Санкт-Петербургский государственный политехнический университет

Институт компьютерных наук и технологий

Кафедра компьютерных систем и программных технологий

РЕФЕРАТ

по дисциплине: «Современные проблемы информатики и вычислительной техники»

Тема работы: «Автоматизация миграции программного кода на новый набор библиотек»

Работу выполнил студент

63501/3 *Алексюк А.О.*

Преподаватель

_____ *Мелехин В.Ф.*

Санкт-Петербург
2016

1. Введение

В рамках курсового проекта студент выбирает один из алгоритмов, для которого выполняется:

1) разработка алгоритма
2) создание последовательной программы, реализующей алгоритм
3) разработка тестового набора, оценка тестового покрытия
4) выделение частей для параллельной реализации, определение общих данных, анализ потенциального выигрыша
5) выбор способов синхронизации и защиты общих данных
6) разработка параллельной программы
7) отладка на имеющихся тестах, анализ и доработка тестового набора с учетом параллелизма
8) измерение производительности, сравнение с производительностью последовательной программы
9) анализ полученных результатов, доработка параллельной программы
10) написание отчета и презентации
11) защита проекта

Мною было выбрано задание №10: Определить частоту встречи слов в тексте на русском языке.

2. Обзор методов миграции программ

3. Обзор инструментов миграции программ

Инструмент миграции программ является частным случаем транспайлера (transpiler transcompiler, source-to-source compiler) - компилятора, результатом работы которого является код на высокоуровневом языке (в отличие от обычных компиляторов, которые на выходе дают программу в машинных кодах или на языке ассемблера). Помимо миграции, транспайлеры используются для решения следующих задач:

- Трансляция программ на другой язык программирования. Например, первый компилятор языка C++ (Cfront, 1983 г.) являлся именно транспайлером. Сначала программа на C++ транслировалась в программу на C, после чего собиралась одним из существующих компиляторов языка C.

- Рефакторинг программ (процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения). Многие современные среды разработки, такие как IntelliJ IDEA или Eclipse, включают в себя системы автоматического рефакторинга программ.

- Перенос программ на другую версию языка. Пример инструмента для переноса на более новую версию языка - скрипт 2to3, транслирующий программу на языке Python 2 в программу на языке Python 3. Третья версия языка Python не имеет обратной совместимости со второй версии, и поэтому разработчикам необходимо переписывать старые программы, чтобы иметь возможность запускать их на новых версиях интерпретатор. Скрипт 2to3 позволяет во многом автоматизировать перенос. Пример переноса на более старую версию языка - инструмент Babel, позволяющий писать программы на языке ECMAScript 6 и исполнять их, даже если интерпретатор поддерживает только ECMAScript 5 или 3.

Для разбиения текста на слова воспользуемся функцией strtok_r (реентерабельный вариант функции strtok). В качестве разделителя укажем пробел и знаки препинания. Вызывая каждый раз функцию strtok_r, мы будем получать новое слово.

Считанные слова будут помещаться в ассоциативный массив (C++ контейнер map), где ключом будет слово, а значением - число появлений этого слова.

Такой ассоциативный массив очень легко заполнять, но не очень удобно обрабатывать. Например, в нем довольно сложно найти 5 самых популярных слов. Для этого преобразуем данные в другой формат, тоже ассоциативный массив, но такой, в котором ключом будет число появлений слова, а значением - само слово. В C++ для этого подойдет контейнер multimap (обычный map не подойдет, так как в наборе может быть несколько слов с одинаковой частотой появления, т.е. несколько записей, имеющих одинаковый ключ).

Выведем результаты на экран. Multimap обычно реализован с помощью бинарного дерева поиска, поэтому нет необходимости вручную сортировать результаты. Возьмем 5 самых популярных слов.

Функция main:

Для сборки воспользуемся утилитой CMake. Ниже приведен сценарий сборки.

В качестве тестовых данных возьмем роман-эпопею Льва Николаевича Толстого «Война и мир». Напишем скрипт, который скачивает её из Интернета, объединяет два тома и преобразует кодировку. Результат её работы - файл book.txt, содержащий чуть менее полумиллиона слов и занимающий 5 мегабайт.

Результат работы программы:

```
Stats:
20304 и
10198 в
8428 не
7822 что
6453 на
```

3.1. Тестирование

Для запуска тестов воспользуемся системой Google Test. Сценарий сборки ожидает, что исходный код фреймворка будет находиться рядом с исходным кодом в директории googletest-release-1.7.0.

Напишем простейший тест - пусть программа берет данные из константной строки, считает в ней частоту появления слов и сравнивает результаты с заранее известными.

Напишем более сложный тест - будем считывать данные из файла.

Результат запуска:

```
artyom@artyom-H97-D3H:~/Projects/ParallelComputing$ build/gcc/runTest
[==========] Running 8 tests from 4 test cases.
[----------] Global test environment set-up.
[----------] 2 tests from WordCountTest
[ RUN      ] WordCountTest.ConstStringTest
[       OK ] WordCountTest.ConstStringTest (0 ms)
[ RUN      ] WordCountTest.FileTest
[       OK ] WordCountTest.FileTest (388 ms)
[----------] 2 tests from WordCountTest (388 ms total)
```

3.2. Разработка параллельной реализации

Перед написанием параллельной реализации создадим промежуточный, «псевдопараллельный» вариант программы. Пусть в процессе своей работы программа разбивает исходные данные на блоки, обрабатывает их последовательно, а в конце объединяет результаты.

Блоком в данном случае будет часть текста. Все блоки будут иметь примерно одинаковый размер. К сожалению, если просто взять длину текста и разделить её на число блоков, если вероятность того, что граница между блоками ляжет посредине слова. Поэтому необходимо передвинуть каждую границу так, чтобы она не разбивала слово пополам (например, чтобы она лежала на пробеле).

Для подсчета частоты появления слов в блоке воспользуемся уже разработанной функцией countWords. Результат её работы - ассоциативный массив (словарь). Для объединения результата блоков достаточно объединить их словари, а если одно и то же слово имеется и в одном, и в другом блоке, нужно сложить количество раз, сколько они появлялись в каждом блоке. Параметр blockCount задает количество блоков.

Напишем тесты для этой реализации. Во-первых, проверим работу функции для файла. Во-вторых, сравним результаты работы новой реализации и старой. Для этого пройдем по словарю и сравним попарно все элементы (как уже говорилось выше, словарь заведомо отсортированный).

Результат тестов:

```
[----------] 2 tests from WordCountBlockwiseTest
[ RUN      ] WordCountBlockwiseTest.FileTest
[       OK ] WordCountBlockwiseTest.FileTest (415 ms)
[ RUN      ] WordCountBlockwiseTest.FileParAndSeqTest
```

```
5 [       OK ] WordCountBlockwiseTest.FileParAndSeqTest (806 ms)
6 [----------] 2 tests from WordCountBlockwiseTest (1239 ms total)
```

3.3. Распараллеливание с помощью OpenMP

"OpenMP (Open Multi-Processing) — открытый стандарт для распараллеливания программ на языках Си, Си++ и Фортран. Дает описание совокупности директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с общей памятью."

Для управления многопоточным выполнением в OpenMP используется директива #pragma. Например,

Код этой реализации практически аналогичен предыдущей.

Тесты аналогичны предыдущей реализации. Для получения количества процессоров воспользуемся функцией omp_get_num_procs(). Результаты запуска тестов:

```
1 [----------] 2 tests from WordCountOpenMPTest
2 [ RUN      ] WordCountOpenMPTest.FileTest
3 [       OK ] WordCountOpenMPTest.FileTest (180 ms)
4 [ RUN      ] WordCountOpenMPTest.FileParAndSeqTest
5 [       OK ] WordCountOpenMPTest.FileParAndSeqTest (589 ms)
6 [----------] 2 tests from WordCountOpenMPTest (769 ms total)
```

3.4. Распараллеливание с помощью POSIX Threads

Код тестов аналогичен предыдущим реализациям. Так как в Pthreads нет функции для получения количества процессоров, используется либо соответствующая функция из OpenMP (если он присутствует в системе), либо функция std::thread::hardwa из состава C++11. Помимо этих способов, при желании можно использовать и другие источники - WinAPI, Linux-специфичные вызовы и т.д. Результаты запуска тестов:

```
1 [----------] 2 tests from WordCountPthreadsTest
2 [ RUN      ] WordCountPthreadsTest.FileTest
3 [       OK ] WordCountPthreadsTest.FileTest (188 ms)
4 [ RUN      ] WordCountPthreadsTest.FileParAndSeqTest
5 [       OK ] WordCountPthreadsTest.FileParAndSeqTest (583 ms)
6 [----------] 2 tests from WordCountPthreadsTest (774 ms total)
```

3.5. Измерение производительности

Для измерения производительности был создан отдельный модуль проекта. Основа модуля - функция doBench, занимающаяся измерением времени выполнения. В процессе своей работы функция использует класс std::chrono::steady_clock из C++11, предоставляющий доступ к монотонному и высокоточному системному таймеру.

В качестве аргумента функции передается функтор (например, лямбда-выражение), благодаря чему можно использовать эту функцию для измерения различных реализаций. Для повышения точности измерения функтор вызывается несколько раз, конкретное количество итераций цикла задается в помощью глобальной переменной runTimes и в данный момент равняется 100. Время выполнения каждой итерации записывается в массив.

Над собранными данными проводится статистическая обработка. Вычисляются следующие параметры выборки:

- Математическое ожидание (из предположения о том, что данные описываются нормальным распределением)
- Среднеквадратичное отклонение
- Доверительный интервал

Сначала измеряется время выполнения последовательной реализации, после чего происходит замер времени выполнения реализаций на основе OpenMP и POSIX Threads. Результаты выводятся на консоль и сохраняются в файл.

Для первого замера воспользуемся компьютером с процессором Intel Core i5 4690 (Haswell, 4 ядра, 4 потока, 3,5 ГГц, двухканальная память DDR3 8 ГБ). Используется ОС Ubuntu 15.10 с ядром 4.2, компилятор GCC 5.2.1.

Рис. 1

Рис. 2

OpenMP:

Кол-во потоков	Время, мс	Margin	СКО	Ускорение
1	415.70	0.42	2.15	1.00
2	235.92	0.19	0.95	1.76
3	182.10	0.20	1.00	2.28
4	168.89	5.38	27.47	2.46
5	182.44	4.11	20.98	2.28
6	171.79	1.39	7.09	2.42
7	152.63	3.08	15.74	2.72
8	139.19	0.18	0.93	2.99

Pthreads:

Кол-во потоков	Время, мс	Margin	СКО	Ускорение
1	413.10	0.31	1.57	1.00
2	236.06	0.17	0.88	1.75
3	181.63	0.23	1.19	2.27

4	161.65	0.21	1.08	2.56
5	149.97	0.38	1.93	2.75
6	145.56	0.60	3.05	2.84
7	150.71	1.00	5.11	2.74
8	161.04	1.85	9.43	2.57

Рис. 3

Проведем аналогичный тест, но с использованием компилятора Clang 3.8.

Рис. 4

Рис. 5

Рис. 6

По сравнению с GCC немного увеличилась производительность, но соотношения остались примерно теми же.

Тестирование на машине с Core i3 5010u (Broadwell, 2 ядра, 4 потока, 2,1 GHz, одноканальная память DDR3 4 Гб) с аналогичным набором ПО:

Рис. 7

Рис. 8

Рис. 9

OpenMP:

Кол-во потоков	Время, мс	Margin	СКО	Ускорение
1	771.91	0.35	1.77	1.00
2	429.25	1.71	8.72	1.80
3	387.76	8.80	44.89	1.99
4	373.89	9.67	49.34	2.06
5	408.68	8.39	42.79	1.89
6	354.32	9.85	50.26	2.18
7	340.76	0.56	2.88	2.27
8	314.36	0.26	1.33	2.46

Pthreads:

Кол-во потоков	Время, мс	Margin	СКО	Ускорение
1	765.86	0.44	2.27	1.00
2	508.68	12.78	65.23	1.51
3	382.24	1.16	5.91	2.00
4	348.25	0.54	2.74	2.20
5	331.02	1.18	6.02	2.31
6	333.93	1.64	8.35	2.29
7	338.16	1.27	6.46	2.26
8	341.49	2.25	11.50	2.24

Что можно заметить:

- HyperThreading дает определенные результаты, скорость работы 4 ядер выше, чем 2. Таким образом, удается более эффективно использовать блоки ЦПУ и память.
- Производительность i3 в целом меньше в 1,5 раза по сравнению с i5 за счет более низкой частоты.
- OpenMP имеет проблемы при 5 потоках. Возможно это связано со сложностями планирования.
- При 8 потоках OpenMP быстрее Pthreads. Возможно это связано с тем, что OpenMP использует некоторый пул потоков, а в случае Pthreads потоки создаются каждый раз заново и на это требуется время

4. Выводы

- Основная идея, примененная при решении этой задачи - разбить задачу на несколько частей (блоков), обрабатывать задачи параллельно, а потом объединить результаты
- Для предоставление эксклюзивного доступа к разделяемому массиву в OpenMP были использованы критические секции, а в Pthreads - мьютексы.
- При увеличении количества потоков производительность растет не линейно - появляются некоторые затраты на синхронизацию. Кроме того, некоторые узлы компьютера (например, интерфейс к памяти) становятся «бутылочным горлышком» и препятствуют дальнейшему ускорению. Решение - переход к других архитектурам, например, NUMA, а также переработке программ.
- Для правильной оценки производительности нужно использоваться стат. обработку
- OpenMP значительно проще в программировании, но допускает меньше контроля и в некоторых ситуациях медленнее