Отчет о выполнении 1 задания практикума кафедры СКИ

Р.М. Куприй, 423 группа

Факультет ВМК МГУ имени М.В. Ломоносова

1. Задание

Задание состоит в освящении проблемы снижения производительности в результате многократного порождения и уничтожения OpenMP нитей. Хотя накладные расходы и ресурсы для создания и уничтожения нитей обычно крайне малы в сравнении с объемом проводимых вычислений, в некоторых случаях это может привести к замедлению некоторой области вычислительного кода. В качестве примера рассматривается многократное умножение плотной матрицы на вектор: $y = y + A * x$. Представлено два варианта распараллеливания гнезда циклов: с созданием и уничтожением нитей для выполнения каждого умножения и с созданием паралельной области, в которой выполняются последовательные призведения.

2. Реализация оптимизации

В первом варианте программы 1 в рассматриваемой проблемной области есть внешний цикл, итерации которого выполняются последовательно. Каждая итерация соответствует накапливанию результата умножения матрицы на вектор: двойной цикл `for`, который распараллелен соответствующей прагмой `#pragma omp parallel for`. В результате такого варианта параллелизации OpenMP нити создаются и уничтожаются на каждой итерации.

Во втором варианте программы 2 OpenMP нити создаются на входе в параллельную область `#pragma omp parallel`. Таким образом, каждая итерация внешнего цикла выполняется каждой нитью. Для распараллеливания внутренних циклов применяется соответствующая инструкция `#pramga omp for`. Такое использование OpenMP инструкций позволяет не порождать и уничтожать нити на каждой итерации, а использовать имеющиеся нити в параллельной области.

```
for (it = 0; it < niters; it++) {
#pragma omp parallel for shared(val_ptr, x_ptr, y_ptr, size) private(i,
    j) schedule(static)
  for (i = 0; i < size; i++) {
    for (j = 0; j < size; j++) {
      y_ptr[i] += val_ptr[i * size + j] * x_ptr[j];
    }
  }
}
```

Рис. 1. Базовый вариант программы

```
#pragma omp parallel shared(val_ptr, x_ptr, y_ptr, size, niters) private
    (i, j, it)
{
for (it = 0; it < niters; it++) {
#pragma omp for schedule(static)
  for (i = 0; i < size; i++) {
    for (j = 0; j < size; j++) {
      y_ptr[i] += val_ptr[i * size + j] * x_ptr[j];
    }
  }
}
}
```

Рис. 2. Оптимизированный вариант программы

3. Методика тестирования

Для оценки эффективности оптимизированного варианта программы сравнивается общее время вычисления многократного умножения матрицы на вектор. Размер матрицы и число итераций выбраны для большей наглядности так, что число итерации много больше чем размер матрицы.

Для тестирования производительности использовалась параллельная вычислительная система Polus, с 3 вычислительными узлами, в каждом из которых 2 10 ядерных процессора IBM POWER8. Для компиляции использовался компилятор `xlc++` с соответствующими флагами опимизации: `-O5 -qsmp=omp -qarch=pwr8`.

Для замеров использовался один узел кластера. Для исключения выбросов и получения равномерной оценки по запускам, каждая версия программы запускалась 6 раз, среди всех запусков выбиралось минимальное время работы. Для привязки нитей к ядрам использовался вспомогательный предоставленный скрипт, а число ядер задавалось строкой `#BSUB -R "affinity[core(x)]`. Далее представлены результаты для плотной квадратной матрицы со стороной 100 и для числа итераций 10^7 и для матрицы размером 500 и числом итераций $2 \cdot 10^6$.

4. Оценки эффективности OpenMP программ

Для рассматриваемой проблемы ожидается нарастание проблемы с масштабируемостью, поскольку при росте числа нитей увеличиваются накладные расходы на их создание и уничтожение. Это подтверждается проведенными экспериментами и наглядно видно в таблицах 1, 2. В таблицах приведено общее время выполнения всех итераций, для базового и для оптимизированного варианта программы соответственно, а также значение ускорения времени расчётов для улучшенной версии программы.

5. Анализ полученных результатов

В результате оптимизации способа использования OpenMP параллелизма, вычисления ускоряются в среднем на 20% для первой конфигурации и на 3% для второй конфигурации. Такая значительная разница в получаемом ускорении обусловлена тем, что объем вычислений для плотной матрицы размером 500, в 25 раз больше, чем объем вычислений для матрицы размером 100 строк и столбцов. Поэтому, вклад

Таблица 1. Сравнение времени выполнения базовой (base time) и оптимизированной (optimized time) программы для матрицы размерности 100 и числом итераций 10^7

nthreads	ncores	base time, sec	optimized time, sec	speedup
1	1	219	211	3.6%
2	2	121	109	11.3%
4	4	73	62	16.8%
8	8	68	59	15.8%
16	8	82	70	16.8%
16	16	77	62	23.8%
32	8	78	75	3.7%
32	16	80	66	20.7%
64	8	74	48	55.4%
64	16	72	52	38.4%

Таблица 2. Сравнение времени выполнения базовой (base time) и оптимизированной (optimized time) программы для матрицы размерности 500 и числом итераций $2 \cdot 10^6$

nthreads	ncores	base time, sec	optimized time, sec	speedup
1	1	1068	1066	0.2%
2	2	537	534	0.6%
4	4	274	271	1%
8	8	142	139	2.2%
16	8	78	74	4.3%
16	16	79	76	3.6%
32	8	91	89	3.1%
32	16	50	46	8.4%
64	8	252	236	6.6%
64	16	92	89	4.1%
128	8	553	543	1.9%
128	16	320	311	3%

времени на создание и уничтожение нитей на каждой итерации пропорционально тем меньше, чем больше размер матрицы и соответственно, объем вычислений в одной итерации.

Применение данной оптимизации не решает основную проблему масштабирования, лишь убирает конкретную специфичную проблему с накладными расходами на создание и уничтожение нитей. Однако, базовые ограничения, например, нехватка слотов для выполнения операций с плавающей точкой или перегрузка шины памяти – остаются неизменными и определяют характер параллелизма для конкретной задачи.