**深 圳 大 学 实 验 报 告**

|  |  |
| --- | --- |
| **课程名称** | **：并行计算** |
| **实验项目名称** | **：前缀和的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04月24日** |
| **实验报告提交时间** | **：2024年04月25日** |

**教务部制**

## 一、实验目的

1. 设计前缀和的并行算法；

2. 掌握for编译制导语句和数据划分方法；

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.cpp）；

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，求数组*a*的前缀和，保存在数组*b*中，即*b*[*i*]=。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，误差在1e-4以内。

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100000000，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>

using namespace std;

int main() {

int n = 1e8, p = 8;

double\* a = new double[n]; //数组a 原始数组

double\* b = new double[n]; //数组b 前缀和数组

double\* temp = new double[p]; //数组temp 存放第i个子数组的总和

srand(0);

for (int i = 0; i < n; i++) //初始化数组a

a[i] = rand() / (double)RAND\_MAX;

omp\_set\_num\_threads(p); //设置线程数为p

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

// 并行求前缀和

#pragma omp parallel

{

//根据线程id求处理的子数组开始坐标start和结束坐标end

int thread\_id = omp\_get\_thread\_num();

int block\_size = n / p;

int start = thread\_id \* block\_size;

int end = (thread\_id == p - 1) ? n : start + block\_size;

double sum = 0.0;

for (int i = start; i < end; i++) { //求子数组的前缀和

sum += a[i];

b[i] = sum;

}

temp[thread\_id] = sum; //将本块的总和存入temp数组

//确保每个线程都已经求取了各自的子数组前缀,每个块的总和已经存入了temp数组

#pragma omp barrier //设置路障，等待各个线程执行完毕

//此时只有第一个子块是正确的结果，其他子块都需要加上之前所有块的总和。

//此时temp中是每个块各自的总和，累加求得前i块的总和。

//累加的动作使用single语句由一个线程完成

#pragma omp single

{

//此时的temp[i]就是第i块及前面所有块的总和

for (int i = 1; i < p; i++) {

temp[i] += temp[i - 1];

}

}

//single语句后有一个隐藏的路障，确保temp[i]是第i块及前面所有块的总和

//除了第一块以外，其他所有数字都需要加上temp[thread\_id-1]

if (thread\_id > 0) {

for (int i = start; i < end; i++) {

b[i] += temp[thread\_id - 1];

}

}

}

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

cout << "Parallel time is " << t1 - t0 << " seconds" << endl;

// 验证代码,串行累加，判断与b数组的结果是否一致

int tag = 0;

double sum\_serial = 0.0;

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

sum\_serial += a[i];

if (abs(sum\_serial - b[i]) > 1e-4) {

tag = 1;

break;

}

}

if (tag == 1)cout << "WRONG!" << endl;

else cout << "RIGHT!" << endl;

delete[] a; //释放a

delete[] b; //释放b

delete[] temp; //释放temp

return 0;

}

## 五、实验结果和分析

测试n固定为100000000，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。

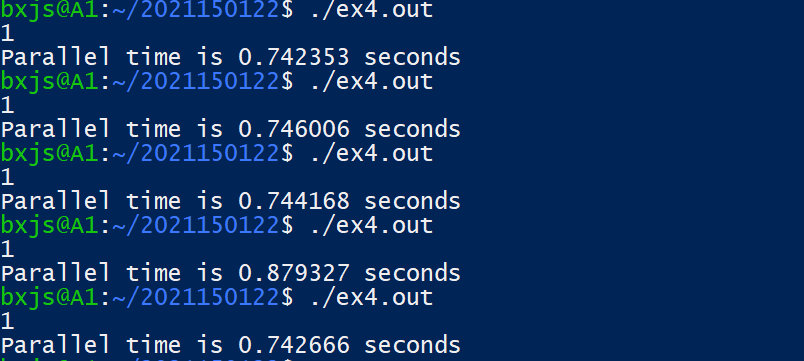
**

图1 线程数为1 时的执行时间

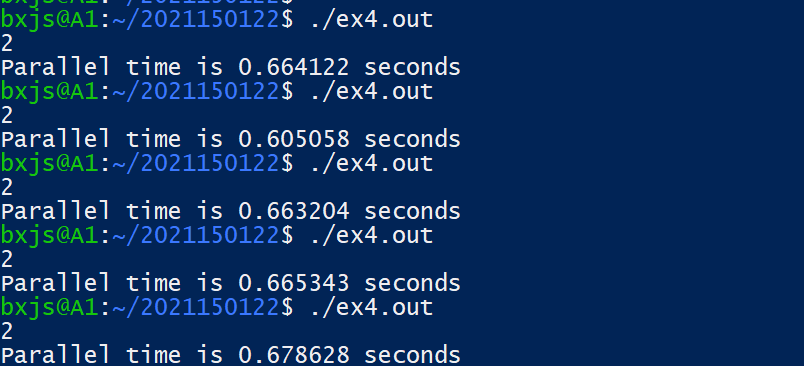
**

图2 线程数为2 时的执行时间

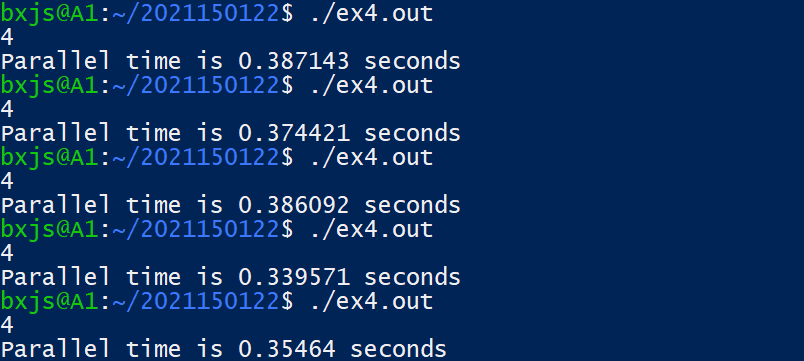
**

图3 线程数为4时的执行时间

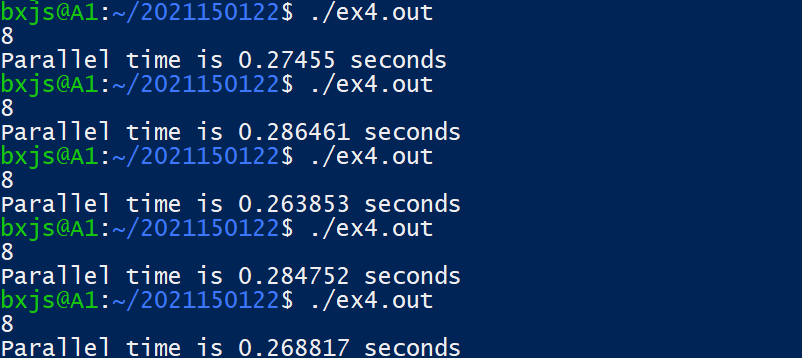
**

图4 线程数为8时的执行时间

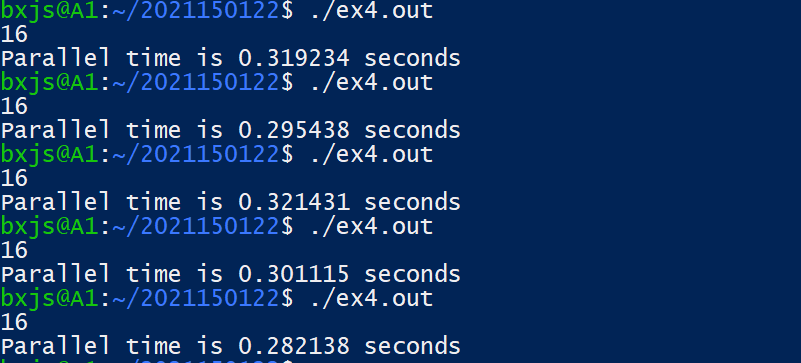
**

图5 线程数为16 时的执行时间

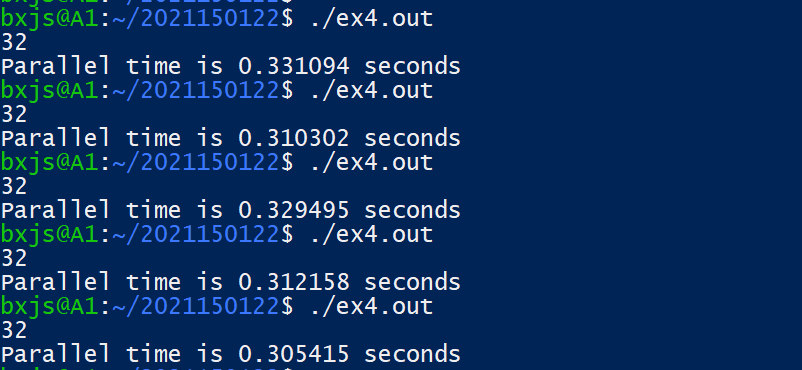
**

图6 线程数为32时的执行时间

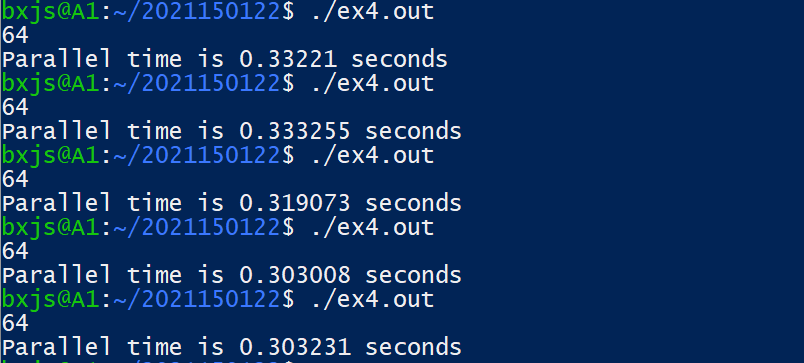
**

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*=100000000）

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 线程数  执行时间 | 1 | 2 | 4 | 8 | 16 | 32 | 64 |
| 第1次 | 0.742353 | 0.664122 | 0.387143 | 0.27455 | 0.319234 | 0.331094 | 0.33221 |
| 第2次 | 0.746006 | 0.605058 | 0.374421 | 0.286461 | 0.295438 | 0.310302 | 0.333255 |
| 第3次 | 0.744168 | 0.663204 | 0.386092 | 0.263853 | 0.321431 | 0.329495 | 0.319073 |
| 第4次 | 0.879372 | 0.665343 | 0.339571 | 0.284752 | 0.301115 | 0.312158 | 0.303008 |
| 第5次 | 0.742666 | 0.678628 | 0.35464 | 0.268817 | 0.282138 | 0.305415 | 0.303231 |
| 平均值 | 0.770913 | 0.655271 | 0.368373 | 0.275687 | 0.303871 | 0.317693 | 0.318155 |
| 加速比 | 1 | 1.17648 | 2.092749 | 2.796388 | 2.536973 | 2.426599 | 2.423071 |

**实验结果**：从上表1可以看出，线程数从1增加到8，执行时间逐渐减少。然后线程数从8增加到64，执行时间逐渐增加，加速比反而下降。线程数为32时和线程数为64时的执行时间几乎相等。在本实验中，最佳线程数是 8。

**分析和综合概括**：从线程数为1到线程数为8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8个线程的情况下，加速比最高，达到了2.79，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。

但是从8线程往后，加速比有所下降。这是因为，我的并行程序其中设置了路障barrier，这会使先完成的线程必须等待最后一个完成的线程。之后又只使用single语句让一个线程去执行temp数组累加的操作，single语句末尾又有一个隐藏的路障，使得其他所有线程都在等待。最后再次并行运算。在single语句中的代码段相当于串行执行，p越大，执行时间反而越长。

更严谨地，代码的时间复杂度为,不考虑常数为所以在已经足够小的时候，p越大，执行时间越长。所以加速比不会随线程数线性增长。同时也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，第一个难点就是算法的设计。很容易想到并行地求取各个块的前缀和，但是如何与其他块的结果合并是一个问题。因此先记录下各个块的总和，然后串行地累加得到第i块及前面所有块的总和，最后再次并行地加上之前所有块的总和。

首先在使用for编译制导语的时候需要进行数据划分，我掌握到可以根据当前线程的线程ID来进行数据划分，这样就可以确保不会数据不会重叠，最终也验证了结果是正确的；另外，在求取子数组前缀和之后，使用barrier语句，用来同步一个线程组中所有的线程,先到达的线程在此阻塞，等待其他线程。之后在累加temp时，使用了OpenMP的single语句，single语句指定内部代码只有线程组中的一个线程执行除非使用了nowait子语，否则在结束处有一隐含的路障。

最后，通过分析并行程序的运行时间复杂度，解释了加速比先增后降的原因。总之，通过本次实验学习到single语句、barrier语句、以及分析并行算法复杂度的方法。