Федеральное государственное бюджетное образовательное учреждение высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»
(СибГУТИ)

Кафедра вычислительных систем

ОТЧЕТ
по лабораторной работе номер 5
по дисциплине «**Параллельные вычислительные технологии (ПВТ)**»

Выполнил:
студент гр. ИВ-221
«__» мая 2024 г. _____ Бойваленко Н. Е.

Проверил:
преподаватель
«__» мая 2024 г. _____ Челканова Т. В.

Новосибирск 2024

ОГЛАВЛЕНИЕ

Введение

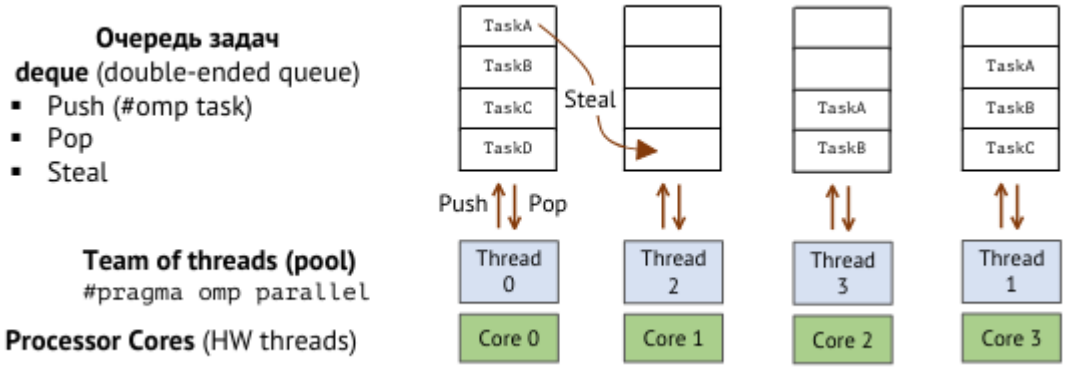
```
#pragma omp parallel
{
        #pragma omp sections
        {
                #pragma omp section
                {
                        // Section 1
                }

                #pragma omp section
                {
                        // Section 2
                }

                #pragma omp section
                {
                        // Section 3
                }

        } // barrier
}
```

pragma omp parallel порождает пул потоков (team of threads) и набор задач (set of tasks).
Каждая секция выполняется отдельным потоком в контексте некоторой задачи.
Не гарантируется, что все секции будут выполняться разными потоками Один поток может выполнить несколько секций



У каждого потока поддерживается двухсторонняя очередь (deque).
Каждый поток имеет двустороннюю очередь (дек) о с нижнего конца поток добавляет новые задачи (#pragma omp task, push) о с нижнего конца поток извлекает задачи для выполнения (pop).
Если некоторый поток выполнил все задачи, которые у него были в очереди, то он генерирует псевдослучайное число (номер потока) и с верхнего конца извлекает задачу для выполнения.

Задача

1) 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.
2) Выполнить анализ масштабируемости алгоритма для различного числа сортируемых элементов и порогового значения THRESHOLD.

Ход работы

Функция getRand() возвращает псевдослучайное число

```
int getRand()
{
    return rand();
}
```

Функция omp_get_wtime() возвращает текущее время. Необходима для замера времени работы алгоритма.

```
double
omp_get_wtime()
{
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return ts.tv_sec + ts.tv_nsec * 1E-9;
}
```

Функция swap() меняет местами переменные. Используется в quickSort.

```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

В начале функция partition() выбирает опорный элемент pivot, путем деления массива пополам. Затем все элементы, которые меньше pivot переносятся влево, а которые больше - вправо

```
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 quickSort обычная последовательная сортировка. Рекурсивно обходит левую и правую часть массива, тем самым формируя отсортированный массив. Нужная для сравнения с распараллеленной версией.

```
void quicksort(int *v, int low, int high)
{
    int i = 0;
    int j = 0;
    partition(v, &i, &j, low, high);
    if (low < j)
        quicksort(v, low, j);
    if (i < high)
        quicksort(v, i, high);
}
```

Функция quicksort_tasks() является распараллеленной версией обычной сортировки quicksort. Если подзадачи большие, то левый подмассив обрабатывается в отдельной подзадачи, а правый в текущей задачи. Если количество элементов дойдет до порогового значения treadshold, то программа начнет выполняться последовательно. Treadshold подбирается экспериментально.

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

В функции main с помощью деректривы pragma omp parallel создается параллельный регион. pragma omp single указывает, что код выполняется одним потоком, а уже в функции quicksort_tasks() происходит создание задач, которые распределяются между всеми потоками(steal).

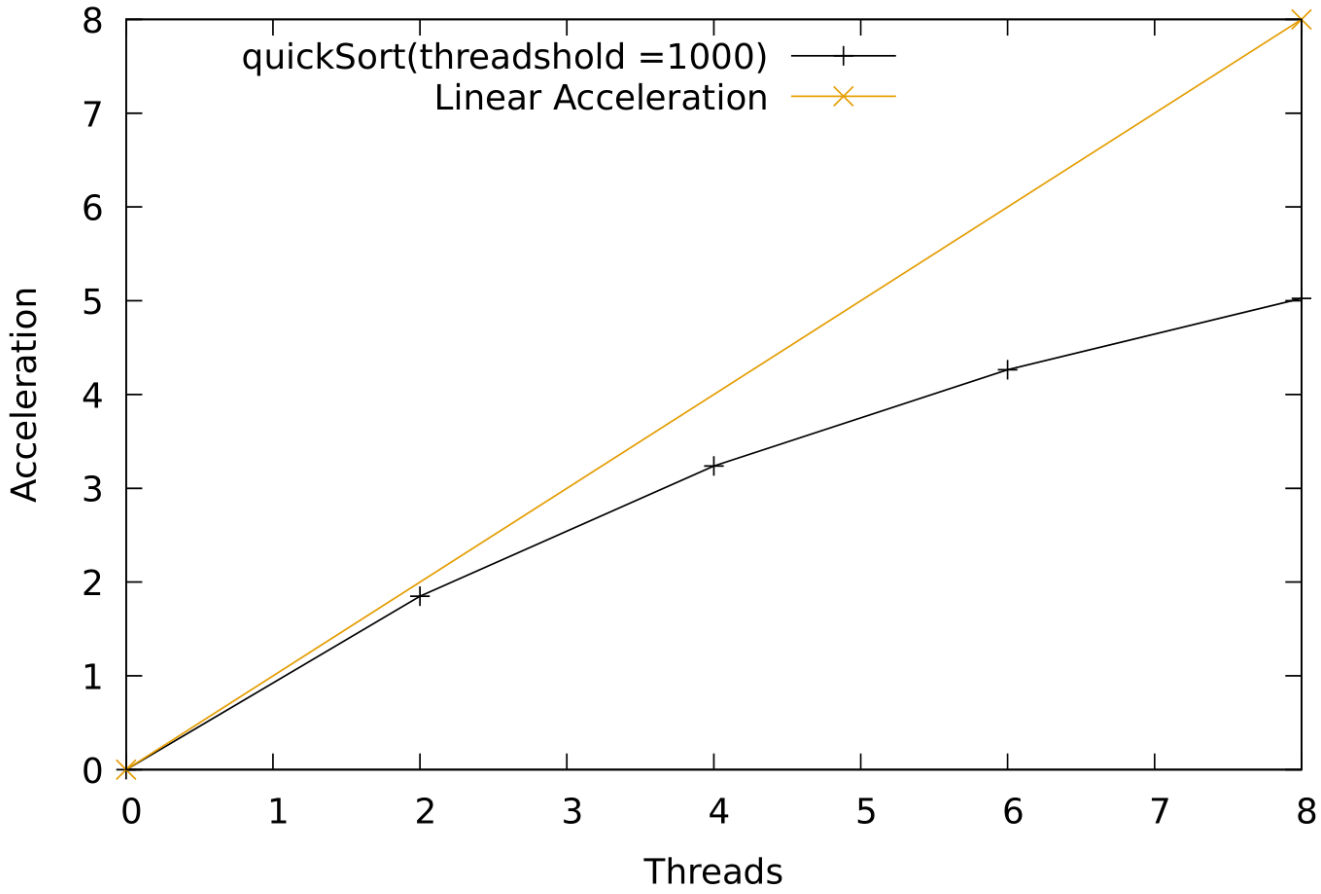
```
    for (int i = 2; i <= 8; i += 2)
    {
        for (int i = 0; i < n; i++)
        {
            array[i] = getRand();
        }
        parallelTime = omp_get_wtime();
#pragma omp parallel num_threads(i)
        {
#pragma omp single
            quicksort_tasks(array, 0, n - 1);
        }
        parallelTime = omp_get_wtime() - parallelTime;
        printf("Tasks time: %.6f\n", parallelTime);
        fprintf(file, "%d   %f\n", i, serialTime / parallelTime);
    }
```

Итог

Результат выполнения программы, при n = 1000000 (количество элементов в массиве).

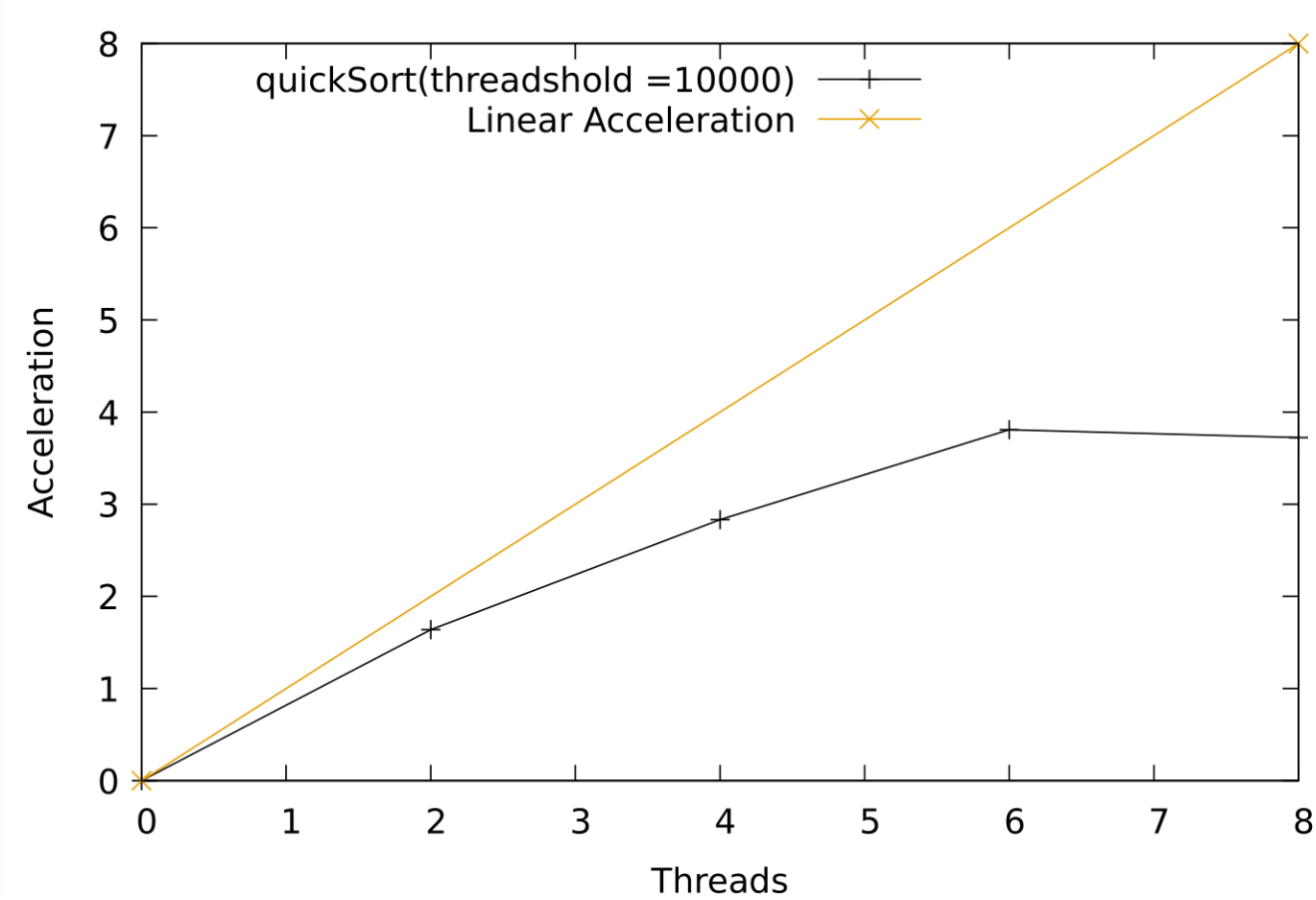
1) threadshold = 1000

```
Non parallel time: 0.142338
Tasks time: 0.069431
Tasks time: 0.039583
Tasks time: 0.031395
Tasks time: 0.026349
major@Major:~/study/PVT/pct-spring-lab5$
```



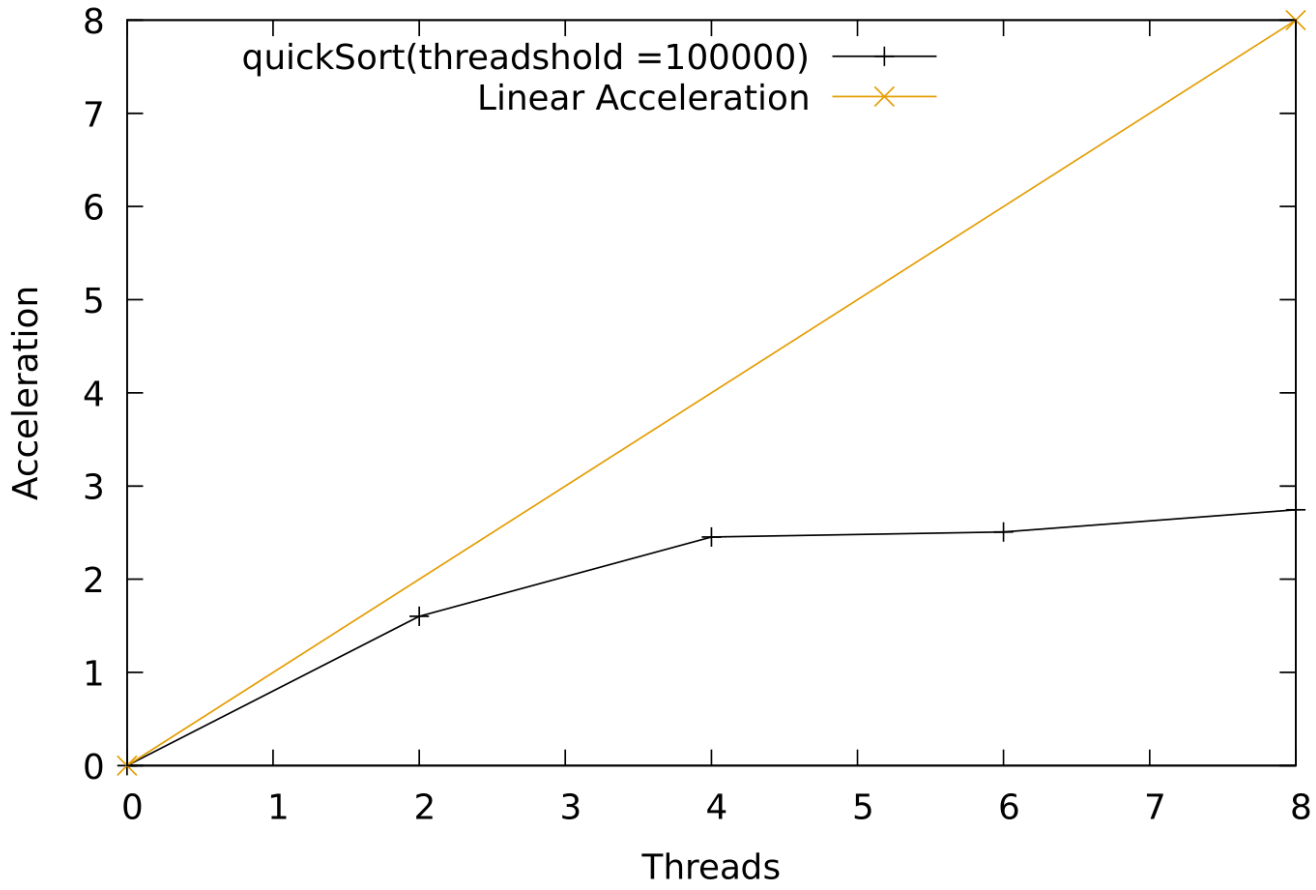
2) threadshold = 10000

```
Non parallel time: 0.127490
Tasks time: 0.071272
Tasks time: 0.040690
Tasks time: 0.034117
Tasks time: 0.028422
major@Major:~/study/PVT/pct-spring-lab5$
```



3) threadshold = 100000

```
Non parallel time: 0.125879
Tasks time: 0.076147
Tasks time: 0.048229
Tasks time: 0.056571
Tasks time: 0.066126
major@Major:~/study/PVT/pct-spring-lab5$
```



Таким образом, при увеличении threadshold эффективность алгоритма падает, ввиду того, что часть времени сортировка работает в последовательном режиме.

Приложение

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int threadshold = 100000;
int getRand()
{
return rand();
}
double
omp_get_wtime()
{
struct timespec ts;
clock_gettime(CLOCK_MONOTONIC, &ts);
return ts.tv_sec + ts.tv_nsec * 1E-9;
}
void swap(int *a, int *b)
{
int tmp = *a;
*a = *b;
*b = tmp;
}
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);
}
void quicksort(int *v, int low, int high)
{
int i = 0;
int j = 0;
partition(v, &i, &j, low, high);
if (low < j)
quicksort(v, low, j);
if (i < high)
quicksort(v, i, high);
}
```

```
52. void quicksort_tasks(int *v, int low, int high)
53. {
54. int i, j;
55. partition(v, &i, &j, low, high);
56. if (high - low < threadshold || (j - low < threadshold || high - i
    < threadshold))
57. {
58. if (low < j)
59. quicksort_tasks(v, low, j);
60. if (i < high)
61. quicksort_tasks(v, i, high);
62. }
63. else
64. {
65. #pragma omp task
66. {
67. quicksort_tasks(v, low, j);
68. }
69. quicksort_tasks(v, i, high);
70. }
71. }
72. int main()
73. {
74. srand(time(NULL));
75. FILE *file;
76. file = fopen("quickSort.dat", "w");
77. double serialTime = 0;
78. double parallelTime = 0;
79. int n = 1000000;
80. int *array = malloc(sizeof(int) * n);
81. for (int i = 0; i < n; i++)
82. {
83. array[i] = getRand();
84. }
85. serialTime = omp_get_wtime();
86. quicksort(array, 0, n - 1);
87. serialTime = omp_get_wtime() - serialTime;
88. printf("Non parallel time: %.6f\n", serialTime);
89. for (int i = 2; i <= 8; i += 2)
90. {
91. for (int i = 0; i < n; i++)
92. {
93. array[i] = getRand();
94. }
95. parallelTime = omp_get_wtime();
96. #pragma omp parallel num_threads(i)
97. {
98. #pragma omp single
99. quicksort_tasks(array, 0, n - 1);
100.     }
101.     parallelTime = omp_get_wtime() - parallelTime;
102.     printf("Tasks time: %.6f\n", parallelTime);
103.     fprintf(file, "%d %f\n", i, serialTime / parallelTime);
104.     }
```

```
105.      fclose(file);
106.      free(array);
107.      return 0;
108.      }
```

Список литературы

- Шамим Эхтер, Джейсон Робертс. Многоядерное программирование. - СПб.: Питер, 2010.
- Maurice Herlihy, Nir Shavit. The Art of Multiprocessor Programming, Morgan Kaufmann, 2012
- Эндрюс Г. Основы многопоточного, параллельного и распределенного программирования. - М.: Вильямс, 2003.
- Расс Миллер, Лоренс Боксер. Последовательные и параллельные алгоритмы. - М.: Бином, 2009
- Уильямс Э. Параллельное программирование на C++ в действии. Практика разработки многопоточных программ. - М.: ДМК Пресс, 2012.