Федеральное государственное автономное образовательное   
учреждение высшего образования  
Санкт-Петербургский политехнический университет Петра Великого

Физико-механический институт  
Высшая школа прикладной математики и вычислительной физики

**КУРСОВАЯ РАБОТА**

**Разработка параллельных программ численного решения двумерного уравнения Пуассона на основе технологии MPI**

по дисциплине

«Вычисления на многопроцессорных компьютерах»

Выполнил работу:

студент группы 5040301/40501

Кучиев Д.Ю.

Преподаватель:

доцент ВШПМиВФ, А.Г. Абрамов

“\_\_\_\_” \_\_\_\_\_\_\_\_\_\_\_ 2025 г.

Санкт-Петербург

2025

1. **Введение**
   1. **Цель работы**

Практическое применение освоенной в рамках учебного курса технологии параллельного программирования MPI для разработки программы, распараллеливающей традиционные численные алгоритмы решения двумерного уравнения Пуассона.

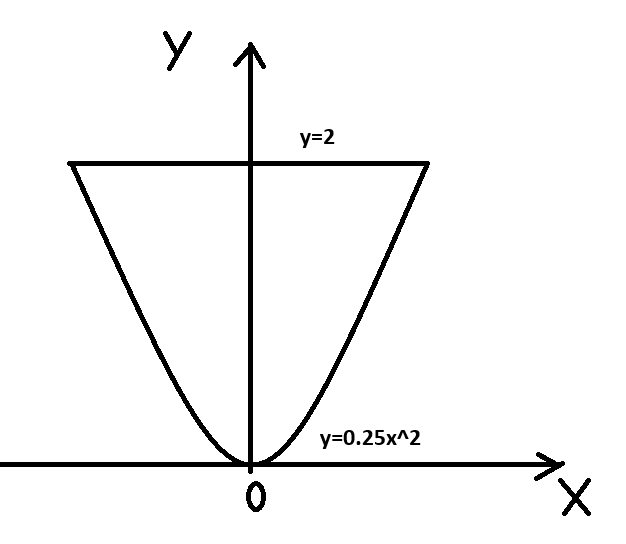
* 1. **Задание к работе**

Разработать программы, параллелизующую с помощью технологии MPI метод Якоби (простой итерации) численного решения двумерного уравнения Пуассона.

Вариант задания – 4*:* 1D декомпозиция, использование для обмена данными между соседними процессами процедур MPI\_ISEND и MPI\_IRECV (неблокирующие обмены), а таже процедуры MPI\_WAITALL для блокирующего ожидания завершения всех обменов.

Вид уравнения и граничных условий:

Граница области, в которой ищется решение уравнения Пуассона изображена на рисунке 1.

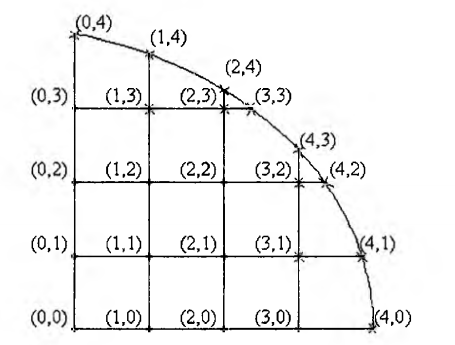
  
Рис. 1. Граница области, в которой ищется решение уравнения для

* 1. **Метод сеток**

Для численного решения уравнения Пуассона используется метод сеток, описанный в [1]. Суть метода заключается в следующем: область, в которой решается задача, заключают в прямоугольник, полностью её охватывающий. Этот прямоугольник равномерно разбивается сеткой — по каждому направлению проводится заданное количество разбиений (параметры NI и NJ подбираются с учётом требуемой точности). В результате получается сетка, узлы которой обозначаются парами индексов (i, j).

Часть узлов располагается непосредственно на границе области Γ (граничные узлы), часть — внутри области, но вблизи границы, на расстоянии менее шага сетки (приграничные узлы), другие — внутри области и на расстоянии не менее одного шага от границы (внутренние узлы). Оставшиеся узлы находятся за пределами области и не участвуют в вычислениях.

Пример такого сеточного разбиения приведён на рисунке 2. Значения функции в граничных узлах определяются граничными условиями и могут быть вычислены точно. Значения функции в остальных (внутренних) узлах находятся приближённо с использованием численных схем, описанных в следующем разделе.

  
Рис. 2. Пример сеточного разбиения для четверти круга [1]

* 1. **Приближенное решение уравнения Пуассона**

Искомая непрерывная функция заменяется на сеточную функцию принимающую дискретное количество значений в узлах расчетной сетки . Для поиска этих значений во внутренних узлах выполняется дискретизация уравнения Пуассона: точные значения вторых производных заменяются на приближенные, вычисленные на основе узловых значений с некоторой точностью.

Подставляя эти выражения в уравнение Пуассона, записанное для внутреннего узла , получаем со вторым порядком точности:

Отсюда можно выразить приближенное значение через значения сеточной функции в прилежащих узлах с погрешностью :

Значения в граничных узлах сразу определяются через функцию , а величины в приграничных узлах ищутся по формуле Коллатца [1]:

где – точка пересечения и сеточной линии , – расстояние между точкой и узлом . Здесь в приграничном узле ищется вдоль линии , по аналогии можно искать вдоль линии через значение . Стоит учесть, что при использовании данной формулы локальное направление возрастания координат и (а, соответственно, и индексов и ) может не совпадать с глобальным.

В методе Якоби на первой итерации функции во внутренних узлах присваиваются некоторые значения, а затем определяются новые величины по формулам выше. Далее происходит итерационное уточнение функции. В данном методе значения на следующей итерации вычисляются только по значениям на предыдущей итерации.

* 1. **Характеристики компьютера и программного обеспечения, использованных в работе**

Количество ядер компьютера: 8

Объем оперативной памяти: 16 Гбайт

**Исходные коды разработанных программ**

* 1. **Последовательная версия программы численного решения двумерного уравнения Пуассона методом Якоби**

Код последовательной программы, реализующей метод Якоби представлен в приложении А.

Данная последовательная программа осуществляет приближенное решение методом простых итераций (методом Якоби) дифференциального уравнения в частных производных (уравнения Пуассона), представленного в виде . Для численного определения значений функции во внутренних и приграничных узлах сетки используются формулы из пункта 1.4. Выход из итерационного процесса происходит либо при достижении максимального количества итераций, либо при уменьшении максимальной по модулю разницы между соответствующими узловыми значениями функции для двух последовательных итераций до заданной величины.

**1.7 Параллелизация метода Якоби с помощью MPI**

Код параллельной программы, реализующей метод Якоби, представлен в приложении Б.

В параллельной программе параметры сетки и количество итераций считываются из файла только процессором с номером 0. Эти значения рассылаются всем другим процессам с помощью процедуры MPI\_BCAST. Координаты узлов сетки записываются в массив только в нулевом процессе, начальное приближение для искомой функции также определяется только в нем. Далее в этом же процессе определяются индексы приграничных узлов, которые затем рассылаются в виде массивов всем прочим процессам снова путем применения MPI\_BCAST.

Одномерная декомпозиция осуществляется путем предварительного определения оптимального количества узлов расчетной области (рабочих узлов) на 1 процесс. Далее в цикле по индексу (по горизонтальным линиям) на основе полученной величины для каждого процесса определяется номер первой и последней линий блока, с которым тот будет работать. Помимо индексов линий определяются смещение и количество элементов для каждого блока для последующего неравномерного распределения между процессами. Индексы крайних линий записываются в массивы процесса с номером 0, поэтому для передачи другим процессам выполняются их разбиение и отправка с помощью процедур MPI\_SCATTER. Неравномерное распределение сеточной функции осуществляется применением процедуры MPI\_SCATTERV, для которой можно определить различные смещения и размеры для блоков элементов, отправляемых процессам.

Затем программа входит в цикл по итерациям, выход из которого происходит либо при достижении максимального заданного числа итераций, либо при достижении необходимой точности. Внутри цикла на каждой итерации находится новое приближение по методу Якоби для локальной для данного процесса части сеточной функции . Одна итерация по методу Якоби для блока, находящегося в данном процессе, состоит из следующих этапов: определение значений функции в приграничных узлах; нахождение нового приближения для внутренних узлов блока на основе приближения с предыдущей итерации; обмен соседних блоков значениями в узлах на крайних горизонтальных линиях друг с другом; вычисление новых значений функции на этих линиях; определение максимальной по модулю ошибки между значениями сеточной функции для двух последовательных итераций.

Вычисление значений сеточной функции во внутренних и приграничных (в отношении всей расчетной области) узлах выполняется с помощью формул, приведенных в пункте 1.4. Для этого также используются массивы с индексами приграничных узлов.

Обмен блоков значениями в узлах на крайних горизонтальных линиях осуществляется с помощью процедур неблокирующих приема и передачи MPI\_ISEND и MPI\_IRECV. Крайний нижний блок обменивается только с верхним блоком, крайний верхний – только с нижним. Для блокирующего ожидания всех начатых обменов используется процедура MPI\_WAITALL. После обмена в блоке данного процесса можно определить новые значения функции в узлах на крайних горизонтальных линиях (только с одной стороны для крайних блоков).

Наконец, после обновления сеточной функции для всех узлов блока можно определить максимальную по модулю ошибку среди всех узлов для приближений, полученных на двух последовательных итерациях. Для определения ошибки, максимальной для всей области (то есть для всех процессов), и ее записи в переменную каждого процесса используется процедура MPI\_ALLREDUCE с операций MPI\_MAX.

Таким образом, после выполнения итерации в каждом процессе содержится обновленная сеточная функции для данного блока и ошибка, максимальная для всей области. Если ошибка становится меньше требуемой, то все процессы выходят из цикла. Иначе цикл продолжается, пока этого не случится или не пройдет максимальной количество итераций.

После выхода из цикла в процессе с номером 0 происходит сборка всех локальных блоков (массивов локальных сеточных функций) в массиве значений искомой функции с помощью функции MPI\_GATHERV на основе смещений и размеров блоков, вычисленных ранее. Затем в нулевом процессе происходит запись решения в файл и вывод некоторых характеристик итерационного процесса (времени, числа итераций, количества рабочих узлов).

1. **Исследование последовательного алгоритма**
   1. **Верификация**

Для проверки правильности реализации последовательного алгоритма сравнивались значения искомой функции в точке (0; 1) для сеток разной размерности. Рассматривались сетки с размерностями : , , , . Данным сеткам соответствовало количество рабочих узлов : , , , . На основании полученных данных была заполнена таблица 1, в которой представлены значения для указанной точки для всех сеток, а также относительная разница между значениями в точках для текущей и последующей сеток.

Таблица 1. Сходимость к точному решению последовательного алгоритма для метода Якоби при измельчении сетки.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Координаты точек | Значение - разница значений | Количество рабочих узлов | | | |
|  |  |  |  |
|  |  | 1.046095 | 1.047908 | 1.048996 | 1.049325 |
|  | 0.17 | 0.10 | 0.03 | - |

Из таблицы 1 видно, что при увеличении числа рабочих узлов сетки в 4 раза (для последней сетки в 2 раза), разница в значениях функции уменьшается для двух соседних сеток примерно в 2-3 раза. Это говорит о том, что рассмотренный численный метод решения двумерного уравнения Пуассона действительно обеспечивает сходимость к точному решению при измельчении расчетной сетки и что он был реализован корректно.

* 1. **Изучение сходимости**

В предыдущем пункте было установлено, что при измельчении сетки метод Якоби сходится к точному решению, однако при работе алгоритма с каждой конкретной сеткой получается свое приближенное решение, которое находится, когда норма разности текущего и предыдущего приближений меньше выбранной величины ошибки. Исследуем влияние количества рабочих узлов сетки на сходимость к соответствующим приближенным решениям, рассмотрев зависимость от размерности сетки количества итераций, времени выполнения основной области программы и количества итераций в секунду для обоих методов. Указанные данные приведены в таблице 2.

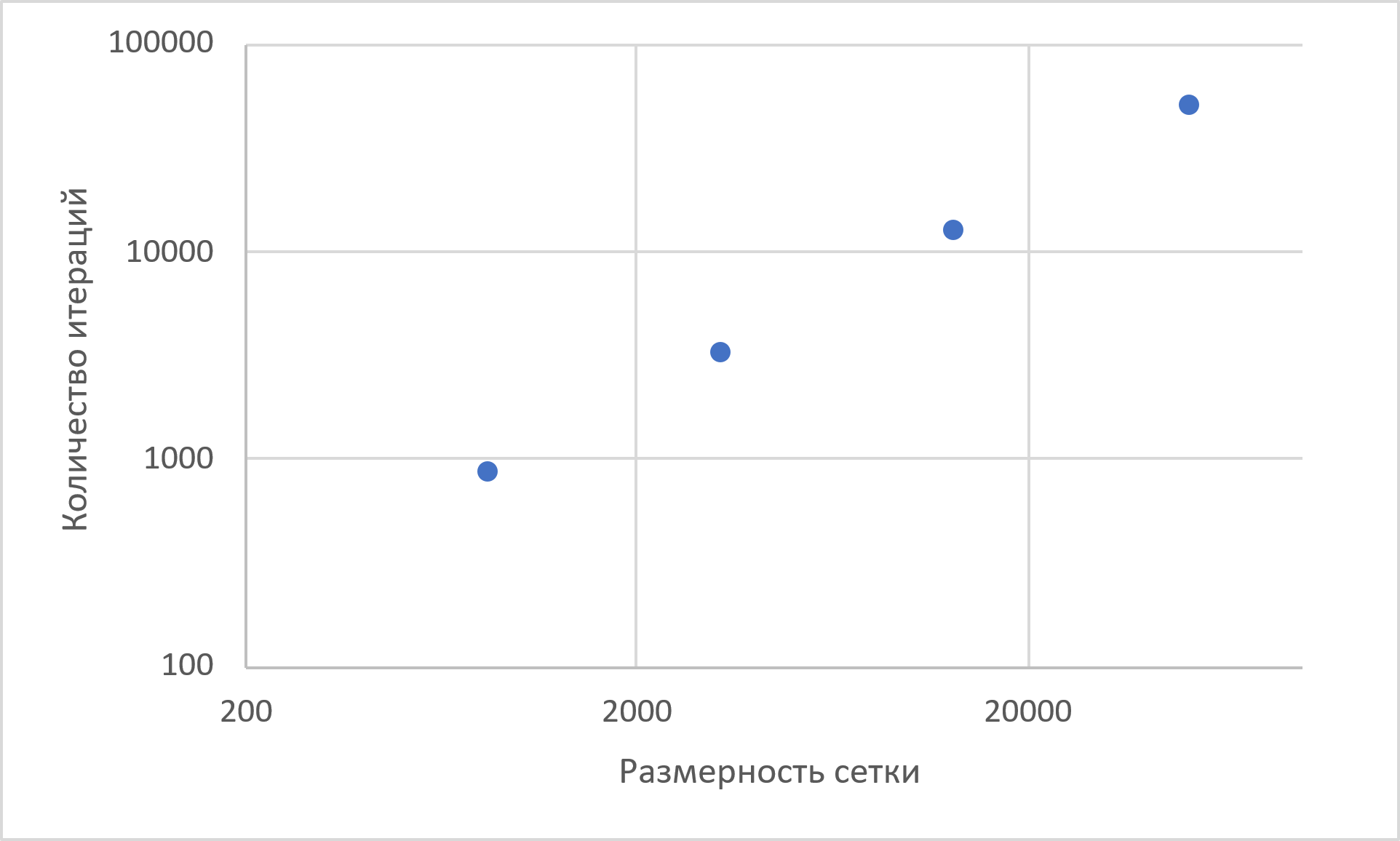
Таблица 2. Влияние размерности сетки на скорость сходимости к решению для последовательного алгоритма

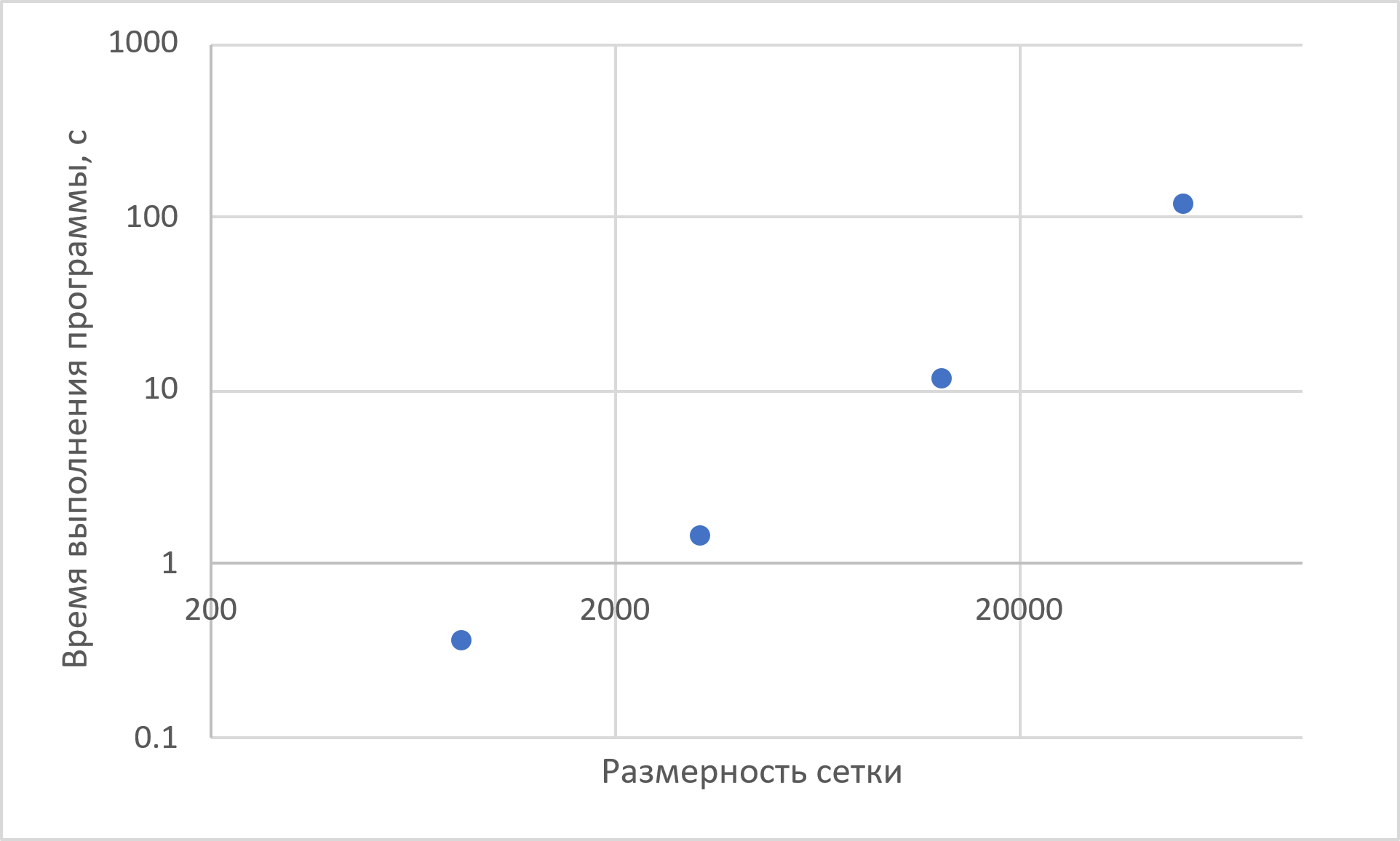
|  |  |  |  |
| --- | --- | --- | --- |
| Количество рабочих узлов | Количество итераций | Время выполнения , с |  |
| 831 | 887 | 0.38 | 2334 |
| 3261 | 3377 | 1.52 | 2221 |
| 12905 | 13138 | 12.03 | 1092 |
| 51401 | 51874 | 123.09 | 421 |

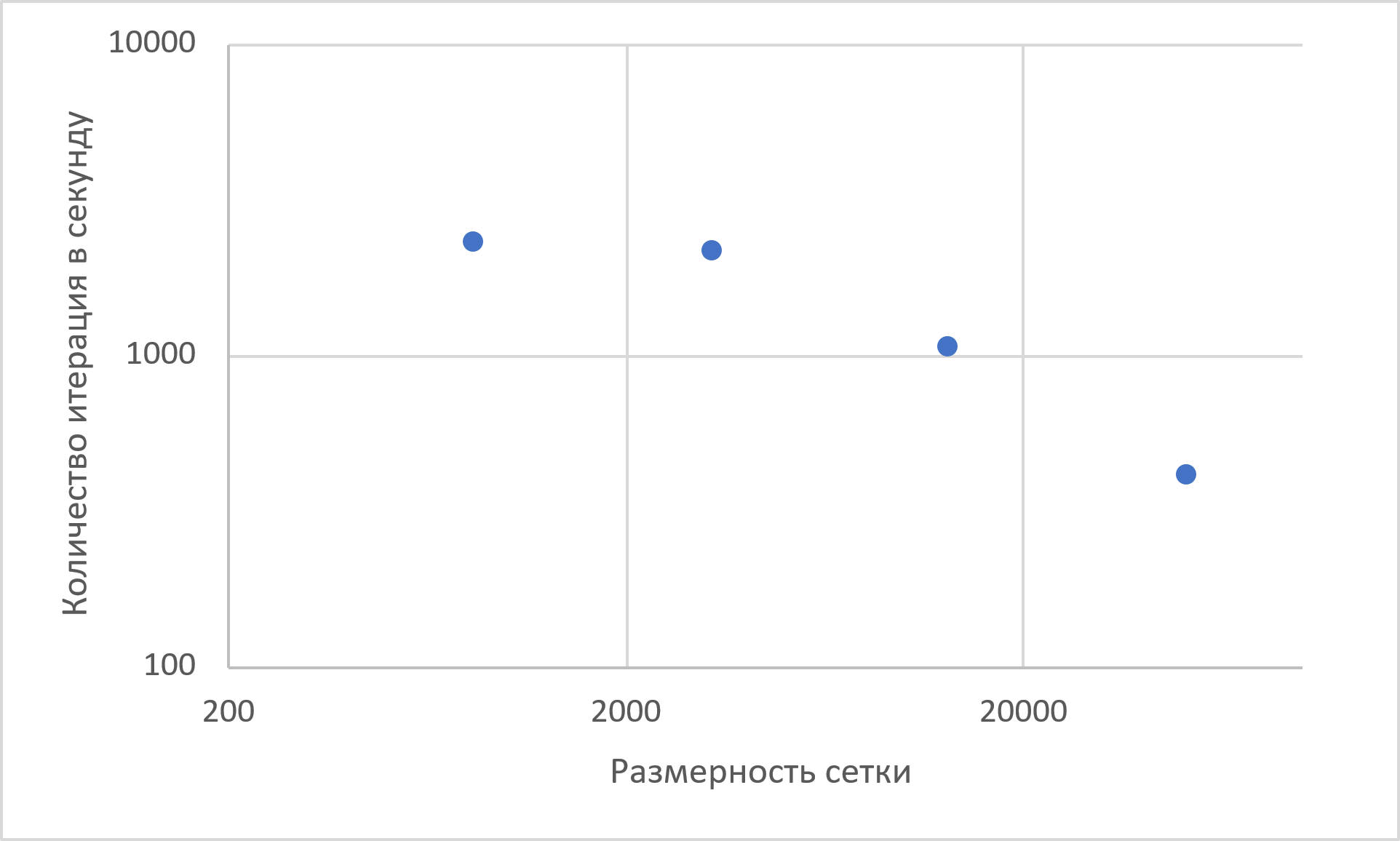
На основании таблицы 3 были построены графики зависимостей всех характеристик сходимости от количества узлов сетки, изображенные на рисунках 3, 4 и 5.

Из рисунка 3 видно, что число итераций в логарифмическом масштабе растет линейно, то есть данная величина возрастает примерно в то же число раз, во сколько раз увеличивается количество узлов. Это связано с тем, что изменение значения функции пропорционально шагу сетки в квадрате, поэтому при его уменьшении в два раза, количество итераций растет примерно в 3-4, но зато возрастает точность решения.

На основании рисунка 4 можно установить, что время выполнения программ зависит от качественно абсолютно также, как и количество итераций, только оно уже увеличивается более чем на порядок при увеличении количества рабочих узлов в 4 раза. Это объясняется тем, что и увеличиваются в 2 раза, поэтому в каждой итерации число обрабатываемых элементов массива и, соответственно, число операций возрастают в 4 раза наряду с тем, что и общее число итераций вырастает в 3-4. Отсюда следует и уменьшение примерно в 4 раза скорости выполнения итераций (рисунок 5), что соответствует возрастанию количества итераций и времени.

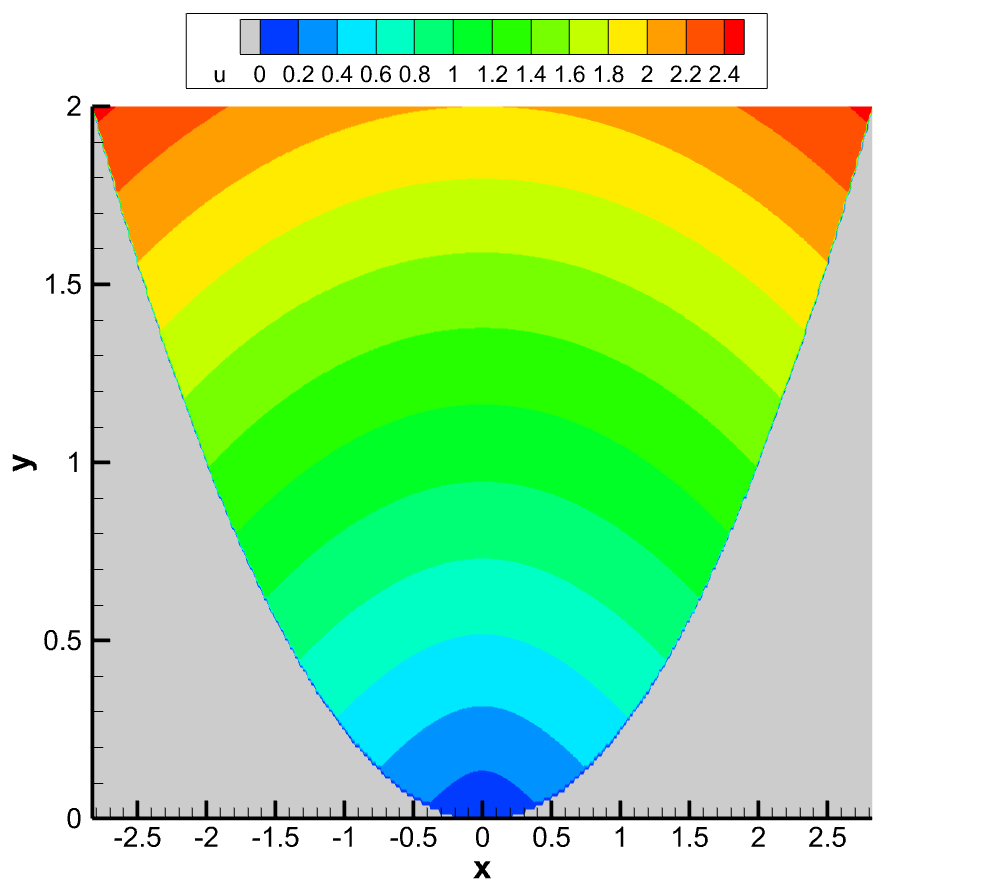
  
Рис. 3. Зависимость числа итераций в последовательной программе от размерности сетки

  
Рис. 4. Зависимость времени выполнения основной области последовательной программы от размерности сетки

  
Рис. 5. Зависимость количества итераций, выполняемых в секунду в последовательной программе, от размерности сетки

* 1. **Визуализация результата**

Так как является функцией двух переменных и , то для визуализации решения была построено поле, представленное на рисунке 6. При этом, стоит отметить, что на границе Г наблюдается негладкая линия, вызванная дискретизацией при помощи прямоугольников. Строго говоря на рисунке представлена вся расчетная область, просто область за Г закрашена в серый цвет.

  
Рис. 6. Поверхность, построенная по значениям в узлах расчетной сетки

1. **Исследование параллелизации**
   1. **Верификация параллелизации**

Для проверки корректности распараллеливания метода Якоби при количестве процессов, равном 4, сравнивались значения функции (в точке, в которой проверялась последовательная версия программы), полученные для сеток разных размерностей, а также относительные отличия для двух последовательных сеток. Указанные данные представлены в таблице 3.

Таблица 3. Сходимость к точному решению параллельного алгоритма для метода Якоби при 4 задействованных процессах при измельчении сетки.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Координаты точек | Значение / разница значений | Количество рабочих узлов | | | |
|  |  |  |  |
|  |  | 1.043658 | 1.047213 | 1.048490 | 1.048988 |
|  | 0.34 | 0.12 | 0.05 | - |

Из таблицы 3 видно, что для параллельного алгоритма также наблюдается сходимость метода Якоби. При сопоставлении с таблицей 1 также можно обнаружить, что программы Jacobi\_serial и Jacobi\_MPI дают близкие результаты для самой мелкой сетки. Отличия могут быть связаны с тем, что в параллельной программе выполняется большее количество вычислений, из-за чего имеется больше источников погрешности.

* 1. **Изучение сходимости параллельного алгоритма**

По аналогии с ситуацией для последовательной программы исследуем влияние количества рабочих узлов сетки на сходимость к приближенному решению в параллельной программе, рассмотрев зависимость от размерности сетки количества итераций и количества итераций в секунду при 4 процессах, а также времени выполнения при числе процессов, равном 1, 2, 4 и 8. Указанные данные представлены в таблицах 4, 5, 6. Стоит отметить что при размере рабочих узлов 831 скорость выполнения параллельной версии программы сильно большая, поэтому этот вариант не рассматривался. То есть время сводилось к погрешности.

Таблица 4. Влияние размерности сетки на скорость сходимости к решению для параллельного алгоритма при 4 процессах

|  |  |  |
| --- | --- | --- |
| Количество рабочих узлов | Количество итераций |  |
| 831 | 1220 | 262777 |
| 3261 | 4245 | 67495 |
| 12905 | 14298 | 19234 |
| 51401 | 46619 | 5394 |

Таблица 5. Зависимость времени работы от количества процессов и сетки

|  |  |  |  |
| --- | --- | --- | --- |
| **p\N** | 3261 | 12905 | 51401 |
| 1 | 0.13 | 1.77 | 22.45 |
| 2 | 0.08 | 1.03 | 13.21 |
| 4 | 0.04 | 0.74 | 8.24 |
| 8 | 0.05 | 0.54 | 6.61 |

Таблица 6. Зависимость ускорения работы от количества процессов и сетки

|  |  |  |  |
| --- | --- | --- | --- |
| **p\N** | 3261 | 12905 | 51401 |
| 1 | 1.00 | 1.00 | 1.00 |
| 2 | 1.63 | 1.72 | 1.70 |
| 4 | 3.25 | 2.39 | 2.72 |
| 8 | 2.60 | 3.28 | 3.40 |

Таблица 7. Зависимость эффективности работы от количества процессов и сетки

|  |  |  |  |
| --- | --- | --- | --- |
| **p\N** | 3261 | 12905 | 51401 |
| 1 | 1.00 | 1.00 | 1.00 |
| 2 | 0.81 | 0.86 | 0.85 |
| 4 | 0.81 | 0.60 | 0.68 |
| 8 | 0.33 | 0.41 | 0.42 |

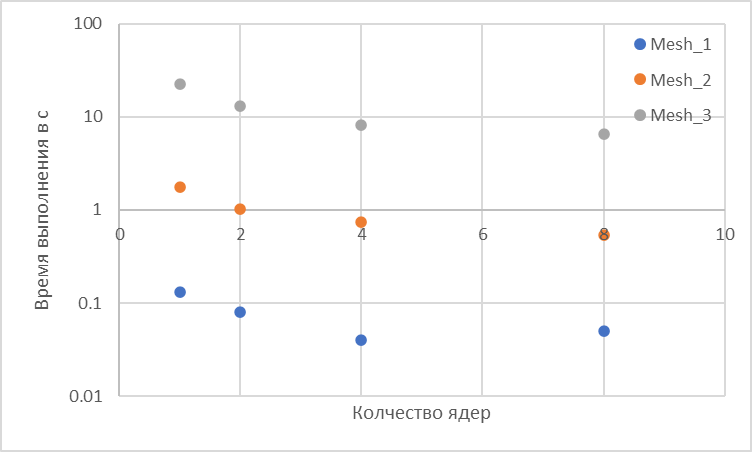
****

Рис. 7 Зависимость времени выполнения от количества ядер

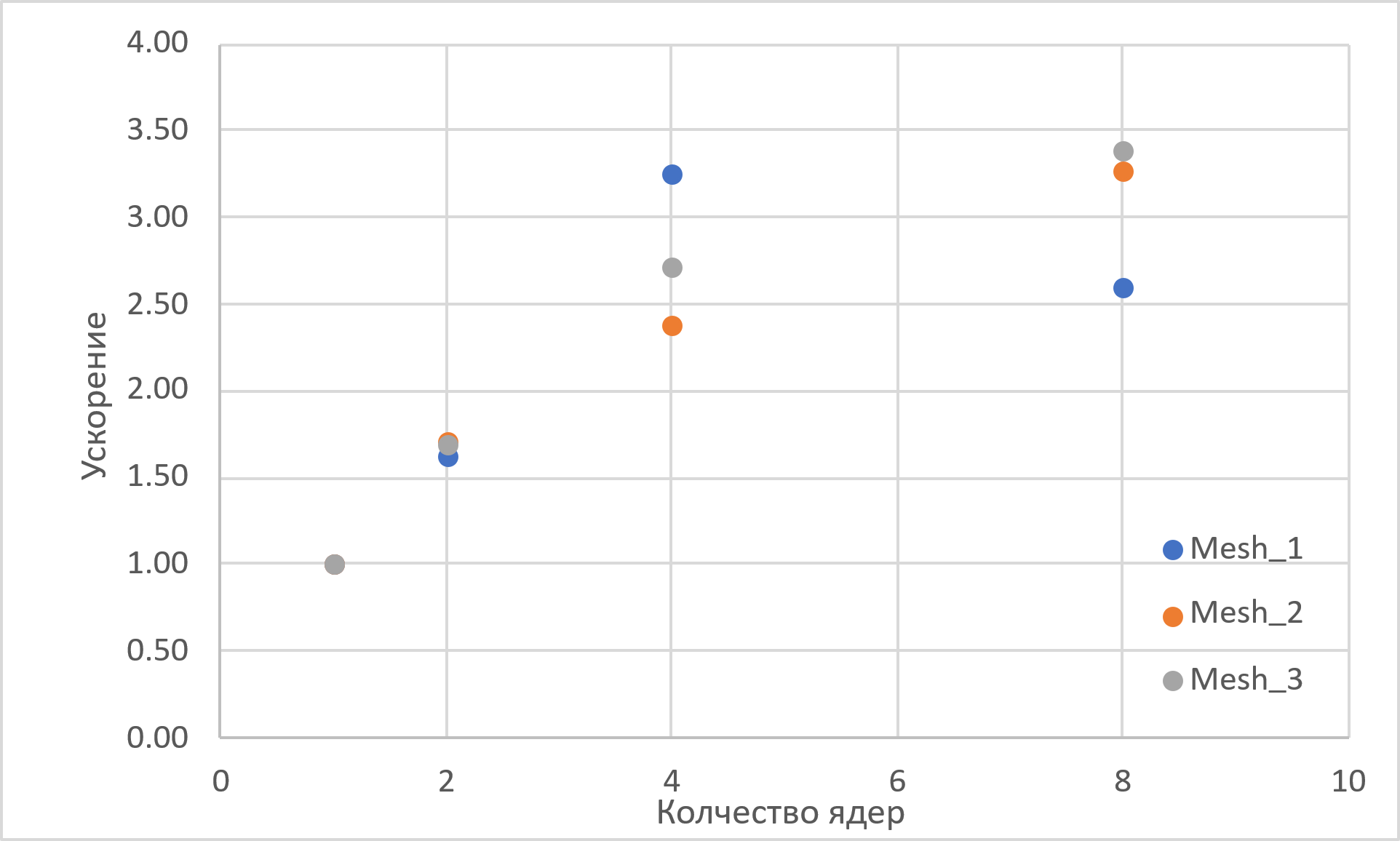
****

Рис. 8 Зависимость ускорения от количества ядер

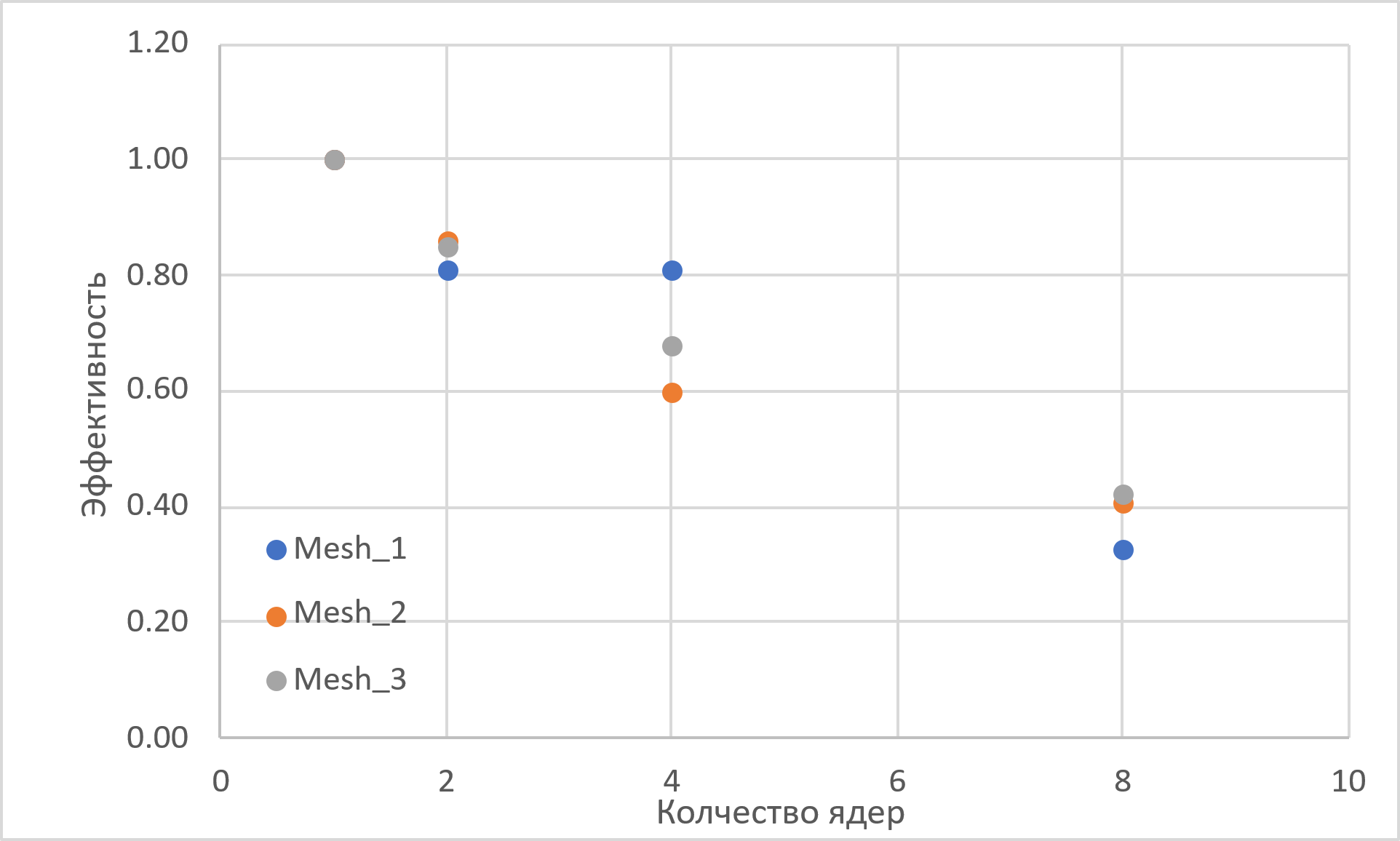
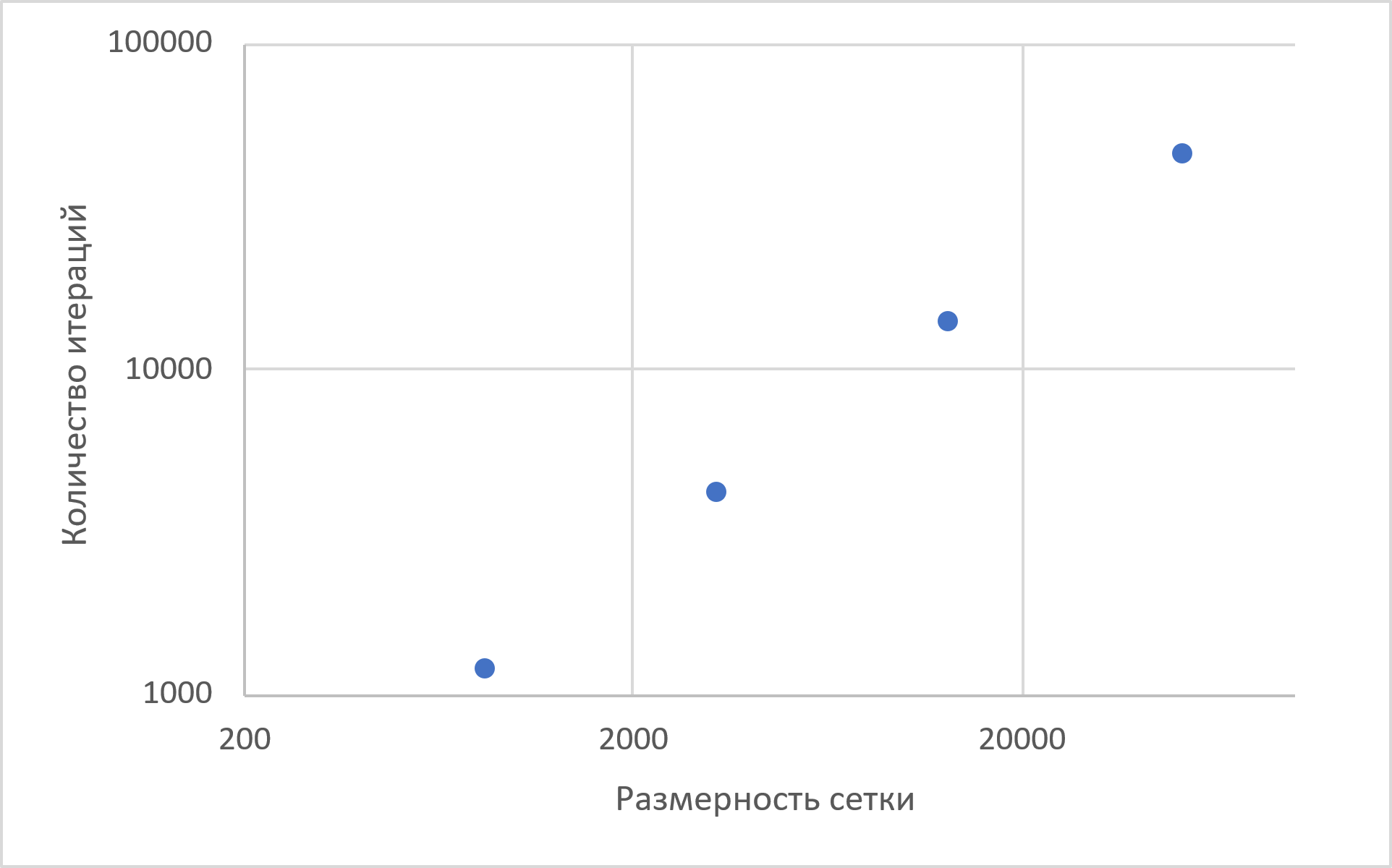
****

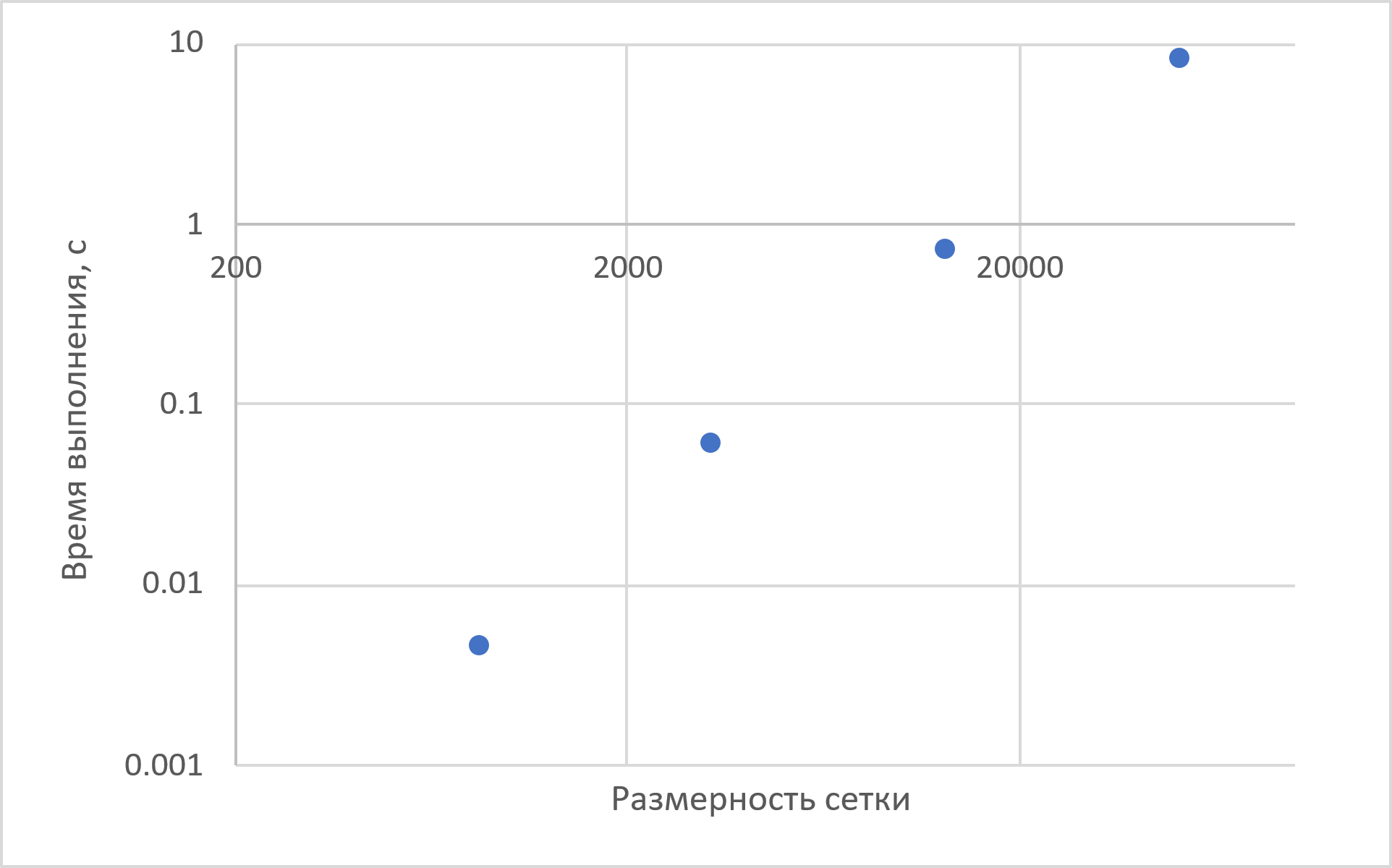
Рис. 9 Зависимость эффективности от количества ядер

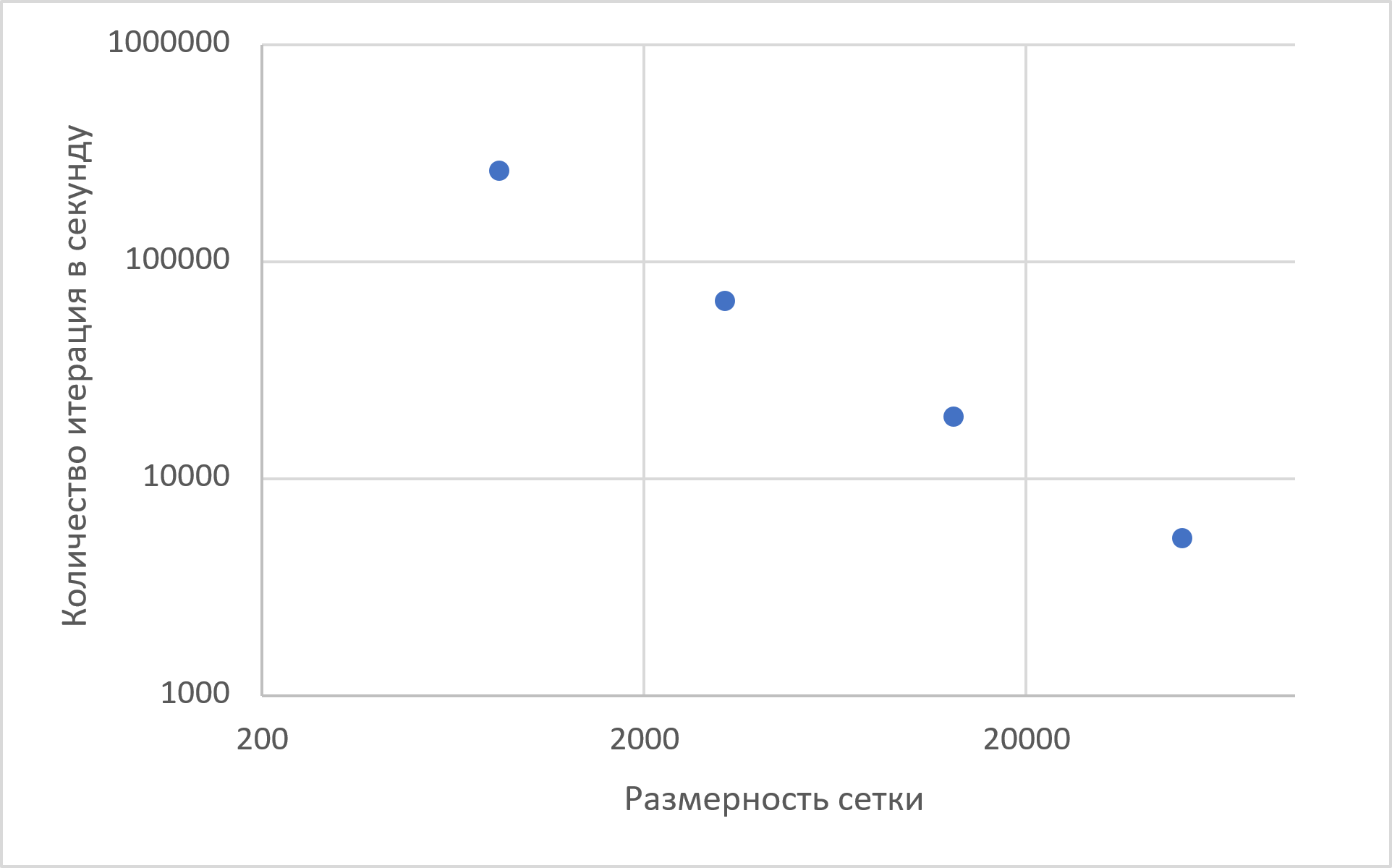
Видно, из таблицы 6, что ускорение , при увеличении количества ядер увеличивается, но при 8 ядрах не так сильно как при изменении с 2 до 4. Также стоит отметить что при малой сетке имеем понижение ускорения при увеличении количество ядер до 8.

При рассмотрении эффективности, , наблюдаем примерно монотонную картину изменения при увеличении количества ядер. Стоит отметить, что эффективность падает ниже единицы с увеличением ядер, что может быть связано с плохой работой с кэш памятью.

Также представим в графическом представлении зависимости времени, количества итераций и количества итераций в секунду от размера сетки на рисунках 10, 11, 12. Поведение аналогично тому, которое было в случае последовательной программы, однако, порядок величин другой. Хоть количество итераций для достижения точности осталось примерно таким же, время выполнения программы сильно уменьшилось, что в свою очередь сильно увеличило количество итераций выполняемых в секунду и при этом сохранило тенденцию аналогичную последовательной программе.

  
Рис. 10. Зависимость числа итераций в последовательной программе от размерности сетки

  
Рис. 11. Зависимость времени выполнения основной области последовательной программы от размерности сетки

  
Рис. 12. Зависимость количества итераций, выполняемых в секунду в последовательной программе, от размерности сетки

1. **Выводы**

В результате выполнения данной курсовой работы были написаны последовательная и распараллеленная с помощью технологии MPI программы, осуществляющие нахождение приближенного решения двумерного уравнения Пуассона с неоднородными граничными условиями для криволинейной области методом Якоби.

Эксперименты с изменением плотности расчётной сетки демонстрируют, что оба алгоритма — последовательный и параллельный — обеспечивают численные решения, приближающиеся к точному по мере её измельчения. Это подтверждается многократным уменьшением расхождений в значениях искомой функции между соседними по размеру сетками. При этом результаты для функции u, полученные разными методами, остаются близкими, что свидетельствует о корректности реализации обоих подходов.

Количество итераций, требуемое для достижения заданной точности, оказалось сопоставимым для обеих программ и растёт пропорционально числу узлов сетки. Однако производительность алгоритмов различается: для последовательной версии скорость выполнения итераций обратно пропорциональна увеличению N, тогда как в параллельной зависимости сложнее из-за динамики эффективности распределения вычислений — от низкой на малых сетках до высокой на крупных.

Время работы ключевых модулей программ демонстрирует квадратичную зависимость от роста числа узлов. Это объясняется одновременным увеличением как количества итераций, так и объёма обрабатываемых данных. Каждое укрупнение сетки приводит к более чем десятикратному замедлению вычислений, что подчёркивает важность оптимизации для задач высокой размерности.

Параллелизация демонстрирует низкое ускорение при малом числе узлов из-за преобладания накладных расходов на синхронизацию процессов. С увеличением размера сетки растёт размер блоков данных, обрабатываемых отдельными процессами, что снижает долю служебных операций в общем времени вычислений. Это приводит к нелинейному росту ускорения, особенно заметному на крупных задачах.

Эффективность параллельной реализации коррелирует с ускорением: она повышается при увеличении размерности сетки, но снижается с ростом числа задействованных процессов (p). Для задач с малым N это падение происходит быстрее, так как коммуникационные задержки начинают доминировать над полезными вычислениями. Таким образом, выбор оптимального числа процессов требует учёта баланса между размером задачи и накладными расходами, чтобы максимизировать производительность системы.

**Приложение А. Последовательный код программы.**

#include <iostream>

#include <fstream>

#include <vector>

#include <cmath>

#include <algorithm>

#include <iomanip>

#include <omp.h>

#include <numeric>

using namespace std;

const int maxit = 100000;

const double errmax = 1e-7;

inline double xbound(double y) {

    return 2.0 \* sqrt(y);

}

inline double ybound(double x) {

    return 0.25 \* x \* x;

}

inline double f(double x, double y) {

    return 0.25;

}

inline double phi(double x, double y) {

    return sqrt(0.25 \* x \* x + y \* y);

}

void init(vector<vector<double>> &u,

          const vector<vector<double>> &x,

          const vector<vector<double>> &y,

          int ni, int nj) {

    for (int i = 0; i < ni; ++i)

        for (int j = 0; j < nj - 1; ++j)

            u[i][j] = 1.0;

    for (int i = 0; i < ni; ++i)

        u[i][nj - 1] = phi(x[i][nj - 1], y[i][nj - 1]);

}

void bcond(vector<vector<double>> &u,

           const vector<vector<double>> &x,

           const vector<vector<double>> &y,

           const vector<int> &Imin,

           const vector<int> &Imax,

           const vector<int> &Jmin,

           double hx, double hy,

           int ni, int nj) {

    for (int i = 1; i < ni - 1; ++i) {

        double xm = x[i][Jmin[i]];

        double ym = y[i][Jmin[i]];

        double delta = ym - ybound(xm);

        u[i][Jmin[i]] = (hy \* phi(xm, ybound(xm)) + delta \* u[i][Jmin[i] + 1]) / (hy + delta);

    }

    for (int j = 0; j < nj; ++j) {

        if (Jmin[Imin[j]] < j) {

            double xm = x[Imin[j]][j];

            double ym = y[Imin[j]][j];

            double delta = xm + xbound(ym);

            u[Imin[j]][j] = (hx \* phi(-xbound(ym), ym) + delta \* u[Imin[j] + 1][j]) / (hx + delta);

        }

        if (Jmin[Imax[j]] < j) {

            double xm = x[Imax[j]][j];

            double ym = y[Imax[j]][j];

            double delta = xbound(ym) - xm;

            u[Imax[j]][j] = (hx \* phi(xbound(ym), ym) + delta \* u[Imax[j] - 1][j]) / (hx + delta);

        }

    }

}

void Jacobi(vector<vector<double>> &u,

            const vector<vector<double>> &x,

            const vector<vector<double>> &y,

            const vector<int> &Imin,

            const vector<int> &Imax,

            const vector<int> &Jmin,

            double hx, double hy,

            int ni, int nj,

            double &error1,

            int iter) {

    vector<vector<double>> u0 = u;

    for (int k = 0; k < iter; ++k) {

        bcond(u, x, y, Imin, Imax, Jmin, hx, hy, ni, nj);

        u0 = u;

        for (int j = 1; j < nj - 1; ++j) {

            for (int i = Imin[j] + 1; i < Imax[j]; ++i) {

                if (j == Jmin[i]) continue;

                u[i][j] = (hy \* hy \* (u0[i + 1][j] + u0[i - 1][j]) +

                           hx \* hx \* (u0[i][j + 1] + u0[i][j - 1]) -

                           hx \* hx \* hy \* hy \* f(x[i][j], y[i][j])) /

                          (2 \* (hx \* hx + hy \* hy));

                if (k == iter - 1)

                    error1 = max(error1, fabs(u0[i][j] - u[i][j]));

            }

        }

    }

}

int main() {

    int iter, ni, nj;

    double Xmin, Xmax, Ymin = 0.0, Ymax = 2.0;

    double error1 = 1.0;

    int l = 0;

    ifstream input("input.txt");

    input >> iter >> ni >> nj;

    input.close();

    Xmax = sqrt(8.0);

    Xmin = -Xmax;

    double hx = (Xmax - Xmin) / (ni - 1);

    double hy = (Ymax - Ymin) / (nj - 1);

    vector<vector<double>> x(ni, vector<double>(nj));

    vector<vector<double>> y(ni, vector<double>(nj));

    vector<vector<double>> u(ni, vector<double>(nj, 0.0));

    vector<int> Imin(nj, 0), Imax(nj, ni - 1), Jmin(ni, nj - 1);

    for (int j = 0; j < nj; ++j)

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

            x[i][j] = Xmin + i \* hx;

            y[i][j] = Ymin + j \* hy;

        }

    init(u, x, y, ni, nj);

    for (int i = 0; i < ni; ++i)

        for (int j = 0; j < nj - 1; ++j) {

            if (y[i][j] > ybound(x[i][j])) {

                Jmin[i] = j;

                break;

            } else {

                u[i][j] = 0.0;

            }

        }

    for (int j = 1; j < nj - 1; ++j) {

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

            if (x[i][j] > -xbound(y[i][j])) {

                Imin[j] = i;

                break;

            }

        }

    }

    for (int j = 1; j < nj - 1; ++j) {

        for (int i = ni - 2; i >= 0; --i) {

            if (x[i][j] < xbound(y[i][j])) {

                Imax[j] = i;

                break;

            }

        }

    }

    double t1 = omp\_get\_wtime();

    while (l <= maxit && error1 > errmax) {

        error1 = 0.0;

        Jacobi(u, x, y, Imin, Imax, Jmin, hx, hy, ni, nj, error1, iter);

        l += iter;

        cout << "iter: " << l << " error: " << error1 << endl;

    }

    double t2 = omp\_get\_wtime();

    ofstream output("output.plt");

    output << "Variables = \"x\", \"y\", \"u\"\n";

    output << "Zone i = " << ni << ", j = " << nj << "\n";

    for (int j = 0; j < nj; ++j)

        for (int i = 0; i < ni; ++i)

            output << fixed << setprecision(5)

                   << x[i][j] << " " << y[i][j] << " " << u[i][j] << "\n";

    output.close();

    cout << "time: " << t2 - t1 << "; l = " << l

         << "; l/t = " << l / (t2 - t1)

         << "; N = " << ni + nj \* (ni - 2)

         - accumulate(Jmin.begin() + 1, Jmin.end() - 1, 0) << endl;

    int i\_center = (ni - 1) / 2;

    int j\_center = (nj - 1) / 2;

    int j\_high = ((nj - 1) \* 9) / 10;

    cout << " x = " << x[i\_center][j\_center]

         << "; y = " << y[i\_center][j\_center]

         << "; u = " << u[i\_center][j\_center] << endl;

    cout << " x = " << x[i\_center][j\_high]

         << "; y = " << y[i\_center][j\_high]

         << "; u = " << u[i\_center][j\_high] << endl;

    return 0;

}

**Приложение Б. Параллельный код программы.**

#include <mpi.h>

#include <vector>

#include <cmath>

#include <iostream>

#include <fstream>

#include <algorithm>

#include <string>

#include <numeric>

using namespace std;

double xbound(double y) {

    return 2.0 \* sqrt(y);

}

double ybound(double x) {

    return 0.25 \* x \* x;

}

double f(double x, double y) {

    return 0.25;

}

double phi(double x, double y) {

    return sqrt(0.25 \* x \* x + y \* y);

}

void init(vector<double>& u, int ni, int nj, const vector<double>& x, const vector<double>& y) {

    fill(u.begin(), u.end(), 1.0);

    for (int i = 0; i < ni; ++i) {

        int j = nj - 1;

        u[i + j \* ni] = phi(x[i + j \* ni], y[i + j \* ni]);

    }

}

void decomp(const vector<double>& u, vector<double>& uloc, int ni, int nj, int pid, int procs,

            vector<int>& disp, vector<int>& scount, int& jp1, int& jp2,

            const vector<int>& Imin, const vector<int>& Imax) {

    vector<int> jpd(procs), jpu(procs);

    if (pid == 0) {

        int opt = (1 + ni \* (nj - 1) + 2 \* (nj - 2 - accumulate(Imin.begin() + 1, Imin.end() - 1, 0))) / procs;

        int sp = 0, i = 0;

        jpd[0] = 1;

        disp[0] = 0;

        for (int j = 1; j <= nj; ++j) {

            sp += Imax[j-1] - Imin[j-1] + 1;

            if (sp >= opt || j == nj || (procs - i - 1) == (nj - j)) {

                jpu[i] = j;

                if (i < procs - 1) {

                    jpd[i+1] = j + 1;

                    disp[i+1] = j \* ni;

                }

                scount[i] = (jpu[i] - jpd[i] + 1) \* ni;

                i++;

                sp = 0;

                if (i >= procs) break;

            }

        }

    }

    MPI\_Scatter(jpd.data(), 1, MPI\_INT, &jp1, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

    MPI\_Scatter(jpu.data(), 1, MPI\_INT, &jp2, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

    jp1--;

    jp2--;

    int local\_j = jp2 - jp1 + 1;

    uloc.resize(ni \* local\_j);

    if (pid == 0) {

        MPI\_Scatterv(u.data(), scount.data(), disp.data(), MPI\_DOUBLE,

                     uloc.data(), ni \* local\_j, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

    } else {

        MPI\_Scatterv(nullptr, nullptr, nullptr, MPI\_DOUBLE,

                     uloc.data(), ni \* local\_j, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

    }

}

void bcond(vector<double>& u, const vector<int>& Imin, const vector<int>& Imax, const vector<int>& Jmin,

           double hx, double hy, int ni, int nj, double Xmin, double Ymin,

           int jp1, int jp2, vector<double>& up) {

    int local\_j = jp2 - jp1 + 1;

    for (int i = max(Imin[jp2], 1); i <= max(Imin[jp1], 1); ++i) {

        int j\_global = Jmin[i] - 1;

        int jr = j\_global - jp1;

        if (jr >= 0 && jr < local\_j) {

            double xm = Xmin + i \* hx;

            double ym = Ymin + j\_global \* hy;

            double delta = ym - ybound(xm);

            double unode = (jr == local\_j - 1) ? up[i] : u[i + (jr + 1) \* ni];

            u[i + jr \* ni] = (hy \* phi(xm, ybound(xm)) + delta \* unode) / (hy + delta);

        }

    }

    for (int i = min(Imax[jp1], ni - 2); i <= min(Imax[jp2], ni - 2); ++i) {

        int j\_global = Jmin[i] - 1;

        int jr = j\_global - jp1;

        if (jr >= 0 && jr < local\_j) {

            double xm = Xmin + i \* hx;

            double ym = Ymin + j\_global \* hy;

            double delta = ym - ybound(xm);

            double unode = (jr == local\_j - 1) ? up[i] : u[i + (jr + 1) \* ni];

            u[i + jr \* ni] = (hy \* phi(xm, ybound(xm)) + delta \* unode) / (hy + delta);

        }

    }

    for (int j = jp1; j <= jp2; ++j) {

        if (Jmin[Imin[j]] - 1 < j) {

            int i = Imin[j];

            double xm = Xmin + i \* hx;

            double ym = Ymin + j \* hy;

            double delta = xm + xbound(ym);

            int jr = j - jp1;

            u[i + jr \* ni] = (hx \* phi(-xbound(ym), ym) + delta \* u[i + 1 + jr \* ni]) / (hx + delta);

        }

        if (Jmin[Imax[j]] - 1 < j) {

            int i = Imax[j];

            double xm = Xmin + i \* hx;

            double ym = Ymin + j \* hy;

            double delta = xbound(ym) - xm;

            int jr = j - jp1;

            u[i + jr \* ni] = (hx \* phi(xbound(ym), ym) + delta \* u[i - 1 + jr \* ni]) / (hx + delta);

        }

    }

}

void InterComm(vector<double>& u\_up, vector<double>& u\_low, vector<double>& up, vector<double>& low,

               int ni, int pid, int procs) {

    MPI\_Request reqs[4];

    MPI\_Status stats[4];

    if (pid > 0) {

        MPI\_Isend(u\_low.data(), ni, MPI\_DOUBLE, pid-1, 0, MPI\_COMM\_WORLD, &reqs[0]);

        MPI\_Irecv(low.data(), ni, MPI\_DOUBLE, pid-1, 1, MPI\_COMM\_WORLD, &reqs[1]);

    }

    if (pid < procs-1) {

        MPI\_Isend(u\_up.data(), ni, MPI\_DOUBLE, pid+1, 1, MPI\_COMM\_WORLD, &reqs[2]);

        MPI\_Irecv(up.data(), ni, MPI\_DOUBLE, pid+1, 0, MPI\_COMM\_WORLD, &reqs[3]);

    }

    if (pid == 0)

        MPI\_Waitall(2, reqs+2, stats+2);

    else if (pid == procs-1)

        MPI\_Waitall(2, reqs, stats);

    else

        MPI\_Waitall(4, reqs, stats);

}

void Jacobi(vector<double>& u, const vector<int>& Imin, const vector<int>& Imax, const vector<int>& Jmin,

            double hx, double hy, int ni, int nj, double Xmin, double Ymin,

            int jp1, int jp2, double& error1, vector<double>& up, int pid, int procs) {

    int local\_j = jp2 - jp1 + 1;

    vector<double> u0(u);

    vector<double> low(ni, 0.0);

    bcond(u, Imin, Imax, Jmin, hx, hy, ni, nj, Xmin, Ymin, jp1, jp2, up);

    for (int j = jp1 + 1; j <= jp2 - 1; ++j) {

        for (int i = Imin[j] + 1; i <= Imax[j] - 1; ++i) {

            if (j == Jmin[i] - 1) continue;

            int jr = j - jp1;

            double x = Xmin + i \* hx;

            double y\_val = Ymin + j \* hy;

            u[i + jr \* ni] = (hy\*hy\*(u0[i+1 + jr\*ni] + u0[i-1 + jr\*ni]) +

                             hx\*hx\*(u0[i + (jr+1)\*ni] + u0[i + (jr-1)\*ni]) -

                             hx\*hx\*hy\*hy\*f(x, y\_val)) / (2\*(hx\*hx + hy\*hy));

        }

    }

    if (procs > 1) {

        vector<double> u\_up(ni), u\_low(ni);

        copy(u.begin() + (local\_j - 1)\*ni, u.begin() + local\_j\*ni, u\_up.begin());

        copy(u.begin(), u.begin() + ni, u\_low.begin());

        InterComm(u\_up, u\_low, up, low, ni, pid, procs);

    }

    double errloc = 0.0;

    for (size\_t i = 0; i < u.size(); ++i)

        errloc = max(errloc, abs(u[i] - u0[i]));

    MPI\_Allreduce(&errloc, &error1, 1, MPI\_DOUBLE, MPI\_MAX, MPI\_COMM\_WORLD);

}

void Sweep(vector<double>& uline, vector<double>& up, vector<double>& low,

           int ni, int nj, int j, const vector<int>& Imin, const vector<int>& Imax,

           const vector<int>& Jmin, double hx, double hy, double Xmin, double Ymin) {

    vector<double> u0 = uline;

    for (int i = Imin[j] + 1; i <= Imax[j] - 1; ++i) {

        if (j == Jmin[i] - 1) continue;

        double x = Xmin + i \* hx;

        double y\_val = Ymin + j \* hy;

        uline[i] = (hy\*hy\*(u0[i+1] + u0[i-1]) + hx\*hx\*(up[i] + low[i]) -

                   hx\*hx\*hy\*hy\*f(x, y\_val)) / (2\*(hx\*hx + hy\*hy));

    }

}

int main(int argc, char\*\* argv) {

    MPI\_Init(&argc, &argv);

    int procs, pid;

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

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

    double Xmin = -sqrt(8.0), Xmax = sqrt(8.0), Ymin = 0.0, Ymax = 2.0;

    int ni = 0, nj = 0, maxit = 0;

    vector<double> x, y, u;

    vector<int> Imin, Imax, Jmin;

    if (pid == 0) {

        ifstream fin("input.txt");

        fin >> maxit >> ni >> nj;

        fin.close();

        x.resize(ni \* nj);

        y.resize(ni \* nj);

        u.resize(ni \* nj);

        double hx = (Xmax - Xmin) / (ni - 1);

        double hy = (Ymax - Ymin) / (nj - 1);

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

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

                x[i + j \* ni] = Xmin + i \* hx;

                y[i + j \* ni] = Ymin + j \* hy;

            }

        }

        init(u, ni, nj, x, y);

        Jmin.resize(ni);

        fill(Jmin.begin(), Jmin.end(), nj - 1);

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

            for (int j = 0; j < nj - 1; ++j) {

                double y\_val = Ymin + j \* hy;

                if (y\_val > ybound(Xmin + i \* hx)) {

                    Jmin[i] = j;

                    break;

                } else {

                    u[i + j \* ni] = 0.0;

                }

            }

        }

        Imin.resize(nj);

        Imax.resize(nj);

        Imin[0] = (ni - 1) / 2;

        Imin[nj - 1] = 0;

        for (int j = 1; j < nj - 1; ++j) {

            double y\_val = Ymin + j \* hy;

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

                double x\_val = Xmin + i \* hx;

                if (x\_val > -xbound(y\_val)) {

                    Imin[j] = i;

                    break;

                }

            }

        }

        Imax[0] = (ni - 1) / 2;

        Imax[nj - 1] = ni - 1;

        for (int j = 1; j < nj - 1; ++j) {

            double y\_val = Ymin + j \* hy;

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

                double x\_val = Xmin + i \* hx;

                if (x\_val < xbound(y\_val)) {

                    Imax[j] = i;

                    break;

                }

            }

        }

    }

    MPI\_Bcast(&ni, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

    MPI\_Bcast(&nj, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

    MPI\_Bcast(&maxit, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

    double hx = (Xmax - Xmin) / (ni - 1);

    double hy = (Ymax - Ymin) / (nj - 1);

    Imin.resize(nj);

    Imax.resize(nj);

    Jmin.resize(ni);

    MPI\_Bcast(Imin.data(), nj, MPI\_INT, 0, MPI\_COMM\_WORLD);

    MPI\_Bcast(Imax.data(), nj, MPI\_INT, 0, MPI\_COMM\_WORLD);

    MPI\_Bcast(Jmin.data(), ni, MPI\_INT, 0, MPI\_COMM\_WORLD);

    vector<int> disp(procs), scount(procs);

    int jp1, jp2;

    vector<double> uloc;

    decomp(u, uloc, ni, nj, pid, procs, disp, scount, jp1, jp2, Imin, Imax);

    vector<double> up(ni, 0.0), low(ni, 0.0);

    double error1 = 1.0, errmax = 1e-7;

    double t1 = MPI\_Wtime();

    for (int l = 1; l <= maxit; ++l) {

        Jacobi(uloc, Imin, Imax, Jmin, hx, hy, ni, nj, Xmin, Ymin, jp1, jp2, error1, up, pid, procs);

        if (pid == 0 && l % 100 == 0)

            cout << "iter: " << l << " error: " << error1 << endl;

        if (error1 < errmax) {

            if (pid == 0)

                cout << "last iter: " << l << " error: " << error1 << endl;

            break;

        }

    }

    double t2 = MPI\_Wtime();

    if (pid == 0) {

        u.resize(ni \* nj);

        MPI\_Gatherv(MPI\_IN\_PLACE, 0, MPI\_DOUBLE, u.data(), scount.data(), disp.data(), MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

        ofstream fout("output.plt");

        fout << "Variables = \"x\", \"y\", \"u\"" << endl;

        fout << "Zone i=" << ni << ", j=" << nj << endl;

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

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

                fout << x[i + j\*ni] << " " << y[i + j\*ni] << " " << u[i + j\*ni] << endl;

            }

        }

        fout.close();

        cout << "Time: " << t2 - t1 << "; iterations: " << maxit << endl;

    } else {

        MPI\_Gatherv(uloc.data(), uloc.size(), MPI\_DOUBLE, nullptr, nullptr, nullptr, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

    }

    MPI\_Finalize();

    return 0;

}