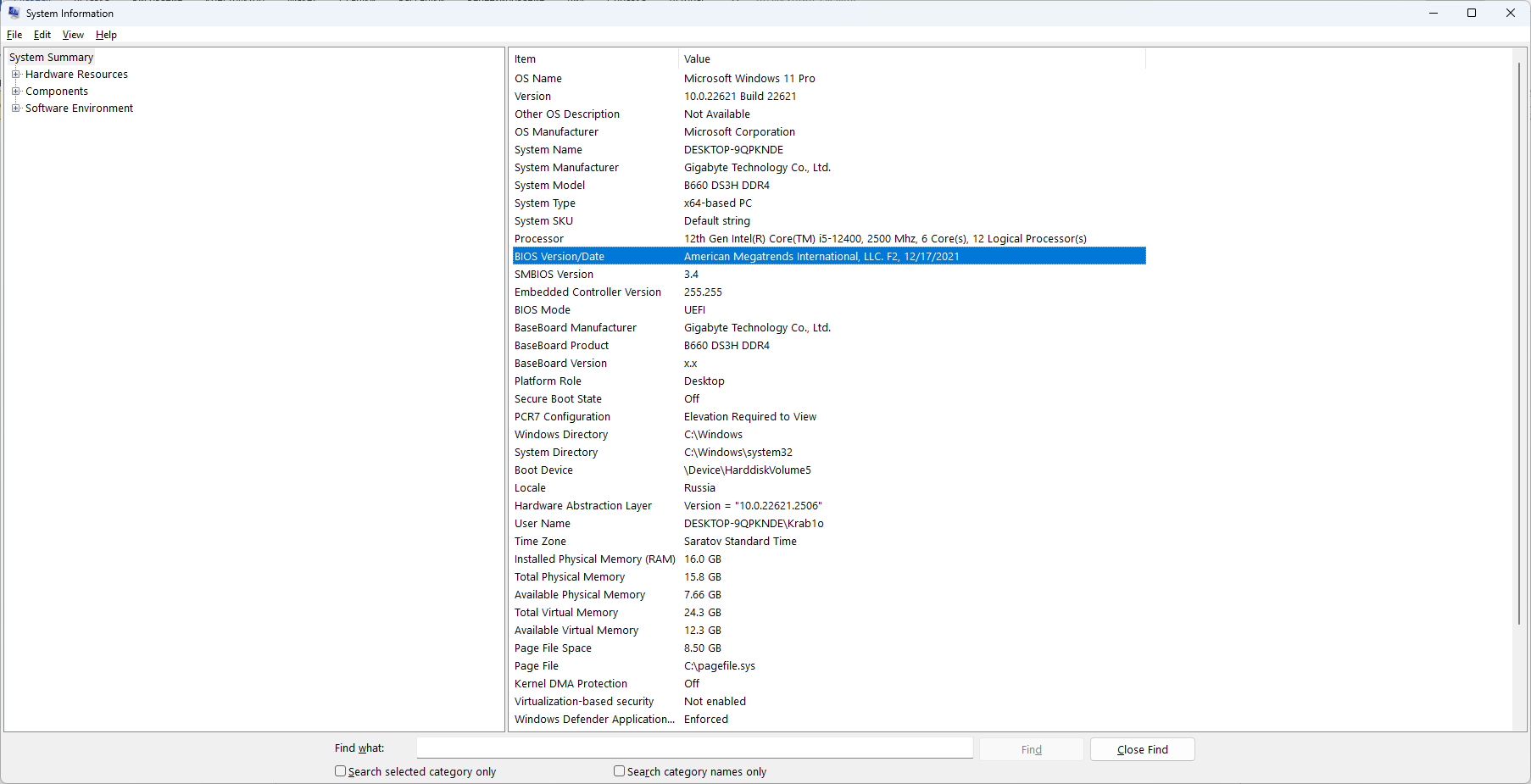
Задание выполнялось на компьютере со следующими параметрами:



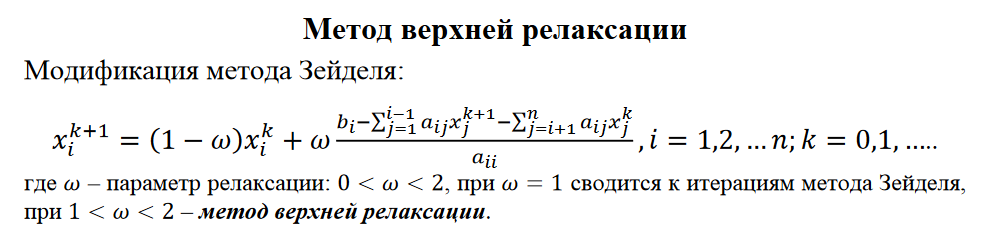
Из важного:

CPU: 12th Gen Intel(R) Core(TM) i5-12400, 2500 Mhz, 6 Core(s), 12 Logical Processor(s)

Motherboard: B660 DS3H DDR4

RAM: 16.0 GB со скоростью 2400MHz

Вариант 3. Реализовать метод верхней релаксации:



|  |  |  |  |
| --- | --- | --- | --- |
| Номер теста | Порядок системы | Ускорение алгоритма Гаусса | Ускорение итерационного алгоритма (верхн. релакс) |
| 1 | 10 | — | — |
| 2 | 100 | 0.5 |  |
| 3 | 500 | 2.342 |  |
| 4 | 1000 | 3.816 |  |
| 5 | 1500 | 4.425 |  |
| 6 | 2000 | 4.530 |  |
| 7 | 2500 | 4.368 |  |
| 8 | 3000 | 4.24 |  |

Код:

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <conio.h>

#include <time.h>

#include <math.h>

#include <omp.h>

// per OpenMP standard, \_OPENMP defined when compiling with OpenMP

#ifdef \_OPENMP

#ifdef \_MSC\_VER

// must use MSVC \_\_pragma here instead of \_Pragma otherwise you get an internal

// compiler error. still an issue in Visual Studio 2022

#define PDDCP\_OMP\_PARALLEL\_FOR \_\_pragma(omp parallel for)

// any other standards-compliant C99/C++11 compiler

#else

#define PDDCP\_OMP\_PARALLEL\_FOR \_Pragma("omp parallel for")

#endif // \_MSC\_VER

// no OpenMP support

#else

#define PDDCP\_OMP\_PARALLEL\_FOR

#endif // \_OPENMP

const double eps = 0.00001;

const double relaxParam = 1;

typedef struct {

int PivotRow;

double MaxValue;

} TThreadPivotRow;

int\* pPivotPos; // The number of pivot rows selected at the iterations

int\* pPivotIter; // The iterations, at which the rows were pivots

// Function for simple initialization of the matrix

// and the vector elements

void DummyDataInitialization(double\* pMatrix, double\* pVector, int

Size) {

int i, j; // Loop variables

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

pVector[i] = i + 1;

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

if (j <= i)

pMatrix[i \* Size + j] = 1;

else

pMatrix[i \* Size + j] = 0;

}

}

}

// Function for random initialization of the matrix

// and the vector elements

void RandomDataInitialization(double\* pMatrix, double\* pVector,

int Size) {

int i, j; // Loop variables

srand(unsigned(clock()));

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

pVector[i] = rand() / double(1000);

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

if (j <= i)

pMatrix[i \* Size + j] = rand() / double(1000);

else

pMatrix[i \* Size + j] = 0;

}

}

}

// Function for memory allocation and definition of the objects elements

void ProcessInitialization(double\*& pMatrix, double\*

& pVector,

double\*& pResult, int& Size) {

// Setting the size of the matrix and the vector

do {

printf("\nEnter size of the matrix and the vector: ");

scanf("%d", &Size);

printf("\nChosen size = %d \n", Size);

if (Size <= 0)

printf("\nSize of objects must be greater than 0!\n");

} while (Size <= 0);

// Memory allocation

pMatrix = new double[Size \* Size];

pVector = new double[Size];

pResult = new double[Size];

// Initialization of the matrix and the vector elements

RandomDataInitialization(pMatrix, pVector, Size);

//DummyDataInitialization(pMatrix, pVector, Size);

}

// Function for formatted matrix output

void PrintMatrix(double\* pMatrix, int RowCount, int ColCount) {

int i, j; // Loop variables

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

for (j = 0; j < ColCount; j++)

printf("%7.4f ", pMatrix[i \* RowCount + j]);

printf("\n");

}

}

// Function for formatted vector output

void PrintVector(double\* pVector, int Size) {

int i;

for (i = 0; i < Size; i++)

printf("%7.4f ", pVector[i]);

}

// Finding the pivot row

int ParallelFindPivotRow(double\* pMatrix, int Size, int Iter) {

int PivotRow = -1; // The index of the pivot row

int MaxValue = 0; // The value of the pivot element

int i; // Loop variable

// Choose the row, that stores the maximum element

#pragma omp parallel

{

TThreadPivotRow ThreadPivotRow;

ThreadPivotRow.MaxValue = 0;

ThreadPivotRow.PivotRow = -1;

#pragma omp for

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

if ((pPivotIter[i] == -1) &&

(fabs(pMatrix[i \* Size + Iter]) > ThreadPivotRow.MaxValue)) {

ThreadPivotRow.PivotRow = i;

ThreadPivotRow.MaxValue = fabs(pMatrix[i \* Size + Iter]);

}

}

//printf("\n Local thread (id = %i) pivot row: %i", omp\_get\_thread\_num(), ThreadPivotRow.PivotRow);

#pragma omp critical

{

if (ThreadPivotRow.MaxValue > MaxValue) {

MaxValue = ThreadPivotRow.MaxValue;

PivotRow = ThreadPivotRow.PivotRow;

}

}

}

return PivotRow;

}

void ParallelColumnElimination(double\* pMatrix, double\* pVector,

int Pivot, int Iter, int Size) {

double PivotValue, PivotFactor;

PivotValue = pMatrix[Pivot \* Size + Iter];

#pragma omp parallel for private(PivotFactor) schedule(dynamic,1)

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

if (pPivotIter[i] == -1) {

PivotFactor = pMatrix[i \* Size + Iter] / PivotValue;

for (int j = Iter; j < Size; j++) {

pMatrix[i \* Size + j] -= PivotFactor \* pMatrix[Pivot \* Size + j];

}

pVector[i] -= PivotFactor \* pVector[Pivot];

}

}

}

void ParallelBackSubstitution(double\* pMatrix, double\* pVector,

double\* pResult, int Size) {

int RowIndex, Row;

for (int i = Size - 1; i >= 0; i--) {

RowIndex = pPivotPos[i];

pResult[i] = pVector[RowIndex] / pMatrix[Size \* RowIndex + i];

#pragma omp parallel for private (Row)

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

Row = pPivotPos[j];

pVector[Row] -= pMatrix[Row \* Size + i] \* pResult[i];

pMatrix[Row \* Size + i] = 0;

}

}

}

// Gaussian elimination

void ParallelGaussianElimination(double\* pMatrix, double\* pVector,

int Size) {

int Iter; // The number of the iteration of the Gaussian

// elimination

int PivotRow; // The number of the current pivot row

for (Iter = 0; Iter < Size; Iter++) {

// Finding the pivot row

PivotRow = ParallelFindPivotRow(pMatrix, Size, Iter);

pPivotPos[Iter] = PivotRow;

pPivotIter[PivotRow] = Iter;

ParallelColumnElimination(pMatrix, pVector, PivotRow, Iter, Size);

}

}

// Function for the execution of Gauss algorithm

void ParallelResultCalculation(double\* pMatrix, double\* pVector,

double\* prevResult, int Size) {

long double\* curResult = new long double[Size];

int t = 0;

//#pragma omp parallel for

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

{

prevResult[i] = 0;

curResult[i] = 0;

}

bool flag = false;

long double sum;

do {

flag = false;

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

curResult[i] = pVector[i];

sum = 0;

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

if (i != j)

sum += pMatrix[Size \* i + j] \* curResult[j];

}

curResult[i] -= sum;

curResult[i] = (1 - relaxParam) \* prevResult[i] + relaxParam \* curResult[i] / pMatrix[Size \* i + i];

}

//Check if difference between current computations and previous ones exceed eps

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

if (fabs(curResult[i] - prevResult[i]) >= eps) flag = true;

prevResult[i] = curResult[i];

}

} while (flag);

delete[] curResult;

}

// Function for computational process termination

void ProcessTermination(double\* pMatrix, double\* pVector, double\*

pResult) {

delete[] pMatrix;

delete[] pVector;

delete[] pResult;

}

// Function for testing the result

void TestResult(double\* pMatrix, double\* pVector,

double\* pResult, int Size) {

/\* Buffer for storing the vector, that is a result of multiplication

of the linear system matrix by the vector of unknowns \*/

double\* pRightPartVector;

// Flag, that shows wheather the right parts

// vectors are identical or not

int equal = 0;

double Accuracy = 1.e-6; // Comparison accuracy

pRightPartVector = new double[Size];

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

pRightPartVector[i] = 0;

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

pRightPartVector[i] +=

pMatrix[i \* Size + j] \* pResult[j];

}

}

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

if (fabs(pRightPartVector[i] - pVector[i]) > Accuracy)

equal = 1;

}

if (equal == 1)

printf("The result of the parallel Gauss algorithm is NOT correct."

"Check your code.");

else

printf("The result of the parallel Gauss algorithm is correct.");

delete[] pRightPartVector;

}

int main() {

double\* pMatrix; // The matrix of the linear system

double\* pVector; // The right parts of the linear system

double\* pResult; // The result vector

int Size; // The sizes of the initial matrix and the vector

double start, finish, duration;

printf("Serial Gauss algorithm for solving linear systems\n");

// Memory allocation and definition of objects' elements

ProcessInitialization(pMatrix, pVector, pResult, Size);

// The matrix and the vector output

printf("Initial Matrix \n");

//PrintMatrix(pMatrix, Size, Size);

printf("Initial Vector \n");

//PrintVector(pVector, Size);

// Execution of Gauss algorithm

start = clock();

ParallelResultCalculation(pMatrix, pVector, pResult, Size);

finish = clock();

duration = (finish - start) / CLOCKS\_PER\_SEC;

TestResult(pMatrix, pVector, pResult, Size);

//Printing the result vector

printf("\n Result Vector: \n");

//PrintVector(pResult, Size);

// Printing the execution time of Gauss method

printf("\n Time of execution: %f\n", duration);

// Computational process termination

ProcessTermination(pMatrix, pVector, pResult);

return 0;

}