|  |  |  |
| --- | --- | --- |
| Лабораторная работа №4  Task-based парралелизм в OpenMP | Выполнил | Доценко Е.В. |
| Группа | САПР 2.1 |
| Преподаватель | Катаев А.В. |
| Дата |  |
| Оценка |  |

Цель: Изучение работы с библиотекой OpenMP на уровне задач.

Язык: С++.

Задание:

Реализовать параллельное вычисление рекурсивного алгоритма с использованием библиотеки OpenMP. Рекомендуется использовать Section или Task для реализации параллелизма, желательно сбалансировать вычисления между потоками, но не делать слишком мелких задач.

Замерить время работы последовательной и параллельной версий алгоритма, добиться ускорения параллельной версии.

Данные генерировать псевдо-рандомно (с использованием зерна) или читать из файла.

Выбранное задание – QuickSort или быстрая сортировка. Суть алгоритма заключается в разделении массива на два подмассива, средней линией считается элемент, который находится в самом центре массива. В ходе работы алгоритма элементы, меньшие чем средний будут перемещены в лево, а большие в право. Такое же действие будет происходить рекурсивно и с подмассива, они будут разделяться на еще два подмассива до тех пор, пока не будет чего разделать (останется один элемент). На выходе получим отсортированный массив.

Сложность алгоритма зависит от входных данных и в лучшем случае будет равняться O(n×2log2n). В худшем случае O(n2).

Далее посмотрим разные ситуации.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № | Кол-во элементов | Последовательная | Распараллеленная | | |
| 2 потока | 3 потока | 4 потока |
| 1 | 10 000 | 0,0005252 | 0,0004198 | 0,000421 | 0,0004208 |
| 2 | 1 000 000 | 0,0423664 | 0,0399445 | 0,058048 | 0,0584309 |
| 3 | 10 000 000 | 0,91013 | 0,521567 | 0,419526 | 0,423247 |

Для анализа были выбраны 3 разные ситуации: маленькое количество элементов(10 000), среднее количество элементов(1 000 000) и большое количество элементов(10 000 000) при диапазоне [0, 1000].

Рисунок 1 – График времени вычисления на 10 000 000 элементов

Из первого случая мы видим, что ускорения даже на 2-х потоках минимальное. Дальше с ростом количества элементов в массиве и с ростом числа потоков ускорение становиться более заметно. Так, при 10 000 00 элементов можно добиться ускорения в 2 раза на 4 потоках.

Вывод: Выполнив данную работу, я изучила принципы работы с библиотекой OpenMP на уровне задач и реализовала быструю сортировку с помощью task.

Код программы

#include <iostream>

#include <omp.h>

#include <algorithm>

#include <stdio.h>

#include <time.h>

#include<stdio.h>

#include<math.h>

#include <stdlib.h>

#include <thread>

using namespace std;

bool Equal(int\* sequential, int\* parallel, int length);

void partition(int\* v, int& i, int& j, int array\_size);

void quicksort(int\* v, int array\_size);

void quicksort\_parallel(int\* v, int array\_size);

int main()

{

setlocale(LC\_ALL, "rus");

int N = 100000000; //количество символов во входном массиве

int\* original = new int[N];

int\* result = new int[N];

srand(time(0));

//генерация данных во входном массиве

for (int i = 0; i < N; i++)

{

original[i] = rand() % 10000 + 1;

}

for (int i = 0; i < N; i++)

{

result[i] = original[i];

}

double start\_time, end\_time;

// Последовательная сортировка

start\_time = omp\_get\_wtime();

quicksort(result, N);

end\_time = omp\_get\_wtime();

double time = end\_time - start\_time;

cout << "Время последовательной сортировки: " << time << endl;

// Параллельная сортировка

for (int numThreads = 2; numThreads <= std::thread::hardware\_concurrency(); numThreads++) {

int\* par\_result = new int[N];

for (int i = 0; i < N; i++)

{

par\_result[i] = original[i];

}

start\_time = omp\_get\_wtime();

omp\_set\_dynamic(0);

omp\_set\_num\_threads(numThreads);

#pragma omp parallel

{

#pragma omp single

quicksort\_parallel(par\_result, N);

}

end\_time = omp\_get\_wtime();

time = end\_time - start\_time;

cout << "Время параллельной сортировки на "<< numThreads << " потоках: " << time << endl;

// Сравнение результатов

if (Equal(result, par\_result, N))

cout << "Массивы совпадают." << endl;

else

cout << "Массивы не совпадают" << endl;

delete[] par\_result;

}

delete[] result;

delete[] original;

}

bool Equal(int\* sequential, int\* parallel, int length)

{

for (int i = 0; i < length; i++)

if (sequential[i] != parallel[i])

return false;

return true;

}

void partition(int\* v, int& i, int& j, int array\_size) {

i = 0;

j = array\_size - 1;

int pivot = v[array\_size / 2];

do {

while (v[i] < pivot) i++;

while (v[j] > pivot) j--;

if (i <= j) {

std::swap(v[i], v[j]);

i++;

j--;

}

} while (i <= j);

}

void quicksort(int\* v, int array\_size) {

int i, j;

partition(v, i, j, array\_size);

if (0 < j)

quicksort(v, j + 1);

if (i < array\_size)

quicksort(v + i, array\_size - i);

}

void quicksort\_parallel(int\* v, int array\_size) {

int i, j;

partition(v, i, j, array\_size);

{

if (array\_size / 2 < 50000) {

#pragma omp task shared(v)

if (j > 0)

quicksort(v, j + 1);

#pragma omp task shared(v)

if (array\_size > i)

quicksort(v + i, array\_size - i);

}

else {

#pragma omp task shared(v)

if (j > 0)

quicksort\_parallel(v, j + 1);

#pragma omp task shared(v)

if (array\_size > i)

quicksort\_parallel(v + i, array\_size - i);

}

#pragma omp taskwait

}

}