МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Системы параллельной обработки данных»

Тема: Использование функций обмена данными «точка-точка»

Студент гр. 9303 _____ Колованов Р.А.

Преподаватель _____ Татаринов Ю.С.

Санкт-Петербург

2023

Цель работы.

Написание программы, использующую функции обмена «точка-точка» с использованием технологии OpenMP. Сравнение двух технологий параллельного программирования MPI и OpenMP.

Формулировка задания.

Процесс 0 генерирует массив и раздает его другим процессам для обработки (например, поиска нулевых элементов), после чего собирает результат.

В качестве обработки был выбран подсчет количество нулей в массиве.

Краткое описание алгоритма.

Для начала исходная программа, написанная с использованием технологии MPI, была переписана с использованием технологии OpenMP. Алгоритм обмена сообщениями представляет собой следующую последовательность действий:

- В самом начале поток 0 генерирует массив случайных чисел (в диапазоне от -5 до 5) заданного размера N;
- Далее поток 0 делит сформированный массив на T – 1 равных частей, где T – количество потоков, после чего части массива отправляются в соответствующие T – 1 потоков, которые их принимают;
- Потоки, получившие соответствующие части массива, осуществляют подсчет количества нулей в части массива, после чего отправляют результат потоку 0;
- В завершение поток 0 принимает от остальных потоков результаты подсчета, суммирует их и выводит на экран.

Если размер массиве делится на N – 1 не нацело, то остаточная часть массива обрабатывается потоком 0.

Поскольку в OpenMP не предусмотрена функциональность обмена сообщениями между процессами, была реализована собственная система обмена сообщениями.

Структура *Message* представляет собой сообщение, отправляемое от одного процесса другому. Сообщение хранит в себе массив данных, информацию о размере массива и размере его элементов, а также информацию об отправителе.

Структура *ThreadInputStorage* представляет собой хранилище входящих сообщений для потока. Каждый поток имеет собственное хранилище. Хранилище представлено в виде связного списка сообщений *Message*. С хранилищем можно выполнять два действия: добавить в него сообщение (отправить сообщение потоку) и взять из него сообщение (получить сообщение). Для обеспечения потокобезопасности при работе с хранилищем был использован замок *omp_lock_t*.

Для отправки и приема сообщений были созданы две функции *sendData* и *receiveData* соответственно. Функции являются блокирующими.

Формальное описание алгоритма.

Формальное описание алгоритма на сетях Петри для не модифицированной программы представлена на рис. 1.

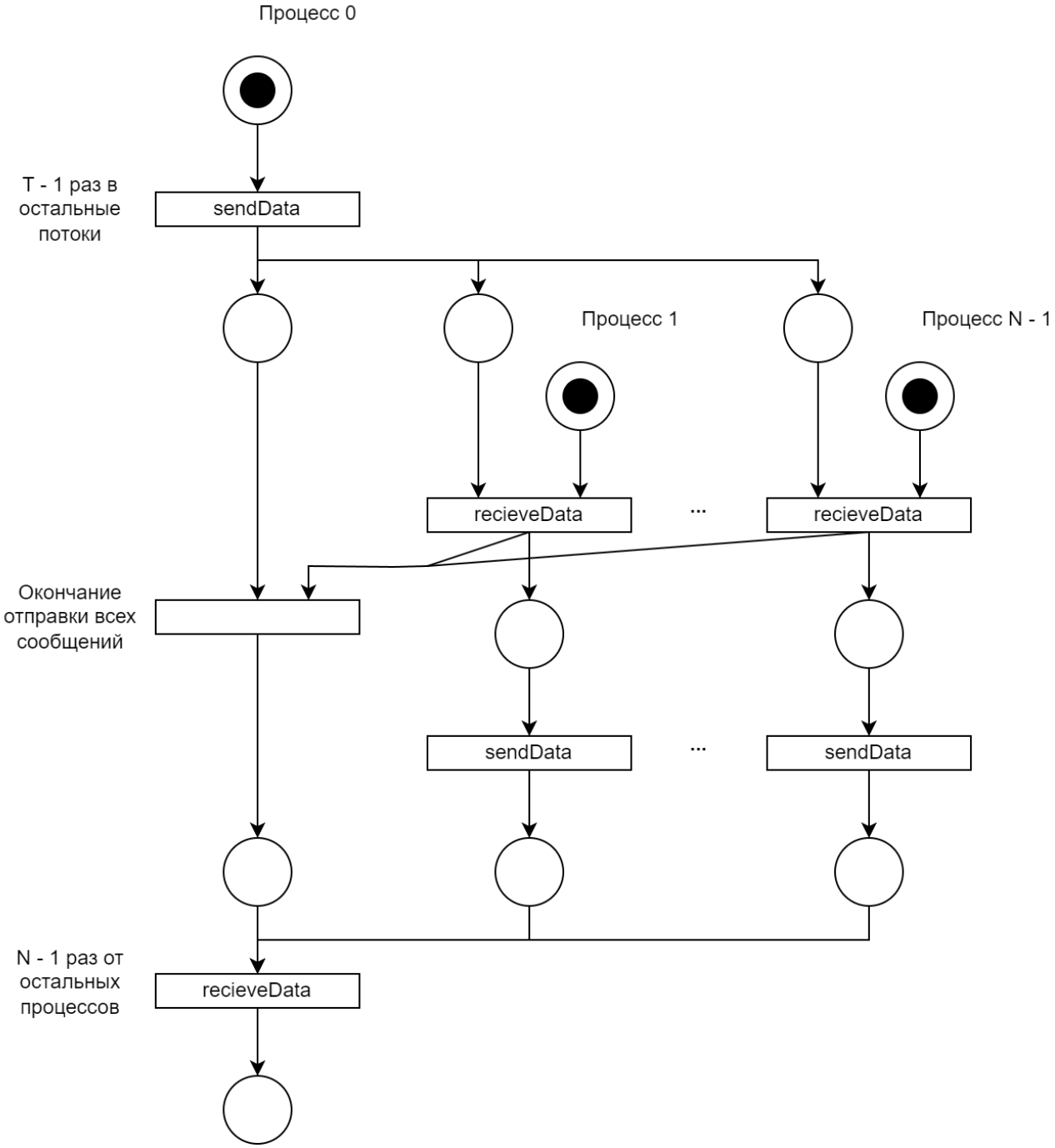

Рисунок 1 – Формальное описание алгоритма на сетях Петри.

Листинг программы.

Исходная программа, использующая технологию MPI, представлена в листинге 1.

Листинг 1. Программа на основе MPI.

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

int main(int argc, char** argv) {
  const int dataSize = 100000000;
  int processNumber, processRank;
  MPI_Status status;

  srand(time(nullptr));

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &processNumber);
  MPI_Comm_rank(MPI_COMM_WORLD, &processRank);

  int dataBlockSize, remainDataSize;
  if (processNumber != 1) {
      dataBlockSize = dataSize / (processNumber - 1);
      remainDataSize = dataSize % (processNumber - 1);
  }
  else {
      dataBlockSize = 0;
      remainDataSize = dataSize;
  }

  if (processRank == 0) {
      int count = 0;
      int* data = new int[dataSize];

      for (int i = 0; i < dataSize; ++i) {
        data[i] = rand() % 11 - 5;
      }

      double startTime = MPI_Wtime();

      for (int i = 1; i < processNumber; ++i) {
        MPI_Send(data + (i - 1) * dataBlockSize, dataBlockSize, MPI_INT, i, 0,
MPI_COMM_WORLD);
      }

      for (int i = dataSize - 1; dataSize - 1 - remainDataSize < i; --i) {
        if (data[i] == 0) {
            ++count;
        }
      }

      for (int result = 0, i = 1; i < processNumber; ++i) {
        MPI_Recv(&result, 1, MPI_INT, i, 0, MPI_COMM_WORLD, &status);
        count += result;
      }

      double deltaTime = MPI_Wtime() - startTime;

      delete[] data;

      std::cout << "Number of '0' in array: " << count << "\n";
      std::cout << "Elapsed time: " << deltaTime << "\n";
  }
  else {
      int count = 0;
      int* data = new int[dataBlockSize];
```

```
      MPI_Recv(data, dataBlockSize, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);

      for (int i = 0; i < dataBlockSize; ++i) {
        if (data[i] == 0) {
            ++count;
        }
      }

      MPI_Send(&count, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
  }

  MPI_Finalize();
  return 0;
}
```

Переписанная программа, использующая технологию OpenMP, представлена в листинге 2.

Листинг 2. Программа на основе OpenMP.

```
#include <iostream>
#include <array>
#include <list>
#include <thread>
#include <omp.h>

/*!
 * \brief Сообщение. Содержит данные и информацию об отправителе.
 */
struct Message
{
  static const int ANY_THREAD = -1;

  Message() :
      data{ nullptr },
      count{ 0 },
      typeSize{ 0 },
      senderId{ ANY_THREAD }
  {}

  Message(const Message&) = delete;
  Message& operator=(const Message&) = delete;
  Message(Message&&) = delete;
  Message& operator=(Message&&) = delete;

  ~Message()
  {
      reset();
  }

  void setData(void* data, int count, int typeSize, int senderId = ANY_THREAD)
  {
      reset();
      this->senderId = senderId;
      this->count = count;
      this->typeSize = typeSize;
      this->data = new char[count * typeSize];
      std::memcpy(this->data, data, count * typeSize);
  }
```

```
  void reset()
  {
      if (data != nullptr)
      {
        delete[] data;
        data = nullptr;
      }
      count = 0;
      typeSize = 0;
      senderId = ANY_THREAD;
  }

  void* data;             // Указатель на данные
  size_t count;           // Количество данных
  size_t typeSize;        // Размер типа данных
  short int senderId; // ID потока-отправителя
};

/*!
 * \brief Хранилище сообщений. Хранит сообщения для определенного потока в виде
связного списка.
 */
struct ThreadInputStorage
{
  explicit ThreadInputStorage() :
      messages{},
      storageLock{ nullptr }
  {
      omp_init_lock(&storageLock);
  }

  ~ThreadInputStorage()
  {
      omp_destroy_lock(&storageLock);
  }

  void pushMessage(Message* message)
  {
      omp_set_lock(&storageLock);
      messages.push_back(message);
      omp_unset_lock(&storageLock);
  }

  Message* popMessage(int senderId = Message::ANY_THREAD)
  {
      Message* result = nullptr;

      omp_set_lock(&storageLock);
      if (!messages.empty())
      {
        for (auto it = messages.cbegin(); it != messages.cend(); ++it)
        {
            if (senderId == Message::ANY_THREAD || senderId == (*it)->senderId)
            {
              result = *it;
              messages.erase(it);
              break;
            }
        }
      }
      omp_unset_lock(&storageLock);

      return result;
```

```
  }

  omp_lock_t storageLock;           // Мьютекс на доступ к списку сообщений
  std::list<Message*> messages;     // Связный список сообщений
};

namespace
{
  constexpr int THREADS = 4;
  const int DATA_SIZE = 100;
  std::array<ThreadInputStorage, THREADS> INPUT_STORAGES;
}

/*!
 * \brief Функция отправки сообщения другому потоку.
 *
 * Функция является блокирующей - освобождается после того, как данные из
входного буффера будут скопированы и отправлены.
 *
 * \param data Указатель на массив данных, который необходимо отправить.
 * \param count Количество элементов в массиве данных
 * \param typeSize Размер одного элемента массива в байтах
 * \param destination ID потока, которому необходимо отправить сообщение
 */
void sendData(void* data, int count, int typeSize, int destination)
{
  if (destination < 0 || destination >= THREADS)
  {
      return;
  }

  auto& storage = INPUT_STORAGES.at(destination);
  auto* message = new Message;
  message->setData(data, count, typeSize, omp_get_thread_num());
  storage.pushMessage(message);
}

/*!
 * \brief Функция приема сообщения от другого потока.
 *
 * Функция является блокирующей - освобождается после того, как данные из
сообщения буду получены.
 *
 * \param data Указатель на массив данных, куда необходимо записать полученные
данные.
 * \param count Количество элементов в массиве данных
 * \param typeSize Размер одного элемента массива в байтах
 * \param source ID потока, от которого необходимо получить сообщение
 */
Message* recieveData(void* data, int count, int typeSize, int source =
Message::ANY_THREAD)
{
  auto& storage = INPUT_STORAGES.at(omp_get_thread_num());
  auto* message = storage.popMessage(source);

  while (message == nullptr)
  {
      message = storage.popMessage(source);
  }

  int size = count * typeSize;
  if (size > message->count * message->typeSize)
  {
```

```
      size = message->count * message->typeSize;
  }

  std::memcpy(data, message->data, size);

  delete message;
}

int main()
{
  srand(time(nullptr));

  #pragma omp parallel num_threads(THREADS)
  {
      const auto threadId = omp_get_thread_num();
      int dataBlockSize, remainDataSize;

      if (THREADS != 1) {
        dataBlockSize = DATA_SIZE / (THREADS - 1);
        remainDataSize = DATA_SIZE % (THREADS - 1);
      }
      else {
        dataBlockSize = 0;
        remainDataSize = DATA_SIZE;
      }

      if (threadId == 0) {
        int count = 0;
        int* data = new int[DATA_SIZE];

        for (int i = 0; i < DATA_SIZE; ++i) {
            data[i] = rand() % 11 - 5;
        }

        double startTime = omp_get_wtime();

        for (int i = 1; i < THREADS; ++i) {
            sendData(data + (i - 1) * dataBlockSize, dataBlockSize, sizeof(int),
i);
        }

        for (int i = DATA_SIZE - 1; DATA_SIZE - 1 - remainDataSize < i; --i) {
            if (data[i] == 0) {
              ++count;
            }
        }

        for (int result = 0, i = 1; i < THREADS; ++i) {
            recieveData(&result, 1, sizeof(int), i);
            count += result;
        }

        double deltaTime = omp_get_wtime() - startTime;

        delete[] data;

        std::cout << "Number of '0' in array: " << count << "\n";
        std::cout << "Elapsed time: " << deltaTime << "\n";
      }
      else {
        int count = 0;
        int* data = new int[dataBlockSize];
```

```
        recieveData(data, dataBlockSize, sizeof(int), 0);

        for (int i = 0; i < dataBlockSize; ++i) {
            if (data[i] == 0) {
              ++count;
            }
        }

        sendData(&count, 1, sizeof(int), 0);
      }
  }

  return 0;
}
```

Результаты работы программы на различном количестве процессов.

Результаты работы программы, использующей технологию MPI, представлена в таблице 3.

Таблица 3 – Результаты работы программы на MPI.

№ п/п	Размер сообщения	Количество процессоров	Результаты работы программы
1.	100	1	Number of '0' in array: 8 Elapsed time: 1.00001e-07
2.		2	Number of '0' in array: 9 Elapsed time: 0.0004116
3.		4	Number of '0' in array: 5 Elapsed time: 0.0005057
4.		6	Number of '0' in array: 9 Elapsed time: 0.0007394
5.		8	Number of '0' in array: 9 Elapsed time: 0.0008611
6.		12	Number of '0' in array: 14 Elapsed time: 0.0011345
7.	10000	1	Number of '0' in array: 916 Elapsed time: 4.4e-06
8.		2	Number of '0' in array: 919 Elapsed time: 0.0003749
9.		4	Number of '0' in array: 877 Elapsed time: 0.0007674
10.		6	Number of '0' in array: 935 Elapsed time: 0.0011171
11.		8	Number of '0' in array: 879 Elapsed time: 0.0013507
12.		12	Number of '0' in array: 890 Elapsed time: 0.0012499

Продолжение таблицы 3.

13.	1000000	1	Number of '0' in array: 90832 Elapsed time: 0.000445
14.		2	Number of '0' in array: 90770 Elapsed time: 0.00199
15.		4	Number of '0' in array: 90847 Elapsed time: 0.0021869
16.		6	Number of '0' in array: 91179 Elapsed time: 0.0031109
17.		8	Number of '0' in array: 90831 Elapsed time: 0.0035063
18.		12	Number of '0' in array: 91256 Elapsed time: 0.0054186
19.	10000000	1	Number of '0' in array: 911129 Elapsed time: 0.0048838
20.		2	Number of '0' in array: 907005 Elapsed time: 0.015893
21.		4	Number of '0' in array: 909393 Elapsed time: 0.0129316
22.		6	Number of '0' in array: 909090 Elapsed time: 0.0115124
23.		8	Number of '0' in array: 909153 Elapsed time: 0.0201529
24.		12	Number of '0' in array: 909415 Elapsed time: 0.0172279

Результаты работы программы, использующей технологию OpenMP, представлена в таблице 4.

Таблица 4 – Результаты работы программы на OpenMP.

№ п/п	Размер сообщения	Количество потоков	Результаты работы программы
1.	100	1	Number of '0' in array: 11 Elapsed time: 1.00001e-07
2.		2	Number of '0' in array: 12 Elapsed time: 4.29e-05
3.		4	Number of '0' in array: 5 Elapsed time: 0.0004236
4.		6	Number of '0' in array: 4 Elapsed time: 0.0001574
5.		8	Number of '0' in array: 11 Elapsed time: 0.0003861

Продолжение таблицы 4.

6.		12	Number of '0' in array: 8 Elapsed time: 0.0007917
7.	10000	1	Number of '0' in array: 872 Elapsed time: 4.3e-06
8.		2	Number of '0' in array: 935 Elapsed time: 4e-05
9.		4	Number of '0' in array: 911 Elapsed time: 4.1e-05
10.		6	Number of '0' in array: 889 Elapsed time: 5.44e-05
11.		8	Number of '0' in array: 918 Elapsed time: 0.0001376
12.		12	Number of '0' in array: 942 Elapsed time: 0.0002663
13.	1000000	1	Number of '0' in array: 90326 Elapsed time: 0.000568
14.		2	Number of '0' in array: 91646 Elapsed time: 0.0023731
15.		4	Number of '0' in array: 91203 Elapsed time: 0.0011393
16.		6	Number of '0' in array: 90594 Elapsed time: 0.0023454
17.		8	Number of '0' in array: 90432 Elapsed time: 0.0022833
18.		12	Number of '0' in array: 90989 Elapsed time: 0.0020991
19.	10000000	1	Number of '0' in array: 908907 Elapsed time: 0.0045666
20.		2	Number of '0' in array: 910044 Elapsed time: 0.0221338
21.		4	Number of '0' in array: 909678 Elapsed time: 0.0115137
22.		6	Number of '0' in array: 909643 Elapsed time: 0.010659
23.		8	Number of '0' in array: 908099 Elapsed time: 0.0093211
24.		12	Number of '0' in array: 909750 Elapsed time: 0.0135737

График зависимости времени выполнения программы от числа процессов для разного объема данных.

Графики зависимости времени выполнения подсчета нулей в массиве от числа процессов для разных размеров массива чисел и для разных технологий MPI и OpenMP представлен на рис. 2, рис. 3, рис. 4 и рис. 5.

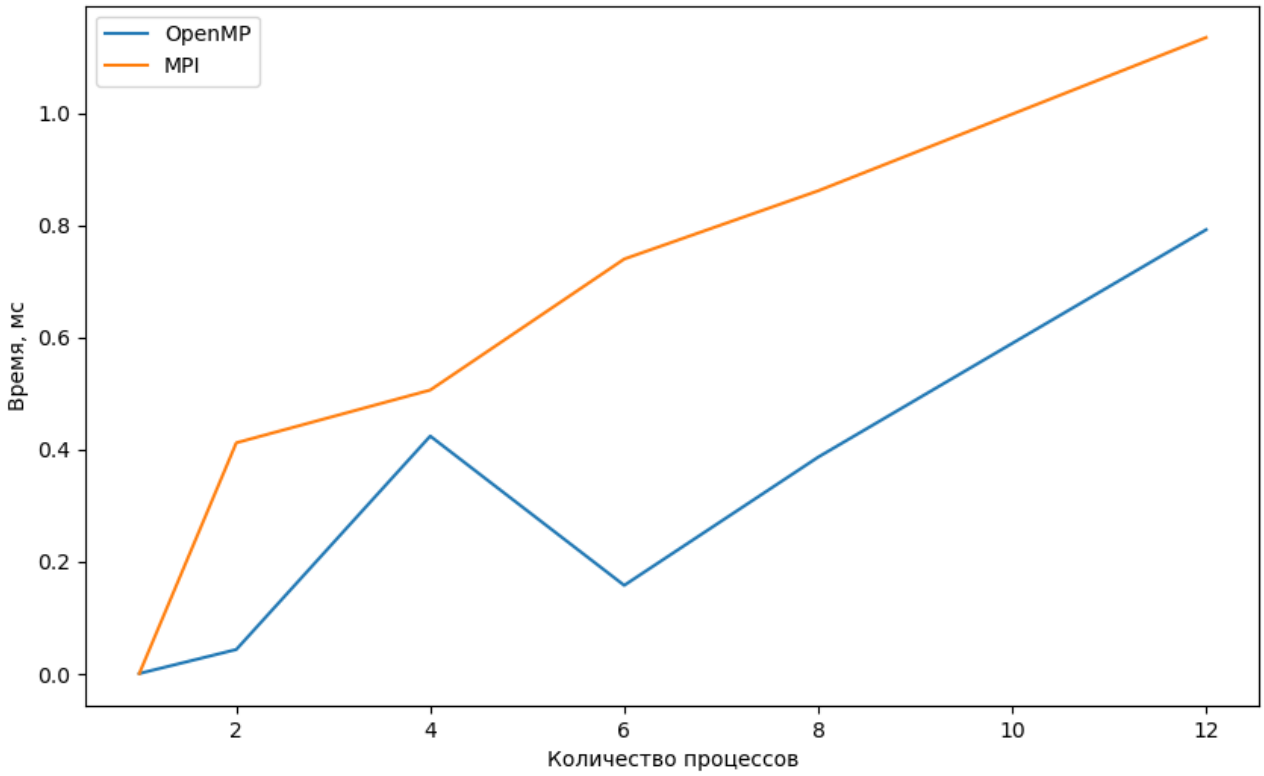

Рисунок 2 – Зависимость времени подсчета от числа процессов при длине массива 100.

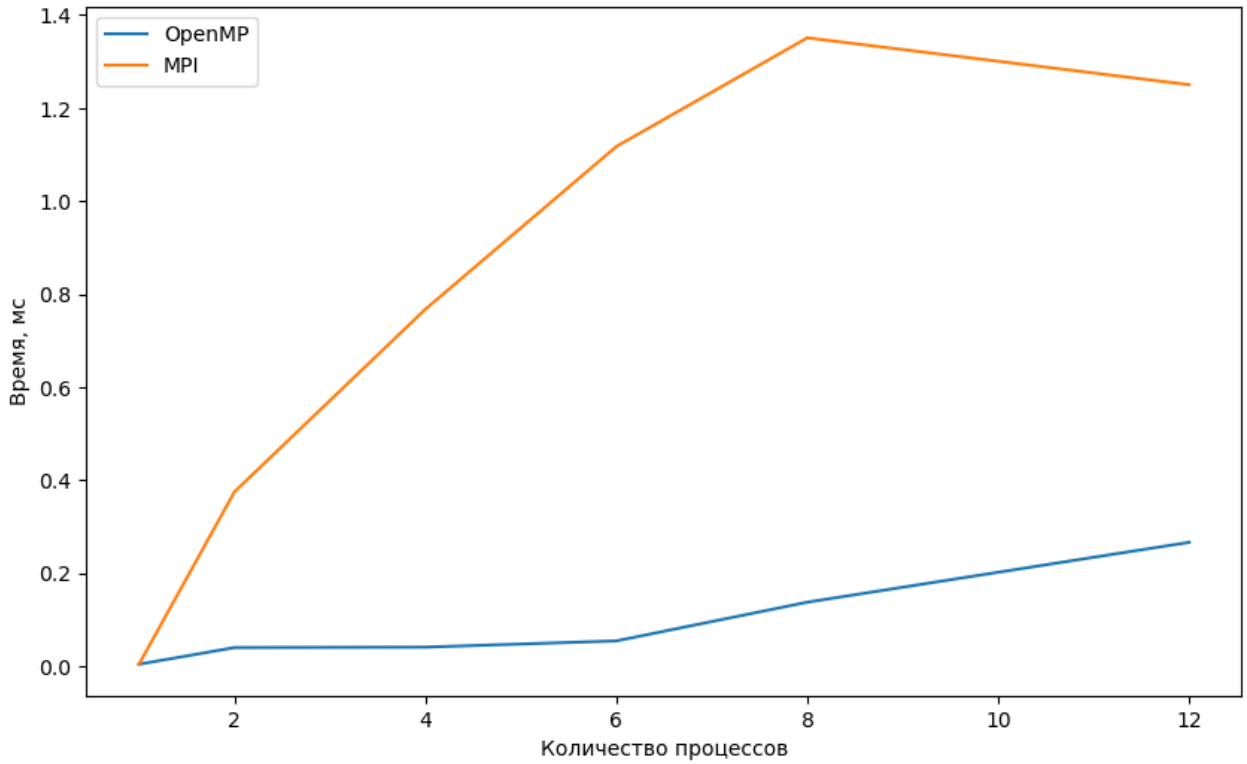

Рисунок 3 – Зависимость времени подсчета от числа процессов при длине массива 10000.

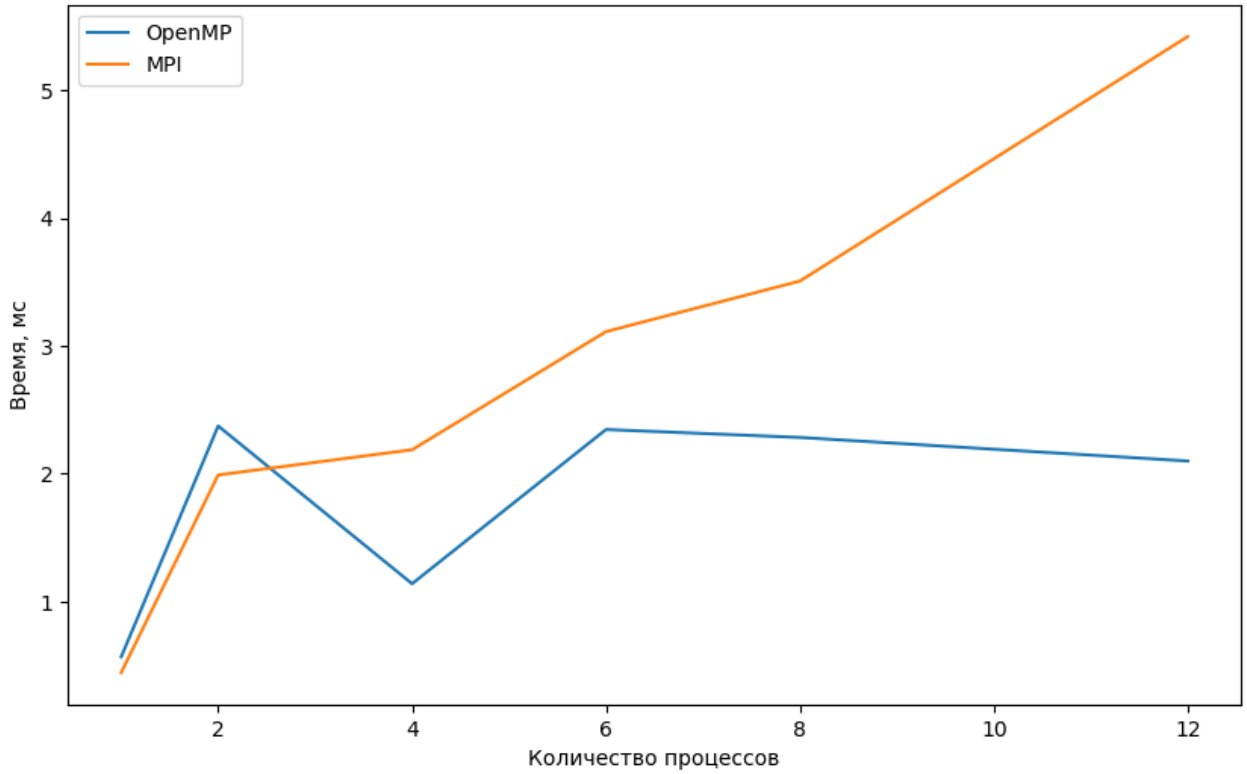

Рисунок 4 – Зависимость времени подсчета от числа процессов при длине массива 1000000.

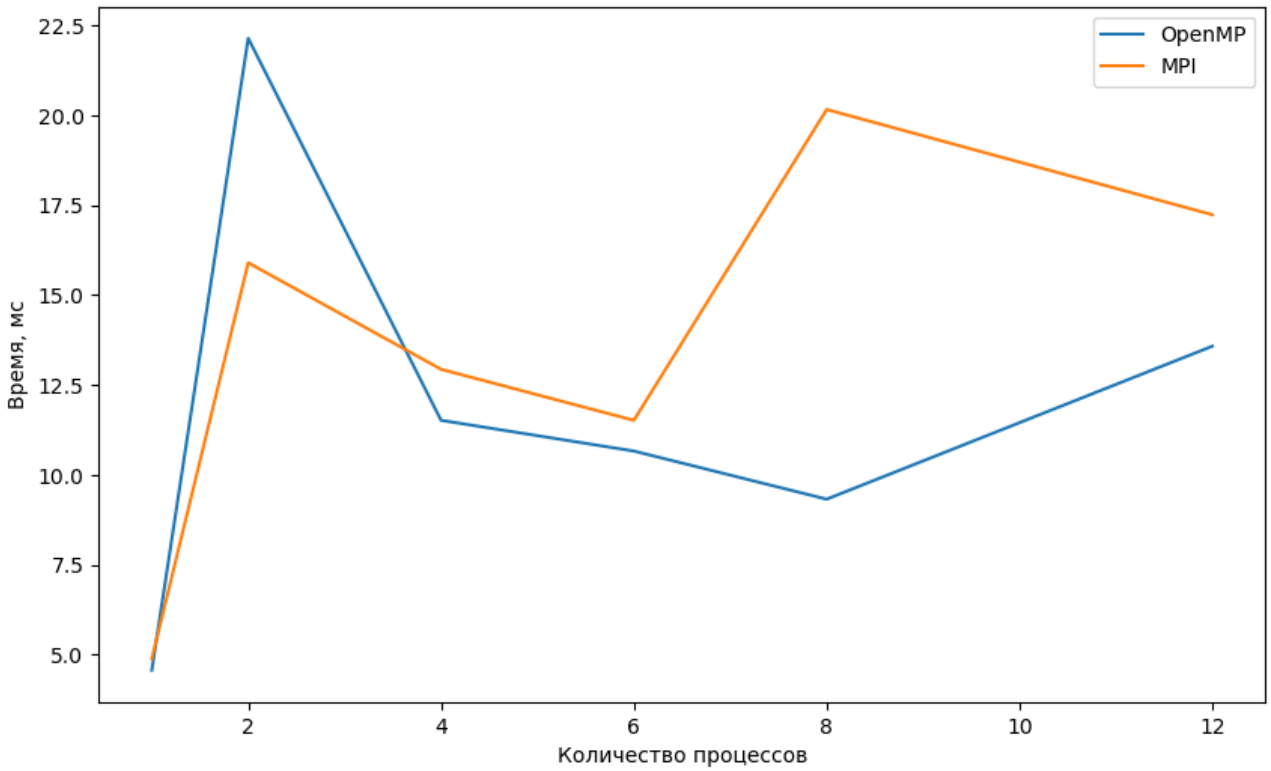

Рисунок 5 – Зависимость времени подсчета от числа процессов при длине массива 10000000.

Для всех графиков характерна одна черта: выполнение задачи на одном процессе всегда быстрее, чем распараллеливание задачи на несколько процессов. Это можно объяснить тем, что задача не является слишком трудоемкой.

Как видно из рисунков 2 и 3, для малых размеров массива (100 и 10000) время выполнения подсчета увеличивается с увеличением количества процессов. Это можно объяснить тем, что накладные расходы на передачу информации между процессами превышают расходы на выполнение задачи подсчета для данного размера массива. При этом, технология OpenMP показывает себя лучше, чем технология MPI. Можно предположить, что это объясняется наличием у потоков в OpenMP общей памяти, за счет которой обмен происходит быстрее.

Как видно из рисунков 4 и 5, для больших размеров массиве ситуация уже несколько иная: с увеличением количества процессов время подсчета сначала начинает уменьшаться, но после 4-8 процессов время опять начинает возрастать. Это можно объяснить тем, что для такого большого массива накладные расходы

на передачу информации между процессами становятся уже не настолько большими по сравнению с расходами на выполнение задачи подсчета. При этом, технология OpenMP в большинстве своем показывает себя лучше, чем технология MPI.

Замеры проводились на ПК со следующими характеристиками:

- Процессор AMD Ryzen 5 5600X (6 физических и 12 логических ядер);
- Оперативная память 32 ГБ 3200 МГц;
- Видеокарта GeForce RTX 3060 Ti.

График ускорения.

Графики ускорения для технологий MPI и OpenMP представлены на рис. 6 и рис. 7 соответственно.

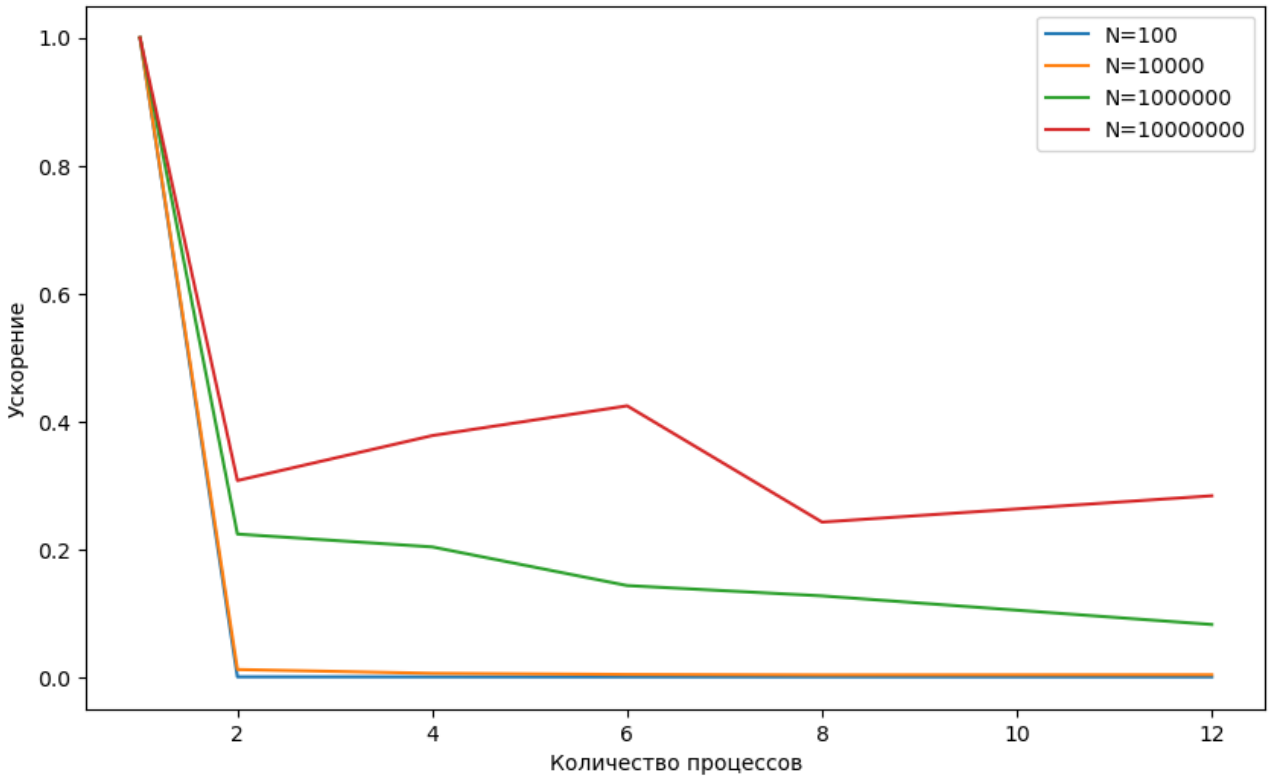

Рисунок 6 – График ускорения для технологии MPI.

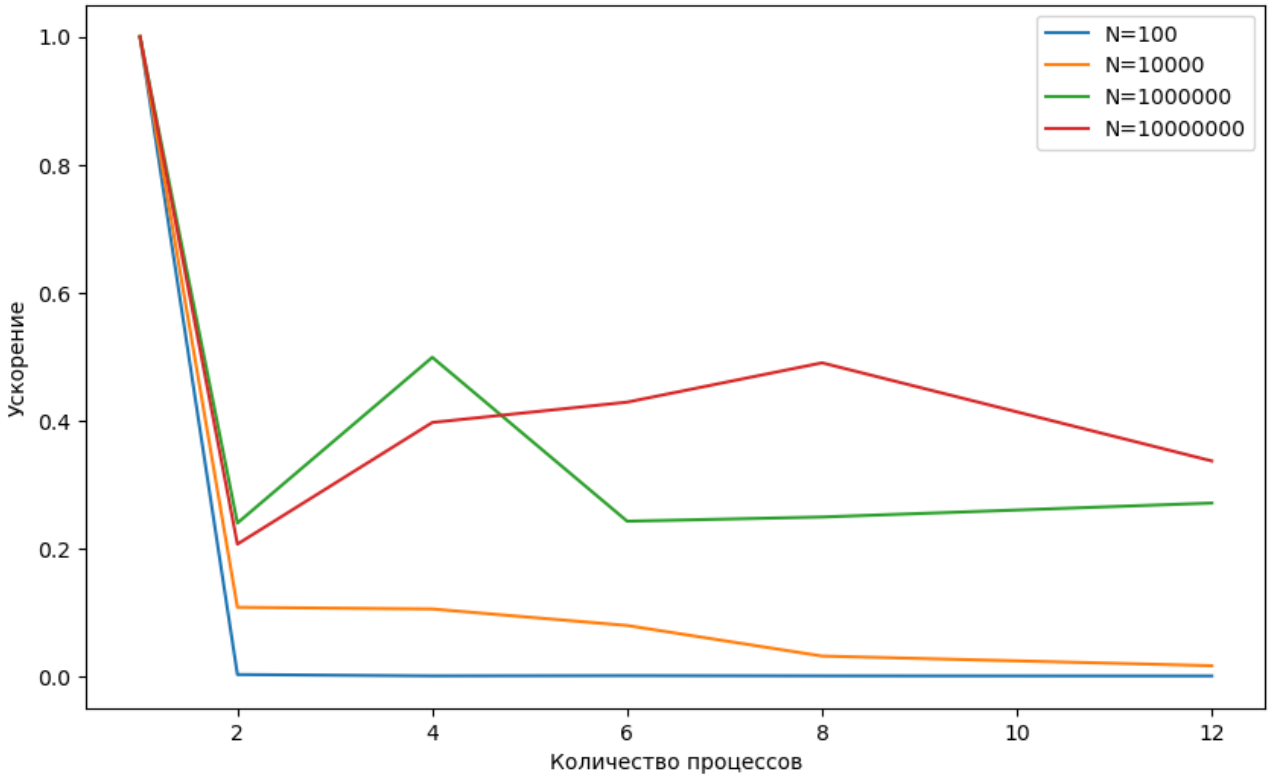

Рисунок 7 – График ускорения для технологии OpenMP.

Графики ускорения на рис. 6 и рис. 7 подтверждают выводы, сделанные в предыдущем пункте. Также из них наглядно видно, что для данной задачи распараллеливание не дает какого-либо ускорения решения задачи. Быстрее будет решить задачу на одном процессе, чем рассылать данные процессам по частям, тратя на этого значительное время (по сравнению со временем решения задачи).

Выводы по работе.

Была написана программа, осуществляющая параллельный подсчет количество нулей в массиве случайных чисел. Было выполнено измерение времени обработки при различных размерах сообщения с разным количеством процессов.

Исходя из полученных графиков видно, что для данной задачи распараллеливание не дает какого-либо ускорения ее решения. Решение задачи на одном процессе будет быстрее, чем решение этой же задачи на нескольких

процессах. Это объясняется тем, что задача подсчета нулей в массиве не сильно трудоемкая. Для небольших размеров массива ускорение близится к 0, а для достаточно больших размеров ускорение становится более существенным (в пределах 0.1 – 0.6), поскольку задача подсчета нулей для огромного массива становится более трудоемкой. Но в обоих случаях ускорение все равно не превышает 1: преимущества распараллеливания задачи не превышают затраты на обмен сообщениями между процессами.

В контексте сравнения технологий MPI и OpenMP, технология OpenMP показала себя лучше, чем MPI. Для данной задачи OpenMP зачастую давал меньшее время решения, чем MPI.