Отчет по курсовому проекту
По дисциплине "параллельные вычисления"

Евсеев Дмитрий

25 мая 2016 г.

Оглавление

Глава 1

Цель работы

1. Ознакомление с основными методами разработки параллельных программ на примере простых тестовых заданий
2. Анализ скорости выполнения кода, выполняющего одно задание с помощью различных библиотек

Глава 2

Ход работы

1. Реализация заданного алгоритма (расчет определителя матрицы) на языке C++
2. Тестирование программы на корректность получаемого ответа
3. Выделение в алгоритме частей, которые могут быть выполнятся параллельно
4. Реализовать параллельный алгоритм с помощью библиотеки *Pthreads*
5. Реализовать параллельный алгоритм с помощью библиотеки OpenMP
6. Проанализировать время выполнения различных реализация

Глава 3

Реализация

3.1 Последовательная программа

Первым шагом был реализован класс *myMatryx* < *typename T* >, реализующий матрицу, состоящую из элементов типа *T*. Класс содержит методы для считывания матрицы из текстового файла, заполнение матрицы случайными значениями. Реализация находится в файле *myMatrix.h*

Был реализован метод вычисления определителя матрицы, основанный на его определении - определитель является суммой по перестановкам всех дополнительных миноров строки или столбца матрицы. Т.е. был выбран алгоритм, который рекурсивно высчитывает определитель дополнительных миноров первой строки, пока размер полученного минора не станет равным единице.

Код последовательного расчета определителя матрицы:

```
T getDet() {
        T det = 0;

        if (size > 1) {
                for (auto i = 0; i <  size; i++) {
                        det += pow(-1, i) * getElement(0, i) *
                            getMinor(0 , i).getDet();
                }
        }
        else if (size == 1)
        det = getElement(0, 0);

        return det;
};
```

Тестирование на данном этапе происходило следующим образом: генерировались случайные матрицы размером до семи элементов и производилось вычисление определителя моей программой и онлайн-сервисом, предоставляющем данную возможность. Размер 7x7 был выбран, т.к. он максимален для найденных мной сервисов. Далее, тестирование производилось и для больших матриц, в этом случае сравнивались результаты, полученные разными методами.

3.2 Библиотека Pthreads

Алгоритм был распараллелен следующим образом: Определители для всех дополнительных миноров входной матрицы считаются параллельно в отдельных потоках. Для полученных миноров используется последовательный алгоритм из предыдущего пункта. Таким образом, было решено использовать N потоков для NxN матрицы.

Для того, чтобы функция-метод класса могла быть запущена в отдельном потоке должна быть:

1. Статической функцией
2. Возвращать тип *void*
3. Принимать на вход только один аргумент типа *void**

Было придумано следующее решение: была реализована структура, которую передаем как параметр в функцию, выполняющуюся в отдельном потоке. В ней содержится указатель на экземпляр класса и расположение в матрицы элемента, дополнительный минор которого надо обработать. Реализация:

```
struct MinorAdr{
        int x, y;
        void* mat;
};
```

Сама функция реализована следующим образом:

```
static void* workingFunc(void *args) {
        MinorAdr *arg = (MinorAdr*) args;
        T tmp;
        myMatrix< T >* mat = (myMatrix< T >*) arg->mat;
        tmp = pow(-1, arg->y) *  mat->getElement(arg->x, arg->y)
             * mat->getMinor(arg->x, arg->y).getDet();

        pthread_mutex_lock(&mat->lock);
        mat->temporaryDet += tmp;
        pthread_mutex_unlock(&mat->lock);
        return SUCCESS;
};
```

mat->temporaryDet - это член класса, в котором хранится промежуточный результат. К этому полю имеют доступ все потоки, поэтому я использую мьютекс для его защиты.

И сам метод, в котором создаются, запускаются, ожидаются потоки:

```
T getPthreadDet() {
        T det = 0;
        temporaryDet = 0;
        int status, status_addr;
        std::vector< MinorAdr > params;
        std::vector< pthread_t > thread;

        for (auto i = 0; i < size; i++){
                pthread_t tmp;
                thread.push_back(tmp);
```

```
                MinorAdr arg;
                arg.x = 0;
                arg.y = i;
                arg.mat = (void*) this;
                params.push_back(arg);
        }

        if (pthread_mutex_init(&lock, NULL) != 0) {
                printf("\n mutex init failed\n");
                exit(0);
        }

        for (auto i = 0; i < size; i++){
                status = pthread_create(&thread.at(i), NULL, &
                    workingFunc, (void*) &params.at(i));
                if (status != 0) {
                        printf("main error: can't create thread,
                             status = %d\n", status);
                        exit(0);
                }
        }

        for (auto i = 0; i < size; i++) {
                status = pthread_join(thread.at(i), (void**)&
                    status_addr);
                if (status != 0) {
                        printf("main error: can't join thread,
                            status = %d\n", status);
                        exit(0);
                }
        }

        pthread_mutex_destroy(&lock);
        det = temporaryDet;
        return det;
};
```

3.3 Библиотека OpenMP

Алгоритм работы программы остался прежним. Однако код стал намного проще благодаря синтаксису библиотеки. В коде указывается количество потоков, которое может быть создано. Для указания частей, кода, которые будут выполнятся параллельно используются препроцессорные директивы. Для синхронизации используется критическая секция.

Код программы:

```
T getOpenMPDet() {
        omp_set_num_threads(8);
        T det = 0;

        #pragma omp parallel
        {
                if (size > 1) {

                        #pragma omp for
                        for (auto i = 0; i <  size; i++) {
                                myMatrix< T > minor = getMinor(0
                                     , i);
```

```
                                T tmp = pow(-1, i) * getElement
                                    (0, i) * getStatDet(&minor);

                                #pragma omp critical
                                {
                                        det += tmp;
                                }
                        }
                }
                else if (size == 1)
                det = getElement(0, 0);
        }
        return det;
};
```

Глава 4

Анализ результатов

Для сравнения методов найдем среднее время нахождения определителя матрицы типа *long long int* размерностью 9x9 при 100 запусках разными методами (таблица 4.1).

Алгоритм работы	Время выполнения, мкс
Последовательный алгоритм	855524, delta = 11798
Pthreads	376735, delta = 12457
OpenMP 2 потока	576536, delta = 10647
OpenMP 4 потока	432814, delta = 11906
OpenMP 8 потоков	421768, delta = 14859

Таблица 4.1: Время выполнения различных вариантов алгоритма

Среднее время выполнения примерно одинаково для *Pthreads* и *OpenMP*, однако вторая библиотека несколько уступает по производительности. Исследование произведено на процессоре *Intel core i3* с двумя ядрами. Таким образом, время выполнения параллельной программы меньше примерно в два раза, чем программы последовательной.

Эксперименты были произведены в ОС *Linux*, в ней лучше сеья проявила библиотека *Pthreads*. *OpenMP*, в свою очередь, показала лучший прирост при работе в восьми потоках.

Глава 5

Вывод

В данной лабораторной работе была проведена разработка параллельных алгоритмов на основе последовательной программы для двух библиотек Pthreads и OpenMP. Реализация для двух библиотек существенна: библиотека OpenMP предоставляет более удобный способ организации параллельных программ за счет наличия готовых синтаксических конструкций параллельного выполнения в составе библиотеки. Реализации параллельного и последовательного алгоритмов отличаются для исходной задачи в нашем случае. Параллельная реализация имеет дополнительные вычисления. Введение параллелизма для последовательной программы возможно как на основе входных данных, так и на основе операций (команд). В полученной реализации содержится параллелизм на основе входных данных, когда параллельное выполнение осуществляется для разных и не связанных частей деревьев. Синхронизация оказала положительное влияние, но потребовала внесения дополнительных вычислений. В общем случае синхронизацию нужно вводить таким образом, чтобы синхронизируемый блок содержал как можно меньше операций. Параллельные программы оказались эффективнее последовательных. В общем случае параллельная реализация будет эффективнее, если правильно будет составлен алгоритм, и синхронизируемые блоки будут занимать меньше вычислительного времени по сравнению с общим потоком вычислений. На эффективность параллельной обработки влияют: - число ядер(аппаратных потоков) вычислительного устройства; - организация алгоритма вычислений; - разделение локальной и общей памяти потоков; - эффект от введенной синхронизации (если она необходима).