#include <iostream>

#include <fstream>

#include <cstdio>

#include <cmath>

#include <ctime>

#include <mpi.h>

using namespace std;

extern "C" {

// для сетки процессоров

void Cblacs\_get(int, int, int \*);

void Cblacs\_gridinit(int \*, const char \*, int, int);

void Cblacs\_pcoord(int, int, int \*, int \*);

void Cblacs\_gridexit(int);

//void Cblacs\_gridinfo(int\*, int\*, int\*, int\*, int\*);

// для распределения матрицы по процессорам

void Cdgerv2d(int, int, int, double \*, int, int, int);

void Cdgesd2d(int, int, int, double \*, int, int, int);

void descinit\_(int \*, int \*, int \*, int \*, int \*, int \*, int \*, int \*, int \*, int \*); // для инициализации матриц (вспом.)

int numroc\_(int \*, int \*, int \*, int \*, int \*); // вычисляет количество строк и столбцов распределенной матрицы на опред. проц.

// scalapack

void pdgehrd\_(int \*, int \*, int \*, double \*, int \*, int \*, int \*, double \*, double \*, int \*, int \*); // приведение матрицы к хессенберговой форме

void pdhseqr\_(char \*, char \*, int \*, int \*, int \*, double \*, int \*, double \*, double \*, double \*, int \*, double \*, int \*, int \*, int \*, int \*); // вычисление собст.зн.матрицы

}

int sizeOfMatrix; // размер матрицы, матрица квадратная (по условию задачи)

int countElements; // общее количество элементов матрицы (sizeOfMatrix \* sizeOfMatrix)

double \* matrix; // матрица

double \* eigenRealValues; // вещественные части собственных значений

double \* eigenImagineValues; // мнимые части собственных значений

void readFile(int argc, char \*\* argv) {

if (argc < 2) {

throw "Please enter source file name";

}

FILE \* inputFile;

if((inputFile=fopen(argv[1], "r"))==NULL) {

throw "Input file is not exist. Please enter right name of input file";

}

fread(&sizeOfMatrix, sizeof(int), 1, inputFile);

countElements = sizeOfMatrix \* sizeOfMatrix;

matrix = new double[countElements];

/\*for (int k = 0; k < countElements; k++) {

if (ftell(inputFile) == SEEK\_END) {

throw "Too few data in input file. Please check the data";

}

double forRead;

fread(&forRead, sizeof(double), 1, inputFile);

matrix[k] = forRead;

}\*/

for (int k1 = 0; k1 < sizeOfMatrix; ++k1)

for (int k2 = 0; k2 < sizeOfMatrix; ++k2) {

if (ftell(inputFile) == SEEK\_END) {

throw "Too few data in input file. Please check the data";

}

double forRead;

fread(&forRead, sizeof(double), 1, inputFile);

matrix[k1 \* sizeOfMatrix + k2] = forRead;

}

fclose(inputFile);

}

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

try {

int sizeProc; // общее количество процессов

int rank; // номер текущего процесса

MPI\_Init(&argc, &argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &sizeProc);

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

if (argc < 3) {

if (rank == 0)

throw "Please enter \"./main [sourceFile] [outputFile]\"";

else {

MPI\_Finalize();

return 1;

}

}

// считывание матрицы из файла

if (rank == 0) readFile(argc, argv);

int context, privateRow, privateCol; // контекст (для сетки), номер строки и столбца процессора в сетке

int sizeOfProcNet = sqrt(sizeProc); // размер сетки процессоров (количество строк = столбцов)

// инициализация сетки процессоров

Cblacs\_get(0, 0, &context);

Cblacs\_gridinit(&context, "row-major", sizeOfProcNet, sizeOfProcNet);

Cblacs\_pcoord(context, rank, &privateRow, &privateCol);

if (rank == 0) {

for (int i = 1; i < sizeProc; ++i)

MPI\_Send(&sizeOfMatrix, 1, MPI\_INT, i, 5, MPI\_COMM\_WORLD);

}

else {

MPI\_Status stat;

MPI\_Recv(&sizeOfMatrix, 1, MPI\_INT, 0, 5, MPI\_COMM\_WORLD, &stat);

}

int sizeOfProcBlock = sizeOfMatrix / sizeOfProcNet;

int zero = 0; // нужна ссылка на 0-й процессор

int numOfRowLocal = numroc\_(&sizeOfMatrix, &sizeOfProcBlock, &privateRow, &zero, &sizeOfProcNet);

int numOfColLocal = numroc\_(&sizeOfMatrix, &sizeOfProcBlock, &privateCol, &zero, &sizeOfProcNet);

int localCount = numOfRowLocal \* numOfColLocal;

double \* matrixLocal = new double [localCount];

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

matrixLocal[i] = 0;

// РАСПРЕДЕЛЕНИЕ МАТРИЦЫ ПО СЕТКЕ

int procRowForSend = 0, procColForSend = 0; // координаты получающего в СЕТКЕ

int rowForRecv = 0, colForRecv = 0; // координаты в ЛОКАЛЬНОЙ матрице, куда будем принимать

for (int i = 0; i < sizeOfMatrix; i = i + sizeOfProcBlock) {

////////////////////////////////////////////////////////////

procColForSend = 0;

int rowsToSend = sizeOfProcBlock; // сколько строк надо отправить за эту итерацию

if (sizeOfMatrix - i < sizeOfProcBlock) rowsToSend = sizeOfMatrix - i; // оставшиеся строки (если нацело количество по сетке не делится)

for (int j = 0; j < sizeOfMatrix; j = j + sizeOfProcBlock) {

int colsToSend = sizeOfProcBlock; // сколько столбцов надо отправить за эту итерацию

if (sizeOfMatrix - i < sizeOfProcBlock) colsToSend = sizeOfMatrix - i; // оставшиеся столбцы (если нацело количество по сетке не делится)

if (rank == 0) {

Cdgesd2d(context, rowsToSend, colsToSend, matrix + i \* sizeOfMatrix + j, sizeOfMatrix, procRowForSend, procColForSend);

}

if (privateRow == procRowForSend && privateCol == procColForSend) {

// принять с нулевого процессора в нужное место

Cdgerv2d(context, rowsToSend, colsToSend, matrixLocal + numOfRowLocal \* rowForRecv + colForRecv, numOfRowLocal, 0, 0);

colForRecv += colsToSend;

if (colForRecv >= numOfColLocal) colForRecv -= numOfColLocal;

}

procColForSend++;

if (procColForSend >= sizeOfProcNet) procColForSend -= sizeOfProcNet;

}

if (privateRow == procRowForSend) {

rowForRecv += rowsToSend;

if (rowForRecv >= numOfRowLocal) rowForRecv -= numOfRowLocal;

}

procRowForSend++;

if (procRowForSend >= sizeOfProcNet) procRowForSend -= sizeOfProcNet;

}

// ПОИСК СОБСТВЕННЫХ ЗНАЧЕНИЙ

// 1. ИНИЦИАЛИЗАЦИЯ ДЕСКРИПТОРОВ

int descMatrix [9], descZ [9]; // дескриптор для матрицы и вспомогательный дескриптор для вычисления

int INFO; // info-переменная

double \* z = new double [localCount]; // вспомогательный массив для вычисления с.зн.

descinit\_(descMatrix, &sizeOfMatrix, &sizeOfMatrix, &sizeOfProcBlock, &sizeOfProcBlock, &zero, &zero, &context, &numOfRowLocal, &INFO);

descinit\_(descZ, &sizeOfMatrix, &sizeOfMatrix, &sizeOfProcBlock, &sizeOfProcBlock, &zero, &zero, &context, &numOfRowLocal, &INFO);

// начало вычислений

double startTime = MPI\_Wtime();

// 2. ПРИВЕДЕНИЕ К ФОРМЕ ХЕССЕНБЕРГА

// вспомогательные переменные для вычисления

int ILO = 1, IHI = sizeOfMatrix, IA = 1, JA = 1;

double \* TAU = new double[sizeOfMatrix -1];

double \* WORK;

int LWORK = -1;

double forComputeLWORK;

// первый вызов для вычисления оптимального LWORK

pdgehrd\_(&sizeOfMatrix, &ILO, &IHI, matrixLocal, &IA, &JA, descMatrix, TAU, &forComputeLWORK, &LWORK, &INFO);

LWORK = (int) forComputeLWORK;

WORK = new double[LWORK];

// второй вызов pdgehrd\_ вычисляет хессенбергову форму

pdgehrd\_(&sizeOfMatrix, &ILO, &IHI, matrixLocal, &IA, &JA, descMatrix, TAU, WORK, &LWORK, &INFO);

delete [] WORK;

delete [] TAU;

// 3. ВЫЧИСЛЕНИЕ СОБСТВЕННЫХ ЗНАЧЕНИЙ

char JOB = 'E', COMPZ = 'N';

ILO = 1, IHI = sizeOfMatrix;

int \* IWORK;

int LIWORK;

LWORK = -1, LIWORK = -1;

double forOptimalLWORK;

int forOptimalLIWORK;

eigenRealValues = new double [sizeOfMatrix];

eigenImagineValues = new double [sizeOfMatrix];

pdhseqr\_(&JOB, &COMPZ, &sizeOfMatrix, &ILO, &IHI, matrixLocal, descMatrix, eigenRealValues, eigenImagineValues, z, descZ, &forOptimalLWORK, &LWORK, &forOptimalLIWORK, &LIWORK, &INFO);

LWORK = (int) forOptimalLWORK;

//LWORK = LWORK \* 3;

LIWORK = forOptimalLIWORK;

//LIWORK = LWORK \* 4;

WORK = new double [LWORK];

IWORK = new int [LIWORK];

pdhseqr\_(&JOB, &COMPZ, &sizeOfMatrix, &ILO, &IHI, matrixLocal, descMatrix, eigenRealValues, eigenImagineValues, z, descZ, WORK, &LWORK, IWORK, &LIWORK, &INFO);

delete [] z;

delete [] WORK;

delete [] IWORK;

// конец вычислений

double endTime = MPI\_Wtime();

// вывод метаинформации и собственных значений

if (rank == 0) {

ofstream out(argv[2]);

out << "Size of matrix = " << sizeOfMatrix << endl;

out << "Processes count = " << sizeProc << endl;

out << "Work time = " << endTime - startTime << endl;

if (sizeOfMatrix < 10) {

out << "Matrix:" << endl;

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

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

out << matrix[i \* sizeOfMatrix + j] << " ";

out << endl;

}

}

else

out << "Matrix is too large for print" << endl;

out << "Eigen values:" << endl;

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

out << eigenRealValues[i];

if (eigenImagineValues[i] > 0) out << "+" << eigenImagineValues[i] << "i";

else if (eigenImagineValues[i] < 0) out << eigenImagineValues[i] << "i";

out << " ";

}

out << endl << "End of report" << endl;

out.close();

}

// освободить память

if (rank == 0) delete [] matrix;

delete [] matrixLocal;

delete [] eigenRealValues;

delete [] eigenImagineValues;

// завершение работы программы

Cblacs\_gridexit(context);

MPI\_Finalize();

return 0;

}

catch(const char \* error) {

cerr << error << endl;

MPI\_Finalize();

return 1;

}

catch(int) {

cerr << "Another error" << endl;

MPI\_Finalize();

return 1;

}

}