МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Государственное образовательное учреждение высшего профессионального образования

«ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ» (ТУСУР)

УТВЕРЖДАЮ

Заведующий кафедрой ЭМИС

_____ И. Г. Боровской

«___» _____ 2014 г.

Е.А. ШЕЛЬМИНА

ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ

Методические указания по выполнению лабораторных работ и самостоятельной работы для студентов 230400.62 «Информационные системы и технологии»

2014

Шельмина Е.А. Параллельное программирование: методические указания по выполнению лабораторных работ и самостоятельной работы для студентов 230400.62 «Информационные системы и технологии» – Томск: Изд-во ТУСУР, 2014. – 22 с.

Пособие посвящено реализации учебно-методической поддержке дисциплины «Параллельное программирование». Данное пособие содержит краткое введение в архитектуру параллельных вычислительных систем, введение в проблематику параллельного программирования для параллельных систем различных классов, сведения о технологии OpenMP; сведения об архитектуре NVIDIA CUDA, сведения об интерфейсе передачи сообщений MPI и методические указания по всем названным темам.

СОДЕРЖАНИЕ ЛАБОРАТОРНЫХ РАБОТ И САМОСТОЯТЕЛЬНОЙ РАБОТЫ
по дисциплине «Параллельное программирование» для студентов 230400.62 «Информационные системы и технологии»

Краткое содержание тем и результатов их освоения

Темы лабораторных занятий	Деятельность студента. Решая задачи, студент:
Моделирование и анализ параллельных алгоритмов	● *изучает принципы построения* параллельных алгоритмов; ● *учиться* оценивать эффективность параллельных алгоритмов;
Этапы разработки параллельных алгоритмов	● *изучает* основные этапы разработки параллельного алгоритма; ● *применяет* полученные знания при решении задач;
Средства разработки параллельных программ	● *изучает* параллельные языки программирования и расширения стандартных языков; ● *получает опыт* работы со средствами автоматического распараллеливания, параллельными компиляторами;
Интерфейс передачи сообщений - MPI	● *изучает* общие принципы построения и реализации MPI;
Технология программирования OpenMP	● *изучает* директивы OpenMP, классы переменных; ● *применяет* функции и переменные окружения для выполнения параллельных программ;
Программирование МВС с графическими процессорами	● *учиться* применять графические процессоры в параллельном программировании;
Типовые параллельные алгоритмы	● *реализует* типовые параллельные алгоритмы (умножение матриц: линейное, блочное разбиение матриц, алгоритмы Фокса и Кеннона, решение систем линейных уравнений, алгоритмы параллельной сортировки);

ХОД ЛАБОРАТОРНЫХ РАБОТ

1. Ознакомиться со справочными интернет-сведениями (СРС)
2. Ознакомиться с указанной темой в основной и дополнительной литературе.

1. Катаев М.Ю. Параллельное программирование [Электронный ресурс]: методические рекомендации к практическим занятиям / М. Ю. Катаев ; Томский государственный университет систем управления и радиоэлектроники (Томск). - Электрон. текстовые дан. - Томск : [б. и.], 2012. - on-line, 8 с. - http://edu.tusur.ru/training/publications/570.

Дополнительная литература

1. Параллельные вычисления на GPU. Архитектура и программная модель CUDA: учебное пособие для вузов / А. В. Боресков [и др.] ; авт. предисл. В. А. Садовничий ; Московский государственный университет им. М. В. Ломоносова (М.). - М. : Издательство Московского университета, 2012. - 336 с. : цв.ил. - (Суперкомпьютерное образование). - Библиогр.: с. 297-300. -ISBN 978-5-211-06340-2 : 5 экз.

3. Ознакомиться с принципом выполнения лабораторных работ.
4. Составить и предоставить преподавателю отчет о работе, если он входит в форму отчетности по данному разделу знаний.

Лабораторные работы

Моделирование и анализ параллельных алгоритмов (2 часа)

Цель работы: изучение моделей, методов и технологий параллельного программировании.

Теоретические сведения

При разработке параллельных алгоритмов решения сложных научно-технических задач принципиальным моментом является анализ эффективности использования параллелизма, состоящий обычно в оценке получаемого ускорения процесса вычислений (сокращения времени решения задачи).

Формирование подобных оценок ускорения может осуществляться применительно к выбранному вычислительному алгоритму (оценка эффективности распараллеливания конкретного алгоритма).

Другой важный подход может состоять в построении оценок максимально возможного ускорения процесса решения задачи конкретного типа (оценка эффективности параллельного способа решения задачи).

Более подробно теоретические сведения по данной теме приведены в основной и дополнительной литературе и в следующих источниках:

1. Гергель В.П. Высокопроизводительные вычисления для многопроцессорных многоядерных систем: учебник для вузов / В. П. Гергель ; Библиотека Нижегородского государственного университета (Нижний Новгород). - М. : Издательство Московского университета, 2010. - 544 с. : ил., табл. - (Суперкомпьютерное образование). - Библиогр.: с. 534-539. - ISBN 978-5-211-05937-5. -ISBN 978-5-9221-1312-0 : 26 экз.

Задание

1. Разработайте модель и выполните оценку показателей ускорения и эффективности параллельных вычислений:
    - для задачи скалярного произведения двух векторов:

$$y = \sum_{i=1}^{N} a_i b_i \ ,$$

- для задачи поиска максимального и минимального значений для заданного набора числовых данных

$$y_{\min} = \min_{1 \le i \le N} a_i ,$$

$$y_{\max} = \max_{1 \le i \le N} a_i$$

- для задачи нахождения среднего значения для заданного набора числовых данных

$$y = \frac{1}{N} \sum_{i=1}^{N} a_i .$$

Этапы разработки параллельных алгоритмов (2 часа)

Цель работы: разработка параллельной программы, которая выполняет умножение двух квадратных матриц.

Теоретические сведения

Более подробное теоретическое обоснование данной темы представлено в следующих источниках:

1. Инструменты параллельного программирования в системах с общей памятью: учебник для вузов / К. В. Корняков [и др.] ; ред. В. П. Гергель ; Нижегородский государственный университет (Нижний Новгород). - 2-е изд., испр. и доп. - М. : Издательство Московского университета, 2010. - 271 с. : ил., табл. - (Суперкомпьютерное образование). - Библиогр. в конце разд. -ISBN 978-5-211-05931-3 (26 экз.)
2. Гергель В.П. Высокопроизводительные вычисления для многопроцессорных многоядерных систем: учебник для вузов / В. П. Гергель ; Библиотека Нижегородского государственного университета (Нижний Новгород). - М. : Издательство Московского университета, 2010. - 544 с. : ил., табл. - (Суперкомпьютерное образование). - Библиогр.: с. 534-539. - ISBN 978-5-211-05937-5. -ISBN 978-5-9221-1312-0 (26 экз.)

Задание

Определить задачу матричного умножения. Реализовать последовательный алгоритм матричного умножения. Разработать параллельный алгоритм умножения матриц.

Средства разработки параллельных программ (6 часов)

Цель работы: изучение параллельных языков программирования и расширения стандартных языков. Получение навыков работы со средствами автоматического распараллеливания, параллельными компиляторами.

Теоретические сведения

В настоящей лабораторной работе рассматривается один из инструментов параллельного программирования, предназначенный для распараллеливания решения задач в системах с общей памятью, – библиотека Intel Threading Building Blocks (TBB).

Основная идея, заложенная в библиотеку, состоит в использовании стандартного высокоуровневого C++ для быстрой разработки кросс-платформенных, хорошо масштабируемых параллельных приложений. Наряду с указанным выше, использование библиотеки TBB предоставляет механизмы абстрагирования от парадигм многопоточного программирования, позволяя сосредоточиться непосредственно на решении прикладной задачи, что является достаточно актуальным.

Изучение принципов функционирования и вопросов эффективного использования TBB проводится в данной лабораторной работе на примере задачи матрично-векторного умножения, ставшей одной из классических задач для формирования навыков параллельного программирования. При этом основной упор делается на ознакомление с распараллеливанием циклов.

Задание

Реализовать параллельный алгоритм матрично-векторного умножения на основе имеющейся реализации последовательного алгоритма. Изменить созданную реализацию на случай неквадратной матрицы. Разработать программу умножения квадратных матриц. Разработать программу умножения прямоугольных матриц.

Интерфейс передачи сообщений – MPI (6 часов)

Цель работы: изучение и разработка интерфейса передачи сообщений для параллельных приложений.

Теоретические сведения

MPI - это стандарт на программный инструментарий для обеспечения связи между ветвями параллельного приложения. MPI расшифровывается как "Message passing interface" ("Взаимодействие через передачу сообщений").

MPI представляет собой библиотеку программирования (заголовочные и библиотечные файлы для языков Си, Си++). Набор функций прост в освоении и позволяет быстро написать надежно работающую программу. Использование всей мощи MPI позволит быстро получить работающую программу при сохранении ее надежности.

Параллельное приложение состоит из нескольких ветвей (процессов или задач), выполняющихся одновременно. Ветви обмениваются друг с другом данными в виде сообщений. Сообщения проходят под идентификаторами, которые позволяют программе и библиотеке связи отличать их друг от друга. Для совместного проведения тех или иных расчетов процессы внутри приложения объединяются в группы. Каждый процесс может узнать у библиотеки связи свой номер внутри группы, и, в зависимости от номера приступает к выполнению соответствующей части расчетов.

Особенность MPI - понятие области связи (communication domains). При запуске приложения все процессы помещаются в создаваемую для приложения общую область связи. При необходимости они могут создавать новые области связи на базе существующих. Все области связи имеют независимую друг от друга нумерацию процессов. Программе пользователя в распоряжение предоставляется коммуникатор - описатель области связи. В программе может существовать несколько коммуникаторов.

Многие функции MPI имеют среди входных аргументов коммуникатор, который ограничивает сферу их действия той областью связи, к которой он прикреплен. MPI_COMM_WORLD - название коммуникатора, создаваемого библиотекой автоматически при инициализации библиотеки MPI параллельным приложением. Он описывает стартовую область связи, объединяющую все процессы приложения.

Все функции MPI условно можно разбить на блокирующие, неблокирующие, локальные и коллективные.

Блокирующие функции останавливают (блокируют) выполнение процесса до тех пор, пока производимая ими операция не будет выполнена. Не блокирующие функции возвращают управление немедленно, а выполнение операции продолжается в фоновом режиме. Локальные функции не инициируют пересылок данных между ветвями. Коллективные функции должны быть вызваны всеми ветвями-абонентами того коммуникатора, который передается им в качестве аргумента.

При запуске приложения все его ветви должны вызвать функцию инициализации библиотеки MPI - MPI_Init.

При завершении приложения все его ветви должны выполнить функцию нормального завершения библиотеки - MPI_Finalize.

Если приложение завершается по причине ошибок времени выполнения, то ветвь, обнаружившая ошибку, должна выполнить функцию MPI_Abort.

При запуске приложения автоматически создается коммуникатор MPI_COMM_WORLD, объединяющий все процессы приложения.

Любой процесс может узнать количество ветвей приложения с помощью функции MPI_size и свой номер в приложении с помощью функции MPI_rank, объединенных этим коммуникатором.

Простейшая схема передачи сообщений - связь "точка-точка" между двумя ветвями. Одна ветвь вызывает функцию передачи сообщения, а другая - функцию приема. В качестве основных параметров функциям задаются буфер приема/передачи, коммуникатор, имя (номер) процесса – источника/адресата сообщения или джокер MPI_ANY_SOURCE, идентификатор сообщения или джокер MPI_ANY_TAG, тип данных. Тип данных задается предопределенными в MPI описателями встроенных типов(MPI_CHAR, MPI_INT, …) или зарегистрированными пользователем описателями типов.

Функции передачи–приема сообщения по схеме «точка – точка»: MPI_Send , MPI_Recv, MPI_Sendrecv.

Для обслуживания приема используются вспомогательные функции:

- MPI_Status – запрос статуса завершения приема,
- MPI_Get_count - запрос количества фактически принятых данных,
- MPI_Probe – предварительный запрос статуса поступившего сообщения перед его приемом.

Коллективные обмены сообщениями осуществляются группой ветвей, объединенных некоторым коммуникатором. Коллективный обмен инициируется в том случае, когда все ветви осуществили вызов одноименной коллективной функции с одинаковыми основными параметрами. Коллективная функция выполняет одновременно и прием, и передачу и имеет большое количество параметров, часть которых нужна для приема, а часть для передачи. В разных ветвях та или иная часть параметров игнорируется, в зависимости от роли ветви в обмене.

Коллективные передачи ведутся в соответствии с одной из схем обмена: «одна – всем», «все – одной», «все – всем».

Коллективные функции обмена:

- MPI_Bcast – передача сообщения из ветви с указанным номером всем остальным ветвям.
- MPI_Gather, MPI_Gatherv - сбор сообщений в указанную ветвь из всех остальных ветвей.
- MPI_Scatter, MPI_Scatterv – передача с распределением сообщения из ветви с указанным номером всем остальным ветвям.
- MPI_Allgather, MPI_Allgatherv - сбор сообщений во всех ветвях из всех ветвей.
- MPI_Alltoall, MPI_Alltoallv - передача с распределением сообщения из всех ветвей всем остальным ветвям.

Функция MPI_Barrier останавливает выполнение вызвавшей ее задачи до тех пор, пока не будет вызвана изо всех остальных задач, подсоединенных к указанному коммуникатору.

Для настройки работы интерфейса MPI с производными типами пользователя используются следующие функции:

MPI_Type_struct , MPI_Type_vector , MPI_Type_indexed, MPI_Type_contiguous - создание описателя пользовательского типа.

MPI_Type_commit - регистрация описателя типа.

MPI_Type_extent - запрос объема памяти, занимаемой переменной производного типа.

MPI_Type_size - запрос минимального («ужатого») объема сообщения с переменной производного типа при передаче ее в сообщении.

MPI_Type_free - сброс описателя пользовательского типа.

Функции выполняют поэлементно заданную операцию над элементами распределенных массивов. Интерфейс MPI содержит 4 распределенные функции:

- MPI_Reduce - массив с результатами размещается в выделенной ветви,
- MPI_Allreduce - массив с результатами размещается во всех ветвях,
- MPI_Reduce_scatter - массив с результатами распределяется между ветвями,
- MPI_Scan : аналогична функции MPI_Allreduce. Но содержимое

массива-результата в задаче i является результатом выполнение операции над массивами из задач с номерами от 0 до i включительно.

В MPI предопределены 12 описателей поэлементных операций:

- MPI_MAX и MPI_MIN –поиск поэлементных максимума и минимума,
- MPI_SUM - сумма векторов;
- MPI_PROD - поэлементное произведение векторов;
- *MPI_LAND, MPI_BAND, MPI_LOR, MPI_BOR, MPI_LXOR, MPI_BXOR - логические и двоичные операции И, ИЛИ, исключающее ИЛИ,*
- MPI_MAXLOC, MPI_MINLOC – поиск

Группа - это некое множество ветвей. Одна ветвь может быть членом нескольких групп.

Область связи ("communication domain") - это абстрактное понятие, связанное с коммуникатором. Области связи автоматически создаются и уничтожаются вместе с коммуникаторами.

Одной области связи могут соответствовать несколько коммуникаторов. Коммуникаторы функционируют независимо и параллельно во времени.

Функции создания .уничтожения групп:

- MPI_Group_xxx - создает новую группу с нужным набором ветвей на базе существующей группы,
- MPI_Comm_group создает группу, на которую указывает соответствующий коммуникатор,
- *MPI_Group_free – уничтожает группу.*

Функции создания коммуникаторов :

- MPI_Comm_dup – копирует один коммуникатор в другой,
- MPI_Comm_split – расщепляет группу существующего коммуникатора на непересекающиеся подгруппы,
- MPI_Comm_create создает коммуникатор для группы.
- MPI_Group_free.

Задание

1. Изучить описание MPI – интерфейса.

2. Разработать библиотеку функций MPI-интерфейса для языка программирования C/C++ .

В библиотеку в обязательном порядке должны быть включены функции MPI_Init(), MPI_Size(), MPI_Rank(). Должна быть реализована минимум одна функция коллективного обмена.

3. Провести комплексную отладку функций библиотеки команд NETBIOS и библиотеки MPI.

4. Провести тестирование надежности и пропускной способности функций библиотеки MPI.

Технология программирования OpenMP (6 часов)

Цель работы: изучить технологию OpenMP и основы разработки параллельных программ для многоядерных процессоров, написать и отладить на языке С параллельную программу для решения поставленной задачи на системе с общей памятью.

Теоретическое обоснование работы

Технология OpenMP в настоящее время является одним из наиболее популярных средств параллельного программирования для компьютеров с общей памятью, базирующейся на традиционных последовательных языках программирования и использовании специальных комментариев. За основу берётся последовательная программа, а для создания её параллельной версии пользователю предоставляется набор директив, функций и переменных окружения. Технология OpenMP нацелена на то, чтобы пользователь имел один и тот же вариант программы как для параллельного, так и для последовательного режима выполнения. Параллелизм в OpenMP реализуется с помощью многопоточности. При запуске программы создается единственный «главный» (master) поток, который затем создает набор «подчиненных» (slave) потоков, и вычисления распределяются между всеми потоками. Предполагается, что потоки выполняются параллельно на машине с несколькими процессорами (ядрами), причём количество процессоров/ядер не обязательно должно быть больше или равно количеству потоков. Другими словами, потоки параллельной программы могут обычным образом конкурировать между собой за время процессора/ядра.

Важным достоинством технологии OpenMP является возможность реализации так называемого инкрементального программирования, когда программист постепенно находит участки в программе, содержащие ресурс параллелизма, с помощью предоставляемых механизмов делает их параллельными, а затем переходит к анализу следующих участков. Таким образом, распараллеленные участки постепенно охватывают всё большую часть программы. Этот подход значительно облегчает процесс адаптации последовательных программ к параллельным компьютерам, а также отладку и оптимизацию программ.

Модель исполнения параллельной программы, подготовленной с помощью технологии OpenMP, можно сформулировать следующим образом:

- Программа содержит набор последовательных и параллельных областей (или секций или регионов).
- В начальный момент времени создается главный поток, выполняющий впоследствии все последовательные области программы.
- При входе в параллельную область главным потоком выполняется операция *fork,* порождающая совокупность подчиненных потоков. Каждый поток имеет свой уникальный числовой идентификатор (главному потоку соответствует 0). При распараллеливании циклов все параллельные потоки исполняют один и тот же код, но с разными данными. В общем случае потоки могут исполнять различные фрагменты кода.

– При выходе из параллельной области всеми потоками выполняется операция *join.* Завершается выполнение всех потоков, кроме главного.

Обычно допускается возможность образования вложенных параллельных областей, когда поток, созданный как дополнительный, становится master-потоком для новой параллельной области.

Под OpenMP понимается совокупность следующих компонент:

– *Директивы компилятора* – используются для создания потоков, распределения работы между потоками и их синхронизации. Директивы включаются программистом в исходный текст программы.
– *Подпрограммы библиотеки времени выполнения* – используются для установки и определения атрибутов потоков. Вызовы этих подпрограмм включаются программистом в исходный текст.
– *Переменные окружения* – используются для управления поведением параллельной программы. Переменные окружения задаются для среды выполнения параллельной программы соответствующими средствами операционной системы.

Использование директив компилятора и подпрограмм библиотеки времени выполнения подчиняется правилам, которые различаются для разных языков программирования. Совокупность таких правил для одного языка программирования называется *привязкой к языку.* Технология OpenMP создана и поддерживается для языков C, C++ и Fortran. В этой лабораторной работе предполагается использование языка C/C++ в среде разработки Microsoft Visual Studio (начиная с версии 2005). Для того, чтобы ее компилятор нужным образом реагировал на директивы OpenMP и строил параллельную программу, в командную строку его запуска должна быть включена опция /openmp (например, путем включения флажка «OpenMP support» в свойствах проекта на закладке Configuration properties\C/C++\Language).

Для того, чтобы в программе на языке C/C++ стали доступны возможности технологии OpenMP, в нее нужно включить заголовочный файл omp.h:

#include <omp.h>

Если эту программу транслировать компилятором, не поддерживающим технологию OpenMP, то она будет построена в обычном последовательном (однопоточном) варианте, поскольку все директивы, задающие распараллеливание, оформляются в виде так называемых прагм (рекомендаций), и просто игнорируются такими компиляторами.

В программах на языке C/C++ все прагмы, имена функций и переменных окружения OpenMP начинаются со строки «omp». Формат директивы:

#pragma omp директива [опция_1[, опция_2, ...]]

Каждая директива вместе со всеми ее опциями обязательно должна занимать ровно одну строку текста. Действие некоторых директив распространяется только на один оператор (или блок операторов, заключенных в фигурные скобки), непосредственно следующий за директивой в тексте программы. Для таких директив будет указан *<структурный блок кода>*.

Перечень **директив OpenMP** включает в себя:

1. Директива задания параллельно выполняемой секции:

#pragma omp parallel [опция_1[, опция_2, ...]]
<структурный блок кода>

С помощью опций этой директивы можно указать:

– требуемое количество потоков n (*num_threads(n)*);
– условие, при котором параллельная область действительно создается (*if(условие)*);
– список общих переменных для всех потоков данной секции (*shared(список переменных)*);

– список переменных, которые будут локальными в каждом потоке секции (*private(список переменных)*), причем их начальные значения не будут определены;
– список переменных, которые будут локальными в каждом потоке секции (*firstprivate(список переменных)*), причем в качестве их начальных значений будут установлены значения одноименных переменных из главного потока;
– способ назначения класса памяти *default(shared|none)* всем переменным потоков, которым класс не назначен явно с помощью опции *shared* (слово *none* означает, что класс памяти всех локальных переменных должен быть задан явно); в реализациях для языка Fortran могут назначаться классы *private* и *firstprivate*;
– список переменных, объявленных директивой *threadprivate* (см. ниже), которые при входе в параллельную секцию инициализируются значениями соответствующих переменных в потоке-мастере;
– оператор сведения и список общих переменных *reduction(оператор : список переменных)*; для каждой указанной в списке переменной создаются локальные копии в каждом потоке; локальные копии инициализируются соответственно типу оператора (для аддитивных операций ноль или его аналоги, для мультипликативных операций единица или её аналоги); над всеми локальными копиями каждой переменной после завершения параллельной секции будет выполнен заданный оператор сведения, результаты будут занесены в одноименные общие переменные; в качестве оператора можно указывать: +, –, *, &, |, ^, &&, ||.

2. Директива определения цикла, итерации которого нужно распределить между параллельно выполняемыми потоками:

#pragma omp for [опция_1[, опция_2, ...]]
<структурный блок кода>

С использованием опций директивы for можно указать:

– список переменных, которые будут локальными в каждом потоке секции (*private(список переменных)*), причем их начальные значения не будут определены;
– список переменных, которые будут локальными в каждом потоке секции (*firstprivate(список переменных)*), причем в качестве их начальных значений будут установлены значения одноименных переменных из главного потока;
– список переменных главного потока (*lastprivate(список переменных)*), которым будут присвоены значения, полученные при выполнении последней итерации цикла;
– оператор сведения и список общих переменных *reduction(оператор : список переменных)*; для каждой указанной в списке переменной создаются локальные копии в каждом потоке; локальные копии инициализируются соответственно типу оператора (для аддитивных операций ноль или его аналоги, для мультипликативных операций единица или её аналоги); над всеми локальными копиями каждой переменной после завершения параллельной секции будет выполнен заданный оператор сведения, результаты будут занесены в одноименные общие переменные; в качестве оператора можно указывать: +, –, *, &, |, ^, &&, ||;
– способ распределения итераций цикла между потоками параллельной секции (*schedule(type[, chunk])*); параметр *chunk* этой опции определяет количество итераций на один поток (по умолчанию 1), параметр *type* указывает тип распределения и может иметь значения:
  - *static* – статический, т.е. при компиляции,
  - *dynamic* – динамический, т.е. при выполнении,
  - *guided* - динамический с уменьшением количества итераций на поток от начального, определяемого автоматически, до значения *chunk*,
  - *auto* – способ распределения выбирается компилятором или исполняющей системой,

- *runtime* – способ распределения задается специальной переменной окружения операционной системы;

– возможность (опция *ordered*) появления в теле цикла директивы *ordered*, которая требует исполнения охваченного ею блока операторов в точности в той же последовательности, какая реализуется в последовательной версии данного цикла;
– отмену (*nowait*) неявной барьерной синхронизации потоков, достигших конца выполнения своей части итераций цикла (при отсутствии этой опции участок программы после цикла будет выполняться только тогда, когда все потоки выполнят все свои итерации;
– глубину n вложенных друг в друга циклов (*collapse(n)*), пространство итераций которых подлежит распределению между параллельными потоками (доступно только в реализации 2.5 технологии OpenMP);

Директивы *parallel* и *for* можно объединять в одну директиву, в которой можно указывать опции как директивы *parallel*, так и директивы *for*:
#pragma omp parallel for [опция_1[, опция_2, ...]]
3. Директива, указывающая на необходимость исполнения охватываемого ею участка кода (части тела цикла) в той последовательности, в которой он выполняется в чисто последовательном режиме
#pragma omp ordered
<структурный блок кода>
4. Директива задания области нециклического параллелизма (участки структурного блока кода, которые могут выполняться параллельно, выделяются с помощью описываемой далее директивы *section*):
#pragma omp sections [опция_1[, опция_2, ...]]
<структурный блок кода>

В качестве опций этой директивы можно указывать *private, firstprivate, lastprivate, reduction* и *nowait*, имеющие в точности такой же синтаксис и тот же смысл, что и у директивы *for.*
5. Директива определения участка нециклического кода для одного потока:
#pragma omp section

С помощью этой директивы внутри участка кода, охваченного директивой *sections*, выделяются отдельные фрагменты для исполнения параллельными потоками. Перед самым первым фрагментом участка нециклического кода директиву *section* можно не указывать.
6. Директива объявления списка локальных переменных потоков
#pragma omp threadprivate(список переменных)

Эта директива позволяет сделать локальные копии для статических переменных языка C/C++ (и COMMON-блоков языка Фортран), которые по умолчанию являются общими.
7. Директива создания отдельной независимой задачи (начиная с версии 2.5 OpenMP):
#pragma omp task [опция_1[, опция_2, ...]]
<структурный блок кода>

Текущий поток создает в качестве задачи ассоциированный с директивой блок операторов. Эта задача может выполняться немедленно после создания или быть отложенной на неопределённое время и выполняться по частям. Размер таких частей, а также порядок выполнения частей разных отложенных задач определяется реализацией OpenMP.

Возможные опции:
– *if, default, private, firstprivate* и *shared* – имеют такой же синтаксис и смысл, как и в предыдущих директивах;

– *untied* – означает, что в случае откладывания задача может быть продолжена любым потоком из числа выполняющих данную параллельную область; если данная опция не указана, то задача может быть продолжена только породившим её потоком;

8. Директива ожидания потоком завершения всех независимых задач, запущенных именно из данного потока:

#pragma omp taskwait

9. Директива, требующая исполнения охваченного ею участка кода в точности одним (любым) потоком:

#pragma omp single [опция_1[, опция_2, ...]]
<структурный блок кода>

Опции директивы позволяют указать:

– списки переменных *private* и *firstprivate*, имеющие такой же синтаксис и семантику, как в ранее описанных директивах;
– список переменных (*copyprivate(список переменных)*), значения которых после выполнения структурного блока, заданного директивой *single*, будут занесены во все одноименные локальные переменные (*private* и *firstprivate*), заданные для охватывающей параллельной секции; эта опция не может использоваться совместно с опцией *nowait*; переменные списка не должны быть перечислены в опциях *private* и *firstprivate* данной директивы *single*;
– отмену (*nowait*) неявной барьерной синхронизации потоков, достигших точки, следующей за блоком операторов, охваченным директивой *single*; при отсутствии этой опции такая синхронизация выполняется.

10. Директива, требующая исполнения охваченного ею участка кода главным потоком программы:

#pragma omp master
<структурный блок кода>

Этот структурный блок будет выполнен только главным потоком программы. Остальные потоки просто пропускают данный участок и продолжают работу с оператора, расположенного следом за ним. Неявной синхронизации данная директива не предполагает.

11. Директива явной барьерной синхронизации:

#pragma omp barrier

Потоки, выполняющие текущую параллельную секцию, дойдя до этой директивы, останавливаются и ждут, пока все потоки не дойдут до этой точки программы, после чего разблокируются и продолжают работать дальше. Кроме того, для разблокировки необходимо, чтобы все синхронизируемые потоки завершили все порождённые ими задачи (директивы *task* и *taskwait*).

12. Директива, объявляющая критическую секцию – участок параллельной области программы, который единовременно может выполняться не более, чем одним потоком:

#pragma omp critical [(<имя_критической_секции>)]
<структурный блок кода>

Если критическая секция уже выполняется каким-либо потоком, то все остальные, выполнившие эту директиву для секции с данным именем, будут заблокированы, пока вошедший в секцию поток не закончит выполнение этого блока кода. Как только это произойдет, один из заблокированных потоков войдет в критическую секцию. Если на входе в критическую секцию стояло несколько потоков, то случайным образом выбирается один из них, а остальные заблокированные продолжают ожидание.

Все неименованные критические секции условно ассоциируются с одним и тем же именем. Все критические секции, имеющие одно и тоже имя, рассматриваются как одна секция, даже если находятся в разных параллельных областях. Побочные входы и выходы из критической секции запрещены.

13. Директива блокировки доступа к общей переменной из левой части оператора присваивания на время выполнения всех действий с этой переменной в данном операторе:

#pragma omp atomic

<оператор присваивания>

Атомарной (блокирующей выполнение остальных потоков) является только работа с переменной из левой части оператора присваивания, при этом вычисления в его правой части, использующие другие переменные, не обязаны быть атомарными.

14. Директива актуализации значений переменных потока:

#pragma omp flush [(список переменных)]

Значения всех переменных (или переменных из списка, если он задан), временно хранящиеся в регистрах и кэш-памяти текущего потока, заносятся в основную память; следовательно, все изменения переменных, сделанные потоком во время работы, станут видимы остальным потокам; если какая-то информация хранится в буферах вывода, то буферы будут сброшены на внешние носители и т.п. Эти действия производятся только с данными выполнившего директиву потока, а данные, изменявшиеся другими потоками, не затрагиваются. До полного завершения этих действий никакие другие операции с участвующими в директиве *flush* переменными не могут выполняться. Поэтому выполнение директивы *flush* без списка переменных может повлечь значительные накладные расходы. Если в данный момент нужна гарантия согласованного представления не всех, а лишь некоторых переменных, то именно их следует явно перечислить в директиве списком.

Переменные окружения

OMP_NUM_THREADS – количество потоков в новой параллельной области.

OMP_NESTED – возможность создания вложенных параллельных областей.

OMP_MAX_ACTIVE_LEVELS – максимальная глубина вложенности параллельных областей.

OMP_DYNAMIC – возможность динамического определения количества потоков (имеет больший приоритет, чем OMP_NUM_THREADS).

OMP_THREAD_LIMIT – максимальное количество потоков программы.

OMP_SCHEDULE – способ распределения итераций цикла для значения *runtime* опции *schedule* директивы *for*.

OMP_STACKSIZE – размер стека каждого потока.

OMP_WAIT_POLICY – выделять или нет кванты процессорного времени ждущим потокам.

Библиотека функций OpenMP:

double omp_get_wtime(void); – возвращает текущее время.

double omp_get_wtick(void); – возвращает размер тика таймера.

int omp_get_thread_num(void); – возвращает номер потока.

int omp_get_max_threads(void); – возвращает максимально возможное количество потоков для следующей параллельной области.

int omp_get_thread_limit(void); – возвращает значение OMP_THREAD_LIMIT.

void omp_set_num_threads(int num); – устанавливает новое значение переменной OMP_NUM_THREADS.

int omp_get_num_procs(void); – возвращает количество процессоров/ядер.

int omp_get_dynamic(void); – возвращает значение OMP_DYNAMIC.

void omp_set_dynamic(int num); – устанавливает новое значение переменной OMP_DYNAMIC.

int omp_get_nested(void); – возвращает значение OMP_ NESTED.

void omp_set_nested(int nested); – устанавливает новое значение переменной OMP_ NESTED.

int omp_in_parallel(void); – возвращает 0, если функция вызвана из последовательной области и 1, если из параллельной.

int omp_get_max_active_levels(void); – возвращает значение переменной OMP_MAX_ACTIVE_LEVELS

void omp_set_max_active_levels(int max);); – устанавливает новое значение OMP_MAX_ACTIVE_LEVELS

int omp_get_level(void); – возвращает глубину вложенности параллельных областей.

int omp_get_ancestor_thread_num(int level); – возвращает номер потока, породившего текущую параллельную область.

int omp_get_team_size(int level); – возвращает для заданного параметром level уровня вложенности параллельных областей количество потоков, порождённых одним родительским потоком.

int omp_get_active_level(void); – возвращает количество вложенных параллельных областей, обрабатываемых более чем одним потоком.

Следующая группа функций используется для синхронизации параллельно выполняющихся потоков с помощью так называемых замков (lock).

*void omp_init_lock(omp_lock_t *lock);* – создать (инициализировать) простой замок.

*void omp_init_nest_lock(omp_nest_lock_t *lock);* – создать замок с множественными захватами.

*void omp_destroy_lock(omp_lock_t *lock);* – уничтожить простой замок (перевести в неинициализированное состояние).

*void omp_destroy_nest_lock(omp_nest_lock_t *lock);* – уничтожить множественный замок (перевести в неинициализированное состояние).

*void omp_set_lock(omp_lock_t *lock);* – захватить простой замок (если замок уже захвачен другим потоком, то данный поток переводится в ждущее состояние).

*void omp_set_nest_lock(omp_nest_lock_t *lock);* – захватить множественный замок (если замок уже захвачен другим потоком, то данный поток переводится в ждущее состояние; если замок захвачен этим же потоком, то увеличивается счетчик захватов).

*void omp_unset_lock(omp_lock_t *lock);* – освободить простой замок (если есть потоки, ждущие его освобождения, то один из них, выбираемый случайным образом, захватывает этот замок и переходит в состояние выполнения).

*void omp_unset_nest_lock(omp_lock_t *lock);* – уменьшить на 1 количество захватов множественного замка (если количество захватов стало равно нулю и есть потоки, ждущие его освобождения, то один из них, выбираемый случайным образом, захватывает этот замок и переходит в состояние выполнения).

*int omp_test_lock(omp_lock_t *lock);* – попытаться захватить простой замок (если попытка не удалась, т.е. замок уже захвачен другим потоком, то возвращается 0, иначе возвращается 1).

*int omp_test_nest_lock(omp_lock_t *lock);* – попытаться захватить множественный замок (если попытка не удалась, т.е. замок уже захвачен другим потоком, то возвращается 0, иначе возвращается новое значение счетчика захватов).

Задание

1. Изучить технологию OpenMP и средства среды разработки используемые при разработке и отладке параллельных программ на основе этой технологии.
2. Разработать последовательный алгоритм решения задачи, написать и отладить последовательную программу.
3. Модифицировать последовательную программу путем вставки директив и вызовов функций OpenMP, получить и отладить параллельную программу решения задачи.

4. Проанализировать результаты решения задачи для последовательного и параллельного вариантов программы, оценить отклонения полученных решений друг от друга и от эталонного (если оно известно), объяснить причины отклонений.

5. Измерить и оценить временные характеристики последовательной и параллельной программы, объяснить полученные соотношения.

Программирование МВС с графическими процессорами (4 часа)

Цель работы: изучить архитектуру CUDA и основы разработки параллельных программ для совместного использования CPU и GPU.

Теоретические сведения

CUDA (Compute Unified Device Architecture) представляет собой совокупность аппаратного (графический процессор – GPU) и программного обеспечения, предоставляющего возможность подготовки и исполнения программ с очень высокой степенью параллелизма.

GPU есть специализированное вычислительное устройство, которое является сопроцессором к основному процессору компьютера (CPU), обладает собственной памятью и возможностью параллельного выполнения огромного количества (тысячи и десятки тысяч) отдельных нитей (потоков) обработки данных согласно модели ОКМД.

Технология OpenMP и CUDA вполне совместимы и могут использоваться в одной параллельной программе для достижения максимально возможного ускорения.

Между потоками, выполняемыми на CPU и потоками, выполняемыми графическим процессором, есть принципиальные различия:

- нити, выполняемые на GPU, обладают крайне низкой «стоимостью» – их создание и управление ими требует минимальных ресурсов (в отличии от потоков CPU)
- для эффективной утилизации возможностей GPU нужно использовать многие тысячи отдельных нитей (для CPU обычно бывает трудно организовывать более, чем 10-20 потоков)

Приложения, использующие возможности CUDA для параллельной обработки данных, взаимодействуют с GPU через программные интерфейсы, называемые CUDA-runtime или CUDA-driver (обычно интерфейсы CUDA-runtime и CUDA-driver одновременно не используются, хотя это в принципе возможно).

Программы для CUDA пишутся на "расширенном" языке C, при этом их параллельная часть (так называемые «ядра») выполняется на GPU, а обычная последовательная часть – на CPU. Компоненты архитектуры CUDA, принимающие участие в подготовке и исполнении приложения, автоматически осуществляют разделение частей и управление их запуском.

С точки зрения разработчика программы для CUDA графический процессор представляет собой одно устройство управления и огромное количество арифметико-логических устройств (АЛУ), каждое из которых обладает собственной регистровой памятью (распределением регистров между АЛУ управляет программное обеспечение CUDA) и имеет доступ к нескольким уровням памяти различного объема и быстродействия. Все арифметико-логические устройства в любой данный момент времени исполняют одну команду, выбранную и декодированную устройством управления. Реально устройств управления несколько (их количество зависит от модели графического процессора). Совокупность из одного устройства управления, связанных с ним арифметико-логических устройств, регистров, быстрой разделяемой памяти, доступной из каждого АЛУ, и более медленной локальной памяти каждого АЛУ образуют так называемый мультипроцессор. Все мультипроцессоры имеют доступ к еще более медленным кэшу текстур, кэшу констант и глобальной памяти устройства, через

которую осуществляется передача данных в/из основную память компьютера. Внутренние элементы памяти мультипроцессора недоступны из CPU.

Отображение одной или нескольких параллельных областей программы на совокупность мультипроцессоров осуществляет программное обеспечение CUDA и является «прозрачным» для программиста. Ему нужно знать только о том, что с логической точки зрения параллельная область программы («ядро» в терминологии CUDA) представляет собой сетку (grid) блоков (block) потоков (thread). Блок потоков – это набор потоков, выполняемых одним мультипроцессором. Потоки одного блока могут взаимодействовать через разделяемую память и синхронизироваться между собой. Сетка и (независимо от нее) каждый из блоков представляют собой одно-, дву- или трехмерную структуру блоков и потоков соответственно. Количество размерностей и размеры каждой из них задает программист. Каждый поток после запуска имеет доступ к переменным (структурам), хранящим его собственные координаты внутри блока и координаты охватывающего блока внутри сетки.

Архитектура CUDA предусматривает широкий набор возможностей и поддерживается в видеокартах нескольких линеек (GEforce, Quadro, NVS, Tesla, ION). Разные модели графических процессоров, функционирующих в таких видеокартах, имеют каждая собственный набор параметров (например – количество регистров, объем разделяемой памяти мультипроцессора) и поддерживают разные наборы возможностей, сгруппированные в версии архитектуры от 1.0 до 2.1 (по состоянию на конец 2011 года). Для каждого устройства производитель указывает поддерживаемую им версию CUDA, по номеру версии можно определить значения параметров и доступные возможности.

Рекомендуемая разработчиками архитектуры технология параллельного программирования состоит в следующем:

1 Спланировать разбиение обрабатываемых данных на фрагменты, целиком помещающиеся в разделяемую память графического процессора.
2 Обеспечить загрузку данных в глобальную память GPU.
3 Запустить ядро, в программе каждого блока:
   3.1. Выполнить перемещение нужных фрагментов данных из глобальной памяти в локальную память блока.
   3.2. Выполнить требуемые вычисления.
   3.3. Скопировать сформированные результаты обработки из разделяемой памяти в глобальную память устройства.
4 Перенести полученные результаты из памяти графического процессора в основную память компьютера, выполнить их окончательную обработку или вывести на внешний носитель.

Расширенная среда программирования для языка C включает:

– Расширения синтаксиса для написания кода для GPU, реализуемые дополнительным препроцессором nvcc.
– Run-time библиотеки:
  • Общая часть – встроенные векторные типы данных и набор функций, исполняемых и на CPU, и на GPU.
  • CPU-компонента, для доступа и управления одним или несколькими GPU (драйвер).
  • GPU-компонента, предоставляющая функции, специфические только для GPU.

Расширения синтаксиса языка включают в себя:

1. Дополнительные виды функций:

	Исполняется на	Вызывается из
__host__ float HostFunc()	CPU	CPU
__global__ void KernelFunc()	GPU	CPU

__device__ float DeviceFunc()	GPU	GPU

Особенности дополнительных видов функций:

Функция, объявленная как __global__, является функцией-ядром, она не может возвращать никакого значения (всегда void).

Функция вида __device__ и выполняется и запускается из функций, исполняемых на GPU, поэтому нельзя получить указатель на такую функцию.

В функциях, исполняемых на GPU:

– Недопустима рекурсия.
– Не может быть статических переменных внутри функций.
– Количество аргументов не может быть переменным.

2. Дополнительные виды переменных:

	Тип памяти	Область видимости	Время жизни
__device__ __shared__ int	разделяемая	блок	блок
__device__ int	глобальная	сетка	ядро
__device__ __constant__ int	кэш констант	сетка	ядро

Ключевое слово *__device__* необязательно, если используется *__shared__* или *__constant__* . Локальные переменные без идентификатора вида хранятся в регистрах, за исключением больших структур или массивов, действительно располагающихся в локальной памяти. Тип памяти указателя адаптируется к типу присвоенного ему выражения

3. Встроенные векторные типы объявляются так:

[u]char[1..4], [u]short[1..4], [u]int[1..4], [u]long[1..4], float[1..4]

Буква *u* в этих объявлениях необязательна, является сокращением от слова *unsigned* и в случае ее указания означает, что все поля данного типа являются беззнаковыми целыми.

Доступ к полям этих типов осуществляется по именам: *x, y, z, w,* например:

```
uint4 param;
int y = param.y;
int abc = param.w;
```

Тип *dim3* является синонимом *uint3*. Обычно тип *dim3* используется для задания параметров сетки.

4. Ядро (собственно программа для GPU) и его запуск:

При запуске ядру должны быть переданы обязательные параметры конфигурации сетки. Для этого в программе для CPU описываются и создаются следующие функции и переменные:

__global__ void KernelFunc(...); – прототип функции, являющейся ядром.

dim3 DimGrid(100, 50); – переменная CPU, определяющая двумерность сетки блоков потоков и ее размеры 100 x 50, т.е. 5000 блоков.

dim3 DimBlock(4, 8, 8); – переменная CPU, определяющая трехмерность совокупности потоков в каждом блок и размеры этой совокупности 4 x 8 x 8, т.е. 256 потоков на блок.

size_t SharedMemBytes = 64; – переменная CPU, определяющая размер разделяемой памяти в 64 байта, выделяемой каждому потоку.

После этих объявлений можно запустить ядро, для этого тоже реализован специальный синтаксис:

KernelFunc<<<DimGrid, DimBlock, SharedMemBytes>>>(...);

Опциональные *SharedMemBytes* байт:

– выделяются в дополнение к статически объявленным разделяемым переменным в ядре;

– отображаются на любую переменную вида: *extern __shared__ float DynamicSharedMem[]*.

Запуск ядра асинхронен, сразу после вызова функции управление немедленно возвращается к следующему оператору программы для CPU.

5. Встроенные переменные, создаваемые и инициализируемые автоматически при запуске ядра:

dim3 gridDim; – размеры сетки в блоках (*gridDim.w* не используется).

dim3 blockDim; – размеры одного блока в потоках.

dim3 blockIdx; – индекс блока внутри сетки.

dim3 threadIdx; – индекс потока внутри блока.

Эти переменные доступны только для кода параллельных ветвей ядра, исполняемого на GPU.

Общая часть Run-time библиотек содержит большое количество функций, необходимых (или опциональных) при написании параллельной программы. Таких функций слишком много для того, чтобы приводить их описание в данном методическом пособии, поэтому здесь приводится описание только тех функций, которые действительно необходимы. Все библиотечные функции можно разделить на несколько групп:

1. Управление потоками CPU и GPU, синхронизация.
2. Обработка ошибок.
3. Управление устройством.
4. Управление событиями.
5. Управление глобальной памятью.
6. Интерфейс с OpenGL и Direct3D версий 9, 10 и 11.
7. Интерфейс с видеоадаптером.
8. Управление текстурами и поверхностями.
9. Интерфейс с драйвером CUDA.

При разработке любой параллельной программы невозможно обойтись без использования функций управления глобальной памятью GPU. Таких функций более 40, из них наиболее употребительны следующие:

cudaError_t cudaMalloc (void ∗∗devPtr, size_t size);

Выделяет блок глобальной памяти устройства размером *size* и заносит адрес этого блока в указатель *devPtr*. Возвращает либо код успешного завершения *cudaSuccess*, либо код ошибки, если блок памяти распределить не удалось.

cudaError_t cudaFree (void ∗devPtr);

Возвращает ранее выделенный блок глобальной памяти в пул свободной памяти. Возвращает либо код успешного завершения, либо код ошибки (например, если этот блок ранее не выделялся).

cudaError_t cudaMemcpy(void ∗dst, const void ∗src, size_t count, enum cudaMemcpyKind kind);

Копирует блок памяти размером *count* байт, расположенный по адресу *src*, в память по адресу *dst*. Один из этих блоков размещается в основной памяти компьютера, второй – в глобальной памяти графического процессора, какой из них где – определяется значением аргумента *kind* (допустимы *cudaMemcpyHostToDevice* и *cudaMemcpyDeviceToHost*). Возвращает либо код успешного завершения, либо код ошибки.

При необходимости использования других функций можно использовать руководства по программированию для CUDA и справочные материалы.

Задание

1. Изучить основы архитектуры CUDA и настройку среды разработки Microsoft Visual Studio для разработки и отладки параллельных программ графического процессора.
2. Разработать план размещения данных решаемой задачи в разделяемой памяти GPU.
3. Разработать ядро (функцию для GPU, выполняющую массовую параллельную обработку матрицы) в соответствии с этим планом.
4. Разработать последовательную программу для CPU, обеспечивающую считывание исходных данных, перенос их в глобальную память GPU, запуск ядра, ожидание его завершения, считывание и окончательную обработку результатов. Отладить совокупность программ для CPU и GPU.
5. Проанализировать результаты решения задачи, оценить отклонение полученных значений от ожидаемых, объяснить причины отклонений.

Типовые параллельные алгоритмы (4 часа)

Цель работы: разработка типовых параллельных алгоритмов.

Теоретические сведения

Более подробное теоретическое обоснование данной темы представлено в следующих источниках:

1. Инструменты параллельного программирования в системах с общей памятью: учебник для вузов / К. В. Корняков [и др.] ; ред. В. П. Гергель ; Нижегородский государственный университет (Нижний Новгород). - 2-е изд., испр. и доп. - М. : Издательство Московского университета, 2010. - 271 с. : ил., табл. - (Суперкомпьютерное образование). - Библиогр. в конце разд. -ISBN 978-5-211-05931-3 (26 экз.)

Задание

Реализовать следующие параллельные алгоритмы: умножение матриц: линейное, блочное разбиение матриц, алгоритмы Фокса и Кеннона, решение систем линейных уравнений, алгоритмы параллельной сортировки.

САМОСТОЯТЕЛЬНАЯ РАБОТА

Темы, выносимые на самостоятельное изучение:

1. Средства разработки распределенных вычислений с использованием параллельной виртуальной машины PVM. Использование функций библиотеки Pvmlib.
2. Протокол и библиотеки MPI. Состав функций, типы данных, организация обмена данными. Пример использования библиотеки MPI. Среда выполнения LAM.
3. Расширения стандартных языков программирования для создания параллельных программ. CILK, mPC. Дополнительные операторы, их синтаксис и семантика. Особенности реализаций и примеры программирования.
4. Язык OCCAM, конструкции языка, понятие процессов, каналов.
5. Удаленные вызовы процедур. Высокоуровневый и низкоуровневый интерфейс программирования. XDR-преобразования, аутентификация, широковещательный режим. Использование технологий DCOM и CORBA для создания распределенных приложений. Язык описания интерфейсов IDL.

Контрольные вопросы

1. Какой минимальный набор средств является достаточным для организации параллельных вычислений в системах с распределенной памятью?
2. В чем состоит важность стандартизации средств передачи сообщений?

3. Что следует понимать под параллельной программой?
4. В чем различие понятий процесса и процессора?
5. Какой минимальный набор функций MPI позволяет начать разработку параллельных программ?
6. Как определить время выполнения MPI программы?
7. Какие режимы передачи данных поддерживаются в MPI?
8. Какие факторы нужно учитывать при оценке производительности вычислительной системы? Какие методики оценки производительности вычислительных систем вы знаете?
9. Что такое степень параллелизма вычислительного алгоритма?
10. Что такое ускорение параллельного алгоритма?
11. Для чего используются блочные матричные алгоритмы?
12. Что такое эффективность параллельного алгоритма?
13. Приведите формулировку закона Амдаля.
14. Во сколько раз нужно ускорить 90% программы, чтобы ускорить всю программу в 5 раз?
15. Что такое синхронизация и для чего она нужна?
16. Каковы области применения, преимущества и недостатки технологии OpenMP?
17. Каковы области применения, преимущества и недостатки библиотеки MPI?
18. Чем отличаются блокирующие и неблокирующие операции в MPI?