Разработка многопоточных приложений с использованием OpenMP
Отчёт

Скрыпина Дарья Кирилловна БПИ 198

Вариант №26

Условие задания:
«*Вторая задача об Острове Сокровищ. Шайка пиратов под предводительством Джона Сильвера высадилась на берег Острова Сокровищ. Несмотря на добытую карту старого Флинта, местоположение сокровищ по-прежнему остается загадкой, поэтому искать клад приходится практически на ощупь. Так как Сильвер ходит на деревянной ноге, то самому бродить по джунглям ему не с руки. Джон Сильвер поделил остров на участки, а пиратов на небольшие группы. Каждой группе поручается искать клад на нескольких участках, а сам Сильвер ждет на берегу. Группа пиратов, обшарив одну часть острова, переходит к другой, еще необследованной части. Закончив поиски, пираты возвращаются к Сильверу и докладывают о результатах. Требуется создать многопоточное приложение с управляющим потоком, моделирующее действия Сильвера и пиратов. При решении использовать парадигму портфеля задач.»*

1) Описание используемой модели вычислений

В данной многопоточной программе используется модель «**Взаимодействующие равные**», а если точнее – парадигма «**портфеля задач**». В данном случае «портфель задач» реализуется с помощью глобальной общей переменной, к которой единовременно имеет доступ только один поток. Эта переменная имеет тип **bool**, называется **hasFoundTreasure** и является признаком того, был ли уже найден клад или нет (изначально имеет значение **False**).

Каждый поток (группа пиратов) проверяет, было ли уже найдено сокровище. Если нет, то поток отправляется исследовать свободную часть острова, информирует пользователя о начале вычислений, выполняет какие-то действия (ищет клад), после чего информирует пользователя о результатах своей работы (клад найден/не найден), и действия повторяются заново. Если при проверке оказалось, что клад уже найден, то поток не делает ничего. Если группа пиратов, образно говоря, «находилась в поисках клада» на своей локации, в то время как другая нашла клад у себя, то данные пираты заканчивают поиски, докладывают о своей неудаче и «возвращаются к капитану» - то есть больше не берутся за задачи.

Выполнение программы начинается с обработки входных данных – двух целых чисел (параметров **pirate_team_count** и **island_parts**), представляющих собой количество потоков, или, в контексте задачи, групп пиратов (см. раздел «Входные данные») и количество частей острова соответственно. После проверки входных данных на корректность с помощью команды **omp_set_num_threads(pirate_team_count)** устанавливается количество потоков, равное количеству групп пиратов.

Далее происходит инициализация блокирующего механизма, обеспечивающего надёжный доступ к потоку вывода в консоль только одному параллельному потоку единовременно. Это необходимо, поскольку вывод текстовой информации в консоль происходит сразу в двух критических секциях – если «лок» убрать, то может возникнуть ситуация, когда один поток выводит информацию из одной критической секции, а второй поток – из другой. Инициализация происходит с помощью команды **omp_init_lock(&write_lock)**, где &write_lock – ссылка на блокирующий механизм.

Далее с помощью генератора случайных чисел выбирается число от 1 до island_parts включительно и записывается в целочисленную глобальную переменную **treasureLocation**. Оно обозначает, на какой части острова спрятано сокровище.

Далее перед началом параллельной секции программы запускается таймер – он нужен для того, чтобы при выводе информации пользователю указывалось время, прошедшее со старта программы.

Далее объявляется цикл for (проходящийся по всем частям острова), перед которым стоит составная конструкция OpenMP: **#pragma omp parallel for schedule(dynamic).**

Часть **#pragma omp parallel** отвечает за создание параллельного региона, ограниченного блоком фигурных скобок, где все встреченные команды исполняются порождёнными параллельными потоками. Их количество

контролируется через переменную окружения **omp_num_threads**, которую мы задали заранее. Часть **for** отвечает за распределение итераций следующего за конструкцией цикла for по потокам. Все потоки, достигнув конца цикла, дожидаются тех, кто ещё не завершился, после чего основная нить продолжает выполняться дальше. Часть **schedule(dynamic)** является условием, контролирующим то, как работа будет распределяться между потоками. В данном случае schedule(dynamic) имеет прямое отношение к заданию - работа распределяется между потоками динамически, каждый поток обрабатывает единичную итерацию цикла, и, как только закончит, захватывает следующую (если клад на тот момент не был найден). Нет чёткого порядка, в котором распределяется выполнение порций между потоками. На каждой итерации цикла вызывается функция **LookForTreasure**, которая будет описана далее.

Функция **LookForTreasure** моделирует поведение пиратов, ищущих сокровище на конкретном участке острова. В качестве входных параметров этой функции передаются номер потока/группы пиратов (получаемый с помощью функции **omp_get_thread_num()**) и номер части острова, которую необходимо исследовать (копия значения переменной-счётчика в цикле, увеличенная на 1 для лучшей читаемости). В начале функции находится критическая секция, объявленная с помощью конструкции **#pragma omp critical {}**. Критическая секция, объявленная с помощью critical, гарантирует, что команды, заключённые в её блоке, будут исполнены только одним потоком единовременно. В данном случае каждая из групп пиратов поочерёдно сообщает о том, что она приступает к поискам на соответствующей локации – в блоке критической секции находится команда вывода текстовой информации в консоль. Кроме этого, с помощью команды **omp_set_lock(&write_lock)** перед выводом устанавливается блокирующий механизм, который не позволяет любому другому потоку выводить информацию в консоль одновременно с данным. После вывода информации в консоль «лок» снимается с помощью команды **omp_unset_lock(&write_lock)**. Аналогичная блокировка и разблокировка происходит в критической секции в конце метода и методе GetElapsedTime().

Кроме этого, при каждом выводе в консоль информации о действиях пиратов выводится и время, прошедшее с начала работы программы – оно рассчитывается с помощью функции **GetElapsedTime()** и там же выводится в консоль (тоже с блокировкой и разблокировкой).

После окончания критической секции идёт часть кода, в которой происходит непосредственно поиск сокровищ. Для каждого потока с помощью локальной переменной и генератора случайных чисел определяется целое число от 34 до 39, которое подаётся на вход вспомогательной функции **DiggingForTreasure()**, моделирующей некоторую временную задержку потока. Данный метод был использован, поскольку при изучении OMP не было найдено аналога thread::sleep_for() из стандартной библиотеки C++. На локальном компьютере возникающая задержка могла в среднем длиться от 0,5 до 4,9 секунд, результаты будут варьироваться на разных машинах и при разных нагрузках на процессор.

Далее следует вторая критическая секция, которая тоже контролирует вывод текстовой информации в консоль – на этот раз о том, что текущая команда нашла/не нашла сокровище на текущей локации. Если номер текущей локации

совпал с номером локации, в которой лежит сокровище (т.е. с заданной заранее переменной treasureLocation), то флаг нахождения сокровища ставится в позицию **True** и выводится информация об успехе данной команды на данной локации. Иначе выводится сообщение о неудаче.

На этом моменте поток завершает выполнение функции LookForTreasure и «забирает» себе новую итерацию цикла (описанного в методе main программы), если сокровище на данный момент ещё найдено не было. Когда сокровище будет найдено, а все команды пиратов доложат о своих последних действиях, программа уничтожает механизм блокировки с помощью команды **omp_destroy_lock(&write_lock)** и завершает свою работу.

2) Входные данные

Для задания входных данных используются параметры командной строки. Нулевым параметром является имя исполняемого файла. Первым – целое число, обозначающее число генерируемых потоков (кол-во групп пиратов). Должно быть в пределах от 1 до 499 включительно. Вторым – целое число, обозначающее количество участков, на которые Джон Сильвер делит остров (тоже от 1 до 499). При получении входных данных в некорректном формате программа выдаёт сообщение об ошибке и завершает работу (см. тестовые примеры).

3) Источники информации, в которых описана данная модель

- http://ccfit.nsu.ru/arom/data/openmp.pdf - основные сведения о параллельном программировании с использованием OpenMP
- http://jakascorner.com/blog/2016/04/omp-introduction.html - введение в OpenMP
- http://jakascorner.com/blog/2016/06/omp-for-scheduling.html - описание конструкций с циклом for и условием schedule, одна из которых используется в данной программе для реализации парадигмы «портфеля задач»
- https://docs.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-directives?view=msvc-160 – директивы OpenMP
- https://docs.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-clauses?view=msvc-160 – параметры OpenMP

4) Тестовые данные

Тестовые наборы и примеры выполнения находятся в формате скриншотов в той же директории, что и данный документ, и имеют название test*.png, где * - номер теста.