РЕФЕРАТ

Отчёт 30 с., 4 лист., 3 рис., 4 ист.

ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ, МЕТОДЫ ОРГАНИЗАЦИИ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ, СИСТЕМА С ОБЩЕЙ ПАМЯТЬЮ, РАСПРЕДЕЛЕННЫЕ СИСТЕМЫ, ГРАФИЧЕСКИЕ ПРОЦЕССОРЫ, OPENCL, MPI, OPENMP.

Цель работы: изучение методов и подходов при организации параллельных вычислений.

В процессе выполнения работы была создана параллельная реализация обучения нейронной сети классификации рукописных чисел.

Объектом исследования являются подходы и методы при организации параллельных вычислений.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В современном мире, когда объёмы данных неуклонно растут, становится все более важным использование параллельных вычислений для обработки информации в различных областях, таких как наука, технологии, бизнес и другие. Однако, чтобы достичь максимальной эффективности при использовании параллельных вычислений, необходимо уметь правильно выбирать методы и подходы при их организации, а также понимать основные принципы параллельного программирования.

В данной работе будут рассмотрены основные подходы к параллельным вычислениям, такие как распараллеливание по данным и задачам, межпроцессное взаимодействие, использование многопоточности и другие, а также будут рассмотрены основные инструменты и технологии, которые используются при организации параллельных вычислений.

1 ТЕОРИЯ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

Параллельные вычисления — способ организации компьютерных вычислений, при котором программы разрабатываются как набор взаимодействующих вычислительных процессов, работающих параллельно (одновременно). Термин охватывает совокупность вопросов параллелизма в программировании, а также создание эффективно действующих аппаратных реализаций. Теория параллельных вычислений составляет раздел прикладной теории алгоритмов [1].

Основная же концепция параллельных вычислений, как раз, реализуется распараллеливанием процесса обработки данных. Программа использующая параллельные вычисления характеризуется как набор взаимодействующих вычислительных процессов.

1.1 Парадигмы параллельных вычислений

Существуют две парадигмы параллельных вычислений: параллелизм данных (data parallelism) и параллелизм задач (task parallelism).

Параллелизм данных заключается в том, что данные разбиваются на несколько небольших блоков, и каждый блок обрабатывается параллельно на разных процессорах или ядрах. Это особенно эффективно для задач, которые могут быть разделены на множество однотипных подзадач, которые можно обрабатывать параллельно, например, при обработке изображений или анализе больших объёмов данных.

Параллелизм задач используется в случаях, когда задачи в программе имеют сложные взаимосвязи и не могут быть легко разделены на независимые блоки данных. В этом случае каждая задача выполняется параллельно на разных процессорах или ядрах, и эти задачи могут взаимодействовать друг с другом при необходимости.

Часто при параллельных вычислениях используются обе парадигмы, чтобы достичь максимальной эффективности и ускорения вычислений. Например, в приложениях для анализа больших данных часто применяется комбинация параллелизма данных и параллелизма задач.

1.2 Конвейер

Конвейер (pipeline) — это ещё одна парадигма параллельных вычислений, которая используется для выполнения вычислительных задач, которые могут быть разделены на набор последовательных шагов. Он позволяет разделить задачу на несколько независимых этапов, которые могут быть выполнены параллельно на разных процессорах или ядрах.

Конвейер работает по принципу последовательного прохождения через стадии обработки, при этом каждая стадия выполняет определённую операцию над данными и передаёт их на следующую стадию. Каждая стадия может работать независимо, что позволяет параллельно выполнять несколько этапов обработки данных.

Преимущества использования конвейера заключаются в том, что он может обеспечивать высокую производительность, ускорение вычислений и более эффективное использование ресурсов. Например, конвейер можно использовать для обработки больших объёмов данных, таких как обработка видео или аудио, обработка изображений, и т.д.

Конвейер может быть использован как в параллелизме задач, так и в параллелизме данных. В случае параллелизма данных, каждый процессор выполняет одну и ту же стадию на разных кусках данных, а в случае параллелизма задач, каждый процессор может работать над разными наборами данных, но проходящими через одни и те же этапы обработки.

1.3 Классификация параллельных систем

Одной из наиболее распространённых классификаций параллельных вычислительных систем является Таксономия Флинна, предложенная Майклом Флинном в 1966 году. Согласно ей, существует четыре типа систем:

- SISD (Single Instruction, Single Data) — вычислительная система с одиночным потоком команд и одиночным потоком данных. Примером таких систем являются обычные последовательные ЭВМ.
- SIMD (Single Instruction, Multiple Data) — вычислительная система с одиночным потоком команд и множественным потоком данных.
- MISD (Multiple Instructions, Single Data) — вычислительная система со множественным потоком команд и одиночным потоком данных.
- MIMD (Multiple Instructions, Multiple Data) - вычислительная система со множественным потоком команд и множественным потоком данных.

1.4 Эффективность параллельных вычислений

При использовании параллельных вычислений, ускорение не всегда пропорционально количеству процессоров, которые используются для выполнения задачи. Эффективность зависит от многих факторов, таких как структура задачи, доступность и производительность процессоров, сложность обработки данных, производительность сети и т.д.

1.4.1 Теорема Брента

Теорема Брента устанавливает ограничение на максимальное ускорение, которого можно достичь при выполнении задачи на параллельном компьютере с p процессорами. В соответствии с этой теоремой, максимальное ускорение при использовании p процессоров составляет $O(\log p)$, то есть ускорение не может быть больше, чем логарифм от количества процессоров. Это означает, что

использование большего количества процессоров может не привести к линейному ускорению.

1.4.2 Закон Амдала

Закон Амдала иллюстрирует иллюстрирует ограничение роста производительности вычислительной системы с увеличением количества вычислителей. Закон утверждает, что максимальное ускорение может быть ограничено той частью задачи, которую невозможно распараллелить. В соответствии с законом Амдала, максимальное ускорение *S(p)* при использовании p процессоров можно вычислить по формуле 1:

$$S(p)=\frac{1}{1-p+\frac{p}{n}}, \tag{1}$$

где *n* - это доля задачи, которую можно распараллелить. Например, если *n* = 0,5, то максимальное ускорение не может превышать 2, даже если используются бесконечно много процессоров.

2 ОРГАНИЗАЦИЯ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

2.1 В системах с общей памятью

Параллельные вычисления на системах с общей памятью осуществляются с использованием многопроцессорных компьютеров или многопроцессорных серверов, где доступ ко всей памяти обеспечивается через единую шину.

Большинство современных компьютеров являются многоядерными, и соответственно обладают несколькими ядрами. Также они поддерживают многопоточность, которая заключается в том, что каждое ядро может выполнять несколько потоков одновременно.

Основная идея здесь заключается в том, что несколько процессоров выполняют одновременно различные задачи на одном и том же наборе данных.

Данные делятся между процессорами таким образом, чтобы каждый процессор обрабатывал свою часть данных.

При параллельном выполнении задач между процессорами необходимо обеспечить согласованность данных и выполнение действий в правильном порядке.

Для организации параллельных вычислений на системах с общей памятью часто используются специализированные программные библиотеки для управления параллелизмом, например, OpenMP или Pthreads.

2.1.1 Разработка программ при помощи библиотеки OpenMP

Библиотека OpenMP (Open Multi-Processing) является одним из наиболее популярных инструментов для создания многопоточных программ.

Разработка параллельных программ при помощи OpenMP осуществляется путем вставки директив препроцессора в код программы. Эти директивы указывают компилятору, какие участки кода должны выполняться параллельно, как распределить нагрузку между потоками и какие данные должны быть общими

для всех потоков. Это позволяет создавать параллельные программы, которые могут выполняться на многопроцессорных и многоядерных системах, ускоряя их выполнение и повышая производительность.

Например, рассмотрим пример распараллеливания алгоритма сложения двух векторов (массивов чисел):

Листинг 1 — Реализация сложения двух векторов при помощи OpenMP [2]

```
double *vectorAdd(double *c, double *c, double *a, double *b, int n)
{
    #pragma omp parallel for
    for (int i = 0; i < n; i++)
        c[i] = a[i] + b[i];
    return c;
}
```

Функция *vectorAdd* принимает в качестве аргументов указатели на векторы *c*, *a* и *b*, а также целочисленное значение *n*, которое указывает размерность векторов. Функция возвращает указатель на вектор *c*.

Директива *#pragma omp parallel for* указывает компилятору на то, что цикл *for* должен быть распараллелен, то есть вычисления должны выполняться одновременно в нескольких потоках. Компилятор автоматически разбивает итерации цикла между потоками и выполняет вычисления параллельно.

Внутри цикла *for* происходит операция сложения векторов *a* и *b*, результат которой записывается в вектор *c*. Каждый поток выполняет часть итераций цикла, что позволяет ускорить выполнение операции сложения векторов. После завершения цикла функция возвращает указатель на вектор *c*, содержащий результат сложения.

2.2 В распределенных системах

В отличии от систем с общей памятью, распределённые системы не обладают общей памятью и используют общую сеть для коммуникации. Для передачи данных используются сообщения.

При организации параллельных вычислений на таких системах наиболее важную роль занимает организация сети, так как зачастую коммуникация занимает большее количество времени, чем сами вычисления.

Производительность сети зависит от нескольких факторов. Наиболее значимые из них это маршрутизация (процесс определения оптимального маршрута данных в сетях связи), контроль потока (механизм, который притормаживает передатчик данных при неготовности приёмника) и топология сети (способ соединения узлов сети).

В распределенных системах часто используется модель на основе сообщений. Каждый узел выполняет свою часть вычислений, а затем передаёт сообщение с результатом вычислений другому узлу. Таким образом, результат вычислений получается путём комбинирования результатов, полученных от разных узлов.

Для организации параллельных вычислений на системах с распределенной памятью был разработан программный интерфейс для передачи информации между процессам MPI.

2.2.1 Разработка параллельных программ при помощи MPI

Message Passing Interface (MPI, интерфейс передачи сообщений) — программный интерфейс (API) для передачи информации, который позволяет обмениваться сообщениями между процессами, выполняющими одну задачу. включает в себя описание функций и процедур для обмена сообщениями между процессами, управления процессами и коммуникаторами, группами процессов, коллективными операциями, обработки ошибок и сбоев в параллельных вычислениях.

MPI имеет множество различных реализаций, таких как OpenMPI, MPICH, Intel MPI, MVAPICH, Cray MPI и др., которые позволяют использовать его практически на любых системах.

Рассмотрим ещё один пример вычисления суммы двух векторов, на этот раз используя MPI (лист. 2).

Листинг 2 — Реализация сложения двух векторов при помощи MPI

```
#include <iostream>
#include <mpi.h>

#define VECTOR_SIZE 10

int main(int argc, char** argv) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int vectorA[VECTOR_SIZE], vectorB[VECTOR_SIZE];
    int localVectorA[VECTOR_SIZE/size],
localVectorB[VECTOR_SIZE/size];

    if (rank == 0) {
        // Fill vectors with data
        for (int i = 0; i < VECTOR_SIZE; i++) {
            vectorA[i] = i;
            vectorB[i] = VECTOR_SIZE - i;
        }
    }

    // Scatter vectors across processes
    MPI_Scatter(vectorA, VECTOR_SIZE/size, MPI_INT, localVectorA,
VECTOR_SIZE/size, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Scatter(vectorB, VECTOR_SIZE/size, MPI_INT, localVectorB,
VECTOR_SIZE/size, MPI_INT, 0, MPI_COMM_WORLD);

    // Compute local sum
    int localSum[VECTOR_SIZE/size];
    for (int i = 0; i < VECTOR_SIZE/size; i++) {
        localSum[i] = localVectorA[i] + localVectorB[i];
    }

    // Gather local sums into final result
    int result[VECTOR_SIZE];
    MPI_Gather(localSum, VECTOR_SIZE/size, MPI_INT, result,
VECTOR_SIZE/size, MPI_INT, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        // Print result
        for (int i = 0; i < VECTOR_SIZE; i++) {
            std::cout << result[i] << " ";
        }
```

Продолжение листинга 2

```
        std::cout << std::endl;
    }

    MPI_Finalize();
    return 0;
}
```

В начале программы мы инициализируем *MPI* с помощью функции *MPI_Init*. Затем мы определяем ранг и общее число процессов с помощью функций *MPI_Comm_rank* и *MPI_Comm_size*.

Далее мы создаём два вектора *vectorA* и *vectorB* длиной 10, которые заполняем данными в процессе с рангом 0. Мы используем условный оператор *if* для того, чтобы заполнить векторы только в одном процессе.

Затем мы используем функцию *MPI_Scatter* для распределения данных векторов по всем процессам. Эта функция отправляет часть массива с данными из корневого процесса (с рангом 0) и разбивает его на части, которые отправляются каждому процессу в отдельности.

Затем мы выполняем сложение локальных векторов *localVectorA* и *localVectorB*, записывая результат в локальный вектор *localSum*. Для этого мы проходимся по элементам векторов и складываем их в соответствующие элементы вектора *localSum*.

Мы используем функцию *MPI_Gather* один раз, чтобы собрать локальные суммы *localSum* из всех процессов в итоговый вектор *result*. Как только данные собраны в корневом процессе (с рангом 0), мы выводим их на экран с помощью оператора вывода *std::cout*.

Наконец, мы завершаем выполнение программы, вызывая функцию *MPI_Finalize*.

2.3 На графических процессорах

Графические процессоры также обладают многопоточной архитектурой, что делает их пригодными для параллельных вычислений. В отличии от центрального процессора, графические обладают большим количеством значительно более слабых ядер и могут быть использованы для выполнения большого количество однотипных низкоуровневых арифметических операции.

Для организации параллельных вычислений на графических процессорах необходимо написать программу с использованием специальных библиотек, таких как CUDA от NVIDIA или OpenCL, которая будет распределять вычислительную нагрузку между ядрами графического процессора.

2.3.1 Параллельные вычисления на графических процессорах при помощи OpenCL

Для разработки программы использующей OpenCL необходимо разработать ядро OpenCL и кода управления.

Ядра OpenCL - это функции, которые будут выполняться на каждом ядре выбранного устройства. Они разрабатываются на специальном языке OpenCL C, который является подмножеством C.

Код управления отвечает за выбор устройства, сборку и запуск ядер, передачу данных в них.

Реализуем алгоритм сложения векторов, используя OpenCL.

Сначала напишем код ядра (лист. 3):

Листинг 3 — Код ядра OpenCL для вычисления суммы векторов

```
__kernel void VectorAdd(__global float *a, __global float *b,
__global float *c, int iNumElements) {
    int iGID = get_global_id(0);
    while (iGID < iNumElements) {
        c[iGID] = a[iGID] + b[iGID];
    }
}
```

Основной цикл этого ядра осуществляется с помощью оператора *while*. Для каждого элемента массивов *a*, *b* и *c*, ядро выполняет сложение соответствующих элементов массивов *a* и *b* и сохранение результата в элементе массива *c*. Для каждой итерации цикла *while*, индекс *iGID* увеличивается на количество потоков, которые были созданы для выполнения данного ядра.

Функция *get_global_id(0)* используется для получения глобального идентификатора потока в одномерном пространстве. Эта функция возвращает целое число, которое идентифицирует текущий поток. В данном случае, используется только одномерное пространство, поэтому значение, возвращаемое функцией *get_global_id*, является индексом элемента массива, который будет обработан данным потоком.

Код ядра хранится в отдельном файле *VectorAdd.cl*.

Далее напишем код основной программы (лист. 4):

Листинг 4 — Код запуска ядра OpenCL

```
#include <iostream>
#include <fstream>
#include <string>
#include <CL/cl.hpp>

int main() {
    // Чтение исходного кода ядра из файла VectorAdd.cl
    std::ifstream kernelFile("VectorAdd.cl");
    if (!kernelFile.is_open()) {
        std::cerr << "Failed to open kernel file!" << std::endl;
        return 1;
    }
     std::string
kernelCode(std::istreambuf_iterator<char>(kernelFile),
(std::istreambuf_iterator<char>()));

    // Инициализация OpenCL
    std::vector<cl::Platform> platforms;
    cl::Platform::get(&platforms);
    if (platforms.empty()) {
        std::cerr << "No OpenCL platforms found!" << std::endl;
       return 1;
    }
```

Продолжение листинга 4

```
    cl::Platform platform = platforms[0];
    std::vector<cl::Device> devices;
    platform.getDevices(CL_DEVICE_TYPE_GPU, &devices);
    if (devices.empty()) {
        std::cerr << "No GPUs found!" << std::endl;
        return 1;
    }
    cl::Device device = devices[0];
    cl::Context context(device);
    cl::Program::Sources sources(1,
std::make_pair(kernelCode.c_str(), kernelCode.length() + 1));
    cl::Program program(context, sources);
    if (program.build({device}) != CL_SUCCESS) {
        std::cerr << "Failed to build program!" << std::endl;
        return 1;
    }

    // Создание буферов и запуск ядра
    const int NumElements = 1024;
    std::vector<float> a(NumElements, 1.0f);
    std::vector<float> b(NumElements, 2.0f);
    std::vector<float> c(NumElements);
    cl::Buffer aBuffer(context, CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, sizeof(float) * NumElements, a.data());
    cl::Buffer bBuffer(context, CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, sizeof(float) * NumElements, b.data());
    cl::Buffer cBuffer(context, CL_MEM_WRITE_ONLY, sizeof(float) *
NumElements);
    cl::CommandQueue queue(context, device);
    cl::Kernel kernel(program, "VectorAdd");
    kernel.setArg(0, aBuffer);
    kernel.setArg(1, bBuffer);
    kernel.setArg(2, cBuffer);
    kernel.setArg(3, NumElements);
    queue.enqueueNDRangeKernel(kernel, cl::NullRange,
cl::NDRange(NumElements));
    queue.enqueueReadBuffer(cBuffer, CL_TRUE, 0, sizeof(float) *
NumElements, c.data());
    // Вывод результата
    for (int i = 0; i < NumElements; i++) {
        std::cout << c[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Программа начинается с чтения содержимого файла *VectorAdd.cl*, в котором определено ядро VectorAdd. Затем программа инициализирует *OpenCL*, используя найденное устройство и исходный код ядра. Далее программа создает три буфера: *aBuffer*, *bBuffer* и *cBuffer*, для передачи входных данных в ядро и получения результата. Векторы *a* и *b* инициализируются значениями 1 и 2 соответственно. *aBuffer* и *bBuffer* создаются с флагом *CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR*, что означает, что они могут быть прочитаны ядром, но не могут быть изменены, и данные из векторов *a* и *b* будут скопированы в буферы при их создании. *cBuffer* создается с флагом *CL_MEM_WRITE_ONLY*, что означает, что данные будут записаны в буфер из ядра.

Далее программа создает объект *cl::CommandQueue*, который используется для запуска команд на устройстве *OpenCL*, и объект *cl::Kernel*, который представляет собой скомпилированное ядро. Каждый аргумент ядра устанавливается с помощью функции *kernel.setArg*.

Затем программа запускает ядро на устройстве, используя *queue.enqueueNDRangeKernel*. *cl::NullRange* означает, что необходимо использовать глобальную конфигурацию запуска ядра, которая была установлена ранее, а *cl::NDRange(NumElements)* определяет количество элементов, которые будут обрабатываться параллельно в ядре. После завершения выполнения ядра, результаты записываются из буфера *cBuffer* в вектор *c*.

В конце программы происходит освобождение ресурсов, занятых объектами OpenCL, и вывод результата сложения векторов *a* и *b* вектора *c* на экран.

3 РАЗРАБОТКА ПРОГРАММНОГО ПРОДУКТА, ИСПОЛЬЗУЮЩЕГО ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ

В ходе выполнения данной ознакомительной практики была разработана программа, которая использует принципы параллельных вычислений для обучения нейронной сети классификации рукописных чисел.

Процесс обучения нейронных сетей часто занимает очень много времени, но используя параллельные вычисления его можно значительно ускорить.

Для этого были использованы принципы параллельных вычислений, алгоритмы распараллеливания и соответствующие техники оптимизации.

Созданная программа обладает интуитивно понятным графическим интерфейсом, который предоставляет пользователю удобные инструменты для настройки процесса обучения и тестирования нейросети классификации рукописных чисел. Графический интерфейс позволяет легко задавать параметры обучения, такие как количество эпох и скорость обучения.

В качестве набора данных для обучения нейросети был использован набор MNIST: http://yann.lecun.com/exdb/mnist/.

Программа позволяет значительно сократить время обучения нейронной сети классификации рукописных чисел, что является существенным преимуществом в современных условиях, где требуется быстрая и точная обработка больших объёмов данных.

3.1 Исходный код

Полный исходный код программы представлен по ссылке https://github.com/mireastudent2312/oznprak.

Программа написана на языке программирования *C++* и библиотеки *OpenMP* для организации параллельных вычислений и *Qt* для реализации графического интерфейса.

Класс *Matrix* содержит различные матричные операции, такие как сложение, умножение и т. д. матриц и скаляров.

В директории *nn* хранятся классы, связанные непосредственно с нейросетью: *NeuralNetwork*, *Layer*, *ActivationFunction* и т. д.

В директории *ui* хранятся классы, связанные с графическим интерфейсом программы: *Window*, *DrawingArea* (компонент для рисования изображений) и т. д.

В директории *mnist* хранятся датасеты для обучения и тестирования нейросети.

Класс *Logic* содержит код связывающий графический интерфейс с нейросетью.

3.2 Организация параллельных вычислений

Параллельные вычисления используются в двух местах: классе *Matrix* и классе *NeuralNetwork*.

В классе *Matrix* параллельные вычисления используются для параллельного выполнения операций сложения, вычитания, умножения, матриц и скаляров, что позволяет значительно ускорить выполнение этих операции.

Например, в метода оператора сложения (см. лист. 5) используется параллельный цикл *for* для поэлементного сложения двух матриц.

Листинг 5 — Код метода оператора сложения для класса Matrix

```
Matrix Matrix::operator+(const double &scalar) const{
    Matrix result(*this);
#pragma omp parallel for default(none) shared(scalar, result)
    for (int i = 0; i < rows_ * cols_; i++) {
        result.data_[i] += scalar;
    }
    return result;
}
```

В классе *NeuralNetwork* параллельные вычисления используются при обучении в функции *train* (см. лист. 6) для одновременного вычисления

градиентов для разных входов и выводов. Директива *#pragma omp critical* используется для избежание одновременной записи данных, что может привести к некоректному результату. Однако из-за этого алгоритм выполняется медленее, чем простой параллелный цикл.

Листинг 6 — Код метода оператора сложения для класса Matrix

```
void NeuralNetwork::train(const std::vector<Matrix> &inputs, const
std::vector<Matrix> &outputs, double learningRate,
                          int epochs,
                          const std::vector<Matrix> &testInputs,
const std::vector<Matrix> &testOutputs) {
    assert(inputs.size() == outputs.size());
    assert(testInputs.size() == testOutputs.size());

    for (int epoch = 0; epoch < epochs; ++epoch) {
#pragma omp parallel for default(none) shared(inputs, outputs,
learningRate)
        for (size_t i = 0; i < inputs.size(); i++) {
            auto grads = trainOne(inputs[i], outputs[i],
learningRate);
#pragma omp critical
            for (size_t k = 0; k < layers.size(); k++) {
                layers[k]->applyGradients(grads[k].first,
grads[k].second);
            }
        }
    }
}
```

3.3 Описание графического интерфейса программы

На рисунках 1-3представлены скриншоты программы.

На рис. 1 изображено главное окно программы. Оно состоит из двух частей. Слева расположены настройки обучения нейросети: количество используемых потоков процессора, количество эпох и коэффициент обучения и кнопка начала обучения. Справа находится область для рисования и кнопка для вычисления результата на основе нарисованного изображения.

При нажатии на кнопку «Train» запускается процесс обучения нейросети с заданными настройками. После завершение обучения, отображаются время,

затраченное на обучение и результат тестирования модели на тестовых данных (рис. 2).

При нажатии на кнопку «Predict» обученная модель выполняется на данных из поля рисования. Результаты отображаются в виде отдельного окна (рис. 3).

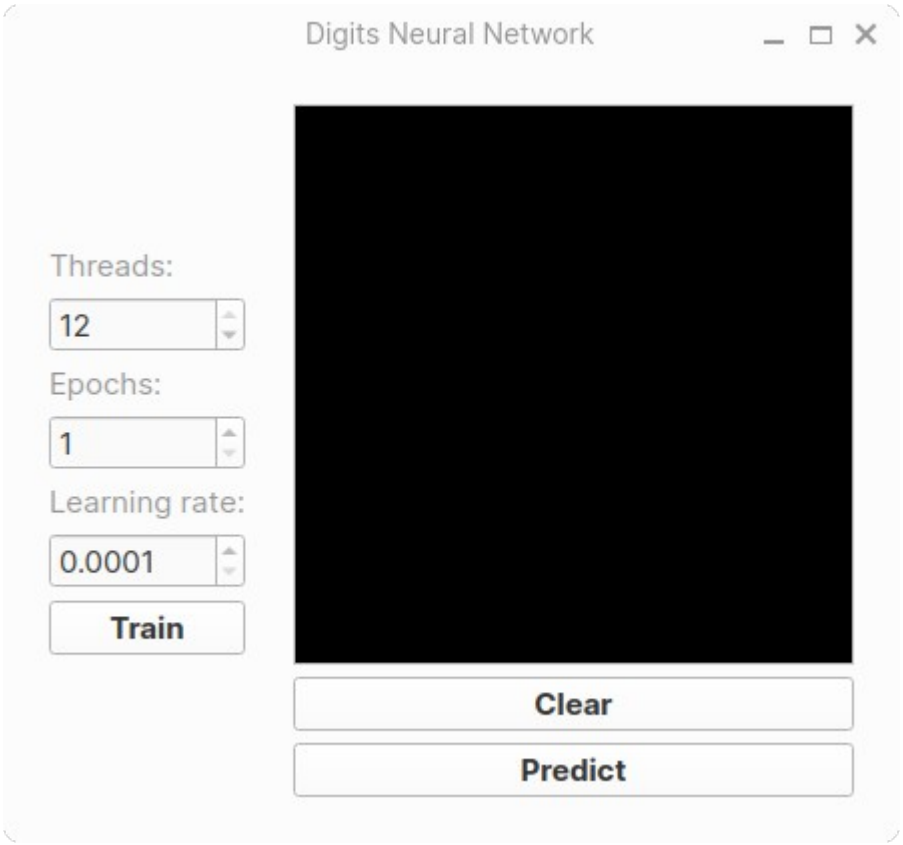

Рисунок 1 — Скриншот главного окна

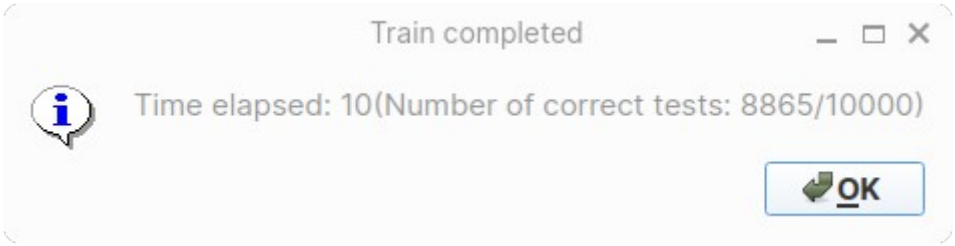

Рисунок 2 — Скриншот окна отображения результатов

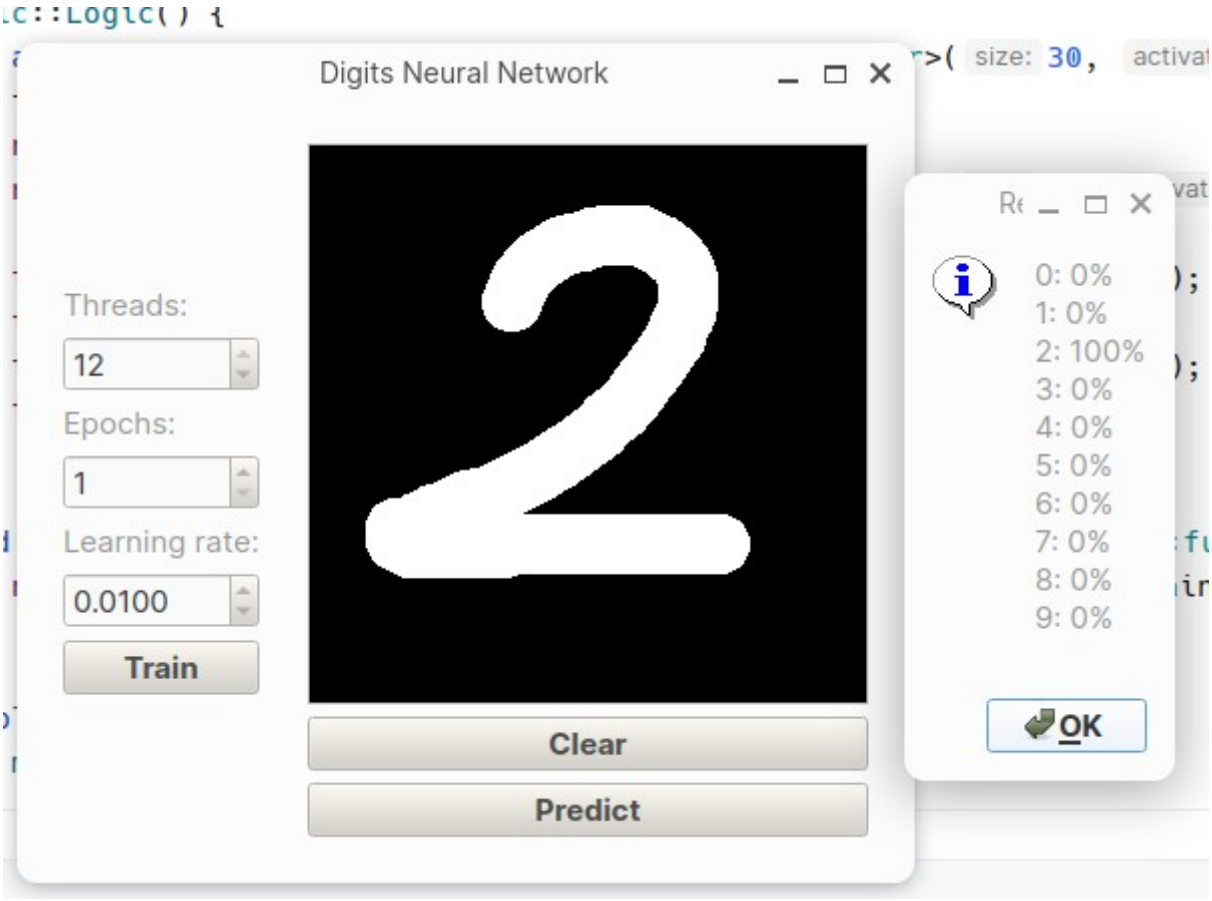

Рисунок 3 — Скриншот окна результатов запуска модели

3.4 Оценка эффективности параллельных вычислений

Данная программа была протестирована на компьютере с 12-поточном процессоре AMD Ryzen 5500U.

При использовании одного потока время выполнения обучения 5 эпох составило 871 секунд (14.5 минут) (рис. 4). При использовании 12 поток га обучение того же количества эпох ушло 237 секунды (3.95 минут) (рис. 5). Т.е. обучение ускорилось в 3.67 раз.

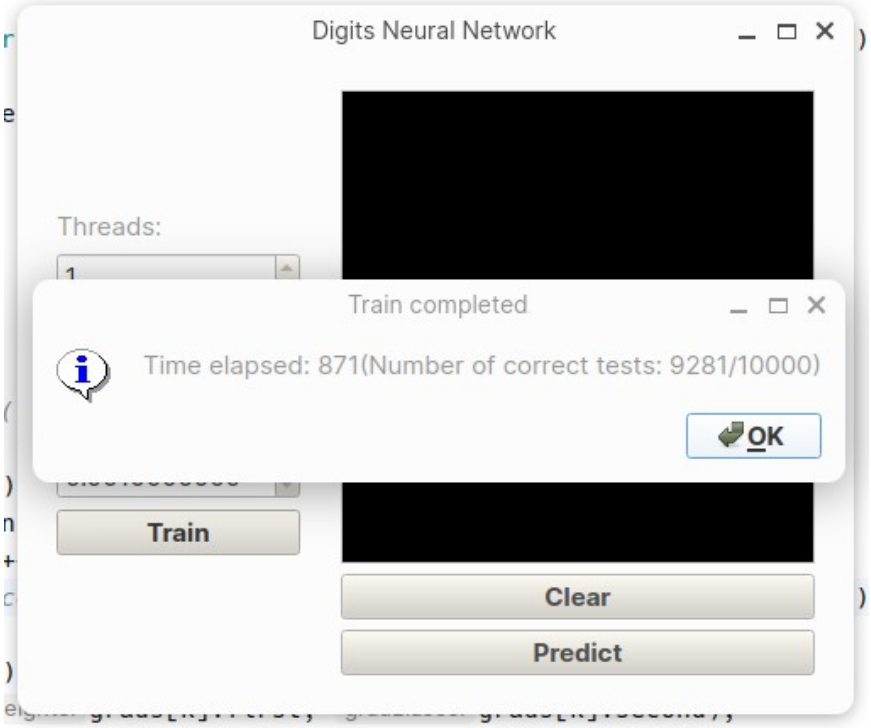

Рисунок 4 — Результат тестирования программы при использовании 1 потока

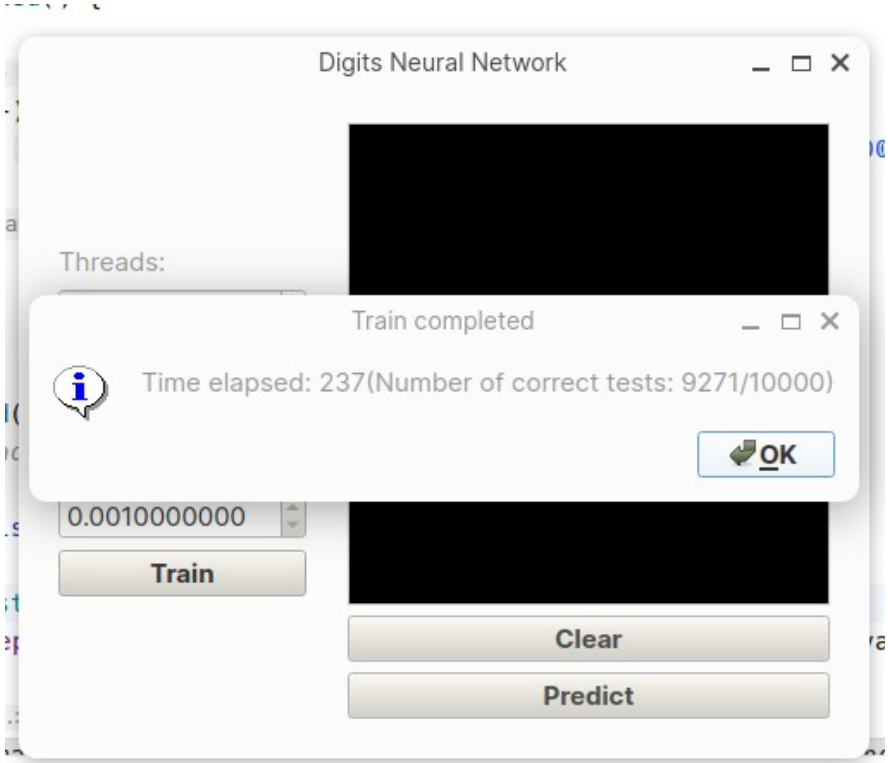

Рисунок 5 — Результат тестирования программы при использовании 12 потоков

ЗАКЛЮЧЕНИЕ

В данной работе была рассмотрена организация параллельных вычислений, которая является одной из ключевых задач в области высокопроизводительного вычисления. Были представлены виды различных параллельных вычислительных систем, описаны их принципы работы и приведены примеры применения.

В ходе работы было выяснено, что выбор подходящей параллельной вычислительной системы зависит от поставленных задач и необходимого уровня производительности. Кроме того, эффективность параллельных вычислений зависит от правильного разбиения задачи на части и управления распределением ресурсов между вычислительными узлами.

Организация параллельных вычислений на различных типах вычислительных систем требует глубоких знаний в области архитектуры вычислительных систем и алгоритмов параллельных вычислений. Но постоянный рост объемов данных и требования к производительности вычислений, а также развитие новых технологий и подходов, делают организацию параллельных вычислений неотъемлемой частью современной вычислительной науки и практики.

Так же была разработана и протестирована программа для обучения нейросети классификации рукописных цифр, использующая параллельные вычисления.

В целом, организация параллельных вычислений играет значительную роль в ускорении вычислительных процессов и повышении эффективности вычислительных систем, а также в решении многих прикладных задач, требующих высокой производительности.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Словарь по кибернетике / Под редакцией академика В. С. Михалевича. — 2-е. — Киев: Главная редакция Украинской Советской Энциклопедии имени М. П. Бажана, 1989. — 751 с. — (С48). — 50 000 экз. — ISBN 5-88500-008-5.

2. Introduction to Parallel Computing / Roman Trobec , Boštjan Slivnik , Patricio Bulić , Borut Robič, — Springer Nature Switzerland AG, 2018 — 256 с.