《并行计算》上机报告

姓名:	李子旸	学号:	PB16110959	日期:	2019.5.4
上机题目:	OpenMP				

实验环境:
CPU:Intel 至强 E5-2650 v2; 内存:64GB; 操作系统:Ubuntu; 软件平台: vim

一、算法设计与分析:

题目:

用 4 中不同的并行方式的 Openmp 实现 pi 的计算
用 Openmp 实现 PSRS 排序

算法设计:
pi 的计算:
串行:

```
for (int i = 1; i <= n; i++){
x = (i - 0.5) / n;
sum += 4 / (1 + x * x);
}
pi = sum / n;
```

改成四种并行:
1.并行域并行化
2.共享任务结构并行化
3.使用 private 子句和 critical 部分并行化
4.使用并行规约
PSRS 排序:
均匀划分->局部排序->选取样本->样本排序->选择主元->主元划分->全局交换->归并排序

二、核心代码:

并行域并行化:

```
#pragma omp parallel
{
int id = omp_get_thread_num();
for (int i = id; i < num_steps; i=i+NUM_THREADS){
double x = (i + 0.5) * step;
sum[id] += 4.0 / (1.0 + x * x);
}
```

```
}
```

共享任务结构并行化：

```
#pragma omp parallel
{
int id = omp_get_thread_num();
#pragma omp for
for (int i = 0; i < num_steps; i++){
double x = (i + 0.5) * step;
sum[id] += 4.0 / (1.0 + x * x);
}
}
```

使用 private 子句和 critical 部分并行化：

```
#pragma omp parallel private(x, sum) // 该子句表示 x,sum 变量对于每个线程是私有的
{
int id;
id = omp_get_thread_num();
for (int i = id, sum = 0.0; i < num_steps; i = i + NUM_THREADS)
{
x = (i + 0.5) * step;
sum += 4.0 / (1.0 + x * x);
}
#pragma omp critical // 指定代码段在同一时刻只能由一个线程进行执行
pi += sum * step;
}
```

使用并行规约：

```
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < num_steps; i++){
double x = (i + 0.5) * step;
sum = sum + 4.0 / (1.0 + x * x);
}
```

PSRS 排序：

```
#pragma omp parallel
{
    int id = omp_get_thread_num();
    sort(number + id * blockSize, number + (id + 1) * blockSize);
    #pragma omp critical
    for (int i = 0; i < NUM_THREADS; i++){
        sampling[++sampling[0]] = number[id * blockSize + i * blockSize / NUM_THREAD
    }
    #pragma omp barrier
    #pragma omp master
    {
        sort(sampling + 1, sampling + sampling[0] + 1);
        for (int i = 0; i < NUM_THREADS - 1; i++)
            pivot[i] = sampling[(i + 1) * NUM_THREADS + 1];
    }
    #pragma omp barrier
    #pragma omp critical        You, a month ago • lab1 100%
    for (int j = 0; j < blockSize; j++){
        for (int i = 0; i < NUM_THREADS; i++){
            if (number[id * blockSize + j] < pivot[i]){
                pivot_array[i].push_back(number[id * blockSize + j]);
                break;
            }
            else if (i == NUM_THREADS - 1){
                pivot_array[i].push_back(number[id * blockSize + j]);
            }
        }
    }

    #pragma omp barrier
    sort(pivot_array[id].begin(), pivot_array[id].end());
    #pragma omp master
    {
        vector<int>::iterator It;
        int cnt = 0;
        for (int i = 0; i < NUM_THREADS; i++){
            for (It = pivot_array[i].begin(); It != pivot_array[i].end(); It++){
                cnt++;
                if (cnt > n)break;
                fileName << (*It) << " ";
            }
        }
        fileName << endl;
    }
}
```

三、结果与分析：

计算 pi 值

n = 100000000

串行：1.723480s

使用 16 个核

并行域并行化：1.398216s

共享结构并行化：1.567573s

使用 private 子句和 critical 部分并行化：0.885023s

使用并行规约：0.17293s

分析：
前两个效果不是很显著，第三个效果稍微显著，最后一个效果最显著。
并行利用率不断提高

PSRS：
n = 10000000
串行归并排序：4.952611s
并行 PSRS 排序：3.843739s

四、备注（* 可选）：
有可能影响结论的因素：
核与核之间的通信耗时，针对特定的环境可以采取特定的并行方法，对实验结果影响较大

总结：
充分发挥多核计算机的特性，对于特定的加法减法操作，可以加速到很快的时间范围内

附录（源代码）

算法源代码（C/C++/JAVA 描述）
并行域并行化：

```
#include <iostream>
#include <cstdio>
#include <omp.h>
#include <ctime>

using namespace std;
const int NUM_THREADS = 16;
int num_steps = 100000000;

int main(){
double sum[NUM_THREADS] = {0.0}, pi = 0, step;
step = (double)1 / num_steps;
double start, end;
start = omp_get_wtime();
```

	omp_set_num_threads(NUM_THREADS); #pragma omp parallel { int id = omp_get_thread_num(); for (int i = id; i < num_steps; i=i+NUM_THREADS){ double x = (i + 0.5) * step; sum[id] += 4.0 / (1.0 + x * x); } } for (int i = 0; i < NUM_THREADS; i++) pi += step * sum[i]; printf("%.8lf\n", pi); end = omp_get_wtime(); printf("time = %lf\n", double(end - start)); return 0; } 共享任务结构并行化: #include <iostream> #include <cstdio> #include <omp.h> #include <ctime> using namespace std; const int NUM_THREADS = 16; int num_steps = 100000000; int main(){ double sum[NUM_THREADS] = {0.0}, pi = 0, step; step = (double)1 / num_steps; double start, end; start = omp_get_wtime(); omp_set_num_threads(NUM_THREADS); #pragma omp parallel { int id = omp_get_thread_num(); #pragma omp for for (int i = 0; i < num_steps; i++){ double x = (i + 0.5) * step; sum[id] += 4.0 / (1.0 + x * x); } }

	for (int i = 0; i < NUM_THREADS; i++) pi += step * sum[i]; printf("%.8lf\n", pi); end = omp_get_wtime(); printf("time = %lf\n", double(end - start)); return 0; } 使用 private 子句和 critical 部分并行化： #include <cstdio> #include <ctime> #include <omp.h> using namespace std; static long num_steps = 100000000; double step; #define NUM_THREADS 4 int main(){ int i; double pi = 0.0; double sum = 0.0; double x = 0.0; step = 1.0 / (double)num_steps; double start, end; start = omp_get_wtime(); omp_set_num_threads(NUM_THREADS); // 设置 2 线程 #pragma omp parallel private(x, sum) // 该子句表示 x,sum 变量对于每个线程是私有的 { int id; id = omp_get_thread_num(); for (i = id, sum = 0.0; i < num_steps; i = i + NUM_THREADS) { x = (i + 0.5) * step; sum += 4.0 / (1.0 + x * x); } #pragma omp critical // 指定代码段在同一时刻只能由一个线程进行执行 pi += sum * step; } printf("%lf\n", pi); end = omp_get_wtime();

	printf("time = %lf\n", double(end - start)); } // 共 2 个线程参加计算,其中线程 0 进行迭代步 0,2,4,... 线程 1 进行迭代步 1,3,5,.... 当被指定为 critical 的代码段正在 //被 0 线程执行时, 1 线程的执行也到达该代码段, 则它将被阻塞知道 0 线程退出临界区。 使用并行规约: #include <cstdio> #include <omp.h> #include <iostream> #include <ctime> using namespace std; const int NUM_THREADS = 16; int num_steps = 100000000; int main(){ double pi, sum = 0.0, x, step; step = 1.0 / (double)num_steps; double start, end; start = omp_get_wtime(); omp_set_num_threads(NUM_THREADS); #pragma omp parallel for reduction(+:sum) for (int i = 0; i < num_steps; i++){ double x = (i + 0.5) * step; sum = sum + 4.0 / (1.0 + x * x); } pi = step * sum; printf("%.10lf\n", pi); end = omp_get_wtime(); printf("time = %lf\n", double(end - start)); return 0; } PSRS 排序: #include <iostream> #include <cstdio> #include <omp.h>

```
#include <fstream>
#include <algorithm>
#include <vector>

using namespace std;
#define NUM_THREADS 8
#define INF 999999999
#define MAXN 10000005
int n;
int number[MAXN];
fstream fileName;

int fileIO(string file){
fileName.open(file);
if (!fileName.is_open()){
printf("File Read Fail\n");
return -1;
}
fileName >> n;
for (int i = 0; i < n; i++)
fileName >> number[i];
for (int i = n; i < (n / NUM_THREADS + 1) * NUM_THREADS; i++)
number[i] = INF;
fileName.close();
return 0;
}

int PSRS(string file){
fileName.open(file);
if (!fileName.is_open()){
printf("File Read Fail\n");
return -1;
}
int blockSize = n / NUM_THREADS + 1;//块大小
int sampling[NUM_THREADS * NUM_THREADS + 1] = {0};//正则采样
int pivot[NUM_THREADS] = {0};//主元
vector<int>pivot_array[NUM_THREADS];
omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{
int id = omp_get_thread_num();
sort(number + id * blockSize, number + (id + 1) * blockSize);
#pragma omp critical
```

```
for (int i = 0; i < NUM_THREADS; i++){
sampling[++sampling[0]] = number[id * blockSize + i * blockSize / NUM_THREADS];
}
#pragma omp barrier
#pragma omp master
{
sort(sampling + 1, sampling + sampling[0] + 1);
for (int i = 0; i < NUM_THREADS - 1; i++)
pivot[i] = sampling[(i + 1) * NUM_THREADS + 1];
}
#pragma omp barrier
#pragma omp critical
for (int j = 0; j < blockSize; j++){
for (int i = 0; i < NUM_THREADS; i++){
if (number[id * blockSize + j] < pivot[i]){
pivot_array[i].push_back(number[id * blockSize + j]);
break;
}
else if (i == NUM_THREADS - 1){
pivot_array[i].push_back(number[id * blockSize + j]);
}
}
}


#pragma omp barrier
sort(pivot_array[id].begin(), pivot_array[id].end());
#pragma omp master
{
vector<int>::iterator It;
int cnt = 0;
for (int i = 0; i < NUM_THREADS; i++){
for (It = pivot_array[i].begin(); It != pivot_array[i].end(); It++){
cnt++;
if (cnt > n)break;
fileName << (*It) << " ";
}
}
fileName << endl;
}
}
fileName.close();
}
```

	`` int main(int argc, char * argv[]){ if (argc <= 2){ printf("Paraments error\n"); return 0; } double start, end; start = omp_get_wtime(); if (fileIO(argv[1]) == -1){ return 0; } // sort(number, number + n); if (PSRS(argv[2]) == -1){ return 0; } end = omp_get_wtime(); printf("time = %lf\n", (double)(end - start)); } ```