

МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**

**ПАРАЛЛЕЛЬНОЕ И РАСПРЕДЕЛЕННОЕ ПРОГРАММИРОВАНИЕ.
WORK13**

ОТЧЕТ О ПРАКТИКЕ

студента 3 курса 311 группы
направления 02.03.02 — Фундаментальная информатика и информационные
технологии
факультета КНиИТ
Вильцева Данила Денисовича

Проверил

Старший преподаватель

М. С. Портенко

СОДЕРЖАНИЕ

1	Work 13.....	3
1.1	Условие задачи.....	3
1.2	Решение. Последовательная версия	3
1.3	Решение. Параллельная версия.....	7
2	Результат работы	15
3	Характеристики компьютера.....	17

1 Work 13

1.1 Условие задачи

Аналогично работе с OMP выполните следующее задание через MPI.

Задайте элементы больших матриц и векторов при помощи датчика случайных чисел. Отключите печать исходных матрицы и вектора и печать результирующего вектора (закомментируйте соответствующие строки кода). Проведите вычислительные эксперименты, результаты занесите в таблицу 1.

Таблица 1. Время выполнения (сек) последовательного и параллельного алгоритмов Гаусса решения систем линейных уравнений и ускорение

Номер теста	Порядок системы	Последовательный алгоритм	Параллельный алгоритм	
			Время	Ускорение
1	10			
2	100			
3	500			
4	1000			
5	1500			
6	2000			
7	2500			
8	3000			

1.2 Решение. Последовательная версия

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>
#include <math.h>

int* pSerialPivotPos; // The Number of pivot rows selected at the
// iterations
int* pSerialPivotIter; // 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_s("%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
    DummyDataInitialization(pMatrix, pVector, Size);
    //RandomDataInitialization(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 FindPivotRow(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
    for (i = 0; i < Size; i++) {
        if ((pSerialPivotIter[i] == -1) &&
            (fabs(pMatrix[i * Size + Iter]) > MaxValue)) {
            PivotRow = i;
            MaxValue = fabs(pMatrix[i * Size + Iter]);
        }
    }
    return PivotRow;
}

// Column elimination
void SerialColumnElimination(double* pMatrix, double* pVector,
    int Pivot, int Iter, int Size) {
    double PivotValue, PivotFactor;
    PivotValue = pMatrix[Pivot * Size + Iter];
    for (int i = 0; i < Size; i++) {
        if (pSerialPivotIter[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];
        }
    }
}

// Gaussian elimination
void SerialGaussianElimination(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 = FindPivotRow(pMatrix, Size, Iter);
        pSerialPivotPos[Iter] = PivotRow;
    }
}

```

```

        pSerialPivotIter[PivotRow] = Iter;
        SerialColumnElimination(pMatrix, pVector, PivotRow, Iter, Size);
    }
}

// Back substitution
void SerialBackSubstitution(double* pMatrix, double* pVector,
    double* pResult, int Size) {
    int RowIndex, Row;
    for (int i = Size - 1; i >= 0; i--) {
        RowIndex = pSerialPivotPos[i];
        pResult[i] = pVector[RowIndex] / pMatrix[Size * RowIndex + i];
        for (int j = 0; j < i; j++) {
            Row = pSerialPivotPos[j];
            pVector[Row] -= pMatrix[Row * Size + i] * pResult[i];
            pMatrix[Row * Size + i] = 0;
        }
    }
}

// Function for the execution of Gauss algorithm
void SerialResultCalculation(double* pMatrix, double* pVector,
    double* pResult, int Size) {
    // Memory allocation
    pSerialPivotPos = new int[Size];
    pSerialPivotIter = new int[Size];
    for (int i = 0; i < Size; i++) {
        pSerialPivotIter[i] = -1;
    }
    // Gaussian elimination
    SerialGaussianElimination(pMatrix, pVector, Size);
    // Back substitution
    SerialBackSubstitution(pMatrix, pVector, pResult, Size);

    // Memory deallocation
    delete[] pSerialPivotPos;
    delete[] pSerialPivotIter;
}

// Function for computational process termination
void ProcessTermination(double* pMatrix, double* pVector, double*
    pResult) {
    delete[] pMatrix;
    delete[] pVector;
    delete[] pResult;
}

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();
SerialResultCalculation(pMatrix, pVector, pResult, Size);
finish = clock();
duration = (finish - start) / CLOCKS_PER_SEC;
// 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;
}

```

1.3 Решение. Параллельная версия

Проведя анализ последовательного варианта алгоритма Гаусса, можно заключить, что распараллеливание возможно для следующих вычислительных операций:

- поиск ведущей строки;
- вычитание ведущей строки из всех строк, подлежащих обработке;
- выполнение обратного хода.

Функция **MPI-Scatter** разбивает сообщение из буфера отправки процесса root на равные части размером sendcount и посылает i-ю часть в буфер приема процесса с номером i (в том числе и самому себе)

Функция **MPI-Gather** производит сборку блоков данных, посылаемых всеми процессами группы, в один массив процесса с номером root. ... То есть данные, посланные процессом i из своего буфера sendbuf, помещаются в i-ю порцию буфера recvbuf процесса root.

Функция **MPI-Allreduce** сохраняет результат редукции в адресном пространстве всех процессов

Функция инициализации параллельной части приложения:

int MPI-Init(int *argc, char** argv)

Функция завершения параллельной части приложения:

int MPI-Finalize(void)

Функция, определяющая количество процессов параллельной программы, входящих в некоторый коммуникатор:

int MPI-Comm-size(MPI-Comm Comm, int *Size)

Функция определения ранга процесса:

int MPI-Comm_{Rank}(MPI – CommComm, int * Rank)

MPI-Bcast отправляет остальным процессам значение заданной пользователем переменной n

int MPI-Bcast(void *buffer, int count, MPI-Datatype datatype, int root, MPI-Comm comm)

MPI-Reduce - объединяет элементы входного буфера каждого процесса в группе, используя операцию op, и возвращает объединенное значение в выходной буфер процесса с номером root.

int MPI-Reduce(void *buf, void *result, int count, MPI-Datatype datatype, MPI-Op op, int root, MPI-Comm comm) Она

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>
#include <math.h>
#include <mpi.h>

int ProcNum; // Number of the available processes
int ProcRank; // Rank of the current process
int* pParallelPivotPos; // Number of rows selected as the pivot ones
int* pProcPivotIter; // Number of iterations, at which the processor
// rows were used as the pivot ones
int* pProcInd; // Number of the first row located on the processes
int* pProcNum; // Number of the linear system rows located on the processes
// Function for random definition of matrix and 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 data initialization
void ProcessInitialization(double*& pMatrix, double*& pVector,
    double*& pResult, double*& pProcRows, double*& pProcVector,
    double*& pProcResult, int& Size, int& RowNum) {
    int RestRows; // Number of rows, that haven't been distributed yet
    int i; // Loop variable
    MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);
    RestRows = Size;
    for (i = 0; i < ProcRank; i++)
        RestRows = RestRows - RestRows / (ProcNum - i);
    RowNum = RestRows / (ProcNum - ProcRank);
    pProcRows = new double[RowNum * Size];
    pProcVector = new double[RowNum];
    pProcResult = new double[RowNum];
    pParallelPivotPos = new int[Size];
    pProcPivotIter = new int[RowNum];

    pProcInd = new int[ProcNum];
    pProcNum = new int[ProcNum];
    for (int i = 0; i < RowNum; i++)
        pProcPivotIter[i] = -1;
    if (ProcRank == 0) {
        pMatrix = new double[Size * Size];
        pVector = new double[Size];
        pResult = new double[Size];
        // DummyDataInitialization (pMatrix, pVector, Size);
        RandomDataInitialization(pMatrix, pVector, Size);
    }
}

// Function for the data distribution among the processes
void DataDistribution(double* pMatrix, double* pProcRows, double* pVector,
    double* pProcVector, int Size, int RowNum) {
    int* pSendNum; // Number of the elements sent to the process
    int* pSendInd; // Index of the first data element sent
    // to the process
    int RestRows = Size; // Number of rows, that have not been
    // distributed yet
    int i; // Loop variable
    // Alloc memory for temporary objects
    pSendInd = new int[ProcNum];
    pSendNum = new int[ProcNum];
    // Define the disposition of the matrix rows for the current process
    RowNum = (Size / ProcNum);

```

```

pSendNum[0] = RowNum * Size;
pSendInd[0] = 0;
for (i = 1; i < ProcNum; i++) {
    RestRows -= RowNum;
    RowNum = RestRows / (ProcNum - i);
    pSendNum[i] = RowNum * Size;
    pSendInd[i] = pSendInd[i - 1] + pSendNum[i - 1];
}
MPI_Scatterv(pMatrix, pSendNum, pSendInd, MPI_DOUBLE, pProcRows,
            pSendNum[ProcRank], MPI_DOUBLE, 0, MPI_COMM_WORLD);
// Define the disposition of the matrix rows for current process
RestRows = Size;
pProcInd[0] = 0;
pProcNum[0] = Size / ProcNum;
for (i = 1; i < ProcNum; i++) {
    RestRows -= pProcNum[i - 1];
    pProcNum[i] = RestRows / (ProcNum - i);
    pProcInd[i] = pProcInd[i - 1] + pProcNum[i - 1];
}
MPI_Scatterv(pVector, pProcNum, pProcInd, MPI_DOUBLE, pProcVector,
↪ pProcNum[ProcRank], MPI_DOUBLE, 0, MPI_COMM_WORLD);
// Free the memory
delete[] pSendNum;
delete[] pSendInd;
}

// Function for gathering the result vector
void ResultCollection(double* pProcResult, double* pResult) {
    //Gather the whole result vector on every processor
    MPI_Gatherv(pProcResult, pProcNum[ProcRank], MPI_DOUBLE, pResult, pProcNum,
↪ pProcInd, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}

// Fuction for the column elimination
void ParallelEliminateColumns(double* pProcRows, double* pProcVector,
    double* pPivotRow, int Size, int RowNum, int Iter) {
    double multiplier;
    for (int i = 0; i < RowNum; i++) {
        if (pProcPivotIter[i] == -1) {
            multiplier = pProcRows[i * Size + Iter] / pPivotRow[Iter];
            for (int j = Iter; j < Size; j++) {
                pProcRows[i * Size + j] -= pPivotRow[j] * multiplier;
            }
            pProcVector[i] -= pPivotRow[Size] * multiplier;
        }
    }
}

}

// Function for the Gaussian elimination
void ParallelGaussianElimination(double* pProcRows, double* pProcVector,
    int Size, int RowNum) {

```

```

double MaxValue; // Value of the pivot element of the process
int PivotPos; // Position of the pivot row in the process stripe
// Structure for the pivot row selection
struct { double MaxValue; int ProcRank; } ProcPivot, Pivot;
// pPivotRow is used for storing the pivot row and the corresponding
// element of the vector b
double* pPivotRow = new double[Size + 1];
// The iterations of the Gaussian elimination stage
for (int i = 0; i < Size; i++) {

    // Calculating the local pivot row
    double MaxValue = 0;
    for (int j = 0; j < RowNum; j++) {
        if ((pProcPivotIter[j] == -1) &&
            (MaxValue < fabs(pProcRows[j * Size + i]))) {
            MaxValue = fabs(pProcRows[j * Size + i]);
            PivotPos = j;
        }
    }
    ProcPivot.MaxValue = MaxValue;
    ProcPivot.ProcRank = ProcRank;
    // Find the pivot process (process with the maximum value of MaxValue)
    MPI_Allreduce(&ProcPivot, &Pivot, 1, MPI_DOUBLE_INT,
↪ MPI_MAXLOC, MPI_COMM_WORLD);
    // Broadcasting the pivot row
    if (ProcRank == Pivot.ProcRank) {
        pProcPivotIter[PivotPos] = i; //iteration number
        pParallelPivotPos[i] = pProcInd[ProcRank] + PivotPos;
    }
    MPI_Bcast(&pParallelPivotPos[i], 1, MPI_INT, Pivot.ProcRank,
↪ MPI_COMM_WORLD);

    if (ProcRank == Pivot.ProcRank) {
        // Fill the pivot row
        for (int j = 0; j < Size; j++) {
            pPivotRow[j] = pProcRows[PivotPos * Size + j];
        }
        pPivotRow[Size] = pProcVector[PivotPos];
    }
    MPI_Bcast(pPivotRow, Size + 1, MPI_DOUBLE, Pivot.ProcRank, MPI_COMM_WORLD);
    ParallelEliminateColumns(pProcRows, pProcVector, pPivotRow, Size, RowNum,
↪ i);
}

// Function for finding the pivot row of the back substitution
void FindBackPivotRow(int RowIndex, int Size, int& IterProcRank,
    int& IterPivotPos) {
    for (int i = 0; i < ProcNum - 1; i++) {

```

```

        if ((pProcInd[i] <=RowIndex) && (RowIndex < pProcInd[i + 1]))
            IterProcRank = i;
    }
    if (RowIndex >= pProcInd[ProcNum - 1])
        IterProcRank = ProcNum - 1;
    IterPivotPos = RowIndex - pProcInd[IterProcRank];
}

// Function for the back substitution
void ParallelBackSubstitution(double* pProcRows, double* pProcVector,
    double* pProcResult, int Size, int RowNum) {
    int IterProcRank; // Rank of the process with the current pivot row
    int IterPivotPos; // Position of the pivot row of the process
    double IterResult; // Calculated value of the current unknown
    double val;
    // Iterations of the back substitution stage
    for (int i = Size - 1; i >= 0; i--) {
        // Calculating the rank of the process, which holds the pivot row
        FindBackPivotRow(pParallelPivotPos[i], Size, IterProcRank,
            IterPivotPos);

        // Calculating the unknown
        if (ProcRank == IterProcRank) {
            IterResult = pProcVector[IterPivotPos] / pProcRows[IterPivotPos *
↵ Size + i];

            pProcResult[IterPivotPos] = IterResult;
        }
        MPI_Bcast(&IterResult, 1, MPI_DOUBLE, IterProcRank, MPI_COMM_WORLD);
        for (int j = 0; j < RowNum; j++)
            if (pProcPivotIter[j] < i) {
                val = pProcRows[j * Size + i] * IterResult;
                pProcVector[j] = pProcVector[j] - val;
            }
    }
}

// Function for the execution of the parallel Gauss algorithm
void ParallelResultCalculation(double* pProcRows, double* pProcVector,
    double* pProcResult, int Size, int RowNum) {
    ParallelGaussianElimination(pProcRows, pProcVector, Size, RowNum);
    ParallelBackSubstitution(pProcRows, pProcVector, pProcResult, Size,
        RowNum);
}

// Function for computational process termination
void ProcessTermination(double* pMatrix, double* pVector, double* pResult,
    double* pProcRows, double* pProcVector, double* pProcResult) {
    if (ProcRank == 0) {
        delete[] pMatrix;
        delete[] pVector;
        delete[] pResult;
    }
}

```

```

    }
    delete[] pProcRows;
    delete[] pProcVector;
    delete[] pProcResult;
    delete[] pParallelPivotPos;
    delete[] pProcPivotIter;
    delete[] pProcInd;
    delete[] pProcNum;
}

// 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
    if (ProcRank == 0) {
        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[pParallelPivotPos[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;
    }
}

void main(int argc, char* argv[]) {
    double* pMatrix; // Matrix of the linear system
    double* pVector; // Right parts of the linear system
    double* pResult; // Result vector
    double* pProcRows; // Rows of the matrix A
    double* pProcVector; // Block of the vector b
    double* pProcResult; // Block of the vector x
    int Size = 15; // Size of the matrix and vectors
    int RowNum; // Number of the matrix rows

```

```

double start, finish, duration;
setvbuf(stdout, 0, _IONBF, 0);
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);

if (ProcRank == 0)
{
    printf("Parallel Gauss algorithm for solving linear systems\n");
    printf("\nChosen size = %d \n", Size);
}

// Memory allocation and data initialization
ProcessInitialization(pMatrix, pVector, pResult,
    pProcRows, pProcVector, pProcResult, Size, RowNum);
// The execution of the parallel Gauss algorithm
start = MPI_Wtime();
DataDistribution(pMatrix, pProcRows, pVector, pProcVector, Size, RowNum);

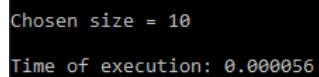
ParallelResultCalculation(pProcRows, pProcVector, pProcResult, Size, RowNum);
ResultCollection(pProcResult, pResult);

finish = MPI_Wtime();
duration = finish - start;
TestResult(pMatrix, pVector, pResult, Size);
// Printing the time spent by Gauss algorithm
if (ProcRank == 0)
    printf("\nTime of execution: %f\n", duration);
// Computational process termination
ProcessTermination(pMatrix, pVector, pResult, pProcRows, pProcVector,
    pProcResult);
MPI_Finalize();
}

```

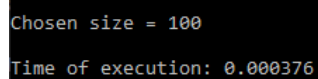
2 Результат работы

Ускорение алгоритма, распараллеленного с помощью MPI, быстрее ускорения алгоритма, распараллеленного с помощью OMP.



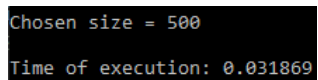
```
Chosen size = 10  
Time of execution: 0.000056
```

Рисунок 1 – Параллельная реализация - Порядок 10



```
Chosen size = 100  
Time of execution: 0.000376
```

Рисунок 2 – Параллельная реализация - Порядок 100



```
Chosen size = 500  
Time of execution: 0.031869
```

Рисунок 3 – Параллельная реализация - Порядок 500

```
Chosen size = 1000
Time of execution: 0.360676
```

Рисунок 4 – Параллельная реализация - Порядок 1000

```
Chosen size = 1500
Time of execution: 1.409349
```

Рисунок 5 – Параллельная реализация - Порядок 1500

```
Chosen size = 2000
Time of execution: 2.668674
```

Рисунок 6 – Параллельная реализация - Порядок 2000

```
Chosen size = 2500
Time of execution: 5.481354
```

Рисунок 7 – Параллельная реализация - Порядок 2500

```
Chosen size = 3000
Time of execution: 8.294152
```

Рисунок 8 – Параллельная реализация - Порядок 3000

Номер теста	Порядок системы	Последовательный алгоритм	Параллельный алгоритм MPI		Параллельный алгоритм OMP	
			Время	Ускорение	Время	Ускорение
1	10	0.0	0.000056	0	0.00001	0
2	100	0.001	0.000376	2,65	0.003785	0.2642
3	500	0.088	0.0318	2,76	0.067338	1.30684012
4	1000	1.0700	0.36	2,972222	0.596365	1.79439
5	1500	3.95400	1.409	2,806	1.951706	2.0259
6	2000	9.47300	2.668674	3,5494	4.547243	2.08326
7	2500	18.30600	5.481354	3,3397	8.735249	2.09565
8	3000	28.7400	8.294152	3,465	13.239482	2.1707

Рисунок 9 – Таблица

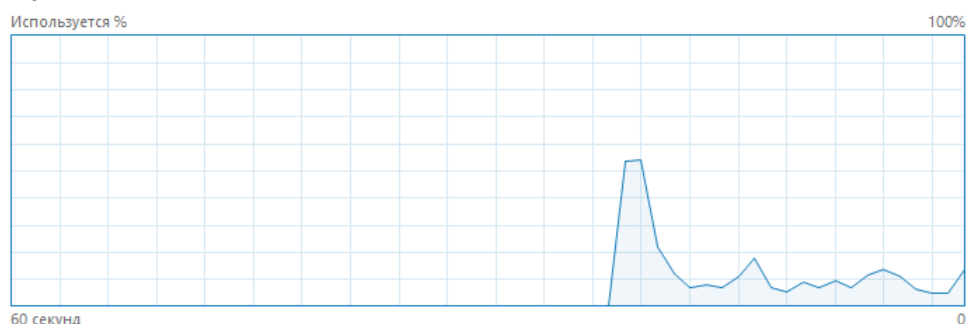
3 Характеристики компьютера

Характеристики устройства

Имя устройства	DESKTOP-MSS8D39
Процессор	Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz 3.20 GHz
Оперативная память	8,00 ГБ
Код устройства	E3BB953D-13B0-42A7-944B-1ED9FD0E C328
Код продукта	00330-80000-00000-AA153
Тип системы	64-разрядная операционная система, процессор x64

ЦП

Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz



Использование	Скорость	Базовая скорости:	3,20 ГГц
14%	3,43 ГГц	Сокетов:	1
Процессы	Потоки	Ядра:	4
220	3285	Логических процессоров:	4
Дескрипторы	Виртуализация:	Включено	
170005	Кэш L1:	256 КБ	
Время работы	Кэш L2:	1,0 МБ	
100:23:51:24	Кэш L3:	6,0 МБ	