Projet de calcul parallèle

Ce sujet est inspiré d'un cours de Patrick Amestoy et Michel Daydé de l'ENSEEIHT.

1 Considérations Numériques

On cherche à résoudre un système linéaire $\mathtt{Ax} = \mathtt{b}$, où $\mathtt{A}$ est une matrice carrée creuse d'ordre N, et où $\mathtt{x}$ et $\mathtt{b}$ sont des vecteurs d'ordre N. La matrice $\mathtt{A}$ est non-symétrique, tri-diagonale par blocs et composée de $Nprocs$ blocs de taille M (voir Figure 1 pour $Nprocs = 4$). On a donc $N = Nprocs * M$. La matrice sera aussi supposée à diagonale strictement dominante. Le découpage en blocs de taille M de la matrice induira un découpage sur les vecteurs $\mathtt{x}$, $\mathtt{b}$ et $\mathtt{r}$ ($\mathtt{r} = \mathtt{b} - \mathtt{Ax}$). Par exemple, x_k correspond au k^{ieme} bloc de taille M du vecteur $\mathtt{x}$ (c.à.d. $\mathtt{x}_k = \mathtt{x}((k-1) * M + 1..k * M)$ avec les notations vectorielles classiques).

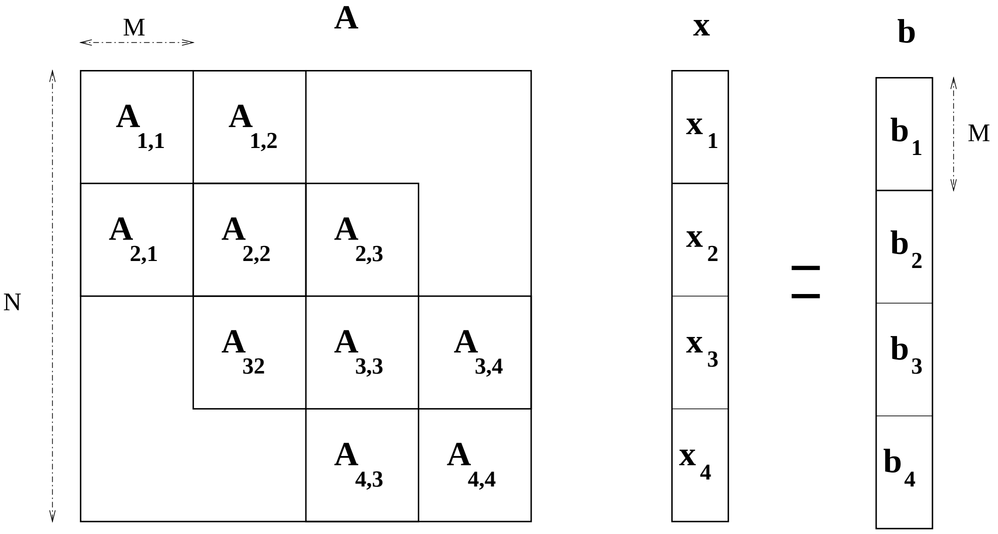

Fig. 1 – Description des structures de données et de leur découpage ($Nprocs = 4$)

On utilise la méthode itérative de Jacobi par blocs. On cherche donc une suite $\{\mathtt{x}^{(Iter)}\}$ de vecteurs d'ordre N convergeant vers la solution du système linéaire quel que soit le vecteur initial $\mathtt{x}^{(0)}$.
La méthode de Jacobi par blocs se déduit naturellement de la méthode de Jacobi. Avec la méthode de Jacobi par blocs le vecteur $\mathtt{x}^{(Iter+1)}$ se déduit du vecteur $x^{(Iter)}$ par la relation simple suivante :

$$\mathtt{x}^{(Iter+1)} \quad = \quad \mathtt{x}^{(Iter)} \quad + \quad \mathtt{D}^{-1} * (\mathtt{b} \; - \; \mathtt{A} * \mathtt{x}^{(Iter)}),$$

où $\mathtt{D}$ désigne la matrice bloc-diagonale constituée des blocs diagonaux $\mathtt{A}_{kk}$ de la matrice $\mathtt{A}$. La méthode de Jacobi par blocs converge en particulier pour des matrices à diagonale strictement dominante.

2 Algorithme séquentiel de Jacobi par blocs

Soient $\mathtt{x}^{(0)} = 0$ le vecteur solution initial, $\mathtt{x}$ l'itéré courant, EPS la précision souhaitée, et $Iter$ le numéro de l'itération courante (on limite ici le nombre maximum d'itérations à $Maxiter$). $\mathtt{y}$ sera un vecteur réel de travail de taille M. On utilisera la norme max pour le test de convergence

$$\|\mathtt{r}\| = \max_{i=1,N} |\mathtt{r}_i|$$

.

Algorithme séquentiel de Jacobi par blocs

Début

- Initialisations : $Iter = 0$ $\mathtt{x} = 0;$ $\mathtt{r} = \mathtt{b}$
- Factorisation de blocs diagonaux :
  Pour $k = 1$ à $Nprocs$ **Faire**
  Copie de $\mathtt{A}_{kk}$ dans $\mathtt{D}_k$
  Factorisation $\mathtt{LU}$ de $\mathtt{D}_k$ ($\mathtt{D}_k = \mathtt{P}_k * \mathtt{L}_k * \mathtt{U}_k$)
  Fin Pour
- **Tant Que** ($Iter < Maxiter$) et ($\|\mathtt{r}\| >$ EPS) **Faire**
  - Calcul du nouvel itéré :
    Pour $k = 1$ à $Nprocs$ **Faire**
    Calcul de $\mathtt{y}$ / $\mathtt{P}_k * \mathtt{L}_k * \mathtt{U}_k\ \ \mathtt{y} = \mathtt{r}_k$ (descente-remontée)
    $\mathtt{x}_k = \mathtt{x}_k + \mathtt{y}$
    Fin Pour
  - Mise à jour du résidu : $\mathtt{r} = \mathtt{b} - \mathtt{A} * \mathtt{x}$
  - Incrémenter $Iter$
- **Fin Tant Que**
- Si le nombre d'itérations est inférieur à $Maxiter$ alors $\mathtt{x}$ contient une approximation, à la précision demandée, de la solution du système.

Fin BlocJacobi séquentiel

2.1 Convention sur le format de A

1. On exploitera la structure tridiagonale de A pour son stockage : seules les valeurs non nulles seront stockées.
2. On stocke les valeurs bloc par bloc.
3. On parcourt les blocs en ligne d'abord.
4. Ainsi, sur l'exemple de la figure 1, les données seront stockées dans l'ordre suivant : $A_{1,1}$, $A_{1,2}$, $A_{2,1}$, $A_{2,2}$, $A_{2,3}$, $A_{3,2}$,$A_{3,3}$, $A_{3,4}$, $A_{4,3}$, $A_{4,4}$.
5. Au sein d'un bloc les valeurs seront stockées en ligne et de manière contigüe en mémoire.

3 Spécification de la version parallèle de Jacobi par blocs

3.1 Remarques

1. On notera que les itérations des deux boucles sur k de l'algorithme séquentiel sont indépendantes (chaque processus peut travailler en parallèle sur une instance de la boucle).
2. De plus, si l'on exploite la structure tri-diagonale par bloc de la matrice $\mathtt{A}$ alors la Mise a jour du résidu peut s'écrire sous la forme (on suppose ici que $Nprocs \geq 2$)
   $\mathtt{r}_1 = \mathtt{b}_1 - \mathtt{A}_{1,1}\,\mathtt{x}_1 - \mathtt{A}_{1,2}\,\mathtt{x}_2$
   Pour $k = 2$ à $Nprocs - 1$ **Faire**
   $\mathtt{r}_k = \mathtt{b}_k - \mathtt{A}_{k,k-1}\,\mathtt{x}_{k-1} - \mathtt{A}_{k,k}\,\mathtt{x}_k - \mathtt{A}_{k,k+1}\,\mathtt{x}_{k+1}$
   Fin Pour
   $\mathtt{r}_{Nprocs} = \mathtt{b}_{Nprocs} - \mathtt{A}_{Nprocs,Nprocs-1}\,\mathtt{x}_{Nprocs-1} - \mathtt{A}_{Nprocs,Nprocs}\,\mathtt{x}_{Nprocs}$
   Si on suppose que chaque processus, k, possède le k^{ieme} bloc de lignes de $\mathtt{A}$ alors il suffit qu'il possède aussi la valeur de la solution de ses voisins pour que le mise à jour de $\mathtt{r}_k$ puisse être parallélisée.

3.2 Structures de données distribuées

1. Chaque processeur k possédera uniquement la partie non-nulle du k^{ieme} bloc de lignes de la matrice $\mathtt{A}$ (matrice ayant M lignes et au maximum de $3 * M$ colonnes)
2. Une copie du k^{ieme} bloc diagonal de la matrice sera rangée dans une matrice $\mathtt{D}$ locale à chaque processus.
3. Un vecteur local x de taille maximum $3 * M$ permettra de ranger à chaque étape la solution locale ainsi que les solutions locales des voisins concernés par la mise à jour du $\mathtt{r}_k$ local.

3.3 Propriétés de l'algorithme parallèle

1. Seul le processus maître (processus 0) possède initialement la matrice tri-diagonale `A` et le second membre `b`. Le maître distribuera les données et assemblera la solution finale `x`. Une fois cette distribution effectuée, il participera au calcul mais ne pourra plus accéder à `A` et `B`.
2. La matrice `A` est d'ordre $N = Nprocs * M$ où $Nprocs$ désigne le nombre de processus MPI et est égal au nombre de blocs diagonaux. Le maître distribuera `A` (et `b`) afin que l'esclave k possède le k^{ieme} bloc ligne de `A` (et `b`).
3. Au cours des itérations chaque processus esclave k ne connaîtra que (et ne pourra communiquer qu'avec) ses processus voisins. (Avec la convention que le voisin de droite de k a pour numéro $k+1$ (ou 1 si $k = Nprocs$) et que le voisin de gauche a pour numéro $k-1$ (ou $Nprocs$ si $k = 1$).) Tout défaut à cette règle (notamment si vous utilisez des communications collectives) devra être clairement explicité.
4. L'esclave k est en charge du calcul de $\mathtt{x}_k$. A chaque étape de l'algorithme itératif il met à jour $\mathtt{x}_k$. Après un échange entre processus voisins des morceaux de solutions mises à jour, il calculera un résidu local $\mathtt{r}_k$ (k^{ieme} bloc ligne de `r`) et sa norme.
5. Comme pour l'algorithme séquentiel, on utilisera la norme max et le même critère de convergence.

4 Travail algorithmique (sur papier)

Écrire l'algorithme **détaillé**. On s'attachera à
- respecter TOUTES les spécifications de l'algorithme (voir section 3 : remarques, structures de données et propriétés),
- décrire l'algorithme retenu pour le calcul de $\mathtt{r}_k$ (on indiquera notamment le nombre de messages envoyés).
- décrire de façon précise les traitements spécifiques liés à des valeurs particulières de $Nprocs$ (par exemple $Nprocs = 1$) et du numéro du processus esclave (par exemple $k = 1$ ou $Nprocs$)

5 Travail sur machine

1. Écrire le programme **MPI** associé à l'algorithme que vous avez specifié.
2. Les bibiothèques d'algèbre linéaire `BLAS` et `LAPACK` vous fourniront les noyaux de calcul dense sur lesquels vous vous appuierez.
3. Vous n'allouerez aucun vecteur sur la pile. Seule l'allocation sur le tas est autorisée et ce par le biais de la fonction **APalloc()** fournie dans *util.c*. Cette fonction encapsule **malloc()** tout en mesurant la consommation mémoire. Voici sa signature :

```
void * APalloc (size_t size);
```

4. On utilisera le type scalaire "double" pour stocker toutes les données réelles.

6 Description des fichiers

Code correspondant à l'implantation de l'algorithme séquentiel et utilitaires fournis :
- **testjacob.c** C'est le programme principal, il génère aléatoirement les données initiales, lance le calcul séquentiel (**jacseq()**), puis le calcul parallèle (**jacpar()**). La solution calculée par chacun des codes est vérifiée, la performance est évaluée et la consommation mémoire mesurée.
- **jacseq.c** (subroutine jacseq).
  Calcul de la solution du système en utilisant l'algorithme bloc-Jacobi séquentiel (Algorithme 1). *C'est le code de référence et il ne faut pas le modifier.*
- **jacpar.c** (subroutine jacpar).
  Calcul de la solution du système en utilisant l'algorithme de bloc-Jacobi parallèle. *C'est le code que vous devrez écrire en respectant la signature de jacpar() initialement fournie.*
- **util.c**
  Ce fichier contient des utilitaires. *A ne pas modifier.*

– **lapack**

Cette bibliothèque contient les codes utiles pour effectuer la factorisation (A = PLU) et la résolution (PLU x = b) de systèmes linéaires pleins. Le code **cblas_dgetrf()** permet d'effectuer la factorisation d'une matrice et le code **cblas_dgetrs()** permet de résoudre un système linéaire. *A ne pas modifier.*
Exemple d'utilisation en (C) sur une matrice D carrée d'ordre M allouée dynamiquement :

```
/* Exemple de resolution d'un systeme lineaire dense D * Y = b
par factorisation LU grace a la bibiotheque Lapack */
#include <clapack.h>
#include ``util.h''
int sampleLU(){
  double *D, *y, *b;
  int *IPIV, M;
  M = 153;
  D = APalloc(M * M * sizeof(double));
  y = APalloc(M * sizeof(double));
  b = APalloc(M * sizeof(double));
  IPIV = malloc(M * sizeof(int));
  ASSERT(D != NULL && y != NULL && b != NULL && IPIV != NULL);
/*      --- Initialiser la matrice et le second membre */
  generateAB(D, b, M, 1, 1.0);
/*      ---------------------------------------------
       factorisation LU de la matrice D (D = P L U)
        ---------------------------------------------*/
  clapack_dgetrf(CblasRowMajor, M, M, D, M, IPIV);
/*         -- En sortie de dgetrf :
C        D(1..M,1..M) contient les matrices L et U
C        IPIV(1..M) contient le vecteur de permutation
C                   associe a la matrice P
C      ---------------------------------------------
C      Resolution : calcul de y tel que  D y = b
C                              (P L U y = b)
C      ---------------------------------------------*/
  cblas_dcopy(M, b, 1, y, 1);
/*      -- En entree y contient le second membre b*/
  clapack_dgetrs(CblasRowMajor, CblasNoTrans, M, 1, D, M, IPIV, y, M);
/*      -- En sortie y contient la solution de D y = b*/
  free(D); free(y); free(b); free(IPIV);
  return 0;
}
```

7 Aller plus loin

Le défaut de la méthode proposée est sa mauvaise scalabilité mémoire. En effet, le processeur maître (processeur 0) possède toujours la matrice A dans son intégralité.

Pour éviter ce goulet d'étranglement mémoire, on peut supposer que la matrice A soit elle-même préalablement calculée de manière distribuée. Écrivez une fonction **generateAB_distentries()** qui génère aléatoirement A et B de manière distribuée. Cette fonction pourra être incluse dans **testjacob.c** et devra respecter la contrainte 1 de la sous-section 3.2. *De plus*, la contrainte suivante devra être respectée :

1. Le second membre b sera distribué sur les processeurs.

Complétez ensuite la fonction **jacpar_distentries()** (incluse dans **jacpar.c**) qui calcule solution du système en utilisant l'algorithme de bloc-Jacobi parallèle avec les données distibuées en entrée.

Les spécifications de la section 3 devront être respectées exceptée la contrainte 1 de la sous-section 3.3 qui *devient* :

1. Aucun processus ne possède entièrement la matrice tri-diagonale A ni le second membre b. Les données (A et b) sont supposées préalablement distibuées. La solution finale x sera elle-même distribuée sur les processeurs.

8 Pièces à rendre

Envoyez-moi par courrier électronique (destinataire : eagullo@ens-lyon.fr ; objet : dmalgopar) une archive contenant les deux fichiers suivants (et uniquement eux) :

1. Votre travail sur papier au format postscript.
2. Le fichier *jacpar.c*.

Notez donc que toute création ou modification d'un fichier autre que *jacpar.c* ne sera pas prise en compte. Il faut donc vous assurer que la compilation et l'exécution de votre programme ne reposent pas sur d'autres fichiers que ceux fournis initialement. L'interface imposée (signatures de **jacpar()** et de **jacpar_distentries()**) devra être scrupuleusement respectée. La fonction **generateAB_distentries()** ne sera pas rendue. En revanche, son inclusion en annexe de votre rapport sera la bienvenue.