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

ОТЧЕТ

по лабораторной работе №3
по дисциплине «Системы параллельной обработки данных»
Тема: Коллективные операции

Студент гр. 9381 _____ Колованов Р.А.

Преподаватель _____ Татаринов Ю.С.

Санкт-Петербург
2023

Цель работы.

Написание программы, использующую коллективные функции обмена с использованием технологии OpenMP. Сравнение двух технологий параллельного программирования MPI и OpenMP.

Формулировка задания.

В каждом процессе дан набор из K + 5 целых чисел, где K — количество процессов. Используя функцию MPI_Reduce для операции MPI_SUM, просуммировать элементы данных наборов с одним и тем же порядковым номером и вывести полученные суммы в главном процессе.

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 массив из K + 5 случайных чисел, где K – количество процессов;
- Далее при помощи функции reduceData с операцией суммирования происходит отправка сгенерированного массива чисел от каждого процесса процессу 0 с последующим суммированием элементов массива с одинаковым индексом;
- После получения данных процесс 0 печатает N + 5 чисел, полученных в результате суммирования элементов массивов с одинаковым индексом.

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*.

Для отправки и приема сообщений были созданы две функции *sendData* и *receiveData* соответственно. Для коллективной операции обмена сообщениями была создана еще одна функция – *reduceData*. Все функции являются блокирующими.

Формальное описание алгоритма.

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

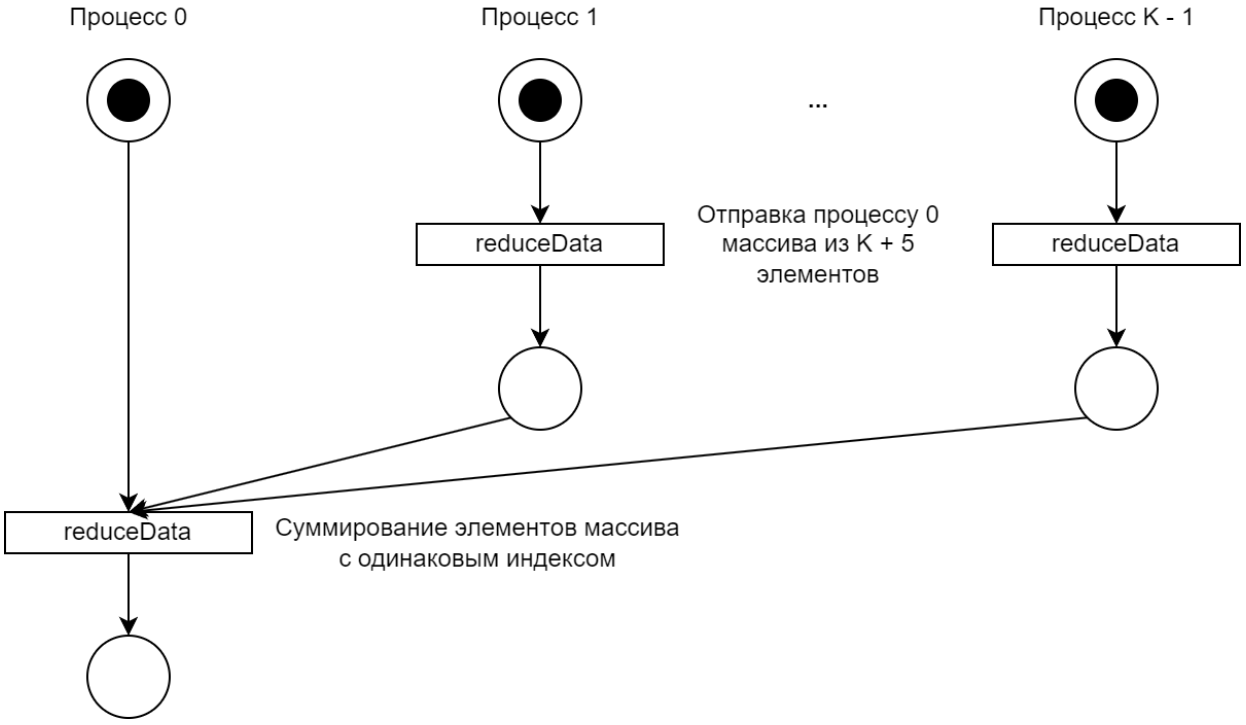

Рисунок 1 – Формальное описание алгоритма на сетях Петри.

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

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

  int* sendBuffer = new int[processNumber + 5];
  int* receiveBuffer = new int[processNumber + 5];

  for (int i = 0; i < processNumber + 5; ++i) {
      sendBuffer[i] = rand() % 11;
  }

  double startTime = MPI_Wtime();

  MPI_Reduce(sendBuffer, receiveBuffer, processNumber + 5, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD);

  double elapsedTime = MPI_Wtime() - startTime;

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

  delete[] sendBuffer;
  delete[] receiveBuffer;

  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 <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;
  }

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

namespace
{
  constexpr int THREADS = 1;
  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);
  }
}

int main()
{
  const int arraySize = THREADS + 5;

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

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

      int* sendBuffer = new int[arraySize];
      int* receiveBuffer = new int[arraySize];

      for (int i = 0; i < arraySize; ++i) {
        sendBuffer[i] = rand() % 11;
      }

      double startTime = omp_get_wtime();

      reduceData<int>(sendBuffer,      receiveBuffer,      arraySize,      0,
OperationType::SUM);
      #pragma omp barrier

      double elapsedTime = omp_get_wtime() - startTime;

      if (threadId == 0) {
        printf("Elapsed time: %.7f", elapsedTime);
```

```
      }

      delete[] sendBuffer;
      delete[] receiveBuffer;
  }

  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: 6.19999e-06
2.	2	Elapsed time: 0.00021
3.	4	Elapsed time: 0.0005363
4.	6	Elapsed time: 0.0005572
5.	8	Elapsed time: 0.0007379
6.	10	Elapsed time: 0.0008233
7.	12	Elapsed time: 0.0009077
8.	16	Elapsed time: 0.0011885
9.	20	Elapsed time: 0.0014865

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: 0.0000017
2.	2	Elapsed time: 0.0000460
3.	4	Elapsed time: 0.0000874
4.	6	Elapsed time: 0.0001338
5.	8	Elapsed time: 0.0002058
6.	10	Elapsed time: 0.0002439
7.	12	Elapsed time: 0.0002370
8.	16	Elapsed time: 0.0002694
9.	20	Elapsed time: 0.0002888

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.

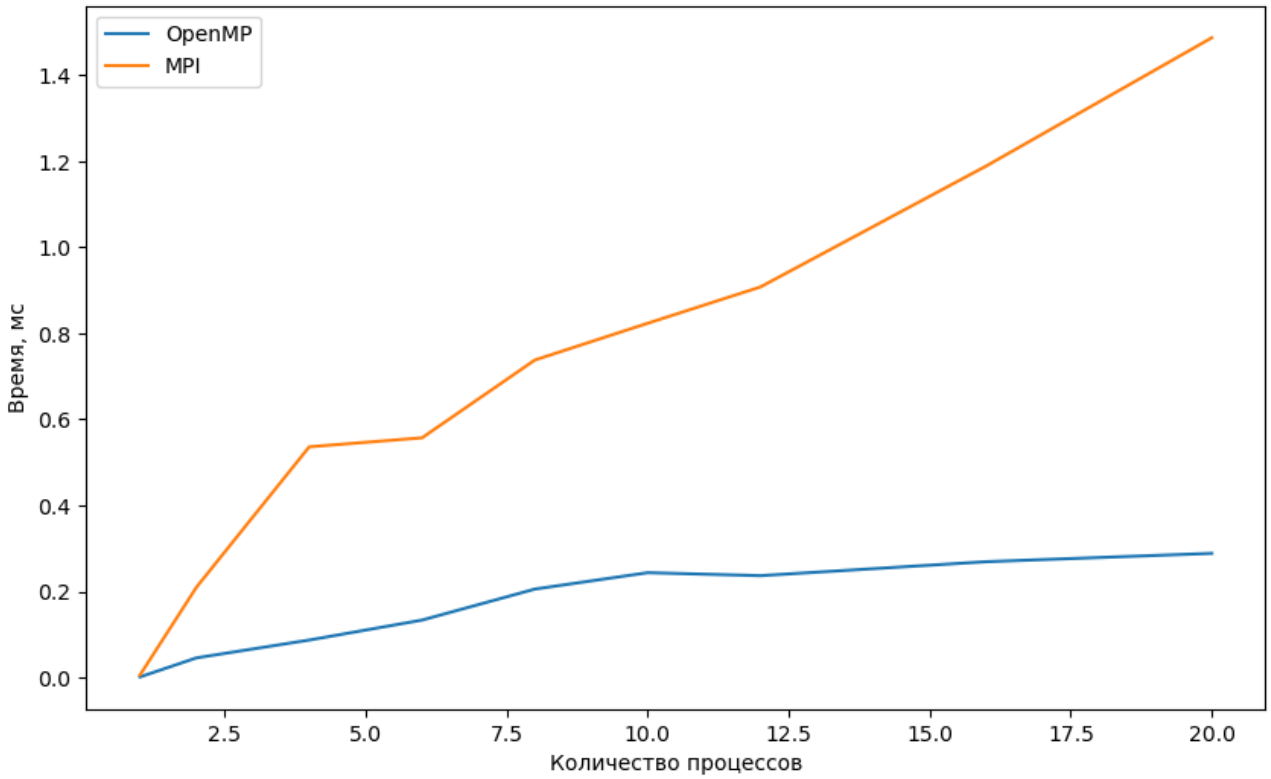

Рисунок 2 – Зависимость времени выполнения коллективной операции от числа процессов.

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 работает быстрее, чем MPI. Это можно объяснить наличием общей памяти, которая ускоряет передачу данных между процессами.

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.

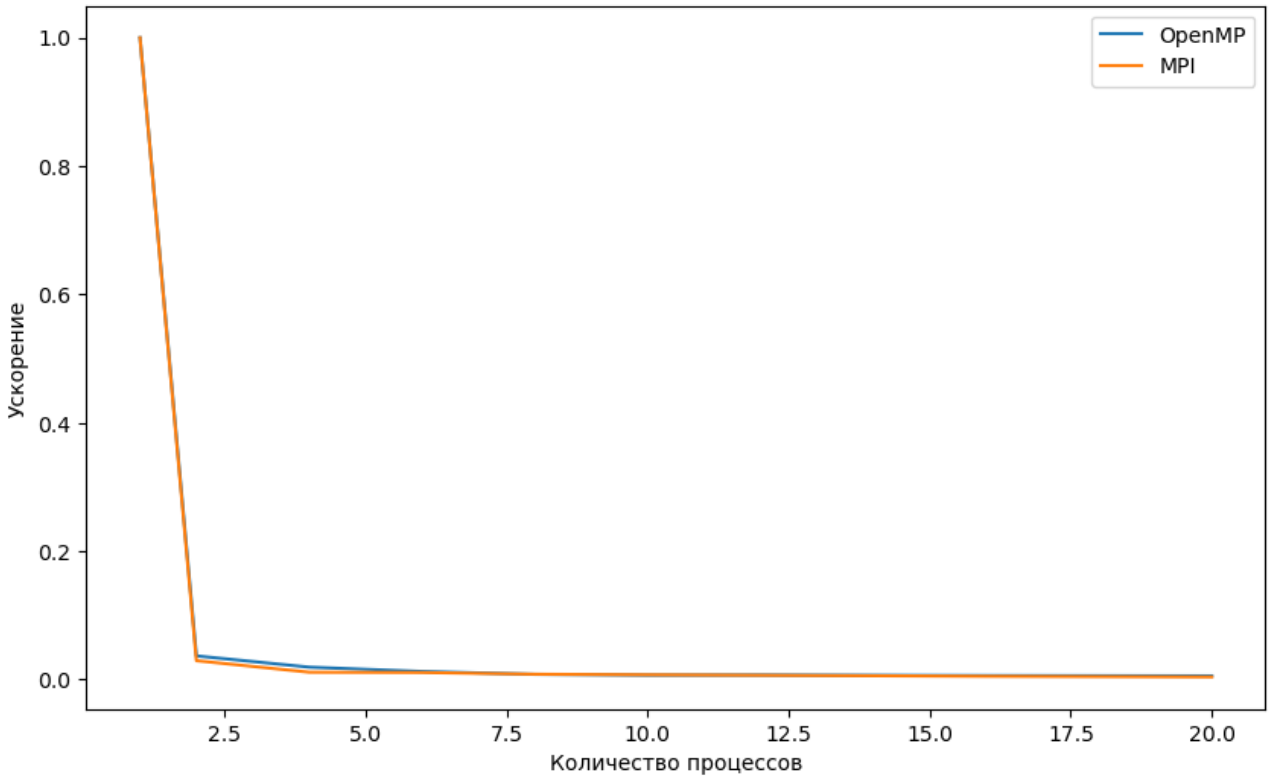

Рисунок 3 – График ускорения для технологий MPI и OpenMP.

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 элементов с одинаковыми индексами в массивах случайных чисел, генерируемых каждым процессом. Было выполнено измерение времени работы с разным количеством процессов.

Из полученных графиков видно, что для технологии 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 во всех рассматриваемых случаях работает быстрее, чем MPI. Это можно объяснить наличием общей памяти, которая ускоряет передачу данных между процессами.

В контексте сравнения технологий MPI и OpenMP, технология OpenMP при показала себя лучше во всех случаях.