Programmation parallèle et distribuée (GIF-4104/7104)

3a - TP #3

Méthode de Gauss-Jordan

(hiver 2014)

UNIVERSITÉ LAVAL

© Marc Parizeau, Département de génie électrique et de génie informatique

Matrice inverse

$$[AI] \Rightarrow A^{-1}[AI] \Rightarrow [I\,A^{-1}].$$

$$\left(A \,\middle|\, I_n\right) = \left(\begin{array}{ccc|ccc} a_{1,1} & \cdots & a_{1,n} & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,n} & 0 & \cdots & 1 \end{array}\right)$$

(Wikipedia)

Algorithme

- l_i^k la ligne i de la matrice A à l'itération k
- a_{ij}^k le scalaire $a_{i\,j}$ de la matrice A à l'itération k

$$\left(\begin{array}{ccc|ccc} a_{1,1} & \cdots & a_{1,n} & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,n} & 0 & \cdots & 1 \end{array}\right)$$

L'algorithme de Gauss-Jordan est le suivant :

Pour k allant de 1 à n

S'il existe une ligne $i \geq k$ telle que $a_{ik}^{k-1} \neq 0$

échanger cette ligne i et la ligne k : $l_i \leftrightarrow l_k$

$$l_k^k \leftarrow \frac{1}{a_{kk}^{k-1}} l_k^{k-1}$$

Pour i allant de 1 à n et $i \neq k$

$$l_i^k \leftarrow l_i^{k-1} - a_{ik}^{k-1} \times l_k^k$$

Étape à paralléliser

(Wikipedia)

Système d'équations

Soit le système d'équations suivant :

$$\begin{cases} x - y + 2z = 5 \\ 3x + 2y + z = 10 \\ 2x - 3y - 2z = -10 \end{cases}$$

On établit la matrice correspondante et on applique la première étape de Gauss-Jordan, le pivot est 1 :

$$\left(\begin{array}{ccc|c} (1) & -1 & 2 & 5 \\ 3 & 2 & 1 & 10 \\ 2 & -3 & -2 & -10 \end{array}\right)$$

On ajoute un multiple de la première ligne aux deux autres lignes pour obtenir des zéros (respectivement $-3 \times l_1$ et $-2 \times l_1$); le nouveau pivot est ensuite 5 :

$$\left(\begin{array}{ccc|c} 1 & -1 & 2 & 5 \\ 0 & (5) & -5 & -5 \\ 0 & -1 & -6 & -20 \end{array}\right)$$

La deuxième ligne est multipliée par 1/5 :

$$\left(\begin{array}{ccc|c} 1 & -1 & 2 & 5 \\ 0 & (1) & -1 & -1 \\ 0 & -1 & -6 & -20 \end{array}\right)$$

On ajoute cette deuxième ligne à la troisième et à la première, le nouveau pivot est -7 :

$$\left(\begin{array}{ccc|c} 1 & 0 & 1 & 4 \\ 0 & 1 & -1 & -1 \\ 0 & 0 & (-7) & -21 \end{array}\right)$$

On divise la 3e ligne par -7 :

$$\left(\begin{array}{ccc|c} 1 & 0 & 1 & 4 \\ 0 & 1 & -1 & -1 \\ 0 & 0 & (1) & 3 \end{array}\right)$$

On utilise la 3e ligne pour éliminer des coefficients dans la première et deuxième ligne. Nous sommes alors en présence d'une forme échelonnée réduite avec la matrice identité d'un côté et la valeur des variables de l'autre :

$$\left(\begin{array}{ccc|c} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \end{array}\right)$$

(Wikipedia)

Élimination de Gauss

$$A^{(k)} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1,k-1} & a_{1k} & \cdots & a_{1n} \\ 0 & a_{22}^{(2)} & \cdots & a_{2,k-1}^{(2)} & a_{2k}^{(2)} & \cdots & a_{2n}^{(2)} \\ \vdots & \ddots & \ddots & \vdots & \vdots & & \vdots \\ \vdots & & \ddots & a_{k-1,k-1}^{(k-1)} & a_{k-1,k}^{(k-1)} & \cdots & a_{k-1,n}^{(k-1)} \\ \vdots & & & 0 & a_{kk}^{(k)} & \cdots & a_{kn}^{(k)} \\ \vdots & & & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 & a_{nk}^{(k)} & \cdots & a_{nn}^{(k)} \end{bmatrix}.$$

Parallel programming, Rauber & Rünger

$$l_{ik} = a_{ik}^{(k)} / a_{kk}^{(k)}, \qquad i = k+1, \ldots, n \ . \tag{7.2}$$

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - l_{ik} a_{kj}^{(k)}, \tag{7.3}$$

$$b_i^{(k+1)} = b_i^{(k)} - l_{ik} b_k^{(k)} \tag{7.4}$$

$$x_k = \frac{1}{a_{kk}^{(n)}} \left(b_k^{(n)} - \sum_{j=k+1}^{n} a_{kj}^{(n)} x_j \right) . \tag{7.5}$$

Parallel programming, Rauber & Rünger

Fig. 7.1

```
double *gauss_sequential (double **a, double *b)
{
  double *x, sum, l[MAX_SIZE];
  int i,j,k,r;

  x = (double *) malloc(n * sizeof(double));
  for (k = 0; k < n-1; k++) { /* Forward elimination */
     r = max_col(a,k);
     if (k != r) exchange_row(a,b,r,k);
     for (i=k+1; i < n; i++) {
        l[i] = a[i][k]/a[k][k];
        for (j=k+1; j < n; j++)
           a[i][j] = a[i][j] - l[i] * a[k][j];
        b[i] = b[i] - l[i] * b[k];
     }
  }
  for (k = n-1; k >= 0; k--) { /* Backward substitution */
     sum = 0.0;
     for (j=k+1; j < n; j++)
        sum = sum + a[k][j] * x[j];
     x[k] = 1/a[k][k] * (b[k] - sum);
  }
  return x;
}
```

1. **Determination of the local pivot element:** Each processor considers its local elements of column k in the rows $k, \ldots, n$ and determines the element (and its position) with the largest absolute value.
2. **Determination of the global pivot element:** The global pivot element is the local pivot element which has the largest absolute value. A single-accumulation operation with the maximum operation as reduction determines this global pivot element. The root processor of this global communication operation sends the result to all other processors.
3. **Exchange of the pivot row:** If $k \neq r$ for a pivot element $a_{rk}^{(k)}$, the row k owned by processor P_q and the pivot row r owned by processor $P_{q'}$ have to be exchanged. When $q = q'$, the exchange can be done locally by processor P_q. When $q \neq q'$, then communication with single transfer operations is required. The elements b_k and b_r are exchanged accordingly.
4. **Distribution of the pivot row:** Since the pivot row (now row k) is required by all processors for the local elimination operations, processor P_q sends the elements $a_{kk}^{(k)}, \ldots, a_{kn}^{(k)}$ of row k and the element $b_k^{(k)}$ to all other processors.
5. **Computation of the elimination factors:** Each processor locally computes the elimination factors l_{ik} for which it owns the row i according to Formula (7.2).
6. **Computation of the matrix elements:** Each processor locally computes the elements of $A^{(k+1)}$ and $b^{(k+1)}$ using its elements of $A^{(k)}$ and $b^{(k)}$ according to Formulas (7.3) and (7.4).

Fig. 7.2

```
double *gauss_cyclic (double **a, double *b)
{
  double *x, l[MAX_SIZE], *buf;
  int i,j,k,r, tag=42;
  MPI_Status status;
  struct { double val; int node; } z,y;
  x = (double *) malloc(n * sizeof(double));
  buf = (double *) malloc((n+1) * sizeof(double));
  for (k=0 ; k<n-1 ; k++) { /* Forward elimination */
    r = max_col_loc(a,k);
    z.node = me;
    if (r != -1) z.val = fabs(a[r][k]); else z.val = 0.0;
    MPI_Allreduce(&z,&y,1,MPI_DOUBLE_INT,MPI_MAXLOC,MPI_COMM_WORLD);
    if (k % p == y.node){ /* Pivot row and row k are on the same processor */
       if (k % p == me) {
          if (a[k][k] != y.val) exchange_row(a,b,r,k);
          copy_row(a,b,k,buf);
       }
    }
    else /* Pivot row and row k are owned by different processors */
      if (k % p == me) {
        copy_row(a,b,k,buf);
        MPI_Send(buf+k,n-k+1,MPI_DOUBLE,y.node,tag,
                             MPI_COMM_WORLD);
      }
      else if (y.node == me) {
        MPI_Recv(buf+k,n-k+1,MPI_DOUBLE,MPI_ANY_SOURCE,
                             tag,MPI_COMM_WORLD,&status);
        copy_exchange_row(a,b,r,buf,k);
      }
```

```
    MPI_Bcast(buf+k,n-k+1,MPI_DOUBLE,y.node,MPI_COMM_WORLD);
    if ((k % p != y.node) && (k % p == me)) copy_back_row(a,b,buf,k);
    i = k+1; while (i % p != me) i++;
    for ( ; i<n; i+=p) {
      l[i] = a[i][k] / buf[k];
      for (j=k+1; j<n; j++)
        a[i][j] = a[i][j] - l[i]*buf[j];
      b[i] = b[i] - l[i]*buf[n];
    }
  }
  for (k=n-1; k>=0; k--) { /* Backward substitution */
    if (k % p == me) {
      sum = 0.0;
      for (j=k+1; j < n; j++)  sum = sum + a[k][j] * x[j];
      x[k] = 1/a[k][k] * (b[k] - sum); }
    MPI_Bcast(&x[k],1,MPI_DOUBLE,k%p,MPI_COMM_WORLD);
  }
  return x;
}
```

- ✓ ***n*** désigne la dimension de la matrice (***n*** lignes par ***n*** colonnes)
- ✓ ***i*** et ***j*** sont des indices de ligne et de colonne respectivement
- ✓ ***k*** est l'indice d'étape de l'algorithme (***n*** étapes au total)
- ✓ ***q*** est l'indice du max (en valeur absolue) dans la colonne ***k***
- ✓ ***r*** est le rang du processus, et ***p*** le nombre total de processus
- ✓ la ligne ***i*** appartient au processus ***r*** si ***i***%***p***=***r*** (décomposition «*row-cyclic*»)

1. Pour chaque étape ***k***: 0..***n***-1 de l'algorithme
   a. Déterminer localement le ***q*** parmi les lignes qui appartiennent à ***r***, puis faire une réduction (All*reduce* avec *MAXLOC*) pour déterminer le ***q*** global
   b. Si la valeur du max est nulle, la matrice est singulière (ne peut être inversée)
   c. Diffuser (B*cast*) la ligne ***q*** appartenant au processus ***r***=***q***%***p*** (***r*** est root)
   d. Permuter localement les lignes ***q*** et ***k***
   e. Normaliser la ligne ***k*** afin que l'élément (***k***,***k***) égale 1
   f. Éliminer les éléments (***i***,***k***) pour toutes les lignes ***i*** qui appartiennent au processus ***r***, sauf pour la ligne ***k***
2. Rapatrier (G*atherv*) toutes les lignes sur le processus 0