Санкт-Петербургский политехнический Университет Петра Великого
Институт компьютерных наук и технологий
Кафедра компьютерных систем и программных технологий

Параллельные вычисления

Отчет по лабораторной работе

“Создание многопоточного приложения средствами OpenMP”

Работу выполнил:
Кисличенко Б.Д.
группа: 3540901/91502
Преподаватель:
к.т.н. Стручков И.В.

Санкт-Петербург
2020

Оглавление

1. Индивидуальное задание

Вариант 10. Определить частоту встречи слов в тексте на русском языке. Реализация на C++ с OpenMP.

2. Используемое окружение

Работа производилась с использованием средства виртуализации Parallels Desktop для MacOS. Версия Parallels Desktop: 15.1.2 (47123).

На виртуальной машине была установлена система с Ubuntu 18.

```
bogdan@bogdan-Parallels-Virtual-Platform:~$ uname -r
4.15.0-101-generic
bogdan@bogdan-Parallels-Virtual-Platform:~$ cat /etc/lsb-release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=18.04
DISTRIB_CODENAME=bionic
DISTRIB_DESCRIPTION="Ubuntu 18.04.4 LTS"
```

Листинг 1 Характеристики окружения

3. Обзор OpenMP

OpenMP – механизм написания параллельных программ для систем с общей памятью. Состоит из набора директив компилятора и библиотечных функций. Позволяет достаточно легко создавать многопоточные приложения на C/C++, Fortran.

Программирование производится путем вставки директив компилятора в ключевые места исходного кода программы. Компилятор интерпретирует эти директивы и вставляет в соответствующие места программы библиотечные вызовы для распараллеливания участков кода.

4. Алгоритм решения

Разработка велась на C++ с использованием OpenMP. Входной текст поступает на вход и разбивается на блоки в зависимости от количества используемых потоков. OpenMP раздает полученные блоки текста потокам, в которых соответственно происходит подсчет количества повторения слов.

Подсчет и вычленение слов было выполнено с помощью функции strtok, которая выполняет поиск лексем в строке string. Последовательность вызовов этой функции разбивает строку string на лексемы, которые представляют собой последовательности символов, разделенных символами разделителями.

На первый вызов, функция принимает строку string в качестве аргумента, чей первый символ используется в качестве начальной точки для поиска лексем. В последующие вызовы, функция ожидает нулевого указателя и использует позицию сразу после окончания последней лексемы как новое местонахождение для сканирования.

```
//подсчет слов путем разбияения текста на блоки
map<string, int> countWordsByBlocksOMP(const char* text, int
number) {
    size_t len = strlen(text);
    map<string, int> tmpMap;
    vector<size_t> indexes;

    //расчитываем размер блока (размер текста на кол-во
потоков)
    size_t blockSize = len/number+1;
    size_t blockStart = 0;
    size_t blockEnd;
    //для каждого блока сохраняем позиции начала и конца
    for (int i = 0; i < number; i++) {
        indexes.push_back(blockStart);
        blockEnd = blockStart + blockSize;

        //проверяем, чтобы блоки не разрезали слова на части,
поэтому
        //смещаем позцию конца блока до первого разделителя
```

```
        while (true) {
            if ((blockEnd > len) || (text[blockEnd] == ' '))
{
                break;
            }
            blockEnd++;
        }
        blockStart = blockEnd + 1;
        if (blockStart > len) {
            break;
        }
    }

    //для каждого блока производим подсчет результатов
    //после подсчета результаты объединяем в одну переменную
    #pragma omp parallel for num_threads(threadNum)
    for (int i = 0; i < number; i++) {
        blockStart = indexes[i];
        blockEnd = i == (number - 1) ? len : indexes[i + 1];

        //считаем слова в блоке текста
        auto localMap = countTextWords(text + blockStart,
blockEnd - blockStart);

        //объединяем результаты
        #pragma omp critical(critsec)
        for (auto& el : localMap) {
            int prevCount = tmpMap.count(el.first) ?
tmpMap[el.first] : 0;
            tmpMap[el.first] = prevCount + el.second;
        }
    }

    return tmpMap;
}
```

Листинг 2 подсчет слов путем разбиения текста на блоки

В данном методе идет разбиение на блоки. При этом делается проверка на то, чтобы при разбиении индекс позиционирования не оказался в каком-нибудь слове, поэтому индекс сдвигается до появления первого заданного разделителя.

В данном коде присутствует директива #pragma omp parallel for num_threads(threadNum), которая показывает, что цикл следует разделить по итерациям между потоками.

Наличие критической секции #pragma omp critical(critsec) в параллельном блоке гарантирует, что она в каждый конкретный момент времени будет выполняться только одним потоком. Т.е. когда один поток находится в критической секции, все остальные потоки, которые готовы в нее войти, находятся в приостановленном состояние.

```
//запуск одного раза OpenMP расчета
double runOneOMP(const char* text, int threadNumber){
     double startTime = omp_get_wtime();
     countWordsByBlocksOMP(text, threadNumber);
     double endTime = omp_get_wtime();
     double result = endTime - startTime;
     return result;
}
```

Листинг 3 Код получения времени за выполнение программы с OpenMP

```
//печать кол-ва слов в тексте по возростанию их встречи
void printSortingMap(map<string, int> map){
     set<pair<int, string>> s;

     for (auto const &kv : map)
         s.emplace(kv.second, kv.first);

     for (auto const &vk : s)
         std::cout  << vk.first  <<" "<< vk.second <<
std::endl;
}
```

Листинг 4 Вывод отсортированного итоговой map

5. Эксперименты

5.1. Проверка работы логики

```
Сибирь. На берегу широкой, пустынной реки стоит город, один из
административных центров России; в городе крепость, в крепости острог. В остроге
уже девять месяцев заключен ссыльнокаторжный второго разряда, Родион
Раскольников. Со дня преступления его прошло почти полтора года.

Судопроизводство по делу его прошло без больших затруднений.
```

Листинг 5 Исходный тестовый текст

```
…..
1пустынной
1разряда
1реки
1ссыльнокаторжный
1стоит
1уже
1центров
1широкой
2в
2его
2прошло

RUN FINISHED; exit value 0; real time: 0ms; user: 0ms; system: 0ms
```

Листинг 6 Пример работы программы

Видно, что программа отработала корректно.

В качестве входных данных использовался текст романа “Преступление и наказание”. Текст содержит около 1073334 символов (177027 слов) с пробелами.

5.2.Увеличение числа потоков для 2х ядерной ВМ

```
bogdan@bogdan-Parallels-Virtual-Platform:~$ cat /proc/cpuinfo|grep processor
processor       : 0
processor       : 1
```

число потоков	OpenMP Ср.время, мсек	Доверительный интервал (95%)	СКО	Дисперсия	Послед. Вып., мсек
1	371,35	371,35 ± 5,31	34,36	1180,76	372,61
2	312,63	312,63 ± 3,37	24,32	591,68	359,83
4	275,71	275,71 ± 4,58	33,09	1095,32	363,52
8	**253,31**	**253,31 ± 7,21**	36,80	1354,60	367,44
16	266,42	266,42 ± 5,10	26,04	678,23	378,61
32	266,54	266,54 ± 7,48	38,17	1457,14	375,71
64	272,19	272,19 ± 5,21	26,62	708,76	375,37
128	322,97	322,97 ± 7,57	38,66	1494,92	374,48
256	331,77	331,77 ± 7,41	37,81	1429,67	370,35

Таблица 1 Зависимость от количества потоков

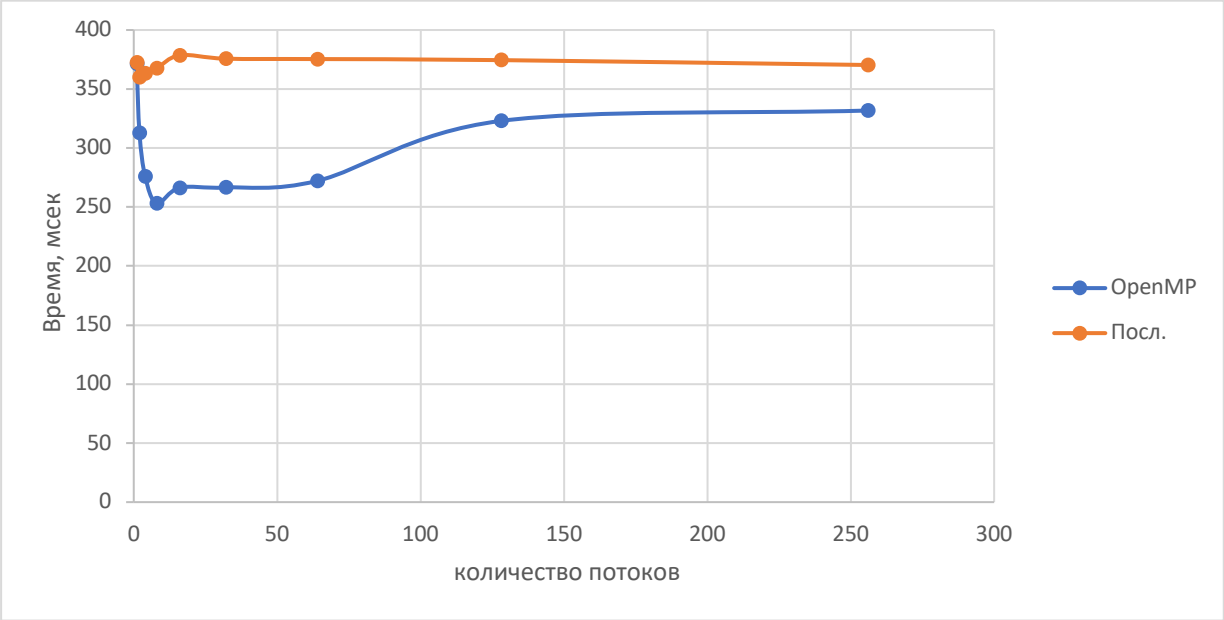

Рис. 1 Ось X - кол-во потоков, Ось Y - время выполнения программы.

Наилучшие показатели при 8 потоках. Лучший прирост производительности составил 1.45 раз.

5.3. Увеличение объемов данных для 2х ядерной ВМ

Для тестирования большими данными были создана соответствующие файлы, в которых роман повторяется 2, 4, 8 и 16 раз.

Количество потоков: 4

Количество слов в тексте: 177027 - 2832432.

Число слов в тексте	OpenMP Ср.время, мсек	Доверительный интервал (95%)	СКО	Дисперсия	Послед. Вып., мсек
177027	392,22	392,22 ± 4,58	33,09	1095,32	387,30
354054	574,19	574,19 ± 10,96	55,94	3130,29	770,08
708108	1010,53	1010,53 ± 18,16	92,68	8589,89	1605,08
1416216	1665,24	1665,24 ± 38,54	137,49	18903,35	3016,42
2832432	3253,34	3253,34 ± 78,44	200,21	40084,04	5840,99

Таблица 2 Зависимость времени выполнения от размера текста

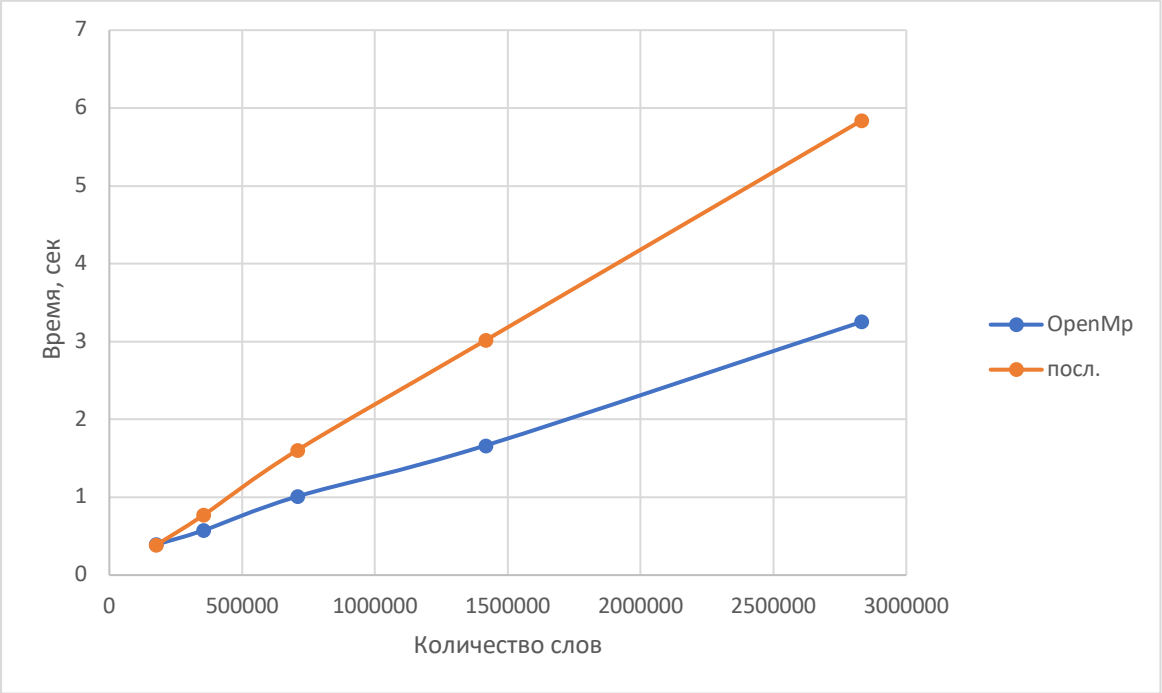

Рис. 2 Зависимость времени подсчета слов от размера входных данных. Ось Y - время выполнения программы, Ось X - кол-во слов во входных данных.

5.4. Увеличение числа потоков для 4х ядерной ВМ

```
bogdan@bogdan-Parallels-Virtual-Platform:~$ cat /proc/cpuinfo|grep processor
processor       : 0
processor       : 1
processor       : 2
processor       : 3
```

Таблица 3 Количество ядер виртуальной машины

Количество потоков: 1, 2, 4, 8, 16, 32, 64, 128 и 256

Количество слов в тексте: 177027.

число потоков	OpenMP Ср.время, мсек	Доверительный интервал (95%)	СКО	Дисперсия	Послед. Вып., мсек
1	368,49	368,49 ± 6,94	35,42	1254,68	366,72
2	217,25	217,25 ± 5,95	33,39	1115,43	369,74
4	177,28	177,28 ± 6,90	35,21	1239,74	370,90

8	**174,95**	**174,95 ± 5,40**	35,55	1264,34	362,67
16	207,86	207,862 ± 6,37	34,53	1192,68	364,37
32	231,16	231,16 ± 7,14	34,45	1186,87	375,54
64	242,65	242,6 ± 6,35	32,40	1049,87	371,38
128	266,12	266,12 ± 4,90	25,04	627,02	373,74
256	308,53	308,53 ± 7,20	36,77	1352,18	383,01

Таблица 4 Зависимость от количества потоков

Зависимость скорости выполнения (по оси Y) от числа потоков (ось X).

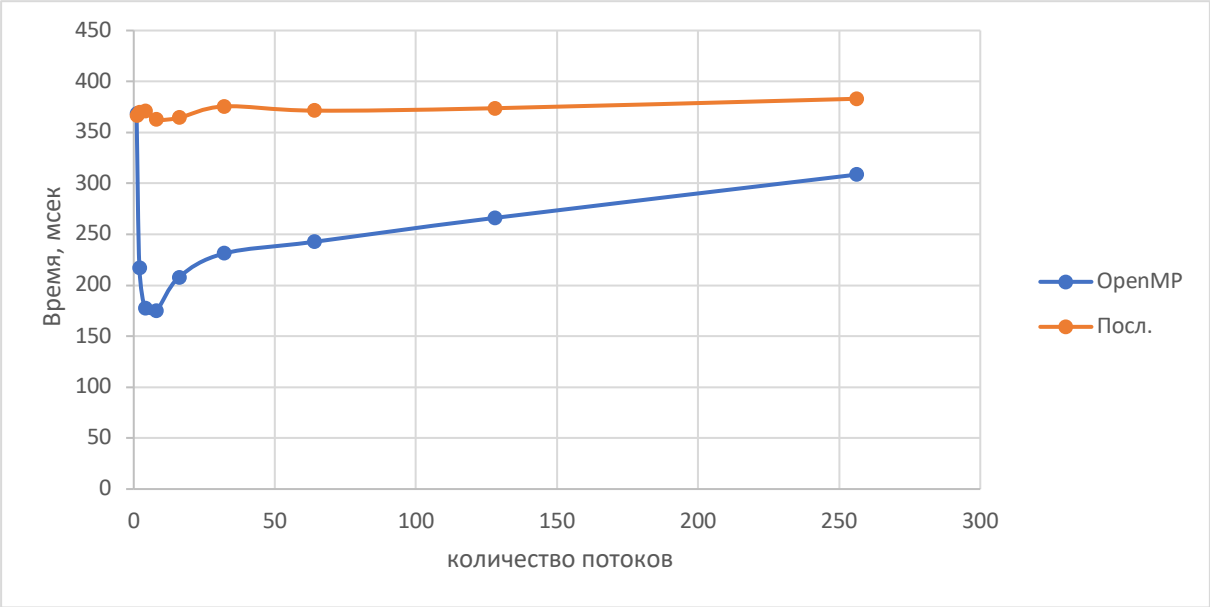

Рис. 3 Ось X - кол-во потоков, Ось Y - время выполнения программы

Из графика видно, что наилучшее время выполнения достигается при количестве потоков равном 4-8. Наилучшие показатели при 8 потоках. Прирост производительности оказался почти в 2 раза.

5.5. Увеличение объемов данных для 4х ядерной ВМ

Для тестирования большими данными были создана соответствующие файлы, в которых роман повторяется 2, 4, 8 и 16 раз.

Количество потоков: 4

Количество слов в тексте: 177027 - 2832432.

Число слов в тексте	OpenMP Ср.время, мсек	Доверительный интервал (95%)	СКО	Дисперсия	Послед. Вып., мсек
177027	193,04	193,04 ± 18,18	25,21	635,54	381,40
354054	375,64	375,64 ± 22,14	35,73	1276,97	781,07
708108	657,13	657,13 ± 23,96	38,67	1495,52	1584,36
1416216	1326,73	1326,73 ± 23,53	99,30	9861,88	2917,67
2832432	2074,74	2074,74 ± 54,46	310,55	96441,30	5744,76

Таблица 5 Зависимость времени выполнения от размера текста

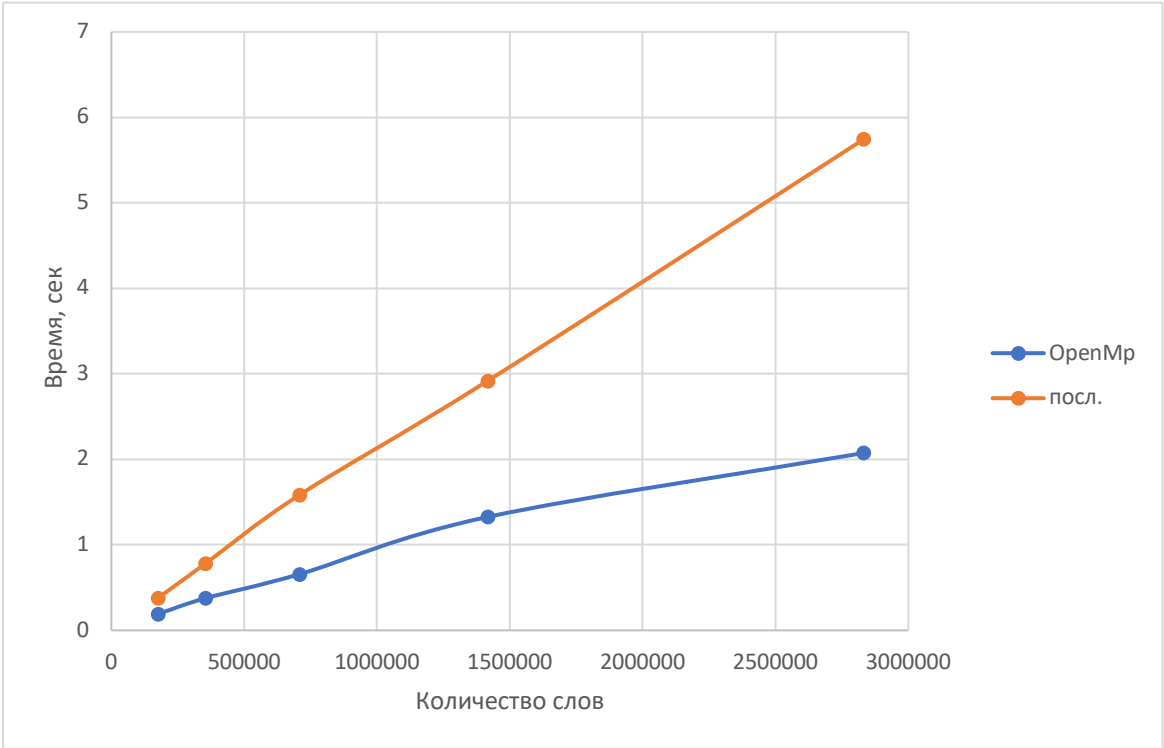

Рис. 4 Зависимость времени подсчета слов от размера входных данных. Ось Y - время выполнения программы, Ось X - кол-во слов во входных данных.

Если сравнивать рис.2 и рис.4, то можно видеть, что при работе на двухъядерном процессоре выполнение производилось примерно в 1.5 раза медленнее.

6. Выводы

В данной лабораторной работе был изучен инструмент распараллеливания программ с разделяемой памятью OpenMP. Добавление данного инструмента в код не составляет труда, нужно только правильно выделить части, которые могут быть распараллелены.

Результаты экспериментов показали, что лучшие показатели получаются при большем числе ядер и при небольшом количестве потоков, т.к. при кратном увеличении потоков кратно увеличиваются накладные расходы.

Лучший прирост производительности оказался примерно в 2 раза лучше при 4х-ядерном процессоре, в 1,45 раз лучше при 2х-ядерном процессоре. Прирост производительности зависит не только от количества ядер и потоков, но и, в соответствии с законом Амдала, от доли распараллеленного кода.