Отчет

по лабораторной работе №5 Параллелизм задач

Выполнил:
Пеалкиви Даниил Яковлевич, гр. ИС-242

Задание

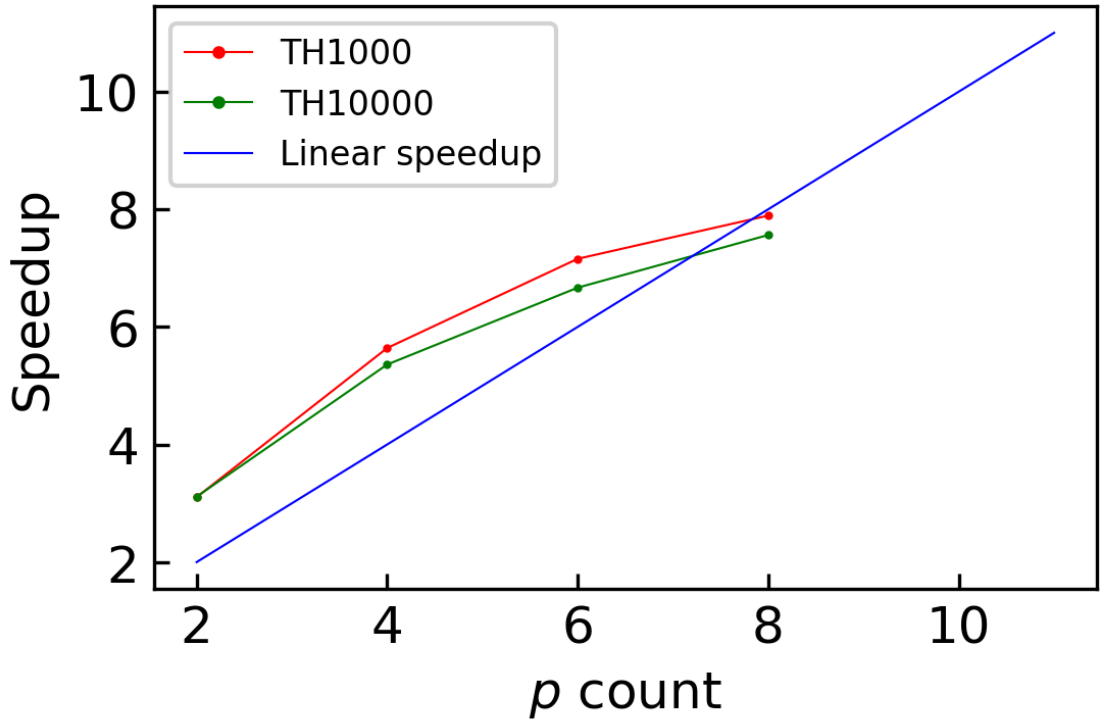
Задание

- На базе директив `#pragma omp task` реализовать многопоточный рекурсивный алгоритм быстрой сортировки (QuickSort). Опорным выбирать центральный элемент подмассива (функция `partition`, см. слайды к лекции). При достижении подмассивами размеров `THREASHOLD = 1000` элементов переключаться на последовательную версию алгоритма.
- Выполнить анализ масштабируемости алгоритма для различного числа сортируемых элементов и порогового значения `THRESHOLD`.

Защита работы

1. Продемонстрировать код программы и графики ускорения
2. Описать суть распараллеливания алгоритма
3. Охарактеризовать эффективность созданной параллельной программы

График



Характеристики процессора:

- Процессор: Intel Core i5-12400F
- Количество ядер: 6 физических ядер (12 потоков)
- Тактовая частота: 2.5 GHz (до 4.4 GHz в режиме Turbo Boost)
- Кэш: L1: 0.470, L2: 7.5, L3: 18 MB Intel Smart Cache

Описание функций

Параллельная реализация сортировки quicksort находится в функции **quicksort_tasks**.

```
void quicksort_tasks(int *v, int low, int high)
{
    int i, j;
    partition(v, &i, &j, low, high);
    if (high - low < THRESHOLD || (j - low < THRESHOLD || high - i < THRESHOLD))
    {
        if (low < j)
            quicksort_tasks(v, low, j);
        if (i < high)
            quicksort_tasks(v, i, high);
    }
    else
    {
#pragma omp task untied
        {
            quicksort_tasks(v, low, j);
        }
        quicksort_tasks(v, i, high);
    }
}
```

1. Создание задач:
   Используется директива **#pragma omp task** для создания параллельной задачи. Это позволяет функции **quicksort_tasks** запускать сортировку подмассива в отдельном потоке.

2. Откреплление задачи:
   Директива **untied** указывает, что задача не привязана к конкретному потоку и может быть выполнена любым доступным потоком.

3. Пороговая величина:
   Для подмассивов, размер которых меньше заданного порога **THRESHOLD,** сортировка выполняется последовательно. Это позволяет избежать создания слишком большого количества задач и снижает накладные расходы на управление потоками.

```c
void swap(int *x, int *y)
{
    int tmp = *x;
    *x = *y;
    *y = tmp;
}
```

Функция для обмена значениями двух целочисленных переменных.

```c
double wtime()
{
    struct timeval t;
    gettimeofday(&t, NULL);
    return (double)t.tv_sec + (double)t.tv_usec * 1E-6;
}
```

Функция для измерения времени с использованием функции **gettimeofday()** для получения текущего времени в секундах с точностью до микросекунд.

```c
void partition(int *v, int *i, int *j, int low, int high)
{
    *i = low;
    *j = high;
    int pivot = v[(low + high) / 2];
    do
    {
        while (v[*i] < pivot)
            (*i)++;
        while (v[*j] > pivot)
            (*j)--;
        if (*i <= *j)
        {
            swap(&(v[*i]), &(v[*j]));
            (*i)++;
            (*j)--;
        }
    } while (*i <= *j);
}
```

Функция разделения массива на две части относительно опорного элемента (pivot). Реализует один шаг алгоритма быстрой сортировки, определяя новое местоположение опорного элемента.

```c
void quicksort(int *v, int low, int high)
{
    int i, j;
    partition(v, &i, &j, low, high);
    if (low < j)
        quicksort(v, low, j);
    if (i < high)
        quicksort(v, i, high);
}
```

Функция **partition** выбирает элемент, называемый опорным элементом (**pivot**), и переставляет элементы массива таким образом, что все элементы, меньшие опорного, перемещаются влево от него, а все элементы, большие опорного, перемещаются вправо. Индексы i и j указывают на границы разделения.

```c
void init(int **arr)
{
    for (int i = 0; i < N; i++)
        (*arr)[i] = rand() % 100;
}
```

Функция инициализации массива случайными значениями от 0 до 99.

```c
void print_arr(int *arr)
{
    for (int i = 0; i < N; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
```

Функция для печати массива на экран.

Основная функция программы. В начале инициализируется массив arr случайными значениями. Затем измеряется время выполнения последовательной версии алгоритма быстрой сортировки и выводится на экран. Затем запускается цикл, в котором параллельная версия алгоритма быстрой сортировки выполняется с разным числом потоков (от 2 до 8). Для каждого числа потоков измеряется время выполнения параллельной версии и выводится на экран, а также вычисляется ускорение (speedup) относительно последовательной версии.

Директива **#pragma omp parallel**

Эта директива создает параллельный регион, в котором создаются i потоки (указанные параметром num_threads(i)). Внутри этого параллельного региона все потоки выполняют код, который находится в его теле.

Директива **#pragma omp single**

Эта директива гарантирует, что следующую область кода (в данном случае, вызов функции **quicksort_tasks**) выполнит только один поток из всех, созданных в параллельном регионе. Другие потоки будут ожидать завершения выполнения этой области.

Работа программы

Threshold - 1000

```
itzavangard@DESKTOP-83TNIRU:~/lab5$ gcc quicksort.c -fopenmp -lm
itzavangard@DESKTOP-83TNIRU:~/lab5$ ./a.out
85.072172 - время последовательной программы
время работы параллельной программы - 27.835480, потоков - 2  speedup:3.056250
время работы параллельной программы - 15.370601, потоков - 4  speedup:5.534733
время работы параллельной программы - 12.061561, потоков - 6  speedup:7.053164
время работы параллельной программы - 10.700575, потоков - 8  speedup:7.950243
```

Threshold - 10000

```
itzavangard@DESKTOP-83TNIRU:~/lab5$ gcc quicksort.c -fopenmp -lm
itzavangard@DESKTOP-83TNIRU:~/lab5$ ./a.out
88.288467 - время последовательной программы
время работы параллельной программы - 29.210534, потоков - 2  speedup:3.022487
время работы параллельной программы - 16.884590, потоков - 4  speedup:5.228938
время работы параллельной программы - 11.707037, потоков - 6  speedup:7.541487
время работы параллельной программы - 10.655688, потоков - 8  speedup:8.285572
```

Вывод

Алгоритм быстрой сортировки (**quicksort**) является одним из алгоритмов, которые могут быть разделены на независимые подзадачи. В его параллельной версии массив разбивается на части, которые затем сортируются одновременно с использованием нескольких потоков. График показывает, что параллельная версия алгоритма демонстрирует хорошее ускорение по сравнению с последовательной версией.