'erreur),

✔ Portabilité (entre toutes les machines parallèles),

✔ Flexibilité (de manière à ce que les utilisateurs puissent développer de nouvelles routines à partir des parties bien conçues),

✔ et Facilité de manipulation (en rendant l'interface de LAPACK et ScaLAPACK très similaires l'une de l'autre).

La plupart de ces objectifs, en particulier la portabilité, sont facilités par la mise au point et promotion de standards, surtout pour les routines de calcul et de communication de

bas niveau. L'utilisation de ScaLAPACK repose sur trois bibliothèques standards appelées BLAS pour Basic Linear Algebra Subprograms, LAPACK et BLACS pour Basic Linear Algebra Communication Subprograms. Ainsi, LAPACK tournera sur des machines où BLAS est disponible, et ScaLAPACK tournera sur des machines où BLAS, LAPACK et BLACS sont disponibles.

b. Les méthodes de factorisation out-of-core et leurs avantages

Concept général:

Les calculs numériques des problèmes d'algèbre linéaire, y compris dans notre cas d'étude la factorisation de QR, jouent des rôles très importants dans de nombreuses recherches scientifiques et applications d'ingénierie. Les ordinateurs de nos jours sont dotés d'au moins deux niveaux de caches entre un processeur central CPU à haute vitesse et une mémoire à vitesse relativement réduite [https://lwn.net/Articles/252125/].

Les caches opèrent à une vitesse proche de celle du CPU. Si les données qui sont en cours de calcul, ou fréquemment utilisées, sont placées dans les caches, le temps d'accès à ces dernières peut être significativement réduit. Par conséquent, l'efficacité globale du calcul va énormément augmenter.

Cela dit, la capacité des mémoires caches est nettement plus petite. Si la quantité de données en cours de calcul est très grande, ce qui est le cas par exemple pour une factorisation de Cholesky pour une gigantesque matrice, il est impossible de charger toutes les données dans le cache. Dans ce cas, le cache ne peut être bien exploité et le calcul est moins efficace.

Il est vrai que les algorithmes utilisés dans LAPACK ont été développées et testés sur des machines datant maintenant de plus de 10 ans, un temps où les ordinateurs avaient de petites mémoires caches, de 64Ko ou 128Ko, mais même avec les ordinateurs récents, y compris les micro-ordinateurs, qui ont des mémoires caches de d'1Mo avec des mémoires vives de l'ordre de quelques gigaoctets, on tombe toujours sur des matrices volumineuses qu'on n'arrive pas à traiter (Voir 2ème chapitre de la partie Applications et Résultats Expérimentaux).

Dans le domaine médical par exemple, les bases de données de patients dépassent facilement les 1Go de taille mémoire. Un constat valable dans plusieurs autres domaines, et qui est accentué par le phénomène du «big data» et l’avènement du «tout numérique» où toute interaction est numérisée, et toute information est numérisable.

Pour remédier à ce problème d'insuffisance de mémoire cache et mémoire vive, on a pensé aux méthodes out-of-core de calcul: la grosse matrice est partitionnée par blocs de sous-matrices de petites tailles. Les calculs sur ces sous-matrices, qui sont bien sûr plus petites, sont plus susceptibles de réussir à l'intérieur des caches. Et on retombe un peu

sur le même concept que les algorithmes par blocs: Traiter une partie de la matrice, puis mettre à jour le reste, et itérer ce procédé autant de fois qu'il y a de parties.

Concrètement, la matrice complète de données est stockée sur disque et les routines de factorisation transfèrent seulement des parties de la matrice à la mémoire vive à l'opposé de la méthode in-core, où l'intégralité de la matrice est passée à la mémoire vive.

En somme, l'intérêt principal de l'utilisation des méthodes out-of-core est de résoudre des problèmes avec de très grandes matrices denses dont les dimensions dépassent de loin la taille de mémoire (RAM) disponible. Mais ce n'est pas tout!

Une autre raison de plus à adopter les méthodes out-of-core, est la différence de coûts de mémoire: En effet, les disques de stockage externes sont moins chers que les mémoires RAM. Pour preuve, un disque dur externe de 250 Go coûte à peu près aussi cher qu'une une mémoire RAM de 4 Go, soit un montant avoisinant les 35 €. Ce qui représente un facteur de gain de 250 / 4 = 62.5 .

En définitive, il semble judicieux de vérifier l'efficacité de la solution out-of-core avec des tests sur des matrices denses volumineuses, ce que l'on traitera dans la partie Applications et Résultats Expérimentaux.

Disposition des données sur disque:

Sur ScaLAPACK, les méthodes out-of-core ont été implémentées en utilisant une interface de mémoire Entrées/Sorties portable et font appel à des routines de factorisation haute-performance de ScaLAPACK comme des noyaux de calcul in-core.

Pour exécuter les méthodes out-of-core de ScaLAPACK, il faut d'abord réussir la répartition de données et leur stockage sur disque en mode de fichier distribué. En théorie, dans le cas général, il faudrait une subroutine qui puisse récupérer une matrice de départ, la découper en blocs, et répartir le tout sur P×Q fichiers sur disque, soit un fichier pour chaque processus si l'on suppose que l'on est dans une configuration distribuée et qu'on dispose d'une grille de (P lignes de processus × Q colonnes de processus) qui vont exécuter les calculs.

Pour ce qui est de la distribution de données, elle est bloc-cyclique pour faire en sorte que les processus aient des charges de travail équilibrées et pour éviter que deux fichiers distincts ne stockent la même partie de la matrice.

Chaque matrice out-of-core est associée à un numéro d'unité ou unité d'appel (compris entre 1 et 99) , tout comme le système d'entrées/sorties de Fortran. Au plus bas niveau d'architecture, chaque opération E/S [Entrée/Sortie] est basée sur des enregistrements. Sur le plan conceptuel, un enregistrement correspond à une matrice ScaLAPACK bloc-cycliquement distribuée de taille MMB×NNB. Par ailleurs, si cette matrice est distribuée sur une grille de P×Q processus, avec comme (MB,NB) comme de taille de blocs, alors ***mod(MMB, P*MB) = mod(NNB, Q*NB) = 0***. C'est-à-dire que les nombres de lignes et de colonnes des enregistrements sont des multiples exacts de P*MB et Q*NB respectivement. Les données à transférer sont d'abord copiées ou assemblées dans un buffer interne temporaire (enregistrement). Cet agencement encourage les transferts de gros blocs contigus. Tous les processus sont impliqués dans chaque opération E/S sur l'enregistrement. Individuellement, chaque processus écrit un bloc de matrice de taille (MMB/P)×(NNB/Q). MMB et NNB peuvent être ajustés pour atteindre une bonne performance E/S avec des transferts de gros blocs contigus ou pour concorder avec la taille d'une bande de blocs disque.

Les E/S supportent une configuration «partagée» ou «distribuée» de la structure du disque. Mais on favorisera plutôt la deuxième option, car elle permet à chaque processeur d'ouvrir un fichier unique sur son disque local et de l'associer à la matrice.

c. *Aspects techniques de programmation out-of-core:*

Comme mentionné dans III.3.b- , la bibliothèque ScaLAPACK fournit des routines pour la résolution out-of-core de systèmes linéaires, dans laquelle les matrices sont sauvegardées sur disque. Un descripteur de tableau particulier est requis pour spécifier un tel stockage de données.

Par rapport au descripteur standard décrit précédemment, le descripteur de tableau pour les matrices out-of-core a des champs en plus pour stocker les paramètres des fichiers associés à la matrice, comme le numéro d'unité E/S, le mode de configuration E/S, l'espace disponible sur le buffer temporaire, le chemin d'accès des fichiers stockant les éléments de la matrice, et les paramètres MMB et NNB.

Quelques astuces pratiques pour le réglage des paramètres:

Une fois le descripteur bien initialisé, il reste tout de même à l'utilisateur de s'assurer que sa machine de calcul utilise un nombre approprié de processeurs et que la matrice est efficacement distribuée. Voici une liste de règles empiriques [32] à respecter:

+ Si l'architecture locale (Nombre de processeurs, nombre de cœurs cpu, etc...) de la station de travail le permet, prendre un nombre de processus:

> nb_proc = $M*N / 10^6$ où M, N sont respectivement le nombre de lignes et le nombre de colonnes de la matrice. Dans ce cas, cela donnerait approximativement une matrice locale de taille 1000×1000 au niveau de chaque processus.

+ Ne pas essayer de résoudre un problème de petite taille sur beaucoup de processeurs.

+ Ne pas dépasser la mémoire physique.

+ Faire en sorte d'avoir une distribution efficace (tailles de blocs optimales : 64 ou 32).

+ Pour les problèmes de moindres carrés, si $M > N$ alors il faudrait prendre $P << Q$:

> Idéalement, si P = 1 et Q = nb_proc , chaque colonne de la matrice sera contenue dans un seul processeur, et ce dernier n'aura pas à communiquer avec les autres processeurs pour accéder aux éléments de cette colonne, voire bloc de colonnes. On profiterait ainsi du column major ordering de Fortran où les éléments consécutifs des colonnes sont contigus en mémoire.

+ Pour les autres problèmes où le nombre de processeurs utilisables est supérieur à 9, il vaudrait mieux privilégier des grilles carrées de processus [32].

IV. Applications et résultats expérimentaux

Type de matrices étudiées:

Pour les tests de résolution de systèmes linéaires issus des problèmes aux moindres carrés, nous avons repris le même type de matrices que les médecins et chercheurs ont à étudier dans le domaine médical, et plus particulièrement celui de l'épidémiologie, à savoir des matrices rectangulaires (sous l'hypothèse nombre de lignes ≥ nombre de colonnes) ou même des fois des matrice rectangulaires allongées (nombre de lignes ≥ carré du nombre de colonnes). En effet, l'épidémiologie consiste à étudier les facteurs influant sur la santé et les maladies de population. Si on s'intéresse par exemple à une forme de fièvre, le vecteur Y second membre de notre système regroupera les températures observées chez une population. Quant à notre notre matrice principale X, on pourra y recenser les données statistiques des potentiels vecteurs principaux de cette maladie (qualité de l'eau, de l'air, du sol, bio-concentration de polluants, de toxines, etc...) . Ici, chaque ligne de la matrice X ou du vecteur Y correspondrait aux données relevées sur un seul individus.

La crédibilité des études statistiques dépendant fortement du nombre de sujets inclus dans les essais, il est nécessaire dans la plupart des cas d'inclure un grand nombre de patients. Voilà pourquoi, on tombe sur des matrices rectangulaires allongées suivant la verticale: Il y a plus d'individus participant à l'étude que de vecteurs principaux.

Tests de régression linéaire sur Matlab & R :

Ces tests ont été effectués sur 'RAVEL' une machine Intel Xeon CPU E5630 dotée de 4 cœurs à 2.53Ghz, avec 5.8Go de mémoire et un disque dur de 301.2 Go. Et il a été procédé de la manière suivante:

Une matrice aléatoire X de taille m×n est générée puis stockée en mode double précision, avec m et n qui varient suivant des puissances de 10: $m=10^i \ ; \ n=\lfloor\sqrt{10^j}\rfloor$

où i∈N ; j∈{1,2,...,2*i}. Ici la notation $\lfloor x \rfloor$ désigne la partie entière de x.

Le second membre utilisé est le vecteur colonne Y qui est pris égal à la somme des colonnes de X pondérées par leurs rangs: $Y=\sum_{k=1}^{n} k*X_k \ ; \ si \ \ X=[X_1,\ldots,X_n]$

Le vecteur B_{exp} obtenu expérimentalement via une facto/résolution QR, est ensuite

comparé à la solution théorique de ce problème de moindres carrés qui vaut: $B_{th} = [1\ 2\ ...\ n]^T$. Une erreur est ainsi calculée comme étant la norme euclidienne de la différence et permet de valider le résultat expérimental: Err = ||X*B – Y|| . Le temps d'exécution est aussi enregistré.

Les tableaux ci-dessous contiennent les temps d'exécution et les erreurs calculés pour chaque (i,j) et les cases grisées indiquent des cas non traités (nombre de lignes < nombre de colonnes) ou des cas pour lesquels les calculs n'aboutissent pas par insuffisance de mémoire.

Résultats expérimentaux obtenus avec Matlab:

	j = 1	j = 2	j = 3	j = 4	j = 5	j = 6	j = 7	j = 8
i=1	T = ε E = ε	T = ε E =3e-29						
i=2	T = ε E = ε	T = ε E = 4.00e-29	T = ε E = 5.03e-27	T = 0.005 E = 2.27e-24				
i=3	T = ε E = ε	T = ε E = 2.30e-28	T = ε E = 9.82e-27	T = 0.02 E = 2.29e-24	T = 0.09 E = 4.96e-22	T = 0.475 E = 1.14e-19		
i=4	T = ε E = 4e-29	T = ε E = 1.58e-27	T = 0.02 E = 5.30e-26	T = 0.155 E = 4.80e-24	T = 0.87 E = 1.22e-21	T = 5.945 E = 1.17e-19	T=48.155 E = 3.16e-17	T = 310.9 E = 7.05e-15
i=5	T = ε E = 3.90e-28	T = 0.03 E = 9.25e-27	T = 0.22 E = 6.03e-25	T = 1.725 E = 7.32e-23	T = 8.695 E = 8.49e-21	T=60.285 E = 1.50e-18	T=496.61 E = 9.96e-17	
i=6	T = 0.065 E = 2.72e-27	T = 0.41 E = 2.33e-25	T = 3.42 E = 8.15e-24	T = 23.33 E = 6.21e-22	T=102.68 E = 9.83e-20			
i=7	T = 0.62 E =	T = 3.99 E =	T = 34.57 E =					

	4.28e-26	1.21e-24	8.08e-23					
i=8	T = 6.72 E = 2e-27							

Résultats expérimentaux obtenus avec R:

	j = 1	j = 2	j = 3	j = 4	j = 5	j = 6	j = 7	j = 8
i=1	T = 0.002 E = 9.86e-31	T = 0.002 E = 7.61e-28						
i=2	T = 0.001 E = 3.94e-31	T = 0.002 E = 5.95e-28	T = 0.002 E = 3.44e-26	T = 0.003 E = 4.69e-22				
i=3	T = 0.003 E = 3.85e-29	T = 0.003 E = 8.08e-27	T = 0.006 E = 2.46e-25	T = 0.02 E = 2.29e-23	T = 0.122 E = 1.26e-20	T = 0.764 E = 3.105e-17		
i=4	T = 0.012 E = 1.106e-27	T = 0.018 E = 2.63e-26	T = 0.045 E = 2.014e-24	T = 0.191 E = 3.59e-22	T = 1.584 E = 4.757e-20	T=11.844 E = 4.91e-18	T=103.66 E = 4.47e-16	T = 764.9 E = 4.52e-11
i=5	T = 0.137 E = 7.99e-27	T = 0.281 E = 2.37e-25	T = 0.737 E = 5.78e-23	T = 2.316 E = 1.25e-21	T=15.008 E = 1.03e-19	T=131.99 E = 2.26e-17		
i=6	T = 1.358 E = 9.93e-26	T = 1.935 E = 1.94e-24	T = 5.469 E = 6.69e-23	T = 26.69 E = 4.28e-20				
i=7	T=13.279 E = 2.28e-25	T=88.569 E = 4.19e-23						
i=8								

Régression linéaire en in-core avec ScaLAPACK

Avant de commencer les tests in-core, j'ai téléchargé des exemples de programmes ScaLAPACK (exemple1.f et exemple2.f) sur le site officiel (http://www.netlib.org/scalapack) que j'ai compilés et exécutés sur mon poste de travail, question de me familiariser un peu avec la bibliothèque ScaLAPACK.

Ensuite, j'ai cherché s'il y avait dans la documentation ScaLAPACK des routines pour faire de la régression linéaire. Je suis tombé sur la routine PDGELS qui est une routine en in-core parallèle, qui résout un problème aux moindres carrés associé à une matrice réelle de rang plein, mais pas nécessairement carrée. Pour cela, la routine calcule la factorisation QR de cette matrice.

D'autre part, en recherchant sur internet [http://www.netlib.org/scalapack/examples/] , j'ai trouvé un programme de test qui consiste à générer une matrice aléatoire A de taille $m \times n$, et un vecteur colonne x aléatoire de taille n. Puis, dans le problème aux moindres carrés que va résoudre le programme, le second membre b est le vecteur colonne de taille m, issu de la multiplication de A par x. Notre solution théorique correspond donc bien au vecteur x généré plus tôt, on peut alors valider le résultat expérimental si le taux d'erreur est inférieur à un seuil pré-déterminé.

J'ai lancé ce programme sur le supercalculateur Neptune du CERFACS mais je me suis vite rendu compte qu'il y avait quelques améliorations à apporter vu que pour une matrice de taille $m = 10^5 \times 10^2$ le temps d'exécution du programme dépassait le temps limite de calcul sur le supercalculateur Neptune qui est de 6 heures. En consultant les temps consacrés à chaque étape du programme, j'ai découvert que c'était l'étape de répartition de la matrice sur la grille de processus qui représentait plus de 95% du temps total d'exécution du programme, ce qui paraissait anormal. En regardant comment cette partie était codée, j'ai constaté que la distribution des données sur les processus n'était pas optimisée en terme de parallélisation.

Ce qui ralentissait en fait l'exécution du code, c'était la pléthore de boucles IF qu'il contenait. En effet, on parcourait un par un les éléments de la matrice (jusqu'ici nulle) avec une double boucle do sur les indices de ligne et de colonne, et pour l'élément (i,j) pointé, il y avait une condition IF qui permettait exclusivement au processus indexé par les indices de grille (myrow, mycol) correspondants de générer la valeur de cet élément

et de la stocker dans sa matrice locale composée de tous les blocs de la matrice qu'il doit gérer. Comme on l'a vu avec la distribution bloc-cyclique des données, il y a un mapping qui lie chaque couple d'indices (i,j) de la matrice à un couple d'indice (myrow, mycol) de la grille de processus. Et c'est ce mapping-là qui est utilisé ici.

En résumé, même si le code devait tourner en parallèle, ces boucles IF représentaient un véritable frein pour les performances, vu que tous les processus parcourent l'intégralité de la matrice, la quasi-totalité d'entre eux le font inutilement, car à chaque élément parcouru seul un processus travaille.

Optimisation de la distribution de données in-core sur ScaLAPACK

Première amélioration:

Par la suite, la première amélioration que j'ai pu apporter à cette version du code était de faire en sorte que la répartition de la matrice se passe vraiment en parallèle avec les processus qui travaillent simultanément et de façon indépendante. Pour cela, je suis parti du principe selon lequel, une fois que tous les processus ont connaissance de leurs blocs respectifs dans la matrice, ils pourront y accéder directement sans que les uns soient "dérangés" par les autres profitant du fait que chaque bloc de la matrice globale est mappé à un seul processus. Pour simplifier l'opération, nous supposons dans un premier temps avoir affaire à une matrice à une seule colonne par exemple et reprenons la notation MB pour la taille de bloc. Chaque bloc colonne est donc composé de MB éléments.

Par conséquent, il suffisait de créer des vecteurs globaux (qu'on a appelés nbloc, ideb, ifin) , accessibles par tous les processus, contenant respectivement le nombre de blocs mappés à chaque processus, les indices globaux de début et de fin de chacun de ces blocs. Ces paramètres sont calculables une fois la taille de la matrice connue, par exemple les indices de début forment une suite arithmétique de raison égale à MB.

Ensuite, il est aussi possible de calculer pour chaque élément d'indice global i dans la matrice colonne, son emplacement local ipos pour le processus auquel il est mappé. C'est cet indice ipos que le processus pourra manipuler pour modifier l’élément global.

Ce principe est aussi valable dans le cas général pour des tableaux de matrices à deux dimensions. Il suffit de l'appliquer sur les lignes et sur les colonnes de la matrice.

Cette première solution nous déleste du fardeau des boucles IF et maintenant les processus n'ont plus à parcourir toute la matrice, mais juste les blocs qui leur sont mappés.

Deuxième amélioration:

L'autre retouche concerne la génération des matrices locales au niveau des processus. Si avant, cette opération se faisait élément par élément, il est tout-à-fait possible de la faire bloc par bloc. Cela fera gagnerait du temps lors de l'exécution.

Pour un bloc de la matrice globale, cela revient à regrouper en une seule boucle "Pour", le calcul des indices locaux de ses éléments, puis à la fin de la boucle do, un simple appel à une subroutine fera la génération de tout le bloc au niveau de la matrice locale au processus.

Pour les matrices générées aléatoirement par exemple, cela revient à faire un seul appel à une subroutine qui génère un bloc de MB×NB nombres aléatoires au lieu de faire MB×NB appels individuels à RANDOM_NUMBER pour chaque élément.

Idem pour les matrices quelconques en entrée, il est mieux d'avoir une subroutine pour les lire bloc par bloc qu'élément par élément.

Troisième amélioration:

En plus, j'ai mis au point mes propres subroutines qui se basent sur les routines incore parallèles de ScaLAPACK, et qui se chargent du réglage des paramètres de parallélisme et distribution de données, et du calcul de la régression linéaire, pour des appels plus simples côté utilisateur:

- param_set : Cette subroutine prend en paramètres d'entrée les dimensions de la matrice A, le nombre de processus, et configure différents paramètres allant de la taille des blocs à considérer, aux paramètres nécessaires à un parallélisme optimal entre processus comme le nombre de blocs associés aux processus et leurs indices de début et de fin. Elle initialise également la grille de processus, et se charge d'allouer de la mémoire pour les matrices et vecteurs.
- randgen_IC : Cette subroutine prend en paramètres d'entrée les dimensions de la matrice A, les tailles de blocs, le nombre de processus, et les autres paramètres de sortie de param_set, effectue les opérations suivantes: initialise les descripteurs de tableaux ScaLAPACK, génère aléatoirement un couple matrice-vecteur aléatoire

(A, x) puis calcule le vecteur b=A*x. Ces 3 variables serviront pour la suite comme entrées de la routine linreg_IC.

- linreg_IC : Prend en entrée les mêmes paramètres que randgen, en plus des données A et b, et calcule en parallèle in-core une régression linéaire de b sur A.

Un programme principal regprog_IC a aussi été implémenté pour lancer quelques tests et tester les capacités de calcul incore. On serait intéressé d'avoir une idée sur la plus grande matrice pouvant être traitée, le temps que pourrait prendre le calcul ou la précision atteinte.

Caractéristiques des machines de calcul utilisées:

Une première série de tests a été lancée sur le supercalculateur NEPTUNE du CERFACS: une machine BULL B510 avec 128 nœuds de calcul bi-socket. Chaque nœud de calcul possède les caractéristiques suivantes:

- deux processeurs Intel Sandy Bridge (E P E5-2670) [8 cœurs 2.6 Ghz (8 flops par cycle par cœur soit 330GFlops/s de performance crête par nœud)]
- 32Go de mémoire par nœud, (mémoire DDR3 cadencée à 1600 Mhz)
- Caches L1 (instruction et données) 32Ko, cache L2 256Ko par cœur
- Cache L3 de 20 Mo partagé par les 8 cœurs de chaque processeur

Quant au réseau d'interconnexion Infiniband, il offre une bande passante de 5 Go/s entre nœuds. La latence MPI est inférieure à 1 micro-seconde.

Les mêmes tests sont ensuite lancés sur un ordinateur portable DELL LATITUDE doté de processeurs Intel Core i5-2520M à 2.50Ghz × 2, avec 3.8Go de mémoire et un disque dur de 476.4 Go.

Résultats expérimentaux sur Neptune:

En reprenant les mêmes notations que pour les jeux de tests sur R et Matlab, avec des dimensions $m=10^i$; $n=\lfloor\sqrt{10^j}\rfloor$, on arrive à tracer les tableaux ci-dessous qui contiennent pour chaque (i,j) les temps d'exécution(temps total Tt, temps d'initialisation Ti, temps de facto/résolution T), le taux d'erreur et aussi la vitesse de calcul estimée en nombre de flops par seconde: le nombre approximatif d'opérations nécessaires à une facto/résolution QR calculé pour chaque (m,n) [suivant la formule énoncée en III.2.b-]

est divisé par le temps d'exécution.

Temps d'exécution (temps total Tt, temps d'initialisation Ti):

	j = 1	j = 2	j = 3	j = 4	j = 5	j = 6	j = 7	j = 8
i=1	Ti =8.841 Tt =8.849	Ti =0.093 Tt =0.191						
i=2	Ti =0.099 Tt =0.106	Ti =0.152 Tt =0.246	Ti =8.445 Tt=8.456	Ti =9.851 Tt =9.914				
i=3	Ti =0.149 Tt =0.152	Ti = 0.15 Tt =0.236	Ti =0.105 Tt =0.188	Ti =0.097 Tt =0.196	Ti =0.165 Tt =0.221	Ti =0.444 Tt =0.649		
i=4	Ti =0.682 Tt =0.687	Ti =0.821 Tt =0.902	Ti =0.309 Tt =0.392	Ti =0.315 Tt =0.412	Ti =0.648 Tt =0.793	Ti =1.928 Tt =2.134	Ti =5.878 Tt =6.527	Ti =18.06 Tt =21.31
i=5	Ti =6.209 Tt =6.213	Ti =7.206 Tt =7.288	Ti =2.383 Tt =2.473	Ti =2.356 Tt =2.483	Ti =5.751 Tt =6.015	Ti =17.95 Tt=18.80	Ti =55.82 Tt =60.91	Ti = 178 Tt =222
i=6	Ti =61.17 Tt=61.17 5	Ti =71.39 Tt=71.49 5	Ti =22.97 Tt=23.13 8	Ti=23.05 Tt=23.75 78	Ti=57.33 Tt=59.51 97	Ti=183.2 Tt=192.1 136	WallTime exceeds Limit	
i=7	T=602.38 Tt =602.395	Ti = 733 Tt= 733.3509	Ti=222.7 Tt= 224.0078	Ti =223.2 Tt =230.728	WallTime exceeds Limit			
i=8	Ti = 5968 Tt= 5968.147	Ti = 7188 Tt=7190. 8996	WallTime exceeds Limit					

Temps de facto/résolution, taux d'erreur et vitesses de calcul:

	j = 1	j = 2	j = 3	j = 4	j = 5	j = 6	j = 7	j = 8
i=1	T = 0.008 TE=0.062 fl=2.03e4	T = 0.098 TE=0.127 fl=1.36e4						
i=2	T = 0.007	T = 0.094	T=1.1e-2	T = 0.063				

	TE = 5e-3 fl=2.76e6	TE = 9e-3 fl=2.06e5	TE = 3e-2 fl=1.57e7	TE=0.043 fl=2.12e7				
i=3	T = 0.003 TE = 9e-4 fl = 6e6	T = 0.086 TE = 1e-3 fl=2.32e6	T = 0.083 TE = 2e-3 fl=2.29e7	T = 0.099 TE = 3e-3 fl=1.95e8	T = 0.056 TE = 8e-3 fl=3.19e9	T = 0.205 TE = 2e-2 fl = 6.5e9		
i=4	T = 0.005 TE = 9e-5 fl = 3.6e7	T = 0.081 TE = 1e-4 fl=2.47e7	T = 0.083 TE = 2e-4 fl=2.31e8	T = 0.097 TE = 2e-4 fl=2.05e9	T = 0.145 TE = 5e-4 fl=1.4e10	T = 0.206 TE = 2e-3 fl=9.4e10	T=0.6489 TE = 2e-3 fl=2.8e11	T=3.2505 TE = 4e-3 fl=4.1e11
i=5	T = 0.004 TE = 2e-6 fl=4.51e8	T = 0.082 TE = 1e-5 fl=2.44e8	T = 0.09 TE = 2e-5 fl=2.14e9	T = 0.127 TE = 3e-5 fl=1.6e10	T = 0.264 TE = 6e-5 fl=7.6e10	T=0.8539 TE = 7e-5 fl=2.3e11	T = 5.095 TE = 1e-4 fl=3.9e11	T=43.989 TE = 5e-4 fl=4.4e11
i=6	T = 0.005 TE = 2e-6 fl=3.6e9	T = 0.105 TE = 1e-6 fl = 1.9e9	T = 0.168 TE = 3e-6 fl=1.1e10	T=0.7078 TE = 4e-6 fl=2.8e10	T=2.1897 TE = 5e-6 fl=9.1e10	T=8.9136 TE = 1e-5 fl=2.2e11	WallTime exceeds Limit	
i=7	T = 0.015 TE = 2e-8 fl=1.2e10	T=0.3509 TE = 5e-7 fl = 5.7e9	T=1.3078 TE = 1e-6 fl=1.5e10	T = 7.528 TE = 1e-6 fl=2.7e10	WallTime exceeds Limit			
i=8	T = 0.147 TE = 1e-7 fl=1.2e10	T=2.8996 TE = 1e-7 fl = 6.9e9	WallTime exceeds Limit					

Taux d'erreur : $TE = ||A*x - b|| / (||x||*||A||*eps*M)$ [ici ||.|| ≡ la norme infinie]
eps : epsilon machine (eps ≈ 1.11E-16)
$M = 10^i$: nombre de lignes de la matrice
$N = E[10^{j/2}]$: nombre de colonnes de la matrice
Les cases où est marqué “WallTime exceeds Limit” indiquent les cas de tests où le temps total d'exécution dépasse le temps maximal autorisé sur le supercalculateur Neptune, qui est de 6 heures.

Réglage des paramètres: Pour l'ensemble des simulations précédentes sur Neptune, nous avons utilisé 4 noeuds de calcul, chacun doté de 16 processus. ==> Un total de 64 processus.

▪ pour j = 1, on a pris MB = NB = 3. (Cases à fond cyan)
▪ pour j = 2, on a pris MB = NB = 5. (Cases à fond vert)
▪ pour (i = 2) et (j = 3 ou 4), on a pris des blocs carrés de taille MB = NB = 2. (Cases à fond jaune)
▪ pour j = 3 et i > 2, on a pris MB = NB = 16. (Cases à fond bleu)
▪ dans les autres cas, on a pris MB = NB = 32 (Cases à fond blanc)
De plus, on adapte à chaque fois la grille de processus avec la forme de la matrice (rectangulaire, rectangulaire allongée, carrée).

Résultats expérimentaux sur Latitude:

Toujours avec des matrices à dimensions $m=10^i$; $n=\lfloor\sqrt{10^j}\rfloor$, on arrive à tracer les tableaux ci-dessous qui contiennent pour chaque (i,j) les temps d'exécution(temps total Tt, temps d'initialisationTi, temps de facto/résolutionT), l'erreur E = ||A*x – b|| et le taux d'erreur TE.

Temps de facto/résolution, erreurs et taux d'erreur:

	j = 1	j = 2	j = 3	j = 4	j = 5	j = 6	j = 7	j = 8
i=1	T =2.7e-3 TE=0.088 E = 7e-16	T =8.5e-4 TE=0.153 E= 1e-14						
i=2	T =5.7e-4 TE=0.024 E = 2e-15	T = 1e-3 TE= 0.02 E = 2e-14	T =7.8e-4 TE=0.025 E = 2e-13	T =3.7e-3 TE=0.199 E = 1e-11				
i=3	T =7.5e-4 TE = 1e-3 E = 1e-15	T =7.8e-4 TE = 4e-3 E = 3e-14	T =4.3e-3 TE = 7e-3 E = 5e-13	T=0.0222 TE = 8e-3 E = 5e-12	T = 0.094 TE=0.027 E = 1e-10	T = 0.681 TE = 0.12 E =7.2e-9		
i=4	T = 1e-3 TE = 7e-4 E = 7e-15	T =3.5e-3 TE = 1e-3 E= 1e-13	T=0.0208 TE = 1e-3 E = 1e-12	T = 0.143 TE = 4e-3 E = 3e-11	T = 0.878 TE = 2e-3 E = 1e-10	T = 3.611 TE = 2e-3 E=1.4e-9	T = 25.91 TE=0.026 E=1.5e-7	T =4.7e-6 TE=0.083 E = 175.1
i=5	T =4.9e-3 TE = 4e-4 E = 3e-14	T=0.0234 TE = 3e-4 E = 3e-13	T = 0.185 TE = 1e-3 E = 9e-12	T = 1.082 TE = 8e-4 E = 6e-11	T = 9.193 TE = 7e-4 E = 4e-10	T=67.986 TE = 5e-4 E = 3e-9	T=641.29 TE = 2e-3 E =1.5e-7	

i=6	T = 3.929 TE = 3e-5 E = 3e-14	T=0.0989 TE = 2e-5 E = 2e-13	T = 1.04 TE=4e-5 E = 3e-12	T = 7.473 TE = 2e-5 E = 2e-11	T=8592.7 TE = 2e-4 E = 1e-9			
i=7	T = 0.163 TE = 3e-6 E = 3e-14	T = 0.935 TE = 3e-6 E = 3e-13						
i=8								

Temps d'exécution (temps total Tt, temps d'initialisation Ti):

	j = 1	j = 2	j = 3	j = 4	j = 5	j = 6	j = 7	j = 8
i=1	Ti =0.055 Tt =0.058	Ti=3.1e-4 Tt=1.2e-3						
i=2	Ti=4.6e-4 Tt = 1e-3	Ti=6.7e-4 Tt=1.7e-3	Ti=5.6e-4 Tt=1.3e-3	Ti=1.8e-3 Tt=5.5e-3				
i=3	Ti=5.9e-3 Tt=6.7e-3	Ti=1.9e-3 Tt=2.6e-3	Ti=6.8e-3 Tt =0.011	Ti =0.017 Tt =0.039	Ti =0.026 Tt = 0.12	Ti =0.065 Tt =0.746		
i=4	Ti =0.045 Tt =0.046	Ti =0.026 Tt =0.029	Ti = 0.05 Tt =0.071	Ti =0.075 Tt =0.218	Ti = 0.15 Tt = 1.03	Ti = 0.33 Tt =3.943	Ti =1.055 Tt =26.96	Ti = 3.23 Tt =178.3
i=5	Ti = 0.29 Tt =0.296	Ti = 0.17 Tt =0.193	Ti =0.155 Tt = 0.34	Ti =0.815 Tt =1.897	Ti =1.517 Tt =10.71	Ti = 3.73 Tt =71.36	Ti =10.85 Tt =652.1	
i=6	Ti = 1.94 Tt = 1.98	Ti = 0.68 Tt = 0.78	Ti = 1.53 Tt = 2.57	Ti = 3.73 Tt = 11.2	Ti =14.17 Tt = 8606.87			
i=7	Ti =14.36 Tt =14.52	Ti = 6.8 Tt = 7.74						
i=8								

Commentaires:

D'abord, les faibles taux d'erreur constatés indiquent que les résultats sont fiables et

valides. Ensuite, on peut aussi dire qu'ils concordent globalement avec les estimations faites par d'autres chercheurs [17,18]sur les capacités de calcul in-core séquentielle et parallèle dont voici le contenu ci-dessous:

Quelques stats et estimations concernant le mode de calcul incore séquentiel:

- Avec un ordinateur linux x86 (32bits) Pentium 4 de fréquence 3.2 Ghz et de 2 Go de RAM, le processeur peut atteindre jusqu'à 5 Gflops/s. La plus grande matrice dense pouvant être traitée est de taille N = 11585. Une factorisation LU par exemple requiert approximativement $O(2.67N^3)$ flops, soit à peu près 14 minutes à 5 Gflops/s. Si le système E/S peut supporter des transferts à 40 Mo/s, alors la lecture/écriture de 2Go prendrait environ une minute.

- Soit M la quantité de mémoire RAM disponible, alors le plus grand problème carré pouvant être traité serait de taille $N = O(\sqrt{M})$. La charge de travail quant à elle, augmente de $O(N^3)$ ou $O(M^{3/2})$. Ainsi, si on augmente 4 fois l'espace mémoire, la charge de travail (ou le temps d'exécution) augmentera à son tour de $4^{3/2} = 8$ fois. Pareillement, multiplier par 9 la quantité de mémoire, revient à multiplier par $9^{3/2} = 27$ la charge de travail.

- Or d'après l'exemple précédent, multiplier 9 fois la quantité de mémoire RAM ne fournirait qu'une capacité de 18 Go et prendrait à peu près $27 \times 14 = 378$ min ou 6.3 h (en négligeant le temps consacré aux accès E/S).

- Il est à noter qu'il faut environ 9 minutes pour lire ou écrire 18Go de données.

Quelques stats et estimations concernant le mode de calcul incore parallèle:

- Supposant qu'on dispose d'un cluster de 32 processus et que chacun a son propre disque local, alors la plus grande matrice carrée pouvant être traitée en incore est de taille N = 92682, et sa factorisation LU par exemple prendrait à peu près 3.7 heures.

- De même, augmenter de 9 fois la quantité de mémoire RAM entraînerait un temps d'exécution de 27×3.7 heures ≈ 4.2 jours.

- Ajouter plus de processeurs nécessiterait de rajouter plus de mémoire, plus de disques, et une bande passante E/S plus élevée.Toutefois, dans le cas de certaines machines à mémoire distribuée, la bande passante E/S est fixée, et ne pourrait évoluer pour supporter plus de processeurs.

Régression linéaire en out-of-core avec des prototypes de ScaLAPACK

Pour ce qui est de la partie out-of-core, j'ai récupéré un programme test de ScaLAPACK qui permet de résoudre des systèmes linéaires carrés [http://www.netlib.org/scalapack/prototype/] .

Ce programme sert en principe à s'assurer des bonnes performances des subroutines ScaLAPACK sur des matrices de grande dimension.

Une matrice carrée A de taille $m \times m$, et un vecteur colonne x de taille m sont générés aléatoirement avec les subroutine PFDMATGEN/PDMATGEN, puis le produit des deux est calculé avec la subroutine PFDGEMV, le résultat est stocké dans un vecteur b.

On résout ensuite le système linéaire associé à la matrice A et de second membre b, afin de retrouver un vecteur colonne r qui va correspondre aux coefficients de régression linéaire entre A et b.

Pour cela, on fait appel aux subroutines de factorisation (QR,LU, Cholesky) et de résolution.

Notre objectif étant de résoudre des problèmes de moindres carrés dans le cas général, il s'avérait nécessaire d'adapter le code récupéré à ce cas, tout en gardant la même méthode de factorisation que celle utilisée en incore: la factorisation QR.

En regardant en détail le code source du programme précédent, il y avait 3 étapes principales:

- Factorisation QR de la matrice A: A = Q*R où Q orthogonale et R triangulaire supérieure, toutes deux de taille m m. Elle se fait via l'appel à la subroutine PFDGEQRF.

- Résolution du système linéaire Q*R*x = b par le biais de la subroutine PFDGEQRS.

Cette deuxième étape inclut deux sous-étapes:

+ Mise à jour de b : $b \leftarrow Q^T * b$ via l'appel à la subroutine PFDORMQR.

+ Résolution du système triangulaire R*x = b par substitution arrière (Backward substitution), via l'appel à la subroutine PFDLATRSM.

Pour les problèmes de moindres carrés avec une matrice triangulaire A de taille $m \times n$ de

rang plein, il a fallu modifier le corps de la subroutine PFDGEQRS:

- Factorisation QR de la matrice A: $A = Q*R = [Q1\ Q2] * [R1'\ 0]^T$ où Q orthogonale de taille m×m et R triangulaire supérieure de taille m×n. Q1 de taille m n; Q2 de taille m×(m-n); R1 triangulaire de taille n×n

La factorisation se fait toujours via l'appel à la subroutine PFDGEQRF.

- Résolution du problème aux moindres carrés $\min ||Q*R*x - b|| = \min ||Q1^T b - R1*x||$ par le biais de la subroutine PFDGEQRS.

+ Mise à jour des n premières composantes de b : $b[1:n] \leftarrow Q1^T * b$ via l'appel à la subroutine PFDORMQR.

+ Résolution du système triangulaire R1*x = b[1:n] par substitution arrière (Backward substitution), via l'appel à la subroutine PFDLATRSM

De façon similaire qu'en in-core, l'utilisateur n'a plus à invoquer directement les routines ScaLAPACK pour lancer un calcul out-of-core de régression linéaire. Les subroutines que j'ai définies, se chargent du réglage des paramètres de configuration out-of-core puis des appels ScaLAPACK:

- randgen : Cette subroutine prend en paramètres entrées les dimensions de la matrice A, les tailles de blocs, le nombre de processus, et un chemin d'accès où stocker les fichiers sur disque, puis effectue les opérations out-of-core suivantes: initialise la grille BLACS de processus et les descripteurs de tableaux ScaLAPACK, génère aléatoirement un couple matrice-vecteur aléatoire (A, x) puis calcule le vecteur b=A*x. Ces 3 variables serviront pour la suite comme entrées de la routine linreg.
- linreg : Prend en entrées les mêmes paramètres que randgen, en plus des données A et b, et calcule en parallèle out-of-core une régression linéaire de b sur A.

Ce sont ces subroutines qui peuvent par la suite être intégrées à R++; quant au programme de test regprog, il a été implémenté à la fois pour servir de mode d'emploi des subroutines randgen et linreg, et a pour but aussi de montrer qu'on arrive bien à s'affranchir des contraintes mémoires rencontrées en in-core et donc de calculer des régressions linéaires (avec la factorisation QR) même sur des matrices de très grande dimension. Les tests ont été lancés sur les mêmes machines décrites précédemment

(l'ordinateur portable Dell Latitude et le supercalculateur Neptune) afin de pouvoir comparer les résultats incore et out-of-core. Ci-dessous les résultats obtenus des tests out-of-core toujours avec les dimensions de matrices en puissances de 10.

Résultats expérimentaux sur Latitude:

Tableau des erreurs et taux d'erreur pour chaque (i,j):

	j = 1	j = 2	j = 3	j = 4	j = 5	j = 6	j = 7	j = 8
i=1	TE=0.031 E = 2e-17	TE=0.619 E = 2e-15						
i=2	TE=0.033 E=2.01e-16	TE=0.029 E=1.18e-15	TE=0.062 E=7.42e-15	TE=0.088 E=2.99e-14				
i=3	TE=0.026 E=2.55e-15	TE = 0.01 E=3.4e-15	TE=0.017 E=1.64e-14	TE=0.016 E=5.89e-14	TE=0.016 E=1.89e-13	TE=0.027 E=1.04e-12		
i=4	TE=2.99e-3 E=2.59e-15	TE=2.75e-3 E=8.63e-15	TE=2.53e-3 E=2.95e-14	TE=3.46e-3 E=1.11e-13	TE=3.07e-3 E=3.58e-13	TE=3.98e-3 E=1.46e-12	TE=4.92e-3 E=5.8e-12	TE=7.54e-3 E=2.82e-11
i=5	TE=5.06e-4 E=2.45e-15	TE=1.71e-4 E=5.58e-15	TE=2.61e-4 E=2.25e-14	TE=1.26e-4 E=4.19e-14	TE=3.18e-4 E=3.57e-13	TE=2.93e-4 E=1.098e-12	TE=1.15e-3 E=1.36e-11	TE=1.25e-3 E=4.63e-11
i=6	TE=4.35e-5 E=4.45e-15	TE=3.5e-5 E=1.18e-14	TE=4.35e-5 E=4.98e-14	TE=1.88e-4 E=6.39e-13	TE=3.41e-4 E=3.83e-12	TE=5.89e-3 E=4.73e-11		
i=7	TE=1.5e-	TE = 1e-5	TE=7.65e					

	5 E = 2e-15	E= 2.96e-14	-6 E = 8e-14					
i=8	TE = 5e-6 E=3.83e-14							

Temps d'exécution (temps total, temps de facto/résolution) pour chaque (i,j):

	j = 1	j = 2	j = 3	j = 4	j = 5	j = 6	j = 7	j = 8
i=1	Tt = ε Tfs = ε	Tt = ε Tfs = ε						
i=2	Tt = ε Tfs = ε	Tt = ε Tfs = ε	Tt = ε Tfs = ε	Tt = 0.01 Tfs = ε				
i=3	Tt = 0.01 Tfs = ε	Tt = 0.01 Tfs = ε	Tt = 0.01 Tfs = ε	Tt = 0.04 Tfs =0.02	Tt = 0.14 Tfs = 0.1	Tt = 0.72 Tfs =0.64		
i=4	Tt = 0.07 Tfs =0.02	Tt = 0.05 Tfs =0.01	Tt = 0.05 Tfs =0.02	Tt = 0.28 Tfs =0.14	Tt = 1.31 Tfs =1.12	Tt = 9.67 Tfs = 9	Tt =74.52 Tfs = 73	Tt =526.7 Tfs =514
i=5	Tt = 0.35 Tfs =0.13	Tt = 0.21 Tfs =0.07	Tt = 0.31 Tfs =0.15	Tt = 2.09 Tfs =1.42	Tt = 13 Tfs =10.6	Tt =109.5 Tfs =91.8	Tt =1536 Tfs=1480	Tt=14253 Tfs=13787
i=6	Tt = 6.5 Tfs =2.35	Tt = 3.16 Tfs =1.16	Tt = 6.14 Tfs =3.23	Tt =115.6 Tfs =98.6	Tt =4270 Tfs=4062	Tt ≈6jours Tfs ≈ 6 j		
i=7	Tt =51.53 Tfs =17.5	Tt =38.43 Tfs =16.5	Tt =336.1 Tfs =281					
i=8	Tt = 1156 Tfs = 818							

Résultats expérimentaux sur Neptune:

Tableau des erreurs et taux d'erreur pour chaque (i,j):

	j = 1	j = 2	j = 3	j = 4	j = 5	j = 6	j = 7	j = 8
i=1	TE=0.052 E = 4e-17	TE=0.414 E = 1e-15						
i=2	TE=0.028 E=1.69e-16	TE=0.029 E=1.22e-15	TE=0.048 E=5.73e-15	TE=0.069 E=2.36e-14				
i=3	TE=0.019 E = 2e-15	TE = 9e-3 E = 3e-15	TE=9e-3 E = 9e-15	TE = 0.01 E = 4e-14	TE=0.012 E = 1e-13	TE = 2e-2 E = 8e-13		
i=4	TE=1.96e-3 E = 1.7e-15	TE=2.47e-3 E=7.75e-15	TE=2.13e-3 E=2.48e-14	TE=2.68e-3 E=8.62e-14	TE=2.6e-3 E=2.9e-13	TE=2.69e-3 E=9.72e-13	TE=3.39e-3 E=3.95e-12	TE=5.25e-3 E=1.93e-11
i=5	TE=6.8e-4 E=3.14e-15	TE=3.63e-4 E=1.56e-14	TE=1.76e-4 E=2.204e-14	TE=7.1e-4 E=2.255e-13	TE=7.76e-4 E=8.34e-13	TE=7.71e-4 E=2.93e-12	TE=8.16e-4 E=9.64e-12	TE=8.77e-4 E=3.28e-11
i=6	TE=1.35e-4 E=1.04e-14	TE=1.82e-5 E=6.34e-15	TE=1.23e-5 E=1.25e-14	TE=2.97e-5 E=1.08e-13	TE=2.91e-4 E=3.1e-12	TE=2.37e-4 E=8.84e-12	TE=2.37e-4 E=2.79e-11	TE=2.54e-4 E=9.34e-11
i=7	TE=5.41e-5 E=7.5e-15	TE=1.07e-6 E = 3.6e-15	TE=8.47e-7 E=9.18e-15	TE=3.21e-6 E=1.02e-13	TE=8.01e-5 E=9.48e-12			
i=8	TE = 6e-8 E=4.33e-	TE = 2e-7 E=1.11e-	TE = 3e-7 E=3.13e-					

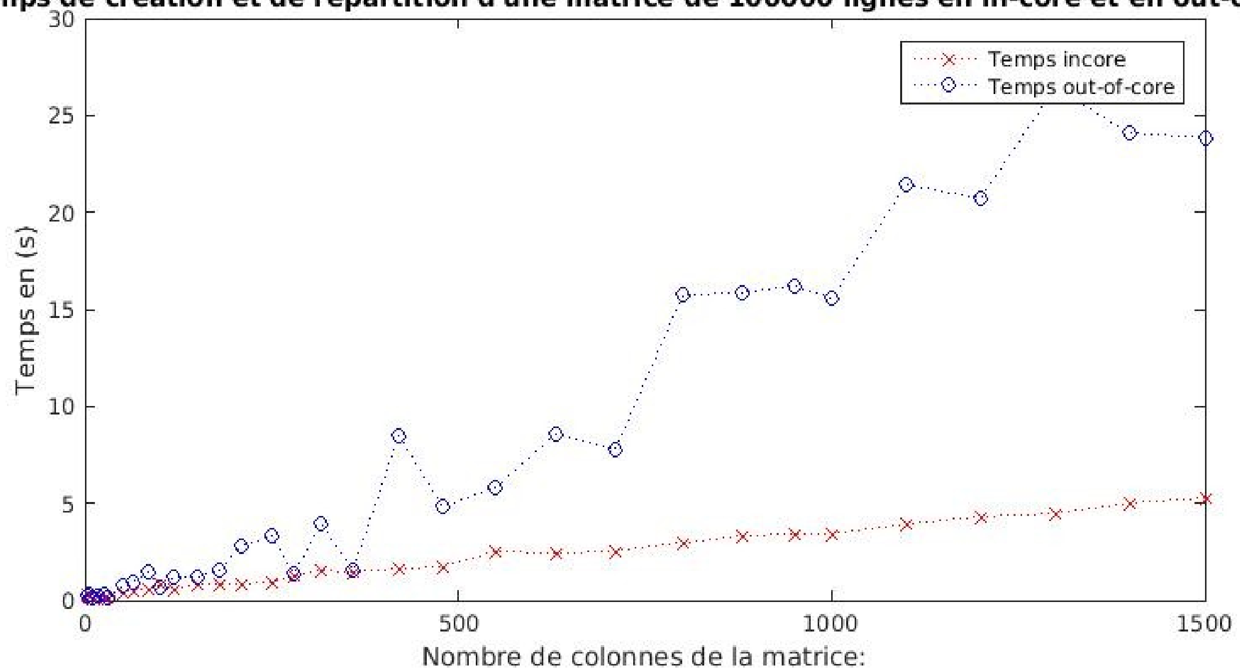
	16	14	14					

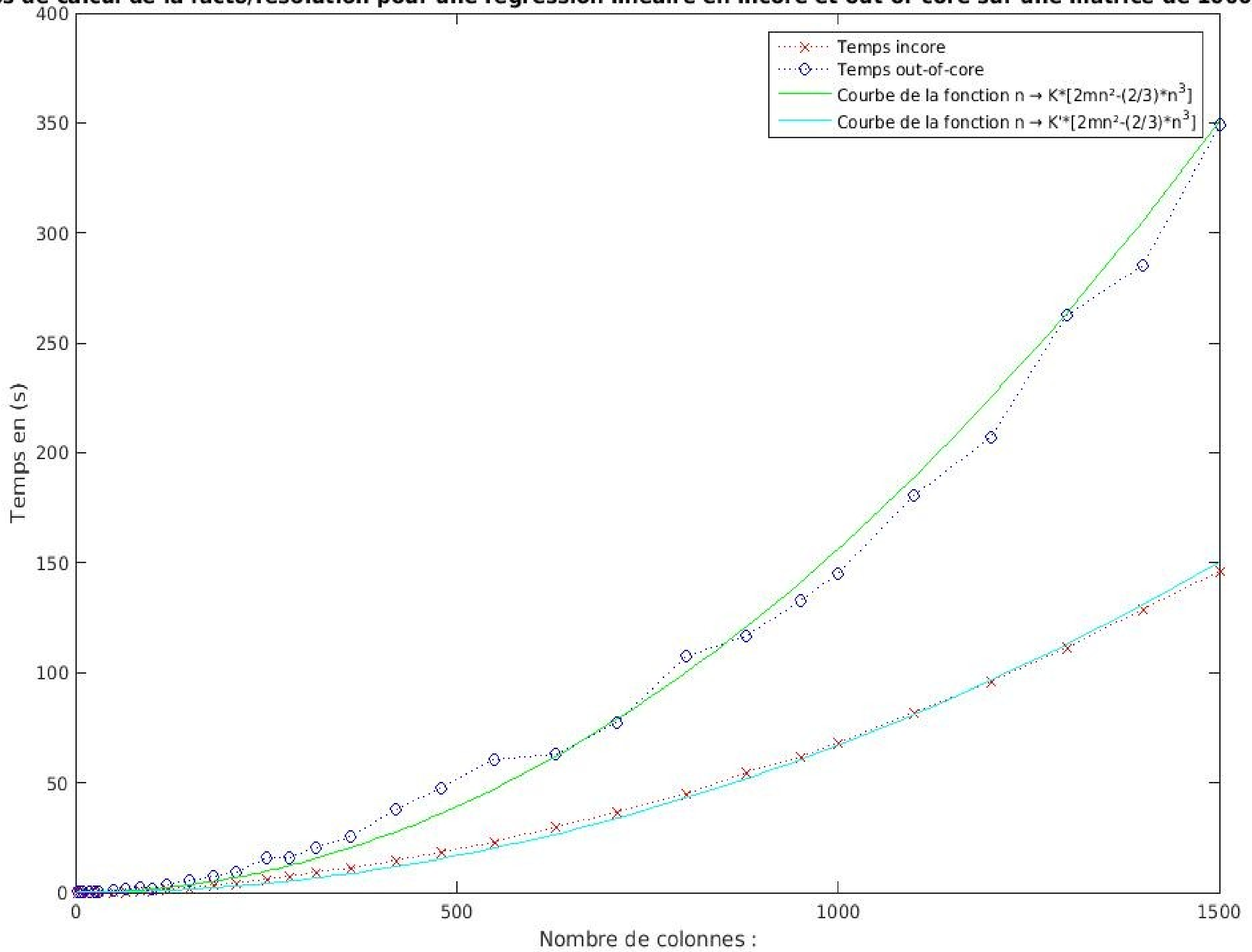
Temps d'exécution (temps total, temps de facto/résolution) pour chaque (i,j):

	j = 1	j = 2	j = 3	j = 4	j = 5	j = 6	j = 7	j = 8
i=1	Ti=1.9e-3 Tf = ε Ts = ε	Ti=7.2e-4 Tf = ε Ts = ε						
i=2	Ti=1.9e-3 Tf = ε Ts = ε	Ti=9.8e-4 Tf = ε Ts = ε	Ti = 2e-3 Tf = ε Ts = ε	Ti=2.4e-3 Tf = ε Ts = ε				
i=3	Ti=8.4e-3 Tf = ε Ts = ε	Ti=4.4e-3 Tf = ε Ts = ε	Ti=4.9e-3 Tf = ε Ts = ε	Ti=0.013 Tf = 0.01 Ts = ε	Ti =0.033 Tf = 0.05 Ts = ε	Ti=0.056 Tf = 0.36 Ts = 0.01		
i=4	Ti =0.044 Tf = 0.01 Ts = ε	Ti =0.025 Tf = ε Ts = ε	Ti=0.039 Tf =0.02 Ts = ε	Ti =0.095 Tf = 0.09 Ts = 0.01	Ti=0.099 Tf = 0.39 Ts = 0.03	Ti=0.321 Tf = 2.40 Ts = 0.1	Ti=1.267 Tf = 5.84 Ts = 0.36	Ti=2.323 Tf = 13.9 Ts = 0.86
i=5	Ti =0.331 Tf = 0.08 Ts = 0.03	Ti =0.196 Tf = 0.05 Ts = 0.01	Ti =0.099 Tf = 0.08 Ts = 0.01	Ti =0.853 Tf = 0.45 Ts = 0.12	Ti =1.264 Tf = 2.44 Ts = 0.49	Ti =6.646 Tf =11.34 Ts = 2.16	Ti =18.18 Tf =62.93 Ts =10.37	Ti = 35.3 Tf =329.1 Ts = 47.4
i=6	Ti =2.115 Tf = 0.37 Ts = 0.14	Ti =0.555 Tf = 0.19 Ts = 0.04	Ti =1.538 Tf = 0.4 Ts = 0.04	Ti =2.924 Tf = 1.79 Ts = 0.42	Ti =15.03 Tt =66.12 Ts = 8.74	Ti =39.89 Tf =304.8 Ts =36.02	Ti =147.1 Tf =937.2 Ts =112.3	Ti =293.7 Tf =7331 Ts =330.1
i=7	Ti =14.41 Tf = 3.92 Ts = 0.7	Ti = 3.21 Tf = 1.42 Ts = 0.59	Ti = 7.29 Tf =10.71 Ts = 1.2	Ti = 24.5 Tf =40.81 Ts =10.66	Ti =153.1 Tf = 567 Ts = 5270	Ti = 1429 Tf = 1709 Ts=14769		
i=8	Ti =56.74 Tf =13.46 Ts = 5.37	Ti =36.36 Tf =38.93 Ts = 10.6	Ti =89.28 Tf=130.6 Ts =27.47					

Constatations:
La comparaison des résultats in-core et out-of-core met en exergue l'intérêt qu'il y a à adopter les méthodes out-of-core et continuer à les développer, non seulement pour les calculs de régression linéaire mais aussi pour d'autres types de problématiques. Comme on peut le constater, on arrive à traiter des matrices de taille plus grande en out-of-core qu'en in-core, (ici, on obtient même un facteur de gain ≈ 10).

Ce gain en taille de matrice traitée a un coût en temps puisque le temps d'exécution (temps de création et répartition de la matrice + temps de calcul de la facto/résolution) en out-of-core est supérieur à celui en in-core, comme le montre les courbes ci-dessous obtenues via des tests sur au plus 8 processus, en prenant des blocs de taille MB = 32:





Les constantes K et K' ont pour valeur: K = 7.84e-10; K' = 3.36e-10 .

On remarque d'abord que si le temps de calcul de la facto/résolution augmente de façon régulière, suivant à un facteur près (K ou K') la fonction estimée du nombre d'opérations (cf III.2.b-): $nb_op(n) = 2mn^2 - 2n^3/3$, le temps de création et répartition de la matrice sur les processus quant à lui connaît plutôt de petites fluctuations au départ puis semble suivre une fonction en escalier. En tout cas, ce temps-là est plutôt faible comparé au temps de la facto/résolution: un peu moins de 10% .

Autre fait marquant: En out-of-core sur l'ordinateur portable Dell Latitude, la plus grande matrice pouvant être traitée correspond à peu près à celle traitée en in-core sur le supercalculateur Neptune. Ce qui est quand même considérable vu la différence de

puissance entre les deux machines!

Cela signifierait que si on pouvait généraliser l'usage out-of-core, on pourrait réaliser sur des machines de travail standards certains calculs qui étaient jusqu'ici réservés aux supercalculateurs.

Difficultés & obstacles:

A propos des difficultés rencontrées, j'en citerai deux majeures:

La première concerne la partie génération et distribution de données out-of-core sur ScaLAPACK que je ne suis pas arrivé à maîtriser à 100%. Par conséquent, je n'ai pour le moment pas pu traiter des matrices quelconques en entrée et j'ai dû me limiter à des tests utilisant des matrices aléatoires. Par faute de temps et par insuffisance de documentation sur ce procédé (la plupart des codes sources des subroutines de base ne sont pas commentés), j'ai été contraint de faire appel à la subroutine PFDMATGEN implémentée sur ScaLAPACK qui génère une matrice aléatoire de taille paramétrable, la découpe et la répartit sur une grille de processus. Les fichiers associés aux processus sont ainsi créés dans un chemin d'accès choisi par l'utilisateur. Ils contiennent tous des portions de la matrice aléatoire générée par PFDMATGEN.

Le deuxième obstacle a été la configuration des fichiers de données out-of-core pour le supercalculateur Neptune. En effet, j'ai constaté des défauts d'exécution lorsque j'ai lancé pour la première fois les codes prototypes out-of-core de ScaLAPACK sur le supercalculateur Neptune. Cela concernait l'ouverture des fichiers de données en configuration distribuée pour les processus lors d'une exécution parallèle. Les processus en charge du calcul n'arrivaient pas à créer leurs fichiers de données. Ce qui se traduisait par des messages d'erreurs qui ne ciblaient pas directement une partie spécifique du code.

Cela m'a pris beaucoup de temps et d'efforts à identifier l'origine de ces erreurs: à peu près un mois à explorer toutes les pistes possibles (problèmes de mémoire, répertoire de fichiers inaccessible au processus, format de fichiers invalide, etc...).

Comme expliqué précédemment, l'exécution de plusieurs processus en parallèle nécessite que chacun crée son propre fichier de données configuré en mode distribué. Cette opération se fait via la fonction open() en C: Elle établit une connexion entre un fichier et un descripteur de fichier (un entier) qui fait office de référence au fichier. Le descripteur de fichier est ainsi utilisé par les autres fonctions E/S qui tentent d'accéder à ce fichier. En principe, un appel à la fonction open() pour ouvrir un fichier retourne le descripteur de fichier à valeur minimale auquel aucun fichier ouvert au niveau de ce processus n'a été associé. Et il existe des descripteurs de fichiers par défaut: par exemple

la valeur 0 pour les entrées (standard inputs), 1 pour les sorties (standard outputs), etc...

Les codes prototypes de ScaLAPACK ayant sans doute été testés et configurés pour des machines standards où l'entrée standard est déjà réservée (clavier), contenaient des boucle IF qui s'assuraient que tout descripteur de fichier créé avait une valeur valide (>=1). Or au niveau d'un processeur du supercalculateur où il n'y a pas d'entrée standard, le premier descripteur de fichier créé se voit attribuer la valeur 0. Une fois le seuil admissible pour les descripteurs de fichier passé de 1 à 0 dans la boucle IF, le problème est réglé.

V. Conclusion & perspectives:

Récapitulatif du travail effectué

Afin d'intégrer des outils de régression linéaire capables de traiter des problèmes à très grande dimension sur R++, nous nous sommes tournés vers ScaLAPACK qui dispose de subroutines out-of-core, parallèles, mais par contre ces dernières sont des routines de base. En effet, il n'y a pas de routines qui prennent directement les matrices d'entrée et calculent les coefficients de régression linéaire. L'utilisateur doit gérer lui-même plusieurs aspects notamment la création d'une grille de processus, la répartition de données sur cette grille, le réglage des paramètres de calcul et ensuite faire les appels à différentes subroutines (PFDGEQRF pour la factorisation QR, PFDGEQRS pour la résolution; cette dernière elle-même inclut 4 ou 5 subroutines internes qu'il a fallu modifier pour traiter des problèmes rectangulaires et non pas seulement des problèmes carrés).

Le travail effectué jusqu'ici, a été d'implémenter une routine qui encapsule les différentes subroutines internes de ScaLAPACK (modifiées pour traiter des problèmes rectangulaires) pour chaque étape de la régression linéaire que ce soit en incore ou en out-of-core sur ScaLAPACK:
- Une routine pour la répartition des données et réglage des paramètres.
- Une deuxième routine pour la partie calcul et résolution.
- Un programme test qui montre comment faire appel à ces routines pour résoudre un problème de régression linéaire.

Avec ces nouvelles routines, l'utilisateur n'a plus à manipuler du ScaLAPACK et faire lui-même les réglages de paramètres indispensables aux appels de routines ScaLAPACK.

Perspectives futures

Les routines implémentées ont été dans un premier temps codées en Fortran, mais seront prochainement transformées en C (avec la collaboration de Joël Falcou du LRI), pour pouvoir être intégrées à R++. Il faudra aussi trouver un moyen de permettre la lecture de matrice quelconque et leur distribution sur les processus. J'envisage de coder une subroutine spécialement dédiée à cette opération, afin de pouvoir l'invoquer lors des tests sur de vraies matrices. Ainsi, il sera possible de faire des tests sur des cas pratiques, notamment avec des matrices non supportées en in-core, et comparer les résultats avec des prévisions ou des résultats obtenus avec d'autres outils pour enfin tirer des

conclusions sur l'efficacité de ces méthodes out-of-core.
Une éventuelle extension de la régression linéaire aux modèles logistiques et linéaires généralisés (GLM) est aussi envisageable.

Un dernier soucis technique est de savoir si les bibliothèques (BLACS, LAPACK, MPI) dont dépend ScaLAPACK, sont facilement installables. Est-ce qu'un utilisateur lambda peut les installer seul, ou bien est-il possible d'inclure ces bibliothèques à l'installation de R++? Dans le cas échéant, un manuel d'installation devra être conçu pour expliquer la procédure d'installation de BLACS, LAPACK et MPI).

Remerciements

Je tiens à témoigner de ma grande reconnaissance à tous ceux qui ont contribué de près ou de loin à la réussite de mon stage.
Je pense en premier lieu à Monsieur Serge Gratton, mon tuteur et responsable du parcours Modélisation et simulation numérique à l'ENSEEIHT, à qui je dois la réussite de ma formation et ce nouveau départ dans la vie professionnelle.
Je remercie aussi Monsieur Christophe Genolini, mon tuteur de stage et chef du projet R++ the next step, pour la confiance qu'il m'a accordée dès mon arrivée. Doté d'exceptionnelles qualités, il est pour moi un exemple à suivre et j'espère continuer à évoluer au sein de son équipe.
Je souhaite exprimer ma profonde gratitude à Monsieur Xavier Vasseur, chercheur au sein de l'équipe ALGO du CERFACS, qui a facilité mon intégration et m'a accompagné tout au long de cette expérience avec patience et pédagogie. Je le remercie pour sa disponibilité et ses conseils qui m'ont été d'une aide précieuse.
Je remercie également les autres membres de l'équipe ALGO, et mes camarades stagiaires et thésards qui ont fait de ce stage une expérience professionnelle et humaine très enrichissante.

Conclusion Générale

Ce stage a été pour moi une réelle source d'enrichissement sur les plans fonctionnel, technique et relationnel.
Sur le plan relationnel, il m'a en effet été offert pendant ces 6 mois de travailler au contact de personnes compétentes et généreuses qui, par leurs carrières et leurs parcours de vie, m'ont aidé à préciser mon projet de vie.
Sur le plan technique, ce stage m'a permis d'approfondir ma connaissance des outils de calcul haute-performance, de renforcer ma maîtrise de certains outils out-of-core. Grâce à cela, mon profil a gagné en crédibilité et j'ai gagné en savoir-faire et en assurance.
D'un point de vue fonctionnel, j'ai enrichi ma culture informatique par des connaissances métiers spécifiques au secteur de la datascience auxquelles s'ajouteront

bientôt, je l’espère, des connaissances dans le secteur de l’E-commerce.
Le fait qu’une offre de thèse se soit présentée à l’issue du stage est aussi un critère de satisfaction. Je suis maintenant prêt à renouer avec la vie active dans la sérénité et la confiance, et je suis plus que jamais déterminé à combler mes lacunes et à fournir un travail de qualité.
En écrivant ces dernières lignes, je repense à tous les obstacles qui ont entravé mon chemin, à tous les moments de doute. Puis, je pense aux personnes à qui je dois ma persévérance, l’aboutissement de ma formation et le début de cette nouvelle étape. Je pense à tous ceux qui m’ont encadré et qui m’ont soutenu à l'ENSEEIHT. A ces gens exceptionnels, je serai toujours reconnaissant.

Références bibliographiques :

[1] J.Chen, Z.Jin, Q.Shi, J.Qiu, W.Liu: "*Block Algorithm and Its Implementation for Cholesky Factorization*", in ICCGI 2013, The Eighth International Multi-Conference on Computing in the Global Information Technology.

[2] J.Choi, J.Dongarra, R.Pozo, and D.Walker: "*ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers*", in Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation, McLean, Virginia, 1992, IEEE Computer Society Press, pp.120-127.

[3] G.Fox, M.Johnson, G.Lyzenga, S.Otto, J.Salmon, and D.Walker: "*Solving Problems on Concurrent Processors*", *Volume 1*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[4] G.Geist and C.Romine: "*LU factorization algorithms on distributed memory multiprocessor architectures*", SIAM J. Sci. Stat. Comput., 9 (1988), pp.639-649.

[5] G.Henry and R.Van De Geijn: "*Parallelizing the QR algorithm for the unsymmetric algebraic eigenvalue problem: Myths and reality*", SIAM J. Sci. Comput., 17 (1996), pp.870-883.

[6] C.Ashcraft: "*The Distributed Solution of Linear Systems Using the Torus-wrap Data mapping*", Tech. Rep. ECA-TR-147, Boeing Computer Services, Seattle, WA, 1990.

[7] R.Brent: "*The LINPACK Benchmark on the AP 1000*", in Frontiers, 1992, McLean, VA, 1992, pp.128-135.

[8] S.Huss-Lederman, E.Jacobson, A.Tsao, and G.Zhang: "*Matrix Multiplication on the Intel Touchstone DELTA*", Concurrency: Practice and Experience, 6 (1994), pp.571-594.

[9] V.Kumar, A.Grama, A.Gupta, and G.Karypis: "*Introduction to Parallel Computing - Design and Analysis of Algorithms*", The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

[10] E.Chu and A.George: "*QR Factorization of a Dense Matrix on a Hypercube Multiprocessor*", SIAM Journal on Scientific and Statistical Computing, 11 (1990), pp.990-1028.

[11] B.Hendrickson and D.Womble: "*The torus-wrap mapping for dense matrix calculations on massively parallel computers*", SIAM J. Sci. Stat. Comput., 15 (1994), pp.1201-1226.

[12] W.Lichtenstein and S.L.Johnsson: "*Block-cyclic dense linear algebra*", SIAM J. Sci. Stat. Comput., 14 (1993), pp.1259-1288.

[13] R.H.Bisseling and J.G.G. Van De Vorst: "*Parallel LU decomposition on a transputer network*", in Lecture Notes in Computer Science, Number 384, G.A. van Zee and J.G.G. van de Vorst, eds., Springer-Verlag, 1989, pp.61-77.

[14] J.Dongarra, R.Van De Geijn, and D.Walker: "*Scalability issues in the design of a library for*

dense linear algebra", Journal of Parallel and Distributed Computing, 22 (1994), pp.523-537.

[15] R.Schreiber and C.Van Loan: *"A Storage Efficient WY Representation for Products of Householder Transformations"*, *SIAM J. of Sci. Stat. Computing*, 10:53-57, 1991.

[16] C.Bischof and C.Van Loan: "*The WY Representation for Products of Householder Matrices"*, SIAM J. of Sci. Stat. Computing, 8:s2-s13, 1987.

[17] E.D'Azevedo: "*Parallel out-of-core extension to ScaLAPACK*", Computer Science and Mathematics Division, Oak Ridge National Laboratory, TN, USA.

[18] E.D'Azevedo and J.Dongarra: "*The design and implementation of the parallel out-of-core ScaLAPACK LU, QR, and Cholesky factorisation routines*", UT, CS-97-347, January 1997.

[19] M.Roux: *"Algorithmes de Classification"*, Editions Masson, Paris 1985.

[20] D.Caumont, J.-L.Chandon: *"Quelques problèmes liés à la validité d'une classification"*, September 1989.

[21] L.Papoz: "*Qualité des données dans les études épidémiologiques"* , Gérontologie et société, Fond. Nationale de gérontologie, 2011/4 (n°99).

[22] S.Ganassali, J.Moscarola: *"Protocoles d'enquêtes et efficacité des sondages par internet"*, Décisions Marketing No. 33 (Janvier-Mars 2004), pp. 63-75.

[23] J.Cohen, B.Dolan, M.Dunlap, J.M.Hellerstein, C.Welton: *"MAD skills: new analysis practices for big data"*, Proceedings of the VLDB Endowment, Volume 2 Issue 2, August 2009, pp.1481-1492.

[24] T.H.Davenport, P.Barth, R.Bean: *"How 'Big Data' Is Different"*, MIT Sloan Management Review, fall 2012, volume n°54.

[25] M.Cox, D.Ellsworth: *"Managing big data for scientific visualization"*, ACM Siggraph, 1997.

[26] Å.Björck: *"Numerical methods for least squares problems"*, SIAM, 1996.

[27] S.Gratton, M.Salaun: *"Cours d'analyse matricielle et d'optimisation"* , 2010, pp 75-78.

[28] P.Besse: "*Régression linéaire multiple ou modèle gaussien"*, WikiStat, 2013.

[29] S.Ostrouchov: *"Cholesky factorization"*, Netlib utk papers, 1995.

[30] S.Ostrouchov: *"QR factorization"*, Netlib utk papers, 1995.

[31] L.N.Trefethen, D.Bau: *"Numerical Linear Algebra"*, SIAM, 1997.

[32] L.S.Blackford, J.Choi, A.Cleary, E.D'Azevedo, J.Demmel, I.Dhillon, J.Dongarra, S.Hammarling, G.Henry, A.Petitet, K.Stanley, D.Walker, R.C.Whaley: "*ScaLAPACK User's Guide"* , SIAM, 1997.