Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и технологий

Кафедра компьютерных систем и программных технологий

**Отчет о курсовом проекте**

**Дисциплина**: Параллельные вычисления

Выполнил студент гр. 13541/2 Котелевский А.С.

(подпись)

Руководитель Стручков И.В.

(подпись)

“ ” 2018 г.

Санкт–Петербург

2018

# Задание

В рамках курсового проекта студент выбирает один из алгоритмов, для которого выполняется:

1) разработка алгоритма

2) создание последовательной программы, реализующей алгоритм

3) создание программы, использующей POSIX-потоки

4) создание программы, использующей OpenMP

5) сравнение результатов работы

Мною было выбрано задание №6: Вычислить сумму ключей в узлах двоичного дерева.

# Ход работы

В ходе выполнения работы было выявлено, что применение многопоточного алгоритма не дает существенного выигрыша. Это происходит из-за того, что обращение к памяти оказывается затратнее самих вычислений. Поэтому исходная задача была переформулирована: вместо простой суммы будем искать сумму sqrt(abs(sin(left)\*sin(left) + cos(right)\*cos(right))) + k, где right сумма в правом поддереве, left сумма в левом и k ключ в текущем узле.

## Разработка последовательной программы

Исходное дерево считывается из файла и сохраняется в массив. Для измерения времени использовался класс std::chrono::steady\_clock из C++11, предоставляющий доступ к монотонному и высокоточному системному таймеру. Значение считывается внешним скриптом для дальнейшей обработки.

Листинг 1. Последовательная программа

#include<fstream>

#include<vector>

#include<iostream>

#include<ctime>

#include<cstdlib>

#include<cmath>

#include <chrono>

using namespace std;

double \*a,\*sum;

int n,k;

double calc(int v){

int leftNode = 2 \* v + 1;

int rightNode = 2\* v + 2;

double leftSum = 0;

double rightSum = 0;

if (leftNode < n)

leftSum = calc(leftNode);

if (rightNode < n)

rightSum = calc(rightNode);

sum[v] = sqrt(fabs(sin(leftSum) \* sin(leftSum) + cos(rightSum) \* cos(rightSum))) + a[v];

return sum[v];

}

int main(int argc, char \*\*argv){

if (argc < 2) return 1;

ifstream in(argv[1]);

in >> n >> k;

a = new double[n];

sum = new double[n];

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

{

in >> a[i];

sum[i] = 0;

}

in.close();

auto begin = chrono::high\_resolution\_clock::now();

calc(0);

auto end = chrono::high\_resolution\_clock::now();

cout<<chrono::duration\_cast<chrono::nanoseconds>(end-begin).count()/1000000.0<< endl;

return 0;

}

## Программа POSIX

Для максимальной эффективности процессы запускаются с одного уровня дерева. После завершения этих процессов, остальная часть дерева досчитывается в основном потоке.

Листинг 2.POSIX программа

#include <iostream>

#include <fstream>

#include <pthread.h>

#include <vector>

#include <cmath>

#include<ctime>

#include<cstdlib>

#include <chrono>

using namespace std;

double \*a,\*sum;

int n, k;

double calcLast(int v, int leftBorder)

{

int leftNode = 2 \* v + 1;

int rightNode = 2 \* v + 2;

double leftSum = 0;

double rightSum = 0;

if (leftNode < n)

{

if (leftNode >= leftBorder)

leftSum = sum[ leftNode- leftBorder];

else

leftSum = calcLast(leftNode , leftBorder);

}

if (rightNode < n)

{

if (rightNode >= leftBorder)

rightSum = sum[rightNode- leftBorder];

else

rightSum = calcLast(rightNode , leftBorder);

}

sum[v] = sqrt(fabs(sin(leftSum) \* sin(leftSum) + cos(rightSum) \* cos(rightSum))) + a[v];

return sqrt(fabs(sin(leftSum) \* sin(leftSum) + cos(rightSum) \* cos(rightSum))) + a[v];

}

double calc(int v)

{

int leftNode = (v \* 2)+1;

int rightNode = (v \* 2)+2;

double leftSum = 0;

double rightSum = 0;

if (leftNode < n)

{

leftSum = calc(leftNode);

}

if (rightNode < n)

{

rightSum = calc(rightNode);

}

return sqrt(fabs(sin(leftSum) \* sin(leftSum) + cos(rightSum) \* cos(rightSum))) + a[v];

}

void \*task(void \*x)

{

pair<int, int> \*attrs = ((pair<int, int>\*)x);

sum[attrs->first] = calc(attrs->second);

}

int main(int argc, char \*\*argv)

{

/\*if (argc < 2)

{

return 1;

}\*/

ifstream in(/\*argv[1]\*/"tests\\input1.txt");

in>>n >> k;

a = new double[n];

sum = new double[n];

int x;

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

{

in >> a[i];

sum[i] = 0;

}

pthread\_t \*threads = new pthread\_t[k];

int level = round((double)log(k) / (double)log(2));

int currVertex = pow(2, level) - 1;

auto begin = chrono::high\_resolution\_clock::now();

for (int i = 0; i < k; i++){

pair<int, int> \*q = new pair<int,int >;

q->first = i;

q->second = currVertex;

currVertex++;

pthread\_create(&threads[i], NULL, task, (void\*)(q));

}

for (int i=0;i<k;i++){

pthread\_join(threads[i], NULL);

}

calcLast(0, pow(2, level) - 1);

auto end = chrono::high\_resolution\_clock::now();

cout<<chrono::duration\_cast<chrono::nanoseconds>(end-begin).count()/1000000.0<< endl;

return 0;

}

Количество потоков должно быть степенью двойки, чтобы покрыть полностью уровень дерева. Программа высчитывает уровень дерева автоматически, исходя из количества запускаемых потоков.

## Программа OpenMP

"OpenMP (Open Multi-Processing) — открытый стандарт для распараллелива- ния программ на языках Си, Си++ и Фортран. Дает описание совокупности ди- ректив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопро- цессорных системах с общей памятью." Для управления многопоточным выполнением в OpenMP используется директива #pragma

Для написания программы с использованием библиотеки OpenMP использовалась директиива parallel for. Алгоритм аналогичен POSIX-потокам.

Листинг 3.Программа на OpenMP

#include <iostream>

#include <fstream>

#include <pthread.h>

#include <vector>

#include <cmath>

#include <omp.h>

#include<ctime>

#include <chrono>

using namespace std;

double \*a;

double \*sum;

int n, k;

double calcLast(int v, int leftBorder)

{

int leftNode = 2 \* v + 1;

int rightNode = 2 \* v + 2;

double leftSum = 0;

double rightSum = 0;

if (leftNode < n)

{

if (leftNode >= leftBorder)

leftSum = sum[ leftNode- leftBorder];

else

leftSum = calcLast(leftNode , leftBorder);

}

if (rightNode < n)

{

if (rightNode >= leftBorder)

rightSum = sum[rightNode- leftBorder];

else

rightSum = calcLast(rightNode , leftBorder);

}

sum[v] = leftSum + rightSum + a[v];

return sqrt(fabs(sin(leftSum) \* sin(leftSum) + cos(rightSum) \* cos(rightSum))) + a[v];

}

double calc(int v)

{

int leftNode = (v \* 2)+1;

int rightNode = (v \* 2)+2;

double leftSum = 0;

double rightSum = 0;

if (leftNode < n)

{

leftSum = calc(leftNode);

}

if (rightNode < n)

{

rightSum = calc(rightNode);

}

sum[v] = leftSum + rightSum + a[v];

return sqrt(fabs(sin(leftSum) \* sin(leftSum) + cos(rightSum) \* cos(rightSum))) + a[v];

}

void task(pair<int, int> attrs)

{

sum[attrs.first] = calc(attrs.second);

}

int main(int argc, char \*\*argv)

{

/\*if (argc < 2)

{

return 1;

}\*/

ifstream in(/\*argv[1]\*/"tests\\input1.txt");

in>>n >> k;

a = new double[n];

sum = new double[n];

int x;

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

{

in >> a[i];

sum[i] = 0;

}

int level = round((double)log(k) / log(2));

int currVertex = pow(2, level) - 1;

auto begin = chrono::high\_resolution\_clock::now();

omp\_set\_num\_threads(k);

#pragma omp parallel for

for (int i = 0; i < k; i++){

pair<int, int> \*q = new pair<int,int >;

q->first = i;

q->second = currVertex + i;

task(\*q);

}

double summa = calcLast(0, pow(2, level) - 1);

auto end = chrono::high\_resolution\_clock::now();

cout<<chrono::duration\_cast<chrono::nanoseconds>(end-begin).count()/1000000.0<< endl;

return 0;

}

## Сравнение результатов

Для тестирования использовался скрипт, написанный на языке Python.

Листинг 4. Скрипт для измерения времени работы

import subprocess

import time

OPEN\_MP = 'openmpthreads2.exe'

POSIX = 'threads.exe'

NORMAL = 'normal.exe'

def getMillis(cmd):

p = subprocess.Popen(cmd, stdout=subprocess.PIPE)

return float(p.stdout.read())

k = 0

for i in range(100):

k+=getMillis(NORMAL + " tests\\input1.txt")

print(k/100.0)

Тесты проводились на компьютере с 4-х ядерным процессором Intel core i5 1.8 ГГц. 6 Гб ОЗУ DDR3. Тесты запускались многократно для получения более точных данных.

* Размер данных 10^4
* Размер данных 10^5
* Размер данных 10^6

Вычислим ускорение программы при использовании потоков.

|  |  |  |  |
| --- | --- | --- | --- |
|  | 10^4 | 10^5 | 10^6 |
| Последовательная программа | 3.7022 | 39.0263 | 377.26188 |
| Pthreads | 1.3507 | 7.5546 | 60.7265 |
| Ускорение | 2.7409 | 5.1658 | 6.212475 |
| OpenMP | 2.5793 | 14.5096 | 117.8782 |
| Ускорение | 1.43535 | 2.689688 | 3.2004 |

Из полученных результатов можно сделать вывод, что наибольшее ускорение достигается при количестве потоков, которое примерно равно количеству ядер на компьютере. При дальнейшем увеличении количества потоков ускорение становится меньше. В целом POSIX – потоки показали лучшие результаты, чем OpenMP, но при увеличении размера входных данных результаты становятся примерно равными.

Сделаем тестовую выборку результатов и посчитаем математическое ожидание и дисперсию.

Тесты проведем для 4 потоков и для разного размера данных.

Размер 10^6

|  |  |  |  |
| --- | --- | --- | --- |
| Характеристика | Последовательный | POSIX | OpenMP |
| Мат. ожидание | 377.26188 | 60.7265 | 117.8782 |
| Дисперсия | 72.8416889616 | 25.34963105 | 55.846914559 |
| Доверительный интервал | [332.11; 422.41] | [45.1; 76.44] | [83.26;152.49] |

Размер 10^5

|  |  |  |  |
| --- | --- | --- | --- |
| Характеристика | Последовательный | POSIX | OpenMP |
| Мат. ожидание | 39.0263 | 7.5546 | 14.5096 |
| Дисперсия | 23.08371 | 0.122941 | 0.8002 |
| Доверительный интервал | [24.72; 53.33] | [7.48; 7.63] | [14.01;15.01] |

Размер 10^4

|  |  |  |  |
| --- | --- | --- | --- |
| Характеристика | Последовательный | POSIX | OpenMP |
| Мат. ожидание | 3.7022 | 1.3507 | 2.5793 |
| Дисперсия | 0.06002 | 0.1026 | 0.13038 |
| Доверительный интервал | [3.66; 3.74] | [1.29; 1.41] | [2.50;2.66] |

# Вывод

В данной лабораторной работе были рассмотрены методы распараллеливания с помощью pthreads и OpenMP.

При использовании OpenMP в общем случае получается более короткий и читаемый код. Результаты исследований показали, что использование многопоточного программирования позволяет достичь ускорения до 4-х раз. Но при увеличении количества потоков, результат начинает ухудшаться, поэтому важно создавать оптимальное количество потоков.

Использование многопоточного программирования оправдано в случае сложных вычислений. Для тривиальных задач лучше использовать однопоточные программы.