МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет информационных технологий

Кафедра параллельных вычислений

ОТЧЕТ

О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ

« Параллельная реализация решения системы линейных алгебраических уравнений с помощью MPI»

студента 2 курса, 18209 группы

Большим Максима Антоновича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
Матвеев А.С.

Новосибирск 2020

СОДЕРЖАНИЕ

ЦЕЛЬ

Реализовать многопоточный алгоритм простой итерации для решения СЛАУ с помощью интерфейса MPI.

ЗАДАНИЕ

Реализовать алгоритм из первой лабораторной работы с помощью MPI. Матрица должна быть загружена в корневом процессе и разослана по частям всем остальным процессам для разделённого вычисления. Решение СЛАУ должно быть собрано в корневом процессе для дальнейшей работы с ним как с единым вектором. Замерить время исполнения, проверить эффективность работы алгоритма от N количества доступных ядер.

КОД ПРОГРАММЫ

```
#include <cstdio>
#include <cmath>
#include <cstdlib>
#include <mpi.h>

void calcMatrixParts(int *vecSize, int *vecStartPos, int *matrixSize, int *matrixBeginPos, int processCount);

void loadData(float **A, float **b);

const int N = 2500;
const float epsilon = 1e-5;
float tau = 0.015;

int main(int argc, char **argv)
{
   int processCount;
   int processRank;
   float start = 0;
   float partNorm = 0;
   float sumNorm = 0;
   float normB = 0;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &processCount);//таблица
   MPI_Comm_rank(MPI_COMM_WORLD, &processRank);

   int *vecSize = static_cast<int*>(malloc(sizeof(int) * processCount));
   int *vecStartPos = static_cast<int*>(malloc(sizeof(int) * processCount));
   int *matrixBeginPos = static_cast<int*>(malloc(sizeof(int) * processCount));
   int *matrixSize = static_cast<int*>(malloc(sizeof(int) * processCount));

   auto *A = static_cast<float *>(malloc(sizeof(float) * N * N));
   auto *b = static_cast<float *>(malloc(sizeof(float) * N));
   auto *x = static_cast<float *>(malloc(sizeof(float) * N));

   if (processRank == 0)
```

```
    loadData(&A, &b);

  calcMatrixParts(vecSize, vecStartPos, matrixSize, matrixBeginPos, processCount);

  printf("I'm %d from %d processes and my lines: %d-%d (%d lines)\n", processRank, processCount,
matrixBeginPos[processRank] / N,
      (matrixBeginPos[processRank] + matrixSize[processRank]) / N, matrixSize[processRank] / N);

  auto *buf0 = static_cast<float *>(malloc(sizeof(float) * vecSize[processRank]));
  auto *buf1 = static_cast<float *>(malloc(sizeof(float) * N));

  auto *partA = static_cast<float *>(malloc(sizeof(float) * vecSize[processRank] * N));
  auto *partB = static_cast<float *>(malloc(sizeof(float) * vecSize[processRank]));

  MPI_Scatterv(A, matrixSize, matrixBeginPos, MPI_FLOAT, partA, matrixSize[processRank], MPI_FLOAT,
0,MPI_COMM_WORLD);
  MPI_Scatterv(b, vecSize, vecStartPos, MPI_FLOAT, partB, vecSize[processRank], MPI_FLOAT, 0,
MPI_COMM_WORLD);

  if (processRank == 0)
  {
    start = MPI_Wtime();

    for (int i = 0; i < N; ++i)
      normB += b[i] * b[i];

    normB = sqrt(normB);
  }

  bool flag = true;
  bool diverge0 = false;
  bool diverge1 = false;
  bool setOld = false;
  float oldValue = 0;
  int divergeCount = 0;
  int normCount = 0;

  while (flag)
  {
    //Ax - b
    for (int i = 0; i < vecSize[processRank]; i++)
    {
      float sum = 0;
      for (int j = 0; j < N; j++)
        sum += partA[i * N + j] * x[j];

      buf0[i] = sum - partB[i];
      partNorm += buf0[i] * buf0[i];//calc norm |Ax-b|
    }

    MPI_Reduce(&partNorm, &sumNorm, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
    MPI_Allgatherv(buf0, vecSize[processRank], MPI_FLOAT, buf1, vecSize, vecStartPos, MPI_FLOAT,
            MPI_COMM_WORLD);

    //x - tau(Ax-b)
    for (int i = 0; i < N; i++)
      x[i] = x[i] - tau * buf1[i];

    if (processRank == 0)
    {
      sumNorm = sqrt(sumNorm);
      flag = sumNorm / normB > epsilon;
```

```
        if (!setOld)
            setOld = true;
        else if (oldValue < sumNorm / normB)
            divergeCount++;
        else
            normCount++;

        if(normCount > 30)
        {
            divergeCount = 0;
            normCount = 0;
        }

        std::cout << sumNorm / normB << std::endl;

        oldValue = sumNorm / normB;

        if (divergeCount > 50)
        {
            if (diverge0)
            {
                diverge1 = true;
                flag = false;
            } else
            {
                diverge0 = true;
                tau *= -1;
                divergeCount = 0;
                normCount = 0;
            }
        }
    }

    MPI_Bcast(&tau, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&flag, 1, MPI_INT, 0, MPI_COMM_WORLD);
    partNorm = 0;
    sumNorm = 0;
}

if (processRank == 0)
{
    float end = MPI_Wtime();
    if(diverge1)
        std::cout << "We haven't solution!" << std::endl;
    else
    {
        FILE *ifsAns = fopen("vecX.bin", "r");
        float a = 0;

        for (int i = 0; i < N; ++i)
        {
            fread(((void *) &a), sizeof(float), 1, ifsAns);
            std::cout << a << " = " << x[i] << std::endl;
        }
    }

    std::cout << "Processes: " << processCount << ", time: " << (end - start) << std::endl;
}


delete[](vecSize);
delete[](matrixSize);
delete[](vecStartPos);
```

```
    delete[](matrixBeginPos);
    delete[](x);
    delete[](buf0);
    delete[](partA);
    delete[](partB);
    if (processRank == 0)
    {
        delete[](A);
        delete[](b);
    }
    MPI_Finalize();
    return 0;
}

void loadData(float **A, float **b)
{
    FILE *A_INP = fopen("matA.bin", "rb");
    FILE *B_INP = fopen("vecB.bin", "rb");

    fread((*A), sizeof(float), N * N, A_INP);
    fread((*b), sizeof(float), N, B_INP);

    fclose(A_INP);
    fclose(B_INP);
}

void calcMatrixParts(int *vecSize, int *vecStartPos, int *matrixSize, int *matrixBeginPos, int processCount)
{
    int curOffset = 0;
    for (int i = 0; i < processCount; i++)
        vecSize[i] = N / processCount;

    for (int i = 0; i < processCount; i++)
    {
        if (i < N % processCount) vecSize[i]++;
        vecStartPos[i] = curOffset;
        curOffset += vecSize[i];
        matrixBeginPos[i] = vecStartPos[i] * N;
        matrixSize[i] = vecSize[i] * N;
    }
}
```

ТАБЛИЦА ПРОИЗВОДИТЕЛЬНОСТИ

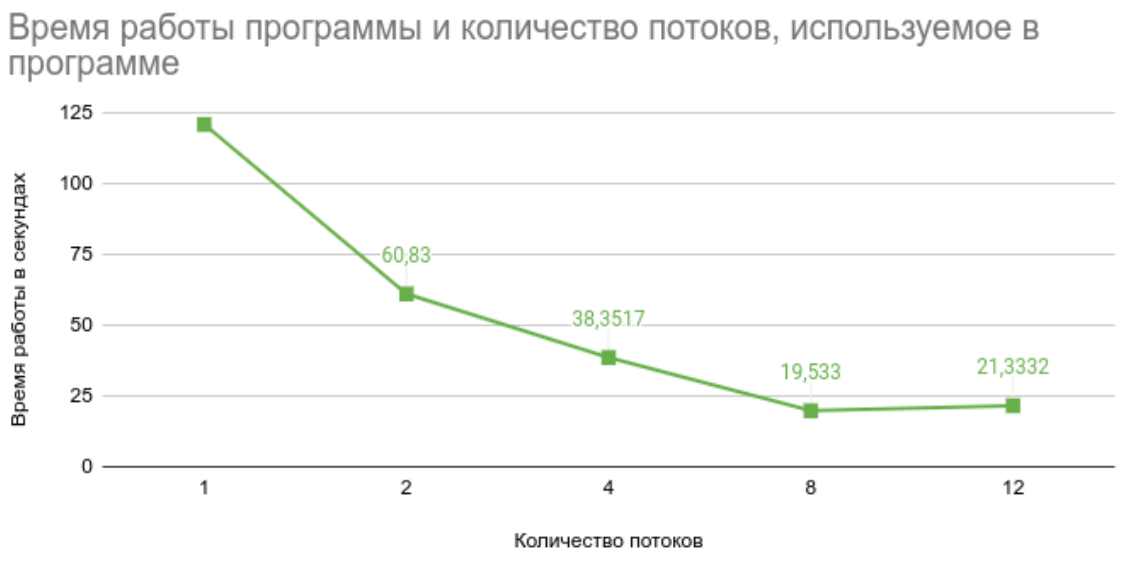

ТАБЛИЦА ЭФФЕКТИВНОСТИ

Ядер	1	2	4	8	12
Эффективность	1	0.99	0,78	0,77	0,47

ЗАКЛЮЧЕНИЕ

В ходе лабораторной работы я научился разделять алгоритм на процессы при помощи MPI. Программа выдает хорошие результаты. Тесты показывают, что MPI может быть хорошим аналогом OpenMP.