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

Кафедра математической кибернетики и компьютерных наук

ОТЧЕТ ПО ПРАКТИКЕ WORK13

ОТЧЕТ

Студента 3 курса 311 группы
направления 02.03.02 — Фундаментальная информатика и информационные технологии
факультета КНиИТ
Забоева Максима Владиславовича

Проверил
Старший преподаватель _____ М. С. Портенко

Саратов 2023

СОДЕРЖАНИЕ

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

2 Практическая часть

Код программы:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>
#include <math.h>
#include <mpi.h>
int ProcNum;
int ProcRank;
int* pParallelPivotPos;
int* pProcPivotIter;
int* pProcInd;
int* pProcNum;
void RandomDataInitialization(double* pMatrix, double* pVector, int Size) {
        int i, j;
        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;
                }
        }
}
void ProcessInitialization(double*& pMatrix, double*& pVector,
        double*& pResult, double*& pProcRows, double*& pProcVector,
        double*& pProcResult, int& Size, int& RowNum) {
        int RestRows;
        int i;
        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];
                RandomDataInitialization(pMatrix, pVector, Size);
        }
}
void DataDistribution(double* pMatrix, double* pProcRows, double* pVector,
        double* pProcVector, int Size, int RowNum) {
        int* pSendNum;
        int* pSendInd;
        int RestRows = Size;
        int i;
```

```
        pSendInd = new int[ProcNum];
        pSendNum = new int[ProcNum];
        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);
        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)
        delete[] pSendNum;
        delete[] pSendInd;
}
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 Gausian elimination
void ParallelGaussianElimination(double* pProcRows, double* pProcVector,
        int Size, int RowNum) {
        double MaxValue;
        int PivotPos;
        struct { double MaxValue; int ProcRank; } ProcPivot, Pivot;
        double* pPivotRow = new double[Size + 1];
        for (int i = 0; i < Size; i++) {
                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;
                MPI_Allreduce(&ProcPivot, &Pivot, 1, MPI_DOUBLE_INT,MPI_MAXLOC, MPI_COMM_WORLD);
                if (ProcRank == Pivot.ProcRank) {
                        pProcPivotIter[PivotPos] = i;
                        pParallelPivotPos[i] = pProcInd[ProcRank] + PivotPos;
                }
```

```
                MPI_Bcast(&pParallelPivotPos[i], 1, MPI_INT, Pivot.ProcRank, MPI_COMM_WORLD);
                if (ProcRank == Pivot.ProcRank) {
                        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);
        }
}
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];
}
void ParallelBackSubstitution(double* pProcRows, double* pProcVector,
        double* pProcResult, int Size, int RowNum) {
        int IterProcRank;
        int IterPivotPos;
        double IterResult;
        double val;
        for (int i = Size - 1; i >= 0; i--) {
                FindBackPivotRow(pParallelPivotPos[i], Size, IterProcRank,
                        IterPivotPos);
                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;
                        }
        }
}
void ParallelResultCalculation(double* pProcRows, double* pProcVector,
        double* pProcResult, int Size, int RowNum) {
        ParallelGaussianElimination(pProcRows, pProcVector, Size, RowNum);
        ParallelBackSubstitution(pProcRows, pProcVector, pProcResult, Size,
                RowNum);
}
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;
}
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);
        finish = MPI_Wtime();
        duration = finish - start;
        // 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();
}
```

3 Результаты работы

3.1 Характеристики компьютера

Процессор — 12th Gen Intel Core i5-12600KF, Базовая скорость 3,70ГГц, Кол-во ядер 10, Кол-во процессоров 16 (включая 4 энергоэффективных ядра). 16гб Оперативной памяти, скорость 3200МГц

3.2 Таблица результатов

Номер теста	Порядок Системы	Последовательный алгоритм	Параллельный алгоритм	
			Время	Ускорение
1	10	0	0,00032	0
2	100	0,001	0,00512	0,1953125
3	500	0,083	0,06352	1,306675063
4	1000	0,61	0,151345	4,030526281
5	1500	1,958	0,329292	5,9460904
6	2000	4,61	0,694422	6,63861456
7	2500	8,841	1,213254	7,28701492
8	3000	15,246	2,854156	5,341684197

3.3 Фото результатов