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

ОТЧЕТ

по лабораторной работе №4
по дисциплине «Системы параллельной обработки данных»
Тема: Группы процессов и коммутаторы

Студент гр. 9303 _____ Колованов Р.А.

Преподаватель _____ Татаринов Ю.С.

Санкт-Петербург
2023

Цель работы.

Написание программы, использующую группы процессов и коммутаторы с использованием библиотеки OpenMP. Сравнение двух технологий параллельного программирования MPI и OpenMP.

Формулировка задания.

Вариант 11.

В каждом процессе дано целое число N, которое может принимать два значения: 0 и 1 (имеется хотя бы один процесс с N = 1). Кроме того, в каждом процессе с N = 1 дано вещественное число A. Используя функцию MPI_Comm_split и одну коллективную операцию редукции, найти сумму всех исходных чисел A и вывести ее во всех процессах с N = 1.

Указание: при вызове функции MPI_Comm_split в процессах, которые не требуется включать в новый коммуникатор, в качестве параметра color следует указывать константу MPI_UNDEFINED.

Краткое описание алгоритма.

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

- Для начала в каждом процессе генерируется число isWorker, которое может быть равно 0 или 1. При этом у 0 процесса isWorker равен 1. Для каждого процесса задано вещественное число data, равное 1.0;
- Далее создается коммутатор workers, в который перемещаются процессы, у которых isWorker = 1;
- Далее для процессов коммутатора workers производится коллективная операция allReduceData, при помощи которой вычисляется сумма исходных чисел data во всех процессах с isWorker = 1.

Была реализована собственная система обмена сообщениями, поскольку в OpenMP не предусмотрена функциональность обмена сообщениями между процессами, в том числе не предусмотрены коллективные операции обмена.

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*.

Структура *Commutator* представляет собой хранилище потоков, которое используется для определения группы потоков, которые будут участвовать в коллективных операциях. С хранилищем можно выполнить одно действие: добавить в него поток. 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. Для коллективных операций обмена сообщениями были созданы функции *reduceData* и *allReduceData*. Все функции являются блокирующими.

Формальное описание алгоритма.

Формальное описание алгоритма на сетях Петри для программы представлено на рис. 1.

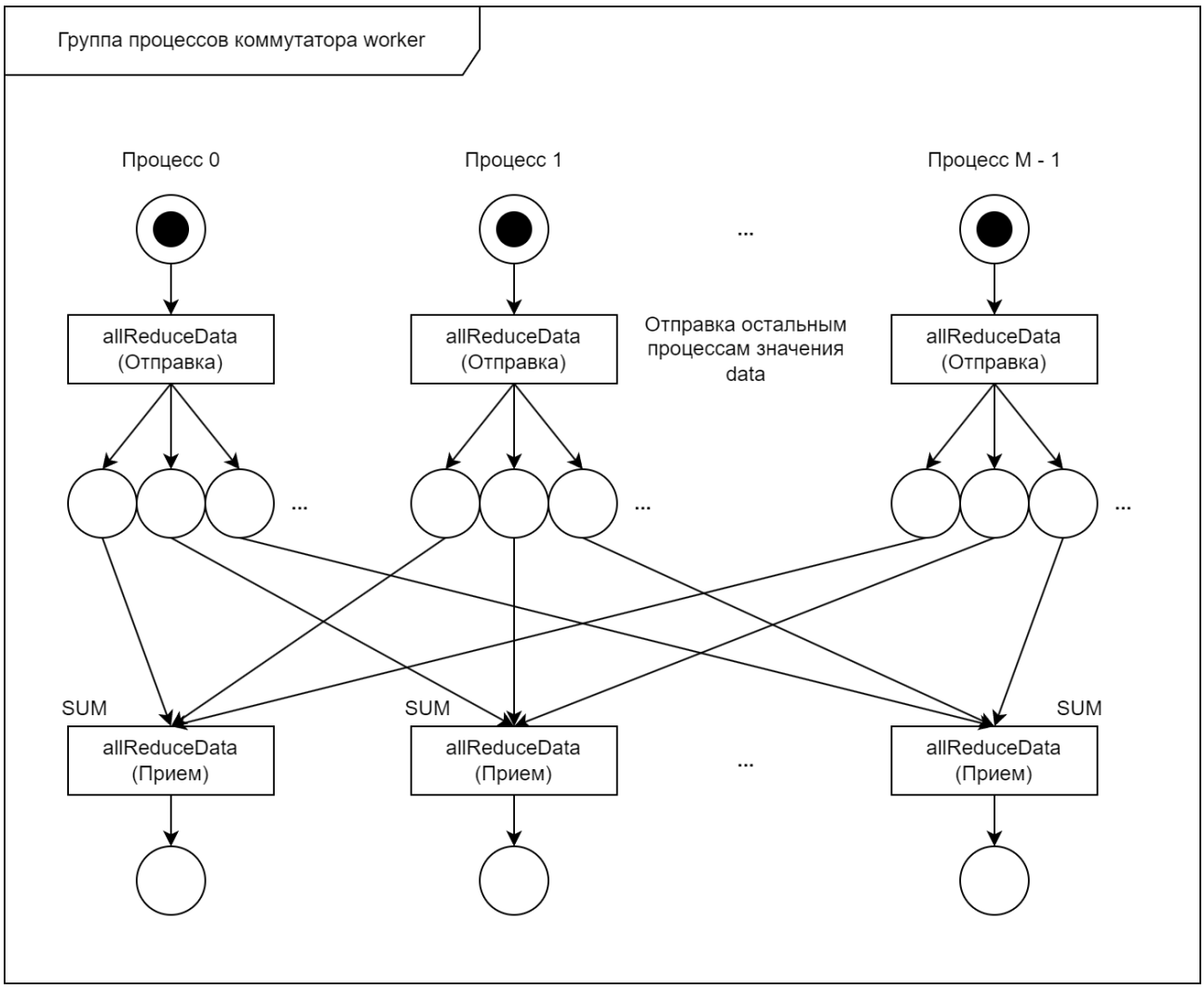

Рисунок 1 – Формальное описание алгоритма на сетях Петри.

Листинг программы.

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

Листинг 1. Программа на основе MPI.

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

int main(int argc, char** argv)
{
  int processNumber, processRank;
  MPI_Comm workers;

  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);

  const int isWorker = (processRank != 0) ? rand() % 2 : 1;
```

```
  double data = 1.0;
  double sum = 0.0;

  MPI_Comm_split(MPI_COMM_WORLD,  (isWorker)  ?  isWorker  :  MPI_UNDEFINED,
processRank, &workers);

  if (isWorker)
  {
      double startTime = MPI_Wtime();
      MPI_Allreduce(&data, &sum, 1, MPI_DOUBLE, MPI_SUM, workers);
      double elapsedTime = MPI_Wtime() - startTime;

      double maxTime;
      MPI_Reduce(&elapsedTime, &maxTime, 1, MPI_DOUBLE, MPI_MAX, 0, workers);

      if (processRank == 0)
      {
        std::cout << "Elapsed time: " << maxTime << "\n";
      }
  }

  MPI_Finalize();

  return 0;
}
```

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

Листинг 2. Программа на основе OpenMP.

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

/*!
 * \brief Тип коллективной операции.
 */
enum OperationType
{
  SUM = 0
};

/*!
 * \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;
  }

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

struct Commutator
{
  explicit Commutator() :
      threads{},
      storageLock{ nullptr }
  {
      omp_init_lock(&storageLock);
  }

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

  void addThread(int threadId)
  {
      omp_set_lock(&storageLock);
      threads.insert(threadId);
      omp_unset_lock(&storageLock);
  }

  std::set<int> threads;
  omp_lock_t storageLock;
};

namespace
{
  constexpr int THREADS = 20;
  int THREAD_GROUP_SIZE = 0;
  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
 */
void 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;
}

/*!
 * \brief Функция коллективного приема сообщений от других потоков и выполнения
операций над данными.
 *
 * Функция является блокирующей - освобождается после того, как сообщение будет
отправлено (для отправителей) или как все сообщения будут получены и над ними
будет выполнена операция (для получателя).
 *
```

```
 * \param sendBuffer Указатель на массив данных, которые нужно отправить
 * \param recvBuffer Указатель на массив данных, куда необходимо записать полученные данные
 * \param count Количество элементов в массиве данных
 * \param root ID потока, который принимает данные
 * \param operation Операция, осуществляемая над данными
 */
template<typename T>
Message* reduceData(void* sendBuffer, void* recvBuffer, int count, int root, const OperationType operation)
{
  const auto threadId = omp_get_thread_num();
  if (root == threadId)
  {
      T* resultBuffer = reinterpret_cast<T*>(recvBuffer);
      T* tempBuffer = new T[count];

      std::memcpy(recvBuffer, sendBuffer, count * sizeof(T));

      for (int i = 0; i < THREADS; ++i)
      {
        if (i == root)
            continue;

        recieveData(tempBuffer, count, sizeof(T), i);

        for (int j = 0; j < count; ++j)
        {
            if (operation == OperationType::SUM)
            {
              resultBuffer[j] += tempBuffer[j];
            }
        }
      }

      delete[] tempBuffer;
  }
  else
  {
      sendData(sendBuffer, count, sizeof(T), root);
  }
}

/*!
 * \brief Функция коллективного приема сообщений всеми потоками коммутатора и выполнения операций над данными.
 *
 * Функция является блокирующей - освобождается после того, как сообщение будет отправлено всеми процессами, после чего все сообщения будут получены и над ними будет выполнена операция.
 *
 * \param sendBuffer Указатель на массив данных, которые нужно отправить
 * \param recvBuffer Указатель на массив данных, куда необходимо записать полученные данные
 * \param count Количество элементов в массиве данных
 * \param operation Операция, осуществляемая над данными
 * \param commutator Коммутатор, процессы которого должны обмениваться сообщениями
 */
template<typename T>
Message* allReduceData(void* sendBuffer, void* recvBuffer, int count, const OperationType operation, const Commutator& commutator)
{
```

```
  const auto threads = commutator.threads;
  for (const auto& thread : threads)
  {
      sendData(sendBuffer, count, sizeof(T), thread);
  }

  std::memset(recvBuffer, 0, count * sizeof(T));

  T* resultBuffer = reinterpret_cast<T*>(recvBuffer);
  T* tempBuffer = new T[count];

  for (const auto& thread : threads)
  {
      recieveData(tempBuffer, count, sizeof(T), thread);

      for (int j = 0; j < count; ++j)
      {
        if (operation == OperationType::SUM)
        {
            resultBuffer[j] += tempBuffer[j];
        }
      }
  }

  delete[] tempBuffer;
}

int main1()
{
  double maxTime = 0.0;
  Commutator workers;

  #pragma omp parallel num_threads(THREADS)
  {
      const auto threadId = omp_get_thread_num();

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

      const int isWorker = (threadId != 0) ? rand() % 2 : 1;
      double data = 1.0;
      double sum = 0.0;

      if (isWorker)
      {
        workers.addThread(threadId);
      }
      #pragma omp barrier

      if (isWorker) {
        double startTime = omp_get_wtime();
        allReduceData<double>(&data, &sum, 1, OperationType::SUM, workers);
        double elapsedTime = omp_get_wtime() - startTime;

        #pragma omp critical
        {
            if (elapsedTime > maxTime)
            {
              maxTime = elapsedTime;
            }
        }
        #pragma omp barrier

        if (threadId == 0) {
```

```
            std::cout << "Elapsed time: " << maxTime << "\n";
        }
      }
  }

  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	Количество процессоров	Результаты работы программы
1.	1	Elapsed time: 7.2e-06
2.	2	Elapsed time: 7.3e-06
3.	4	Elapsed time: 0.0001252
4.	6	Elapsed time: 5.9e-05
5.	8	Elapsed time: 0.0001215
6.	10	Elapsed time: 0.000276
7.	12	Elapsed time: 0.0003326
8.	16	Elapsed time: 0.0003421
9.	20	Elapsed time: 0.0008312

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

Таблица 2 – Результаты работы программы на OpenMP.

№ п/п	Количество потоков	Результаты работы программы
1.	1	Elapsed time: 1.5e-06
2.	2	Elapsed time: 1.9e-06
3.	4	Elapsed time: 1.27e-05
4.	6	Elapsed time: 1.94e-05
5.	8	Elapsed time: 2.05e-05
6.	10	Elapsed time: 4.43e-05
7.	12	Elapsed time: 4.48e-05
8.	16	Elapsed time: 0.0004756
9.	20	Elapsed time: 0.002286

График зависимости времени выполнения программы от числа процессов.

Графики зависимости времени отправки сообщения от числа процессов для разных технологий MPI и OpenMP представлен на рис. 2.

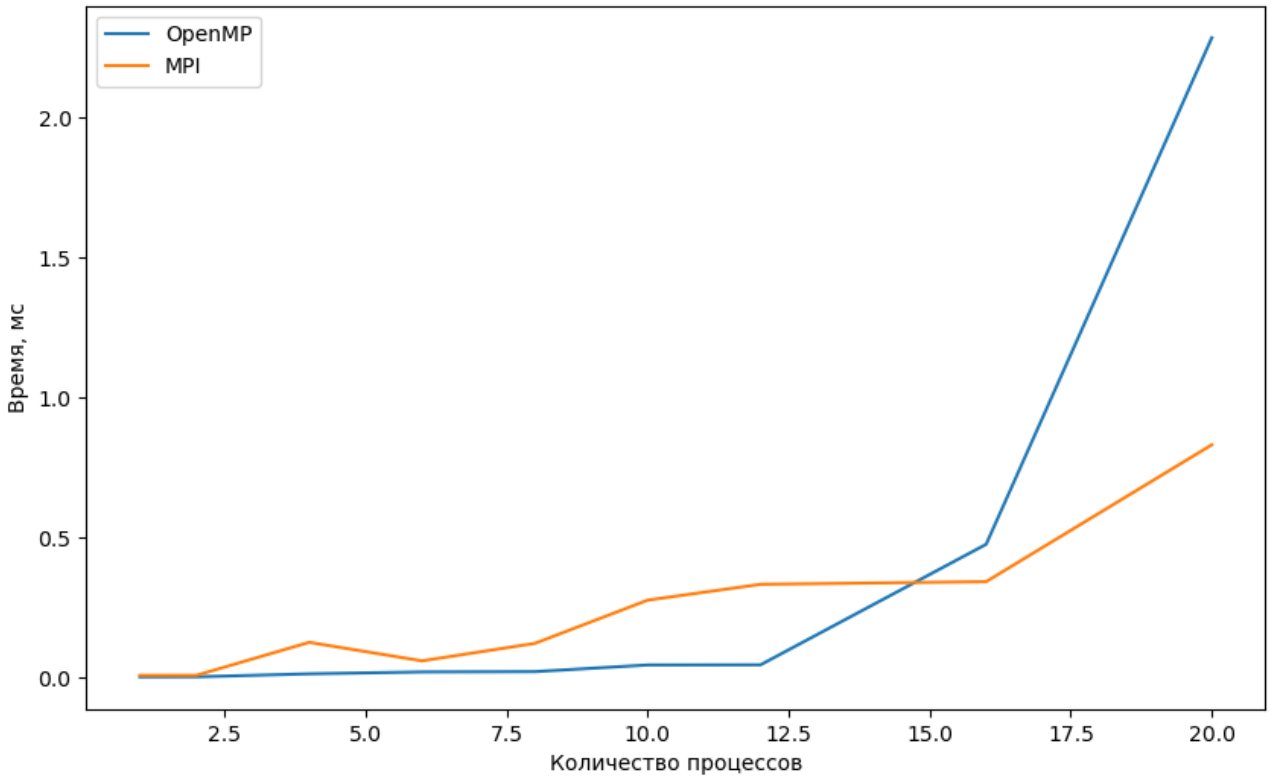

Рисунок 2 – Зависимость времени выполнения коллективной операции от числа процессов.

Очевидно, что при увеличении количества процессов время обмена сообщениями будет в общей тенденции увеличиваться, поскольку трудоемкость самой задачи возрастает из-за роста количества процессов и сообщений (и размера сообщений, соответственно). Это подтверждается графиком на рис. 2.

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.

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.

График ускорения для технологий MPI и OpenMP представлен на рис. 3.

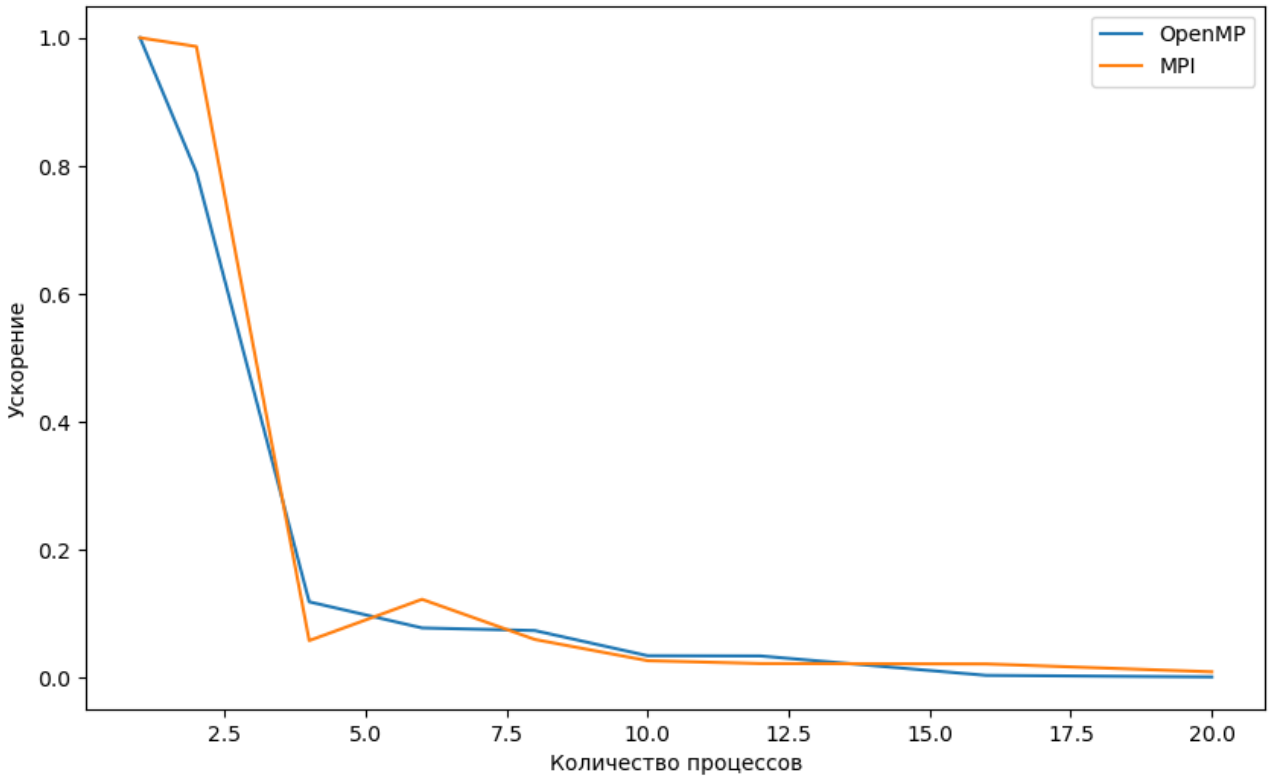

Рисунок 3 – График ускорения для технологий MPI и OpenMP.

Из графика ускорения на рис. 3 видно, что для данной задачи распараллеливание не дает какого-либо ускорения в случае использования MPI и OpenMP.

Выводы по работе.

Была написана программа, осуществляющая суммирование вещественных чисел, получаемых от каждого процесса, относящегося к коммутатору workers, и сохранения результата суммы в каждом процессе, относящегося к коммутатору workers. Было выполнено измерение времени работы с разным количеством процессов.

Из полученных графиков видно, что для технологий MPI и OpenMP при увеличении количества процессов время обмена сообщениями будет в общей тенденции увеличиваться, поскольку трудоемкость самой задачи возрастает из-за роста количества процессов и сообщений (и размера сообщений, соответственно).

По мимо этого, из графиков видно, что OpenMP в случае малого количества процессов работает быстрее, чем MPI. Это можно объяснить наличием общей памяти, которая ускоряет передачу данных между процессами. Но при большом количестве процессов технология MPI работает быстрее, чем OpenMP. Это можно объяснить недостаточной оптимизации кода (блокировки потоков).

В контексте сравнения технологий MPI и OpenMP, технология OpenMP при показала себя лучше при малом количестве процессов, а MPI – при достаточно большом количестве процессов (больше 16).