**深 圳 大 学 实 验 报 告**

|  |  |
| --- | --- |
| **课程名称** | **：并行计算** |
| **实验项目名称** | **：寻找完数的OpenMP并行程序** |
| **学院** | **：计算机与软件学院** |
| **专业** | **：计算机科学与技术** |
| **指导教师** | **：陆克中** |
| **报告人** | **：郑雨婷** |
| **学号** | **：2021150122** |
| **实验时间** | **：2024年03月27日** |
| **实验报告提交时间** | **：2024年03月27日** |

**教务部制**

## 一、实验目的

1. 掌握for编译制导语句和schedule子句；

2. 理解数据竞争，掌握同步结构；

3. 对并行程序进行简单的性能分析。

## 二、实验环境

1. 硬件环境：64核CPU、256GB内存的共享内存并行计算平台；

2. 软件环境：Ubuntu Linux、gcc、g++（g++ -O3 -fopenmp -o a.out a.cc）；

3. 远程登录：本地PowerShell中执行ssh bxjs@hpc.szu.edu.cn；

4. 传输文件：本地PowerShell中执行scp c:\a.cpp [bxjs@hpc.szu.edu.cn:/home/bxjs/](mailto:bxjs@hpc.szu.edu.cn:/home/bxjs/)或<ftp://hpc.szu.edu.cn>。

## 三、实验内容

1. 用OpenMP语言编写程序，求小于等于*n*的所有完数（一个数恰好等于它的因子之和），并存放在数组*a*中，调节for编译制导语句中schedule的参数，使得执行时间最短。为了验证结果正确性，将并行计算结果和串行计算结果相比较。

2. 测试并行程序在不同线程数下的执行时间和加速比（与线程数=1时的执行时间相比）。其中，*n*固定为5000000，线程数分别取1、2、4、8、16、32、64时，为减少误差，每项实验进行5次，取平均值作为实验结果。

## 四、代码描述

#include <omp.h>

#include <iostream>

#include <cmath>

using namespace std;

const int MAX = 100;

//判断x是否为完数

bool check(int x) {

int sum = 1; //1一定是因子，直接加入sum中

for (int i = 2; i \* i <= x; i++) { //找x的因子，从2开始，找到x的平方根即可

if (x % i == 0) { //若能整除，就加到sum上

sum += i;

if (i \* i != x) sum += x / i; //除了平方根以外的因子，总是成对出现，顺便加上

}

}

if (sum == x) return true; //若因子和等于x本身，是完数

return false;

}

int main()

{

int a[MAX], b[MAX]; // a用于存并行计算的结果，b用于存串行计算的结果

int i, n, nthreads, count = 0, count2 = 0;

cin >> n; // 输入待检查的数的范围，实验中固定为5000000

cin >> nthreads; // 输入线程数，可为1.2.4.8.16.32.64

omp\_set\_num\_threads(nthreads); // 设置线程数

double t0 = omp\_get\_wtime(); // 记录并行计算开始时间

//并行代码

#pragma omp parallel shared(count,a) private(i) // 共享变量count和数组a,私有变量 i

{ // 动态调度，chunk为100.每次分配 100 个任务

#pragma omp for schedule(dynamic,100)

for (i = 2; i < n; ++i) {

if (check(i))

// 使用临界区，保证同时只有一个线程修改 count 和 a,防止出现数据竞争

#pragma omp critical

a[count++] = i;

}

}

double t1 = omp\_get\_wtime(); // 记录并行计算结束时间

// 输出并行计算结果和时间

for (i = 0; i < count; i++)

cout << a[i] << " ";

cout << endl;

cout << "parallel computing time is " << t1 - t0 << " seconds" << endl;

//验证代码，串行化计算，结果存入数组

for (int j = 2; j < n; ++j) {

if (check(j)) b[count2++] = j;

}

//输出串行计算结果

for (i = 0; i < count2; i++)

cout << b[i] << " ";

cout << endl;

return 0;

}

## 五、实验结果和分析

测试n固定为5000000，线程数分别取1、2、4、8、16、32、64时并行程序的执行时间，每项实验进行5次，测试结果如图1~图7。

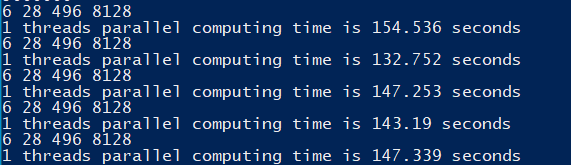
**

图1 线程数为1 时的执行时间

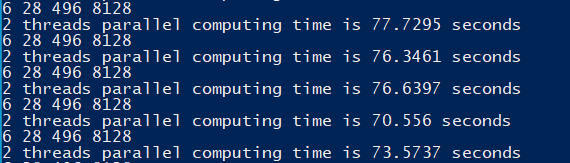
**

图2 线程数为2 时的执行时间

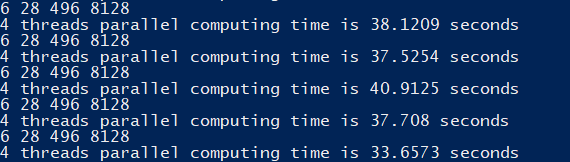
**

图3 线程数为4时的执行时间

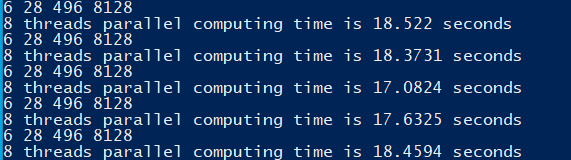
**

图4 线程数为8时的执行时间

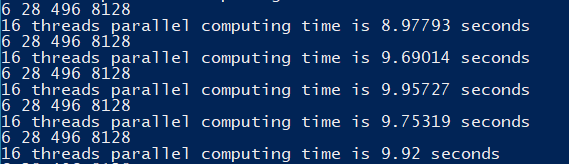
**

图5 线程数为16 时的执行时间

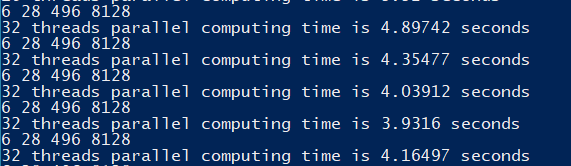
**

图6 线程数为32时的执行时间

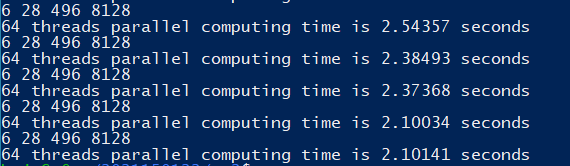
**

图7 线程数为64 时的执行时间

将程序在不同线程数下的执行时间和加速比（与线程数=1时的执行时间相比）记录在下表1 中。

表1 并行程序在不同线程数下的执行时间（秒）和加速比（n=5000000）

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 线程数  执行时间 | 1 | 2 | 4 | 8 | 16 | 32 | 64 |
| 第1次 | 154.536 | 77.7295 | 38.1209 | 18.522 | 8.97793 | 4.89742 | 2.54357 |
| 第2次 | 132.752 | 76.3461 | 37.5254 | 18.3731 | 9.69014 | 4.35477 | 2.38493 |
| 第3次 | 147.253 | 76.6397 | 40.9125 | 17.0824 | 9.95727 | 4.03912 | 2.37368 |
| 第4次 | 143.19 | 70.556 | 37.708 | 17.6325 | 9.75319 | 3.9316 | 2.10034 |
| 第5次 | 147.339 | 73.5737 | 33.6573 | 18.4594 | 9.92 | 4.16497 | 2.10141 |
| 平均值 | 145.014 | 74.969 | 37.58482 | 18.01388 | 9.659706 | 4.277576 | 2.300786 |
| 加速比 | 1 | 1.93432 | 3.858313 | 8.050126 | 15.01226 | 33.90098 | 63.02803 |

实验结果：从上表1可以看出，随着线程数的增加，执行时间逐渐减少。在线程数从1增加到2时，执行时间几乎减半；从2增加到4时，再次减半；以此类推。加速比随着线程数的增加呈指数增长。

分析和综合概括：从线程数为1到线程数为64的实验结果可以看出，随着线程数的增加，加速比逐渐增大。在64个线程的情况下，加速比达到了63.03，说明并行程序在利用多核处理器的同时取得了较好的性能提升。但是加速比并不是与线程数相等。这是受到处理器的硬件结构和内存子系统的限制，虽然处理器具有多个核，但这些核共享内存总线或内存通道，因此同时访问内存的核心数量是有限制的。在线程数过大时，可能还是需要等待排队读写内存。

## 六、实验结论

在本次实验中，一开始寻找完数时判断因子从2到x/2都进行了判断，这样导致n=5000000时运行非常缓慢，后来想到因子总是成对的出现（平方根只算一个因子），将判断完数时的循环缩小了，运行时间明显缩短。

本次实验为了提高实验性能，使用了for语句的schedule子句。type为static时循环被分成大小为 chunk的块，静态分配给线程。type为dynamic时循环被划分为大小为chunk的块，动态分配给线程。我使用schedule (dynamic,100),。因为数字越大，判断是否为完数时循环的次数也就越多，如果静态分配，第一个线程的计算量一定是最小的，最后一个线程的计算量相对来说十分庞大，使得整个程序的效率下降。动态分配时，先完成任务的线程去领取下一次的任务继续计算，就会分配的更加平均。

另外，使用critical语句，使指定代码段在同一时刻只能由一个线程执行。使用critical语句，保护对共享变量a和count的写操作，以确保多个线程不会同时修改a和count，从而避免数据竞争和不确定行为,提高程序的可靠性和正确性。

总之，通过实验学习到并行程序应合理选择线程数，schedule子句可以更合理的给线程分配任务、critical语句可以有效防止数据竞争。