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

ОТЧЕТ

по лабораторной работе №2
по дисциплине «Системы параллельной обработки данных»
Тема: Использование аргументов-джокеров

Студент гр. 9303 _____ Колованов Р.А.

Преподаватель _____ Татаринов Ю.С.

Санкт-Петербург
2023

Цель работы.

Написание программы, использующую функции обмена с применением аргументов-джокеров с использованием технологии OpenMP. Сравнение двух технологий параллельного программирования MPI и OpenMP.

Формулировка задания.

Имитация топологии «звезда» (процесс с номером 0 реализует функцию центрального узла). Процессы в случайном порядке генерируют пакеты, состоящие из адресной и информационной части и передают их в процесс 0 Маршрутная часть пакета содержит номер процесса-адресата. Процесс 0 переадресовывает пакет адресату. Адресат отчитывается перед процессом 0 в получении. Процесс 0 информирует процесс-источник об успешной доставке.

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 процесс 0 начинает принимать сообщения от других процессов, а другие процессы начинают отправлять сообщения, содержащие маршрутную информацию и данные, процессу 0 (сами пакеты хранят ранг некоторого процесса в качестве места назначения), после чего входят в режим ожидания сообщений от процесса 0;
- Процесс 0, принимая сообщения от разных процессов, перенаправляет их процессу, указанному в маршрутной информации пакета.
- Другие процессы, получив сообщение от процесса 0, отправляют сообщение о подтверждении получения обратно процессу, от которого он получил сообщение. Отправка осуществляется аналогичным образом через процесс 0.
- Когда все сообщения будут отправлены, процесс 0 отправит всем процессом сообщение с указанием завершения работы и заканчивает

свою работу. Остальные процессы, получившие сообщения с указанием завершения работы от процесса 0, тоже завершают работу.

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 программы представлено на рис. 1.

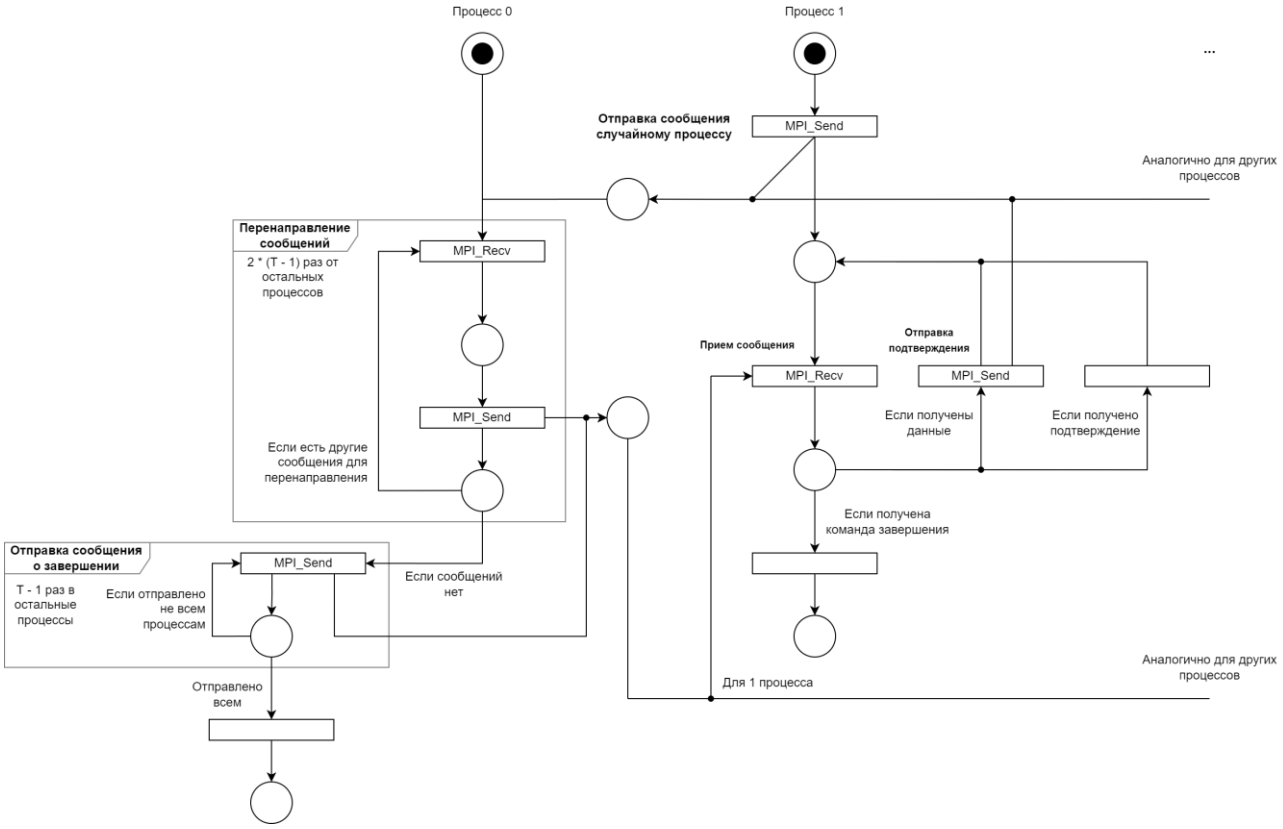

Рисунок 1 – Формальное описание алгоритма на сетях Петри.

Листинг программы.

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

Листинг 1. Программа на основе MPI.

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

namespace {
  const size_t DATA_SIZE = 32;
}

struct Message {
  enum Type {
      Data = 0,
      Confirmation = 1,
      Finish = 2,
      Unknown = -1
  };

  Type type = Type::Unknown;
  int source = -1;
  int destination = -1;
  char data[DATA_SIZE];

  Message() = default;

  Message(const Type type, int source, int destination)
```

```
  {
      this->type = type;
      this->source = source;
      this->destination = destination;
  }
};

void fillArrayWithData(char* data, int count) {
  for (int i = 0; i < count; ++i) {
      data[i] = i % 256;
  }
}

int main2(int argc, char** argv) {
  int processNumber, processRank;
  MPI_Status status;

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

  srand(time(nullptr) + static_cast<time_t>(processRank) * 1000);

  if (processNumber < 2) {
      std::cerr << "At least two processes are required to work.\n";
      MPI_Finalize();
      return 0;
  }

  double elapsedTime = -1;

  if (processRank == 0) {
      Message* message = new Message;

      // Перенаправляем пакеты
      for (int i = 1; i <= (processNumber - 1) * 2; ++i) {
        MPI_Recv(message,    sizeof(Message),    MPI_BYTE,    MPI_ANY_SOURCE,
MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        MPI_Send(message,  sizeof(Message),  MPI_BYTE,  message->destination,
message->type, MPI_COMM_WORLD);
      }

      *message = { Message::Type::Finish, 0, 0 };

      // Отправляем пакеты с запросом на завершение
      for (int i = 1; i < processNumber; ++i) {
        message->destination = i;
        MPI_Send(message,  sizeof(Message),  MPI_BYTE,  message->destination,
message->type, MPI_COMM_WORLD);
      }

      delete message;
  }
  else {
      int destinationProcess = 1 + rand() % (processNumber - 1);

      Message*  message  =  new  Message(Message::Type::Data,  processRank,
destinationProcess);
      fillArrayWithData(message->data, DATA_SIZE);

      double startTime = MPI_Wtime();

      // Отправляем пакет с данными случайному процессу
```

```
      MPI_Send(message,  sizeof(Message),  MPI_BYTE,  0,  Message::Type::Data,
MPI_COMM_WORLD);

      // Принимаем пакеты от 0 процесса и обрабатываем их
      do {
        MPI_Recv(message,    sizeof(Message),    MPI_BYTE,    0,    MPI_ANY_TAG,
MPI_COMM_WORLD, &status);

        if (message->type == Message::Type::Data) {
            *message  =  {  Message::Type::Confirmation,  processRank,  message-
>source};
            MPI_Send(message,        sizeof(Message),        MPI_BYTE,        0,
Message::Type::Confirmation, MPI_COMM_WORLD);
        }

      } while (message->type != Message::Type::Finish);

      elapsedTime = MPI_Wtime() - startTime;

      delete message;
  }

  double maxTime = -1;
  MPI_Reduce(&elapsedTime,    &maxTime,    1,    MPI_DOUBLE,    MPI_MAX,    0,
MPI_COMM_WORLD);
  if (processRank == 0)
  {
      std::cout << "Elapsed time: " << maxTime << "\n";
      std::cout.flush();
  }

  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 = 32;
  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);
  }

  size_t size = count * typeSize;
  if (size > message->count * message->typeSize)
  {
      size = message->count * message->typeSize;
  }

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

  delete message;
}

namespace wrapper
{
  struct Message {
      enum Type {
        Data = 0,
        Confirmation = 1,
        Finish = 2,
        Unknown = -1
      };

      Type type = Type::Unknown;
      int source = -1;
      int destination = -1;
      char data[DATA_SIZE];

      Message() = default;

      Message(const Type type, int source, int destination)
      {
        this->type = type;
        this->source = source;
        this->destination = destination;
      }
  };

  void fillArrayWithData(char* data, int count) {
      for (int i = 0; i < count; ++i) {
        data[i] = i % 256;
      }
  }
}

int main()
{
  if (THREADS < 2)
```

```
  {
      std::cerr << "At least two threads are required to work.\n";
      return 0;
  }

  double maxTime = -1;

  #pragma omp parallel num_threads(THREADS)
  {
      const auto threadId = omp_get_thread_num();
      double elapsedTime = -1;

      srand(time(nullptr) + static_cast<time_t>(threadId) * 1000);

      if (threadId == 0) {
        wrapper::Message* message = new wrapper::Message;

        // Перенаправляем пакеты
        for (int i = 1; i <= (THREADS - 1) * 2; ++i) {
            recieveData(message, 1, sizeof(wrapper::Message));
            sendData(message,     1,     sizeof(wrapper::Message),     message-
>destination);
        }

        *message = { wrapper::Message::Type::Finish, 0, 0 };

        // Отправляем пакеты с запросом на завершение
        for (int i = 1; i < THREADS; ++i) {
            message->destination = i;
            sendData(message,     1,     sizeof(wrapper::Message),     message-
>destination);
        }

        delete message;
      }
      else {
        int destinationProcess = 1 + rand() % (THREADS - 1);

        wrapper::Message*      message      =      new      wrapper::Message{
wrapper::Message::Type::Data, threadId, destinationProcess};
        wrapper::fillArrayWithData(message->data, DATA_SIZE);

        double startTime = omp_get_wtime();

        // Отправляем пакет с данными случайному процессу
        sendData(message, 1, sizeof(wrapper::Message), 0);

        // Принимаем пакеты от 0 процесса и обрабатываем их
        do {
            recieveData(message, 1, sizeof(wrapper::Message), 0);

            if (message->type == wrapper::Message::Type::Data) {
              *message  =  {  wrapper::Message::Type::Confirmation,  threadId,
message->source };
              sendData(message, 1, sizeof(wrapper::Message), 0);
            }
        } while (message->type != wrapper::Message::Type::Finish);

        elapsedTime = omp_get_wtime() - startTime;

        #pragma omp critical
        {
            if(elapsedTime > maxTime)
```

```
            {
              maxTime = elapsedTime;
            }
        }

        delete message;
      }
  }

  printf("Elapsed time: %.7f", maxTime);

  return 0;
}
```

Результаты работы программы на различном количестве процессов.

Результаты работы программы, использующей технологию MPI, представлена в таблице 1.

Таблица 1 – Результаты работы программы на MPI.

№ п/п	Размер сообщения	Количество процессоров	Результаты работы программы
1.	100	1	-
2.		2	Elapsed time: 0.0003458
3.		4	Elapsed time: 0.0005523
4.		6	Elapsed time: 0.0010093
5.		8	Elapsed time: 0.0012355
6.		12	Elapsed time: 0.0016794
7.		16	Elapsed time: 0.0023344
8.	1000	1	-
9.		2	Elapsed time: 0.0003596
10.		4	Elapsed time: 0.0007777
11.		6	Elapsed time: 0.0008874
12.		8	Elapsed time: 0.0012430
13.		12	Elapsed time: 0.0019483
14.		16	Elapsed time: 0.0023739
15.	10000	1	-
16.		2	Elapsed time: 0.0004541
17.		4	Elapsed time: 0.0007068
18.		6	Elapsed time: 0.0009991
19.		8	Elapsed time: 0.0014346
20.		12	Elapsed time: 0.0020509
21.		16	Elapsed time: 0.0026533

Результаты работы программы, использующей технологию OpenMP, представлена в таблице 2.

Таблица 2 – Результаты работы программы на OpenMP.

№ п/п	Размер сообщения	Количество потоков	Результаты работы программы
1.	100	1	-
2.		2	Elapsed time: 0.0000282
3.		4	Elapsed time: 0.0000572
4.		6	Elapsed time: 0.0001125
5.		8	Elapsed time: 0.0001702
6.		12	Elapsed time: 0.0352466
7.		16	Elapsed time: 0.0606740
8.	1000	1	-
9.		2	Elapsed time: 0.0000079
10.		4	Elapsed time: 0.0001856
11.		6	Elapsed time: 0.0001366
12.		8	Elapsed time: 0.0002029
13.		12	Elapsed time: 0.0357934
14.		16	Elapsed time: 0.0650556
15.	10000	1	-
16.		2	Elapsed time: 0.0000454
17.		4	Elapsed time: 0.0001480
18.		6	Elapsed time: 0.0002931
19.		8	Elapsed time: 0.0002976
20.		12	Elapsed time: 0.0425025
21.		16	Elapsed time: 0.0800687

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

Графики зависимости времени отправки сообщения от числа процессов для разных размеров массива чисел и для разных технологий MPI и OpenMP представлен на рис. 2, рис. 3 и рис. 4.

За время отправки сообщения принимается максимальное время среди всех процессов между двумя событиями: начало отправки сообщения процессом и завершение приема сообщений от процесса 0 (после получения сообщения о завершении работы).

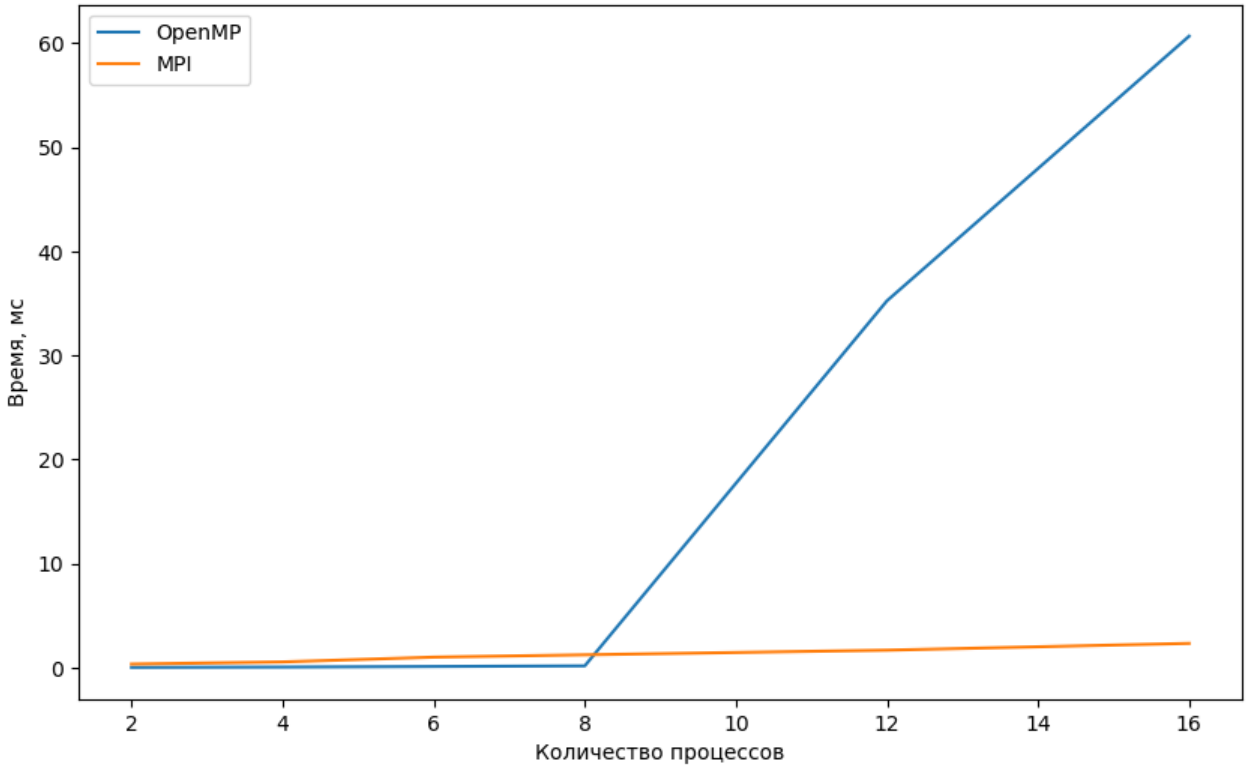

Рисунок 2 – Зависимость времени подсчета от числа процессов при длине массива 100.

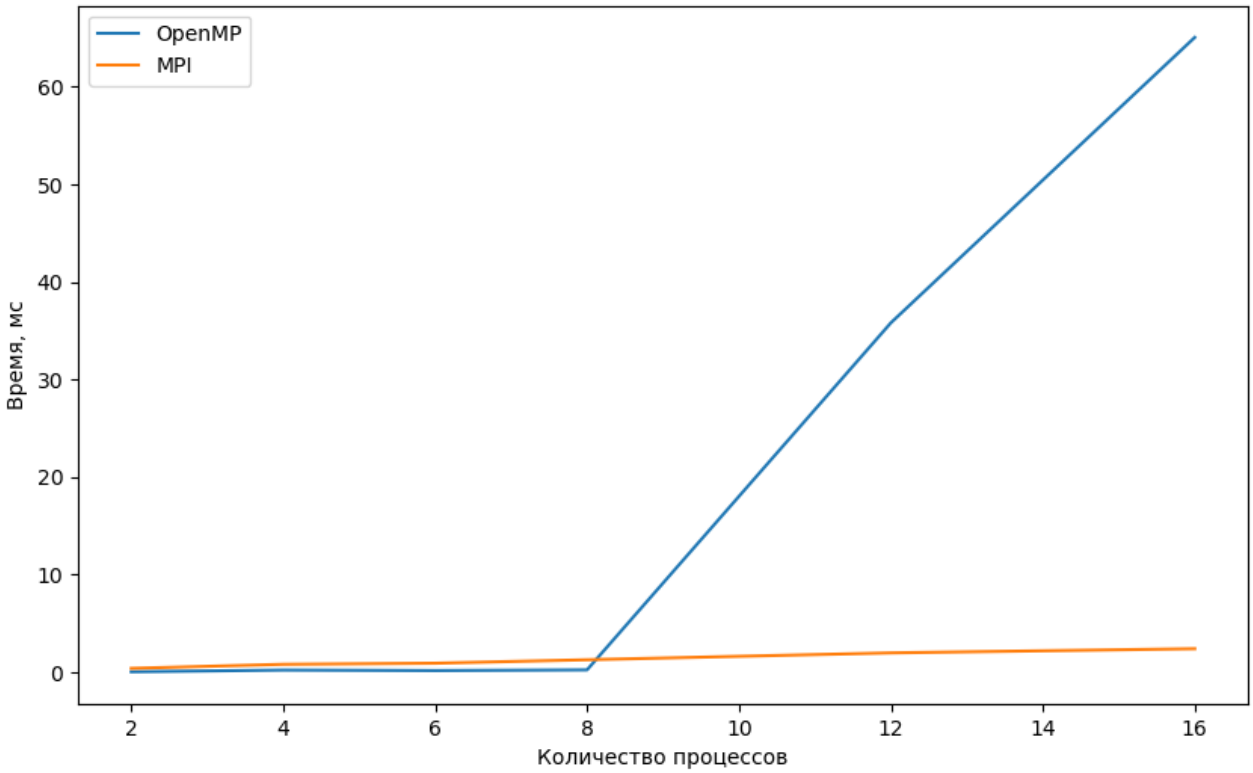

Рисунок 3 – Зависимость времени подсчета от числа процессов при длине массива 1000.

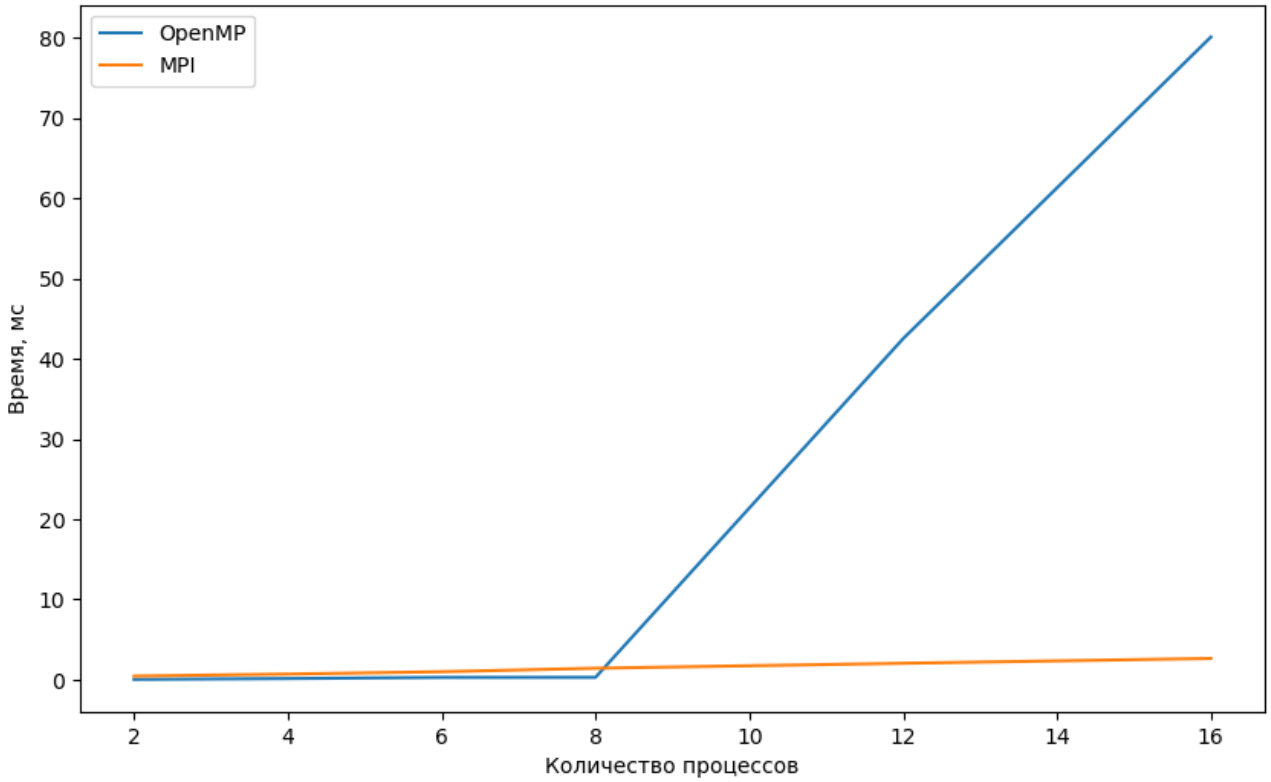

Рисунок 4 – Зависимость времени подсчета от числа процессов при длине массива 10000.

Очевидно, что при увеличении количества процессов время обмена сообщениями будет увеличиваться, поскольку трудоемкость самой задачи только возрастает из-за роста количества процессов (и сообщений, которые нужно отправить, соответственно). Это подтверждается графиками на рис. 2, рис. 3 и рис. 4.

Также можно заметить, что OpenMP на малом количестве процессов работает быстрее, чем MPI. Но при количестве процессов, примерно больших 8, время работы OpenMP значительно возрастает в сравнении с MPI. Можно предположить, что это связано с более оптимальной реализацией обмена сообщениями в библиотект MPI. В текущей реализации обмена сообщениями OpenMP такой эффект может быть следствием недостаточно оптимальной блокировки хранилища сообщений при большом количестве поступающих и извлекаемых сообщений (имеются ввиду сообщения, проходящие через процесс 0).

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 данной задачи строить не имеет смысла по следующим причинам:

- Данная задача предполагает наличие нескольких процессов, поэтому невозможно выполнить задачу на одном процессе для получения времени T_0, которое учувствует в вычислении значений ускорения;
- Очевидно, что увеличение количества процессов не даст ускорения работы, поскольку с их увеличением увеличивается также и количество сообщений, которые нужно отправить.

Выводы по работе.

Была написана программа, осуществляющая обмен сообщениями между процессами через центральный процесс 0. При получении сообщений процессы отправляют отправителю подтверждение.

Из полученных графиков видно, что при увеличении количества процессов время обмена сообщениями увеличивается, поскольку трудоемкость самой задачи только возрастает из-за роста количества процессов (и сообщений, которые нужно отправить, соответственно). Соответственно, для данной задачи увеличение количества процессов не дает какого-либо ускорения.

Из полученных графиков также видно, что OpenMP на малом количестве процессов работает быстрее, чем MPI. Но при количестве процессов, примерно больших 8, время работы OpenMP значительно возрастает в сравнении с MPI. Можно предположить, что это связано с более оптимальной реализацией обмена сообщениями в библиотект MPI. В текущей реализации обмена сообщениями OpenMP такой эффект может быть следствием недостаточно оптимальной

блокировки хранилища сообщений при большом количестве поступающих и извлекаемых сообщений (имеются ввиду сообщения, проходящие через процесс 0).

В контексте сравнения технологий MPI и OpenMP, технология OpenMP при малом количестве процессов (не больше 8) показала себя лучше, чем MPI. Но при большом количестве процессов (больше 8) MPI показала себя значительно лучше, чем OpenMP.