Лабораторная работа №1

«Параллельная реализация решения системы линейных алгебраических уравнений с помощью MPI»

Вариант 1: векторы x и b дублируются в каждом MPI-процессе.

#include<stdio.h>

#include<mpi.h>

#include<math.h>

#include<stdlib.h>

#include<time.h>

void dotProduct(double\* a , double\* b , int size , double\* dest){

double s = 0;

for(int i = 0; i < size; ++i){

s+=(a[i] \* b[i]);

}

\*dest = s;

}

void absoluteValue(double \*a , int size , double\* dest){

double d = 0;

dotProduct(a , a , size , &d);

\*dest = sqrt(d);

}

void multiMatVec(double\* A , double\* x , double\* dest , int A\_rows , int A\_col){

for(int r = 0; r < A\_rows; ++r){

double s = 0;

for(int c = 0; c < A\_col; ++c){

s+=(A[r \* A\_col + c] \* x[c]);

}

dest[r] = s;

}

}

void subtractVec(double\* a , double\* b , double\* dest , int size){

for(int i = 0; i < size; ++i){

dest[i] = (a[i] - b[i]);

}

}

void addVec(double\* a , double\* b , double\* dest , int size){

for(int i = 0; i < size; ++i){

dest[i] = (a[i] + b[i]);

}

}

void fillVector(double\* v , int size , double value){

for (int i = 0; i < size; ++i)

{

v[i] = value;

}

}

void multiOnScalar(double\* v , double scalar , int size){

for (int i = 0; i < size; ++i)

{

v[i]\*=scalar;

}

}

void showVector(double\* v , int size){

for (int i = 0; i < size; ++i)

{

printf("%lf\n", v[i]);

}

}

double checkAccuracy(double\* Ax , double\* b , int size){

double a1 = 0 , a2 = 0;

absoluteValue(Ax , size , &a1);

absoluteValue(b , size , &a2);

if(0 == a2){

printf("|b| = 0?\n");

exit(0);

}

return a1 / a2;

}

void fillVectorM3(double\* v , int size){

for (int i = 0; i < size; ++i)

{

v[i] = 1;

if(!(i%3)) v[i] = (1 - 2\*(i%2))\*50;

else v[i] = 0;

}

}

void fillVectorM3Grid(double\* grid , double\* b , int N\_x , int N\_y){

for (int i = 0; i < N\_y; ++i)

{

for (int j = 0; j < N\_x; ++j)

{

grid[i \* N\_x + j] = b[i \* N\_x + j];

}

}

}

void fillMatrixM3(double\* m , int r , int c , int l\_s\_index , int N\_x){

for(int i = 0; i < r; ++i){

for(int j = 0; j < c; ++j){

if(j == l\_s\_index + i){

m[i \* c + j] = -4;

}else if(j == l\_s\_index + i + N\_x || j == l\_s\_index + i - N\_x){

m[i \* c + j] = 1;

}else if(j == l\_s\_index + i + 1){

if(j && (j % N\_x == 0)){

m[i \* c + j] = 0;

}else{

m[i \* c + j] = 1;

}

}else if(j == l\_s\_index + i - 1){

if(j && ((j + 1) % N\_x == 0)){

m[i \* c + j] = 0;

}else{

m[i \* c + j] = 1;

}

}

else{

m[i \* c + j] = 0;

}

}

}

}

void showMatrix(double\* m , int r , int c){

for (int i = 0; i < r; ++i)

{

for (int j = 0; j < c; ++j)

{

printf("%d ", (int)m[i \* c + j]);

}

printf("\n");

}

}

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

MPI\_Init(&argc , &argv);

srand(time(0));

if(argc != 3){

printf("N e a\n");

return 0;

}

int N\_x = atoi(argv[1]);

int N\_y = atoi(argv[2]);

if(N\_x <= 0 || N\_y <= 0){

printf("Wrong Args\n");

return 0;

}

int MATRIX\_SIZE = N\_x \* N\_y;

int m\_rank = 0 , p\_count = 0;

MPI\_Comm\_size(MPI\_COMM\_WORLD , &p\_count);

MPI\_Comm\_rank(MPI\_COMM\_WORLD , &m\_rank);

if(0 == m\_rank){

printf("Process count: %d\n", p\_count);

}

int task\_per\_process = MATRIX\_SIZE / p\_count;

int overflow\_tasks = MATRIX\_SIZE % p\_count;

int l\_s\_index = 0 , l\_e\_index = 0;

l\_s\_index = m\_rank \* task\_per\_process;

l\_e\_index = l\_s\_index + task\_per\_process - 1;

int\* recvcounts = (int\*)malloc(sizeof(int) \* p\_count);

int\* displs = (int\*)malloc(sizeof(int) \* p\_count);

int i\_sum = 0;

for (int i = 0; i < p\_count; ++i)

{

double s = i \* task\_per\_process + (i < overflow\_tasks ? i : overflow\_tasks);

double e = s + task\_per\_process - 1 + (i < overflow\_tasks ? 1 : 0);

displs[i] = i\_sum;

recvcounts[i] = e - s + 1;

i\_sum+=recvcounts[i];

if(m\_rank == i){

l\_s\_index = s;l\_e\_index = e;

}

}

int l\_matrix\_r = (l\_e\_index - l\_s\_index + 1);

int l\_matrix\_c = MATRIX\_SIZE;

double\* l\_matrix = (double\*)malloc(l\_matrix\_c \* l\_matrix\_r \* sizeof(double));

double\* l\_vec\_keepAx = (double\*)malloc(sizeof(double) \* l\_matrix\_r);

double\* vec\_x = (double\*)malloc(sizeof(double) \* MATRIX\_SIZE);

double\* vec\_b = (double\*)malloc(sizeof(double) \* MATRIX\_SIZE);

double\* vec\_Gathered\_y = (double\*)malloc(sizeof(double) \* MATRIX\_SIZE);

double\* vec\_Gathered = (double\*)malloc(sizeof(double) \* MATRIX\_SIZE);

double t\_factor = 0;

/////////TempTest

fillVectorM3(vec\_b , MATRIX\_SIZE);

fillMatrixM3(l\_matrix , l\_matrix\_r , l\_matrix\_c , l\_s\_index , N\_x);

fillVector(vec\_x , MATRIX\_SIZE , 0);

double start\_t = MPI\_Wtime() , end\_t = 0;

int end = 0;

int count = 0;

while(1){

++count;

multiMatVec(l\_matrix , vec\_x , l\_vec\_keepAx , l\_matrix\_r , l\_matrix\_c);

//Reduce

MPI\_Allgatherv(l\_vec\_keepAx , l\_matrix\_r , MPI\_DOUBLE,vec\_Gathered ,

recvcounts , displs , MPI\_DOUBLE, MPI\_COMM\_WORLD);

//calculate y(n)

subtractVec(vec\_Gathered , vec\_b , vec\_Gathered\_y , MATRIX\_SIZE);

double check = 1;

if(0 == m\_rank){

check = checkAccuracy(vec\_Gathered\_y , vec\_b , MATRIX\_SIZE);

if(check < 0.00001){

end = 1;

printf("res: %lf\n", check);

}else{

//printf("%lf\n", check);

}

}

MPI\_Bcast(&end , 1 , MPI\_INT, 0 , MPI\_COMM\_WORLD);

if(end){

end\_t = MPI\_Wtime();

if(0 == m\_rank){

//showVector(vec\_x , MATRIX\_SIZE);

printf("Time taken: %f , Iterations: %d \n", end\_t - start\_t , count);

}

{

free(l\_matrix);free(l\_vec\_keepAx);

free(vec\_x);free(vec\_b);

free(vec\_Gathered);free(vec\_Gathered\_y);

}

MPI\_Finalize();

return 0;

}

//calculate A \* y(n)

multiMatVec(l\_matrix , vec\_Gathered\_y , l\_vec\_keepAx , l\_matrix\_r , l\_matrix\_c);

MPI\_Allgatherv(l\_vec\_keepAx , l\_matrix\_r , MPI\_DOUBLE,vec\_Gathered ,

recvcounts , displs , MPI\_DOUBLE, MPI\_COMM\_WORLD);

double dotpr = 0;

dotProduct(vec\_Gathered\_y , vec\_Gathered , MATRIX\_SIZE , &dotpr);

t\_factor = dotpr;

dotProduct(vec\_Gathered , vec\_Gathered , MATRIX\_SIZE , &dotpr);

t\_factor/=dotpr;

multiOnScalar(vec\_Gathered\_y , t\_factor , MATRIX\_SIZE);

subtractVec(vec\_x , vec\_Gathered\_y , vec\_x , MATRIX\_SIZE);

}

return 0;

}

Вариант 2: векторы x и b разрезаются между MPI-процессами.

#include<stdio.h>

#include<mpi.h>

#include<math.h>

#include<stdlib.h>

#include<time.h>

void multiMatVecAdd(double\* A , double\* x , double\* dest , int A\_rows , int A\_col , int r\_diff , int c\_diff){

for(int r = 0; r < A\_rows; ++r){

double s = 0;

for(int c = 0; c < A\_col; ++c){

s+=(A[r \* r\_diff + c\_diff + c] \* x[c]);

}

dest[r]+=s;

}

}

void subtractVec(double\* a , double\* b , double\* dest , int size){

for(int i = 0; i < size; ++i){

dest[i] = (a[i] - b[i]);

}

}

void multiOnScalar(double\* v , double scalar , int size){

for (int i = 0; i < size; ++i)

{

v[i]\*=scalar;

}

}

void fillVector(double\* v , int size , double value){

for (int i = 0; i < size; ++i)

{

v[i] = value;

}

}

void showMatrix(double\* A , int r , int c){

for (int i = 0; i < r; ++i)

{

for (int j = 0; j < c; ++j)

{

printf("%lf ", A[c \* i + j]);

}

printf("\n");

}

}

void showVector(double\* v , int size){

for (int i = 0; i < size; ++i)

{

printf("%lf\n", v[i]);

}

}

void dotProduct(double\* a , double\* b , int size , double\* dest , int ra){

double s = 0;

for(int i = 0; i < size; ++i){

s+=(a[i] \* b[i]);

}

\*dest = s;

}

void subToTest(double\* a , double\* b , double\* dest , int size , int b\_start){

for (int i = 0; i < size; ++i)

{

dest[i] = (a[i] - b[i + b\_start]);

}

}

void multiMatVecTest(double\* A , double\* x , double\* dest , int A\_rows , int A\_col){

for(int r = 0; r < A\_rows; ++r){

double s = 0;

for(int c = 0; c < A\_col; ++c){

s+=(A[r \* A\_col + c] \* x[c]);

}

dest[r] = s;

}

}

void multi(double\* l\_matrix , double\* l\_vec\_ , double\* l\_vec\_dest ,\

int\* recvcounts , int\* displs , int l\_matrix\_r , int l\_matrix\_c , int m\_rank , int p\_count , int task\_p\_p){

int block\_i = 0 , exc\_size = task\_p\_p + 1;

MPI\_Status st[2];

for (int i = 0; i < p\_count; ++i)

{

block\_i = (m\_rank + i) % p\_count;

multiMatVecAdd(l\_matrix , l\_vec\_ , l\_vec\_dest , l\_matrix\_r , recvcounts[block\_i] , l\_matrix\_c , displs[block\_i]);

if(p\_count > 1){

MPI\_Sendrecv\_replace(l\_vec\_, exc\_size, MPI\_DOUBLE, (m\_rank-1+p\_count)%p\_count,12345, (m\_rank + 1)%p\_count, 12345, MPI\_COMM\_WORLD, st);

}

}

}

void absolute\_value(double\* l\_vec\_ , double\* dest , int size){

double sum = 0;

for (int i = 0; i < size; ++i){

sum+=(l\_vec\_[i] \* l\_vec\_[i]);

}

MPI\_Reduce(&sum , dest , 1 , MPI\_DOUBLE , MPI\_SUM, 0, MPI\_COMM\_WORLD);

\*dest = sqrt(\*dest);

}

void fillVectorM3(double\* v , int size , int s\_index){

for (int j = 0; j < size; ++j)

{

int i = s\_index + j;

if(!(i%3)) v[j] = (1 - 2\*(i%2)) \* 50;

else v[j] = 0;

}

}

void fillVectorM3Grid(double\* grid , double\* b , int size){

for (int i = 0; i < size; ++i)

{

grid[i] = b[i];

}

}

void fillMatrixM3(double\* m , int r , int c , int l\_s\_index , int N\_x){

for(int i = 0; i < r; ++i){

for(int j = 0; j < c; ++j){

if(j == l\_s\_index + i){

m[i \* c + j] = -4;

}else if(j == l\_s\_index + i + N\_x || j == l\_s\_index + i - N\_x){

m[i \* c + j] = 1;

}else if(j == l\_s\_index + i + 1){

if(j && (j % N\_x == 0)){

m[i \* c + j] = 0;

}else{

m[i \* c + j] = 1;

}

}else if(j == l\_s\_index + i - 1){

if(j && ((j + 1) % N\_x == 0)){

m[i \* c + j] = 0;

}else{

m[i \* c + j] = 1;

}

}

else{

m[i \* c + j] = 0;

}

}

}

}

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

MPI\_Init(&argc , &argv);

srand(time(0));

if(argc != 3){

printf("N e a\n");

return 0;

}

int N\_x = atoi(argv[1]);

int N\_y = atoi(argv[2]);

if(N\_x <= 0 || N\_y <= 0){

printf("Wrong Args\n");

return 0;

}

int MATRIX\_SIZE = N\_x \* N\_y;

int m\_rank = 0 , p\_count = 0;

MPI\_Comm\_size(MPI\_COMM\_WORLD , &p\_count);

MPI\_Comm\_rank(MPI\_COMM\_WORLD , &m\_rank);

if(0 == m\_rank){

printf("Process count: %d\n", p\_count);

}

int task\_per\_process = MATRIX\_SIZE / p\_count;

int overflow\_tasks = MATRIX\_SIZE % p\_count;

int l\_s\_index = 0 , l\_e\_index = 0;

l\_s\_index = 0;

l\_e\_index = 0;

int\* recvcounts = (int\*)malloc(sizeof(int) \* p\_count);

int\* displs = (int\*)malloc(sizeof(int) \* p\_count);

int i\_sum = 0;

for (int i = 0; i < p\_count; ++i)

{

double s = i \* task\_per\_process + (i < overflow\_tasks ? i : overflow\_tasks);

double e = s + task\_per\_process - 1 + (i < overflow\_tasks ? 1 : 0);

displs[i] = i\_sum;

recvcounts[i] = e - s + 1;

i\_sum+=recvcounts[i];

if(m\_rank == i){

l\_s\_index = s;l\_e\_index = e;

}

}

int l\_matrix\_r = (l\_e\_index - l\_s\_index + 1);

int l\_matrix\_c = MATRIX\_SIZE;

int exc\_size = task\_per\_process + 1;

double\* l\_matrix = (double\*)malloc(l\_matrix\_c \* l\_matrix\_r \* sizeof(double));

double\* l\_vec\_x = (double\*)malloc(sizeof(double) \* exc\_size);

double\* l\_vec\_y = (double\*)malloc(sizeof(double) \* exc\_size);

double\* l\_vec\_b = (double\*)malloc(sizeof(double) \* exc\_size);

double\* l\_vec\_dest = (double\*)malloc(sizeof(double) \* exc\_size);

double\* l\_vec\_recv = (double\*)malloc(sizeof(double) \* exc\_size);

double\* answer = (double\*)malloc(sizeof(double) \* MATRIX\_SIZE);

double VEC\_B\_ABSOLUTE = 1;

//TempTest

fillVectorM3(l\_vec\_b , l\_matrix\_r , l\_s\_index);

fillMatrixM3(l\_matrix , l\_matrix\_r , l\_matrix\_c , l\_s\_index , N\_x);

fillVector(l\_vec\_x , l\_matrix\_r, 0);

absolute\_value(l\_vec\_b , &VEC\_B\_ABSOLUTE , l\_matrix\_r);

double start\_t = MPI\_Wtime() , end\_t = 0;

double t\_factor = 0;

int end = 0;

int count = 0;

while(1){

++count;

fillVector(l\_vec\_y , exc\_size , 0);

multi(l\_matrix , l\_vec\_x, l\_vec\_y ,recvcounts , displs , l\_matrix\_r\

, l\_matrix\_c , m\_rank , p\_count , task\_per\_process);

subtractVec(l\_vec\_y , l\_vec\_b , l\_vec\_y , l\_matrix\_r);

double a = 0;

absolute\_value(l\_vec\_y , &a , l\_matrix\_r);

if(0 == m\_rank){

if(a / VEC\_B\_ABSOLUTE < 0.00001){

end = 1;

printf("Acc: %lf\n" , a / VEC\_B\_ABSOLUTE);

}

//printf("Acc: %lf\n" , a / VEC\_B\_ABSOLUTE);

}

MPI\_Bcast(&end , 1 , MPI\_INT, 0 , MPI\_COMM\_WORLD);

if(end){

end\_t = MPI\_Wtime();

if(0 == m\_rank){

printf("Time taken: %f Iterations: %d \n" , end\_t - start\_t , count);

}

{

free(l\_matrix); free(l\_vec\_x);

free(l\_vec\_y); free(l\_vec\_dest);

free(l\_vec\_recv);free(l\_vec\_b);

free(answer);

}

MPI\_Finalize();

return 0;

}

fillVector(l\_vec\_dest , exc\_size , 0);

multi(l\_matrix , l\_vec\_y, l\_vec\_dest , recvcounts , displs\

, l\_matrix\_r , l\_matrix\_c , m\_rank , p\_count , task\_per\_process);

double l\_d1 = 0 , l\_d2 = 0 , d1 = 0 , d2 = 0;

dotProduct(l\_vec\_y , l\_vec\_dest , l\_matrix\_r , &l\_d1 , m\_rank);

dotProduct(l\_vec\_dest , l\_vec\_dest , l\_matrix\_r , &l\_d2 , m\_rank);

MPI\_Reduce(&l\_d1 , &d1 , 1 , MPI\_DOUBLE , MPI\_SUM, 0, MPI\_COMM\_WORLD);

MPI\_Reduce(&l\_d2 , &d2 , 1 , MPI\_DOUBLE , MPI\_SUM, 0, MPI\_COMM\_WORLD);

if(0 == m\_rank){

t\_factor = d1 / d2;

}

MPI\_Bcast(&t\_factor , 1 , MPI\_DOUBLE, 0 , MPI\_COMM\_WORLD);

multiOnScalar(l\_vec\_y , t\_factor , l\_matrix\_r);

subtractVec(l\_vec\_x , l\_vec\_y , l\_vec\_x , l\_matrix\_r);

}

return 0;

}

Зависимости времени работы программы от числа используемых ядер

Зависимости ускорения от числа используемых ядер

Зависимость эффективности распараллеливания программы от числа используемых ядер

Вывод: