#include <iostream>

#include <cmath>

#include <fstream>

#include <vector>

#include <cstdlib>

#include <limits>

#include <mpi.h>

#include <stdio.h>

#include <assert.h>

typedef std::vector<double> Vec;

typedef double(\*Function)(const size\_t&, const Vec&);

typedef double(\*Deriv)(const size\_t&, const size\_t&, const Vec&);

typedef std::vector<Function> Sys\_;

typedef std::vector<std::vector<Deriv> > Derivatives;

typedef std::vector<std::vector<double> > Matrix;

typedef Vec(\*LSSolver)(const Matrix&, const Vec&);

typedef Vec(\*VecSum)(const Vec&, const Vec&);

typedef double(\*VecDiff)(const Vec&, const Vec&);

struct LS {

 Matrix A;

 Vec b;

};

size\_t system\_size = 4096;

double func(const size\_t& i, const Vec& v) {

 return cos(v[i]) - 1;

}

double derivative(const size\_t& i, const size\_t& j, const Vec& v) {

 if (i == j) {

 return -sin(v[i]);

 }

 return 0.0;

}

static void printVec(const Vec& v) {

 std::cout << "size: " << v.size() << std::endl;

 for (size\_t i = 0; i < v.size(); ++i) {

 std::cout << v[i] << " ";

 }

 std::cout << std::endl;

}

Vec gauss(const Matrix& A, const Vec& b) {

 MPI\_Status status;

 int nSize, nRowsBloc, nRows, nCols;

 int Numprocs, MyRank, Root = 0;

 int irow, jrow, icol, index, ColofPivot, neigh\_proc;

 double \*Input\_A, \*Input\_B, \*ARecv, \*BRecv;

 double \*Output, Pivot;

 double \*X\_buffer, \*Y\_buffer;

 double \*Buffer\_Pivot, \*Buffer\_bksub;

 double tmp;

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

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

 if (MyRank == 0) {

 nRows = A.size();

 nCols = A[0].size();

 nSize = nRows;

 Input\_B = (double \*)malloc(nSize \* sizeof(double));

 for (irow = 0; irow < nSize; irow++) {

 Input\_B[irow] = b[irow];

 }

 Input\_A = (double \*)malloc(nSize\*nSize \* sizeof(double));

 index = 0;

 for (irow = 0; irow < nSize; irow++)

 for (icol = 0; icol < nSize; icol++)

 Input\_A[index++] = A[irow][icol];

 }

 MPI\_Bcast(&nRows, 1, MPI\_INT, Root, MPI\_COMM\_WORLD);

 MPI\_Bcast(&nCols, 1, MPI\_INT, Root, MPI\_COMM\_WORLD);

 /\* .... Broad cast the size of the matrix to all ....\*/

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

 nRowsBloc = nSize / Numprocs;

 /\*......Memory of input matrix and vector on each process .....\*/

 ARecv = (double \*)malloc(nRowsBloc \* nSize \* sizeof(double));

 BRecv = (double \*)malloc(nRowsBloc \* sizeof(double));

 /\*......Scatter the Input Data to all process ......\*/

 MPI\_Scatter(Input\_A, nRowsBloc \* nSize, MPI\_DOUBLE, ARecv, nRowsBloc \* nSize, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

 MPI\_Scatter(Input\_B, nRowsBloc, MPI\_DOUBLE, BRecv, nRowsBloc, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

 /\* ....Memory allocation of ray Buffer\_Pivot .....\*/

 Buffer\_Pivot = (double \*)malloc((nRowsBloc + 1 + nSize \* nRowsBloc) \* sizeof(double));

 /\* Receive data from all processors (i=0 to k-1) above my processor (k).... \*/

 for (neigh\_proc = 0; neigh\_proc < MyRank; neigh\_proc++) {

 MPI\_Recv(Buffer\_Pivot, nRowsBloc \* nSize + 1 + nRowsBloc, MPI\_DOUBLE, neigh\_proc, neigh\_proc, MPI\_COMM\_WORLD, &status);

 for (irow = 0; irow < nRowsBloc; irow++) {

 /\* .... Buffer\_Pivot[0] : locate the rank of the processor received \*/

 /\* .... Index is used to reduce the matrix to traingular matrix \*/

 /\* .... Buffer\_Pivot[0] is used to determine the starting value of

 pivot in each row of the matrix, on each processor \*/

 ColofPivot = ((int)Buffer\_Pivot[0]) \* nRowsBloc + irow;

 for (jrow = 0; jrow < nRowsBloc; jrow++) {

 index = jrow\*nSize;

 tmp = ARecv[index + ColofPivot];

 for (icol = ColofPivot; icol < nSize; icol++) {

 ARecv[index + icol] -= tmp \* Buffer\_Pivot[irow\*nSize + icol + 1 + nRowsBloc];

 }

 BRecv[jrow] -= tmp \* Buffer\_Pivot[1 + irow];

 ARecv[index + ColofPivot] = 0.0;

 }

 }

 }

 Y\_buffer = (double \*)malloc(nRowsBloc \* sizeof(double));

 /\* ....Modification of row entries on each processor ...\*/

 /\* ....Division by pivot value and modification ...\*/

 for (irow = 0; irow < nRowsBloc; irow++) {

 ColofPivot = MyRank \* nRowsBloc + irow;

 index = irow\*nSize;

 Pivot = ARecv[index + ColofPivot];

 assert(Pivot != 0);

 for (icol = ColofPivot; icol < nSize; icol++) {

 ARecv[index + icol] = ARecv[index + icol] / Pivot;

 Buffer\_Pivot[index + icol + 1 + nRowsBloc] = ARecv[index + icol];

 }

 Y\_buffer[irow] = BRecv[irow] / Pivot;

 Buffer\_Pivot[irow + 1] = Y\_buffer[irow];

 for (jrow = irow + 1; jrow < nRowsBloc; jrow++) {

 tmp = ARecv[jrow\*nSize + ColofPivot];

 for (icol = ColofPivot + 1; icol < nSize; icol++) {

 ARecv[jrow\*nSize + icol] -= tmp \* Buffer\_Pivot[index + icol + 1 + nRowsBloc];

 }

 BRecv[jrow] -= tmp \* Y\_buffer[irow];

 ARecv[jrow\*nSize + irow] = 0;

 }

 }

 /\*....Send data to all processors below the current processors \*/

 for (neigh\_proc = MyRank + 1; neigh\_proc < Numprocs; neigh\_proc++) {

 /\* ...... Rank is stored in first location of Buffer\_Pivot and

 this is used in reduction to triangular form ....\*/

 Buffer\_Pivot[0] = (double)MyRank;

 MPI\_Send(Buffer\_Pivot, nRowsBloc\*nSize + 1 + nRowsBloc, MPI\_DOUBLE, neigh\_proc, MyRank, MPI\_COMM\_WORLD);

 }

 /\*.... Back Substitution starts from here ........\*/

 /\*.... Receive from all higher processors ......\*/

 Buffer\_bksub = (double \*)malloc(nRowsBloc \* 2 \* sizeof(double));

 X\_buffer = (double \*)malloc(nRowsBloc \* sizeof(double));

 for (neigh\_proc = MyRank + 1; neigh\_proc < Numprocs; ++neigh\_proc) {

 MPI\_Recv(Buffer\_bksub, 2 \* nRowsBloc, MPI\_DOUBLE, neigh\_proc, neigh\_proc,

 MPI\_COMM\_WORLD, &status);

 for (irow = nRowsBloc - 1; irow >= 0; irow--) {

 for (icol = nRowsBloc - 1; icol >= 0; icol--) {

 /\* ... Pick up starting Index .....\*/

 index = (int)Buffer\_bksub[icol];

 Y\_buffer[irow] -= Buffer\_bksub[nRowsBloc + icol] \* ARecv[irow\*nSize + index];

 }

 }

 }

 for (irow = nRowsBloc - 1; irow >= 0; irow--) {

 index = MyRank\*nRowsBloc + irow;

 Buffer\_bksub[irow] = (double)index;

 Buffer\_bksub[nRowsBloc + irow] = X\_buffer[irow] = Y\_buffer[irow];

 for (jrow = irow - 1; jrow >= 0; jrow--)

 Y\_buffer[jrow] -= X\_buffer[irow] \* ARecv[jrow\*nSize + index];

 }

 /\*.... Send to all lower processes...\*/

 for (neigh\_proc = 0; neigh\_proc < MyRank; neigh\_proc++) {

 MPI\_Send(Buffer\_bksub, 2 \* nRowsBloc, MPI\_DOUBLE, neigh\_proc, MyRank, MPI\_COMM\_WORLD);

 }

 /\*.... Gather the result on the processor 0 ....\*/

 Output = (double \*)malloc(nSize \* sizeof(double));

 MPI\_Gather(X\_buffer, nRowsBloc, MPI\_DOUBLE, Output, nRowsBloc, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

 /\* .......Output vector .....\*/

 Vec res(nSize);

 if (MyRank == 0) {

 for (irow = 0; irow < nSize; irow++)

 res[irow] = Output[irow];

 }

 return res;

}

Vec sum(const Vec& lhs, const Vec& rhs) {

 Vec res(lhs.size());

 if (lhs.size() != rhs.size()) {

 return res;

 }

 for (size\_t i = 0; i < lhs.size(); ++i) {

 res[i] = lhs[i] + rhs[i];

 }

 return res;

}

double diff(const Vec& lhs, const Vec& rhs) {

 double res = 0.;

 for (size\_t i = 0; i < lhs.size(); ++i) {

 res += lhs[i] - rhs[i];

 }

 return fabs(res);

}

class Newtone {

 public:

 Newtone(int rank) : m\_rank(rank) {

 m\_h = std::numeric\_limits<double>::epsilon();

 }

 Vec find\_solution(const Sys\_& sys, const Vec& start, const Derivatives& d, LSSolver solver,

 VecSum vec\_summator, VecDiff vec\_differ, const double& eps, const size\_t& max\_iter) {

 size\_t iter\_count = 1;

 double diff = 0.;

 Vec sys\_val(sys.size(), 0);

 if (m\_rank == 0) {

 m\_jac.reserve(sys.size());

 for (size\_t i = 0; i < sys.size(); ++i) {

 Vec v(sys.size());

 m\_jac.push\_back(v);

 }

 compute\_jacobian(sys, d, start);

 for (size\_t i = 0; i < sys.size(); ++i) {

 sys\_val[i] = -sys[i](i, start);

 }

 }

 MPI\_Barrier(MPI\_COMM\_WORLD);

 Vec delta = solver(m\_jac, sys\_val);

 Vec new\_sol(sys.size()), old\_sol(sys.size());

 if (m\_rank == 0) {

 new\_sol = vec\_summator(start, delta);

 old\_sol = start;

 }

 MPI\_Bcast(new\_sol.data(), new\_sol.size(), MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

 MPI\_Bcast(old\_sol.data(), old\_sol.size(), MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

 diff = vec\_differ(new\_sol, old\_sol);

 while (diff > eps && iter\_count <= max\_iter) {

 old\_sol = new\_sol;

 if (m\_rank == 0) {

 compute\_jacobian(sys, d, old\_sol);

 for (size\_t i = 0; i < sys.size(); ++i) {

 sys\_val[i] = -sys[i](i, old\_sol);

 }

 }

 MPI\_Barrier(MPI\_COMM\_WORLD);

 delta = solver(m\_jac, sys\_val);

 if (m\_rank == 0) {

 new\_sol = vec\_summator(old\_sol, delta);

 }

 MPI\_Bcast(new\_sol.data(), new\_sol.size(), MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

 iter\_count++;

 diff = vec\_differ(new\_sol, old\_sol);

 }

 return new\_sol;

 }

 double compute\_derivative(const size\_t& pos, Function func, const size\_t& var\_num, const Vec& point) {

 Vec left\_point(point), right\_point(point);

 left\_point[var\_num] -= m\_h;

 right\_point[var\_num] += m\_h;

 double left = func(pos, left\_point), right = func(pos, right\_point);

 return (right - left) / (2 \* m\_h);

 }

 private:

 void compute\_jacobian(const Sys\_& sys, const Derivatives& d, const Vec& point) {

 for (size\_t i = 0; i < sys.size(); ++i) {

 for (size\_t j = 0; j < sys.size(); ++j) {

 double res\_val;

 res\_val = d[i][j](i, j, point);

 m\_jac[i][j] = res\_val;

 }

 }

 }

 double m\_h;

 int m\_rank;

 Matrix m\_jac;

 Vec m\_right\_part;

};

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

 int rank, size;

 Sys\_ s;

 Derivatives d(system\_size);

 Vec start(system\_size, 0.87);

 for (size\_t i = 0; i < system\_size; ++i) {

 s.push\_back(&func);

 }

 for (size\_t i = 0; i < system\_size; ++i) {

 for (size\_t j = 0; j < system\_size; ++j)

 d[i].push\_back(&derivative);

 }

 MPI\_Init(&argc, &argv);

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

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

 Newtone n(rank);

 MPI\_Barrier(MPI\_COMM\_WORLD);

 double time = MPI\_Wtime();

 Vec sol = n.find\_solution(s, start, d, &gauss, &sum, &diff, 0.0001, 100);

 MPI\_Barrier(MPI\_COMM\_WORLD);

 time = MPI\_Wtime() - time;

 double max\_time;

 MPI\_Reduce(&time, &max\_time, 1, MPI\_DOUBLE, MPI\_MAX, 0, MPI\_COMM\_WORLD);

 if (rank == 0) {

 std::ofstream myfile;

 char filename[32];

 snprintf(filename, 32, "out\_%ld\_%d.txt", system\_size, size);

 myfile.open(filename);

 for (size\_t i = 0; i < sol.size(); ++i) {

 myfile << sol[i] << " ";

 }

 myfile << std::endl;

 myfile << "Time: " << time << " " << max\_time << std::endl;

 myfile.close();

 }

 MPI\_Finalize();

 return 0;

}