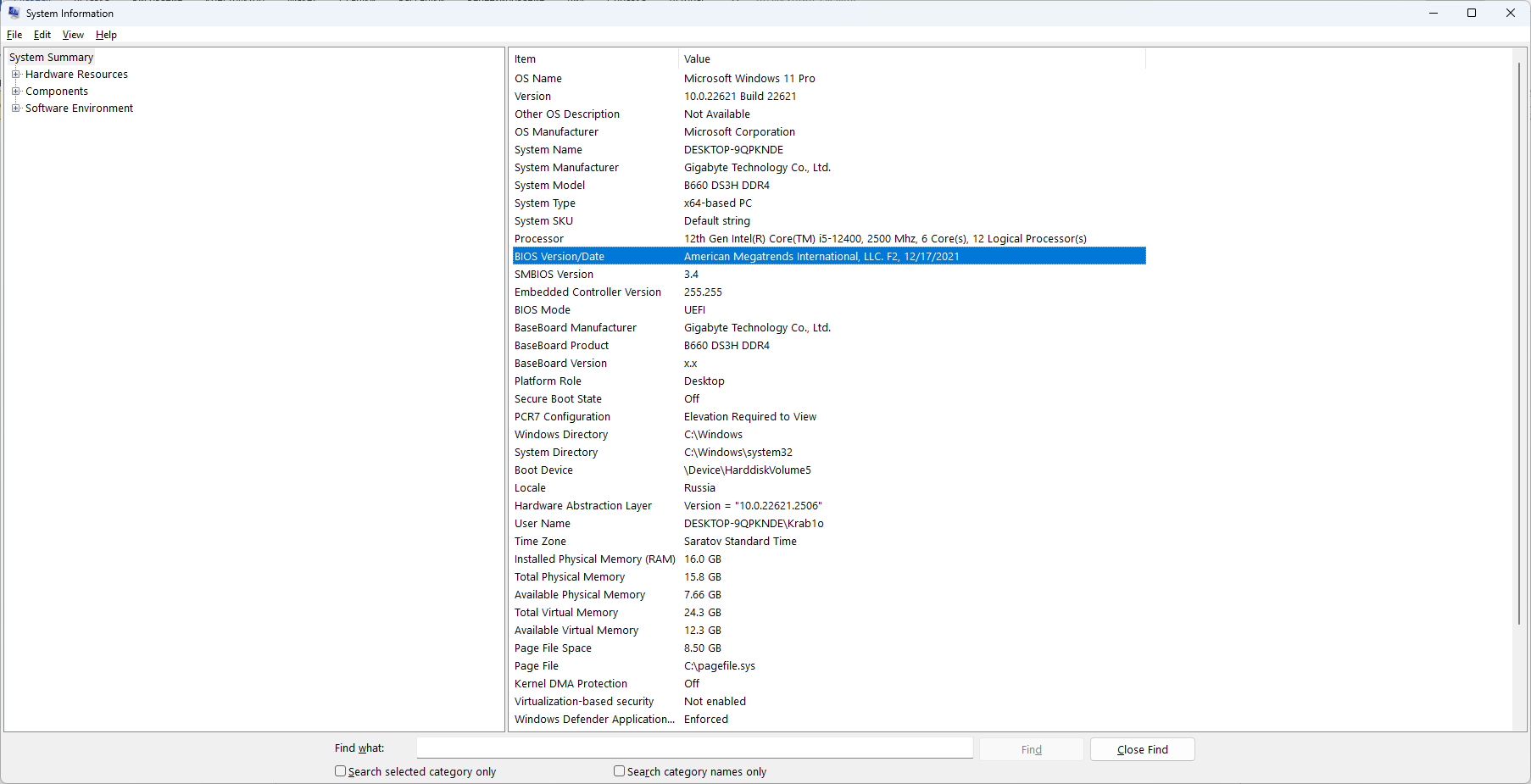
Задание выполнялось на компьютере со следующими параметрами:



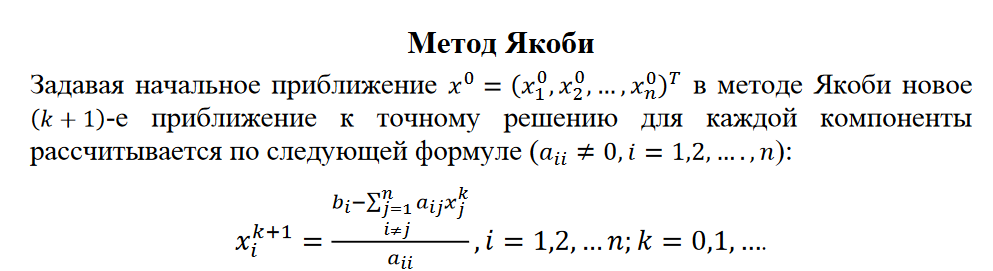
Из важного:

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 Якоби:



Условие сходимости:

Чтобы методы сходились, требуется выполнение некоторых условий. Для Якоби должно быть диагональное преобладание. Для Зейделя — спектральный радиус меньше единицы, диагонального преобладание также достаточно.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Номер теста | Порядок системы | Последовательный алгоритм | Параллельный алгоритм | |
| Время | Ускорение |
| 1 | 10 | — | — | — |
| 2 | 100 | 0.003050 | 0.010387 | 0.2936 |
| 3 | 500 | 0.038860 | 0.026965 | 1.4411 |
| 4 | 1000 | 0.171142 | 0.116945 | 1.4634008 |
| 5 | 1500 | 0.432129 | 0.171327 | 2.522247 |
| 6 | 2000 | 0.642627 | 0.344131 | 1.87639 |
| 7 | 2500 | 1.129990 | 0.567905 | 1.98975 |
| 8 | 3000 | 1.429655 | 0.675155 | 2.1175211 |

|  |  |  |  |
| --- | --- | --- | --- |
| Номер теста | Порядок системы | Ускорение алгоритма Гаусса | Ускорение алгоритма  Якоби |
| 1 | 10 | — | — |
| 2 | 100 | 0.2936 | 0.04839 |
| 3 | 500 | 1.4411 | 0.157 |
| 4 | 1000 | 1.4634008 | 0.5143 |
| 5 | 1500 | 2.522247 | 0.8139 |
| 6 | 2000 | 1.87639 | 0.78764 |
| 7 | 2500 | 1.98975 | 0.96515 |
| 8 | 3000 | 2.1175211 | 0.9734 |

Алгоритм Якоби не ускоряется, так как из-за особенности алгоритма большинство итераций приходится на цикл, который невозможно распараллелить.

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <conio.h>

#include <time.h>

#include <math.h>

#include <mpi.h>

const double eps = 0.000000001;

const double relaxParam = 1;

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(long double\* pMatrix, long 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(long double\*\* pMatrix, long double\* pVector,

int Size) {

long double s;

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++) {

pMatrix[i][j] = rand() / double(1000);

}

}

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

s = 0;

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

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

if (i != j)

s += pMatrix[i][j];

}

pMatrix[i][i] = rand() / double(1000) + s;

}

}

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

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

& pVector, long 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 long double\*[Size];

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

{

pMatrix[i] = new long double[Size];

}

pVector = new long double[Size];

pResult = new long 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(long double\* pVector, int Size) {

int i;

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

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

}

// Function for the execution of Gauss algorithm

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

long double\* pResult, int Size, int NProc, int ProcId) {

double\* g = new double[Size];

double\* d = new double[Size];

int t = 0;

//#pragma omp parallel for

for (int i = 0; i < Size; i++) { pResult[i] = 0; g[i] = 0; }

bool flag = false;

double sum;

double sum1;

do {

//cout << t << endl;

//t++;

flag = false;

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

g[i] = pVector[i];

sum = 0;

sum1 = 0;

int n1 = Size / NProc;

int n2 = (ProcId + 1) \* n1;

if (NProc == ProcId + 1) {

n2 = Size;

}

int st = ProcId \* n1;

for (int j = st; j < n2; j++) {

if (i != j) {

sum += (pMatrix[i][j] \* g[j]);

}

}

MPI\_Reduce(&sum, &sum1, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

MPI\_Bcast(&sum1, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

g[i] -= sum1;

g[i] = (1.0 \* g[i]) / pMatrix[i][i];

if (fabs(g[i] - pResult[i]) >= eps) flag = true;

pResult[i] = g[i];

}

} while (flag);

delete[] g;

}

// Function for computational process termination

void ProcessTermination(long double\*\* pMatrix, long double\* pVector, long double\* pResult, int Size) {

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

{

delete[] pMatrix[i];

}

delete[] pMatrix;

delete[] pVector;

delete[] pResult;

}

// Function for testing the result

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

long 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 \*/

long double\* pRightPartVector;

// Flag, that shows wheather the right parts

// vectors are identical or not

int equal = 0;

long double Accuracy = 0.001; // Comparison accuracy

pRightPartVector = new long 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() {

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

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

long double\* pResult; // The result vector

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

long 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);

int NProc, ProcId;

MPI\_Init(NULL, NULL);

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

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

// Execution of Gauss algorithm

start = clock();

ParallelResultCalculation(pMatrix, pVector, pResult, Size, 4, 1);

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, Size);

MPI\_Finalize();

return 0;

}