***Répartition statique en bloc de lignes***

taille\_bloc = n/p

**Si(my\_rank!=0)**

int fini

MPI\_Recv(fini,0,0)

Tant que non fini

double \*bloc=malloc(taille\_bloc\*n\*size(double)+n\*size(double))

double \*Y = (double)malloc(taille\_bloc\*size(double)+size(double))

MPI\_Recv(bloc,0,1)

Pour i allant de 0 à taille\_bloc

res[i]=0

double norm =0

Pour j allant de 0 à n

Y[i] += A\_i[j] \* X[j]

A\_i += n;

norm += Y[i]^2

MPI\_Send(norm,0,0)

MPI\_Recv(norm,0,2)

double erreur =0

Pour i allant de 0 à taille\_bloc

Y[i] = Y[i]/norm

delta = X[i] - Y[i]

erreur += delta\*delta

MPI\_Send(erreur,0,3)

MPI\_Recv(fini,0,0)

tmp = X

X = Y

Y = tmp

Fin TantQue

MPI\_Send(X,0,4)

Fin Si

Sinon

Pour i allant de 0 à p

MPI\_send(A[i\*n\*taille\_bloc:(i+1)\*n\*taille\_bloc],i,1)

erreur = 1

Tant que erreur>1E^-9

norme = 0

erreur=0

Pour i allant de 0 à p

MPI\_Recv(norm\_inter,i,1)

norm += norm\_inter

Fin Pour

norm = sqrt(norm)

MPI\_IBcast(norm,,2) #broadcast non bloquant pour continuer les calculs

Pour i allant de 0 à p

MPI\_Recv(erreur\_inter,i,1)

erreur += erreur\_inter

Fin Pour

MPI\_IBcast(erreur ,,3)#broadcast non bloquant pour continuer les calculs

Fin Tantque

X = (double\*)maloc(n\*sizeof(double))

bloc = (double\*)maloc(taille\_bloc\*sizeof(double))

Pour i allant de 0 à p

MPI\_Recv(bloc+i\*taille\_bloc,i,0)

Fin Pour

Affichage Résultat

Fin Sinon

FIN ALGO

**Matrice A[n,n] ;**

**Vecteur x[n], y[n] ;**

**Tant que erreur > 10^{-9} :**

**y = A.x / norme2(A.x)**

/\* Partie la plus coûteuse

Pour chaque proc : reçoit bloc de ligne A[ip à ip+1]

+ bloc de ligne x[ip à ip+1] du processeur 0

calcule y[ip à ip+1]

calcule une partie de la norm : y[ip à ip+1]²

envoie y[ip à ip+1]² au processeur 0

Processeur 0 : reçoit y[ip à ip+1]² de tous les processeurs

calcule (au fur et à mesure des réceptions) la somme des y[ip à ip+1]² pour avoir y²

calcule norm = racine\_carré(y²)

envoie norm à tous les processeurs

Pour chaque proc : reçoit norm du processeur 0

calcule y[ip à ip+1] = y[ip à ip+1] / norm

\*/

**erreur = norme2(x - y)**

/\*

Pour chaque proc : calcule une partie de l’erreur :

(x[ip à ip+1]-y[ip à ip+1])²

envoie (x[ip à ip+1]-y[ip à ip+1])² au processeur 0

Processeur 0 : reçoit (x[ip à ip+1]-y[ip à ip+1])²

de tous les processeurs

calcule (au fur et à mesure des réceptions) la somme des (x[ip à ip+1]-y[ip à ip+1])²

pour avoir (x[i]-y[i])²

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

**Si** erreur > 10-9 alors

envoie erreur à tous les processeurs

**Sinon** Arrêt des processeurs :

Arrêt des processeurs :

\*/

**x <- y**

/\*

Pour chaque proc : reçoit erreur du processeur 0

remplacer les valeurs de x[ip à ip+1]

par celles de y[ip à ip+1]

\*/

**Fin tant que**

Afficher: le nombre d’itérations, l’erreur et norme2(A.x) = norme2(λx) = |λ|

Sauvegarder: n et x dans “result.out”

==============================================================================================================================================

/\* indent -nfbs -i4 -nip -npsl -di0 -nut iterated\_seq.c \*/

/\* Auteur: C. Bouillaguet et P. Fortin (Univ. Lille) \*/

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <sys/time.h>

#define PRNG\_MAX 0x0007FFFFFFFll

#define PRNG\_1 0x00473EE661Dll

#define PRNG\_2 0x024719D0275ll

#define RANGE 101

double my\_gettimeofday(){

struct timeval tmp\_time;

gettimeofday(&tmp\_time, NULL);

return tmp\_time.tv\_sec + (tmp\_time.tv\_usec \* 1.0e-6L);

}

// Initialise la ligne N°i (pointée par A\_i) de la matrice de taille n x n :

void init\_ligne(double \*A\_i, long i, long n){

for (long j = 0; j < n; j++) {

A\_i[j] = (((double)((i \* i \* PRNG\_1 + j \* j \* PRNG\_2) & PRNG\_MAX)) / PRNG\_MAX) / n;

}

for (long k = 1; k < n; k \*= 2) {

if (i + k < n) {

A\_i[i + k] = ((i - k) \* PRNG\_2 + i \* PRNG\_1) % RANGE;

}

if (i - k >= 0) {

A\_i[i - k] = ((i + k) \* PRNG\_2 + i \* PRNG\_1) % RANGE;

}

}

}

int main(int argc, char \*\*argv){

long i, j, n;

long long size;

double norm, inv\_norm, error, start\_time, total\_time, delta;

double \*A, \*A\_i, \*X, \*Y;

int n\_iterations;

FILE \*output;

if (argc < 2) {

printf("USAGE: %s [n]\n", argv[0]);

exit(1);

}

n = atoll(argv[1]);

size = n \* n \* sizeof(double);

printf("taille de la matrice : %.1f G\n", size / 1073741824.);

/\*\*\* allocation de la matrice et des vecteurs \*\*\*/

A = (double \*)malloc(size);

if (A == NULL) {

perror("impossible d'allouer la matrice");

exit(1);

}

X = malloc(n \* sizeof(double));

Y = malloc(n \* sizeof(double));

if ((X == NULL) || (Y == NULL)) {

perror("impossible d'allouer les vecteur");

exit(1);

}

/\*\*\* initialisation de la matrice et de x \*\*\*/

A\_i = A;

for (i = 0; i < n; i++) {

init\_ligne(A\_i, i, n);

A\_i += n;

}

for (i = 0; i < n; i++) {

X[i] = 1.0 / n;

}

start\_time = my\_gettimeofday();

error = INFINITY;

n\_iterations = 0;

#tag=0 -> termine,1->puissance,2->normalisation\_part1,

Si(my\_rank!=0)

double \*val1=(double)malloc(n\*sizeof(double),val2;

int fini = MPI\_Recv(&fini,source = 0,tag = 0)

Tant que non fini

MPI\_Recv(,source=0,MPI\_ANY\_TAG,status)

Si status.MPI\_TAG==

while (error > 1e-9) {

printf("iteration %4d, erreur actuelle %g\n", n\_iterations, error);

------DIFFICILEMENT PARALL2LISABLE-------------

/\*\*\* y <--- A.x \*\*\*/

A\_i = A;

for (i = 0; i < n; i++) {

Y[i] = 0;

-----------------PARTIE NON PARALL2LISABLE-----------------

for (j = 0; j < n; j++) {

Y[i] += A\_i[j] \* X[j];

}

-----------------------------------------------------------

A\_i += n; /\*Avancer d’une ligne\*/

}

------FACILEMENT PARALL2LISABLE-------------

MPI any source

/\*\*\* norm <--- ||y|| \*\*\*/

norm = 0;

for (i = 0; i < n; i++) {

norm += Y[i] \* Y[i];

}

---------------------------------------------

norm = sqrt(norm);

------FACILEMENT PARALL2LISABLE-------------

/\*\*\* y <--- y / ||y|| \*\*\*/

inv\_norm = 1.0 / norm;

for (i = 0; i < n; i++) {

Y[i] \*= inv\_norm;

}

---------------------------------------------

------FACILEMENT PARALL2LISABLE-------------

MPI any source

/\*\*\* error <--- ||x - y|| \*\*\*/

error = 0;

for (i = 0; i < n; i++) {

delta = X[i] - Y[i];

error += delta \* delta;

}

---------------------------------------------

error = sqrt(error);

/\*\*\* x <--> y \*\*\*/

double \*tmp = X; X = Y ; Y = tmp;

n\_iterations++;

}

total\_time = my\_gettimeofday() - start\_time;

printf("erreur finale après %4d iterations: %g (|VP| = %g)\n", n\_iterations, error, norm);

printf("time : %.1f s MFlops : %.1f \n", total\_time, (2.0 \* n \* n + 7.0 \* n) \* n\_iterations / 1048576. / total\_time);

/\*\*\* stocke le vecteur propre dans un fichier \*\*\*/

output = fopen("result.out", "w");

if (output == NULL) {

perror("impossible d'ouvrir result.out en écriture");

exit(1);

}

fprintf(output, "%ld\n", n);

for (i = 0; i < n; i++) {

fprintf(output, "%.17g\n", X[i]);

}

fclose(output);

free(A);

free(X);

free(Y);

}