**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** | **：何泽锋** |
| **学号** | **：2022150221** |
| **实验时间** | **：2025年04月02日** |
| **实验报告提交时间** | **：2025年04月03日** |

**教务部制**

## 一、实验目的

1. 设计前缀和的并行算法；

2. 掌握barrier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。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

*在下面写出完整的程序代码（文本格式，不可截图），对于其中的关键代码，以注释方式给出必要的描述。*

*法一 分块法*

|  |
| --- |
| #include <omp.h>  #include <iostream>  #include <vector>  #include <cstring>  #include <math.h>  #include <iomanip>  using namespace std;  int n = 1e8, p = 8;  double t\_s; // 串行执行时间  double table[70][10]; // 结果记录表  double Serial(double\*a,double\*b) { // 串行计算前缀和  double t0 = omp\_get\_wtime();  b[0] = a[0];  for (int i = 1; i < n; i++) {  b[i] = b[i - 1] + a[i];  }  double t1 = omp\_get\_wtime();  return t1 - t0;  }  bool Check(double\* a, double\* b) { // 比较串行和并行结果  for (int i = 0; i < n; i++) {  if (fabs(a[i] - b[i]) > 1e-4) {  cout << "Mismatch at index " << i << ": " << a[i] << " vs " << b[i] << endl;  return false;  }  }  return true;  }  void Parallel(int threads, double\* a, double\* b) { // 分块法  cout << "Threads: " << threads << endl;  omp\_set\_num\_threads(threads);  double total\_time = 0;  double\* temp = new double[n];  for (int run = 0; run < 5; run++) {  memcpy(temp, a, n \* sizeof(double)); // 临时数组  double t0 = omp\_get\_wtime();  #pragma omp parallel  {  int tid = omp\_get\_thread\_num();  int chunk\_size = (n + threads - 1) / threads; // 按照线程数计算块大小  int start = tid \* chunk\_size;  int end = min(start + chunk\_size, n); // 最后一块特殊处理  for (int i = start + 1; i < end; i++) { // 块内计算前缀和  temp[i] += temp[i - 1];  }  }  double prefix = 0; // 记录当前块之前的所有数之和  for (int t = 0; t < threads; t++) {  int chunk\_size = (n + threads - 1) / threads;  int start = t \* chunk\_size;  int end = min(start + chunk\_size, n);  if (end > start) {  double last = temp[end - 1]; // 本块最后一个值  #pragma omp parallel for  for (int i = start; i < end; i++) {  temp[i] += prefix; // 加上先前块的总和  }  prefix += last; //更新累加和  }  }  double t1 = omp\_get\_wtime();  if (run == 0) {  if (Check(temp, b)) cout << "Pass Check" << endl;  }  printf("Parallel time: %lf s \n", t1 - t0);  table[threads][run] = t1 - t0;  total\_time += t1 - t0;  }  delete[] temp;  double avg\_time = total\_time / 5;  table[threads][5] = avg\_time;  table[threads][6] = t\_s / avg\_time;  cout << "Average time: " << avg\_time << " s" << endl;  cout << "Speedup: " << t\_s / avg\_time << "\n" << endl;  }  int main(){  double\* a = new double[n];  double\* b = new double[n];  double\* c = new double[n];  srand(0);  for (int i = 0; i < n; i++) {  a[i] = rand() / (double)RAND\_MAX;  b[i] =c[i] = 0;  }  t\_s = Serial(a, b);  printf("Serial time: %lf s\n", t\_s);  Parallel(1, a, b);  Parallel(2, a, b);  Parallel(4, a, b);  Parallel(8, a, b);  Parallel(16, a, b);  Parallel(32, a, b);  Parallel(64, a, b);  int thread[] = {1,2,4,8,16,32,64};  for (int i = 0; i < 7; i++) {  if (i < 5)printf("第%d次 ", i + 1);  else if (i == 5)printf("均 值 ");  else printf("加速比 ");  for (int j : thread) {  cout <<fixed<<setprecision(6) << table[j][i] << " ";  }  cout << endl;  }  delete[] a;  delete[] b;  delete[] c;  return 0;  } |

*法二 Blelloch并行扫描法*

|  |
| --- |
| void Parallel1(int threads, double\* a, double\* c, double\* b) { // 扫描法  cout << "Tread: " << threads << endl;  omp\_set\_num\_threads(threads);  double total\_time = 0;  // 计算不小于n的最小2的幂，用于分配空间  int next\_pow2 = 1;  while (next\_pow2 < n) next\_pow2 <<= 1;  vector<double> temp(next\_pow2);  for (int run = 0; run < 5; run++) {  // 拷贝数据到临时数组  copy(a, a + n, temp.begin());  fill(temp.begin() + n, temp.end(), 0);  double t0 = omp\_get\_wtime();  // 上扫阶段  for (int stride = 2; stride <= next\_pow2; stride \*= 2) {  int half\_stride = stride / 2;  #pragma omp parallel for schedule(dynamic)  for (int i = stride - 1; i < next\_pow2; i += stride) {  temp[i] += temp[i - half\_stride];  }  }  // 下扫阶段  temp[next\_pow2 - 1] = 0;  for (int stride = next\_pow2; stride >= 2; stride /= 2) {  int half\_stride = stride / 2;  #pragma omp parallel for schedule(dynamic)  for (int i = half\_stride - 1; i < next\_pow2; i += stride) {  double t = temp[i];  temp[i] = temp[i + half\_stride];  temp[i + half\_stride] += t;  }  }  // 转换为包含前缀和  #pragma omp parallel for  for (int i = 0; i < n; i++) {  c[i] = temp[i] + a[i];  }  double t1 = omp\_get\_wtime();  if (run == 0 && Check(c, b)) {  cout << "Pass Check" << endl;  }  total\_time += t1 - t0;  table[threads][run] = t1 - t0;  printf("Parallel time: %lf s \n", t1 - t0);  }  double avg\_time = total\_time / 5;  table[threads][5] = avg\_time;  table[threads][6] = t\_s / avg\_time;  cout << "Average time: " << total\_time / 5 << " s" << endl;  cout << "Speedup: " << t\_s / (total\_time / 5) << endl;  } |

## 五、实验结果和分析

*实验结果以及对实验结果的比较分析和综合概括。*

表1 并行程序（法一）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 | 0.72323 | 0.48209 | 0.44390 | 0.31189 | 0.24451 | 0.32346 | 0.28443 |
| 第2次 | 0.71012 | 0.48497 | 0.23630 | 0.21236 | 0.24324 | 0.23215 | 0.26289 |
| 第3次 | 0.69876 | 0.49397 | 0.26363 | 0.22818 | 0.23630 | 0.28465 | 0.25602 |
| 第4次 | 0.73308 | 0.49147 | 0.27499 | 0.21629 | 0.26282 | 0.29062 | 0.41272 |
| 第5次 | 0.70938 | 0.48584 | 0.25418 | 0.22346 | 0.22533 | 0.19308 | 0.25491 |
| 平均值 | 0.71491 | 0.48767 | 0.29460 | 0.23844 | 0.24244 | 0.26480 | 0.29419 |
| 加速比 | 1.00000 | 1.46599 | 2.42674 | 2.99835 | 2.94883 | 2.69988 | 2.43009 |

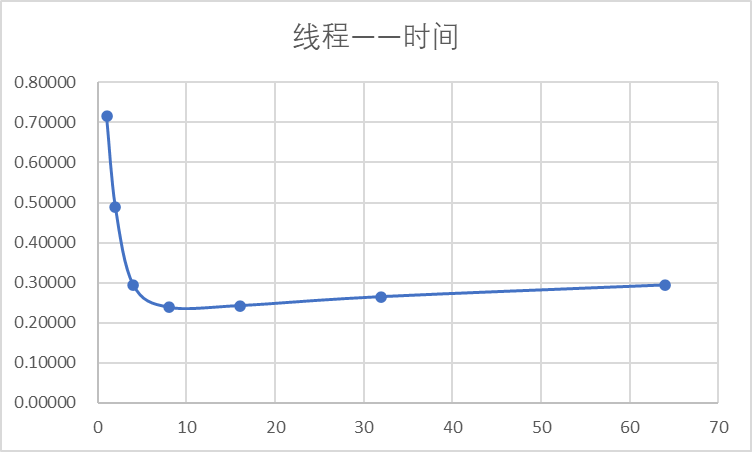
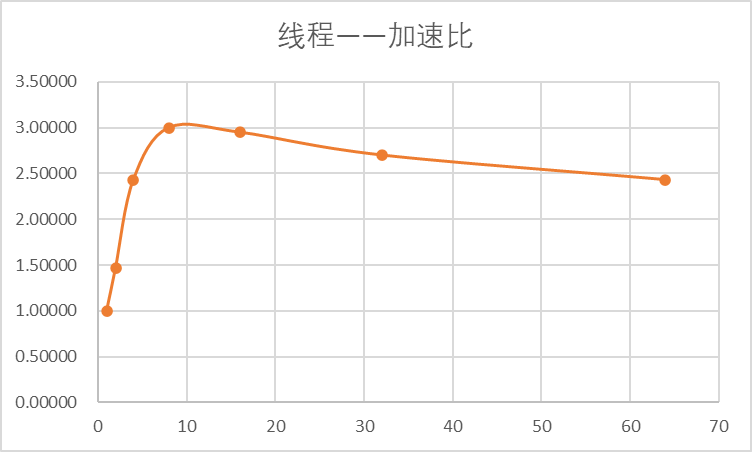
 

图1 法一线程——时间图 图2 法一线程——加速比

实验分析：根据实验数据结合图表分析可知，随着线程数量的增加，并行计算前缀和的执行时间明显减少，其下降趋势在开始时较快，后续趋于平缓。同理可以看到加速比与线程数的关系，随着线程数的增加，加速比先上升的较快，后续趋于平缓。结合上述两个分析可以知道，适当的使用多线程可以有效的提升程序运行效率，但是当线程数超过一定数量后，加速比的提升会逐渐放缓。分析并行的时间复杂度，分块的情况下，第一阶段计算各个块的时间复杂度为，第二阶段累加前面的块的和时间复杂度为，因此总的复杂度约为，但实验数据显示，加速比远没有达到理想情况，综合分析，可能是因为cache未命中等其他硬件原因

表2 并行程序（法二）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 | 3.87054 | 2.78095 | 2.12362 | 2.07138 | 2.20846 | 2.21147 | 2.10321 |
| 第2次 | 3.83935 | 2.81322 | 2.15463 | 2.24403 | 1.98784 | 2.22671 | 2.14260 |
| 第3次 | 3.85582 | 3.00038 | 2.09154 | 1.71356 | 1.94734 | 2.35720 | 1.68484 |
| 第4次 | 3.87202 | 2.48199 | 2.08697 | 1.69847 | 1.95124 | 2.53028 | 1.60538 |
| 第5次 | 3.84458 | 2.64871 | 2.12542 | 1.33840 | 1.70294 | 2.00845 | 1.42696 |
| 平均值 | 3.85646 | 2.74505 | 2.11643 | 1.81317 | 1.95956 | 2.26682 | 1.79260 |
| 加速比 | 1.00000 | 1.40488 | 1.82215 | 2.12692 | 1.96802 | 1.70126 | 2.15132 |

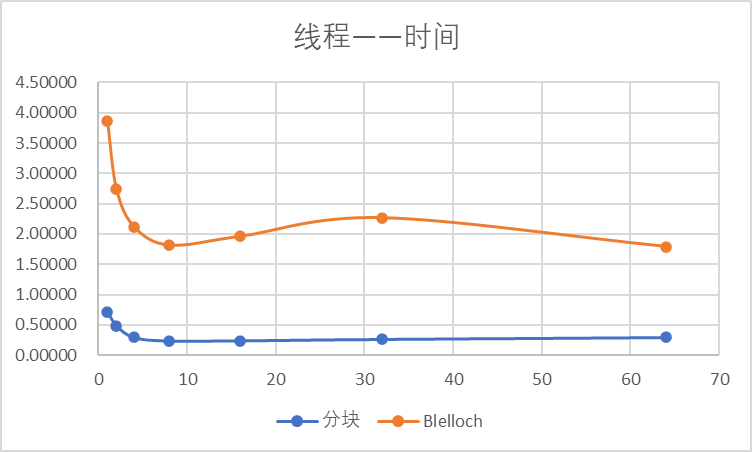
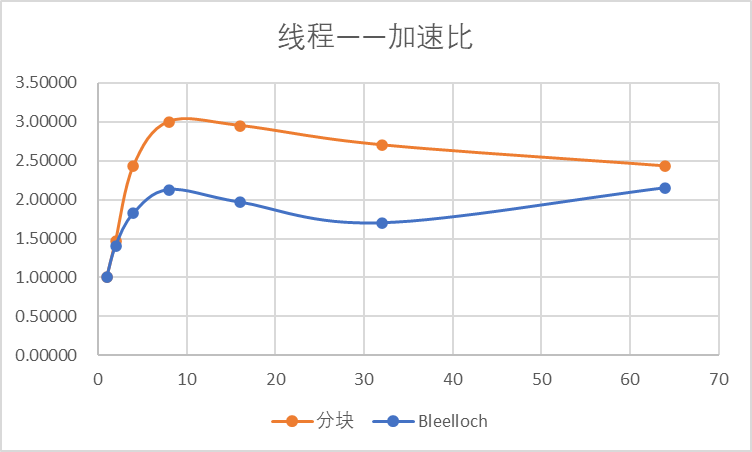
 

图3 法二线程——时间图 图4 法二线程——加速比图

实验分析：本次实验还使用了另外一种方法Blelloch并行扫描法，简单的来说是通过二叉树的方式来计算前缀和，分为两个阶段，向上阶段和向下阶段：

（1）向上（Upsweep）阶段：

·该阶段是通过一棵隐含的二叉树自底向上构建累积和。

·在此过程中，每一个节点汇聚子节点的值。

·经过 log(n) 次迭代，最终根节点处存储了整个数组的总和。

（2）向下（Downsweep）阶段：

·在这个阶段，算法从树的根节点开始，自顶向下计算前缀和，同时传播累积和到子节点。

·在每一层中，左子节点继承父节点的值，而右子节点加上父节点的值。

如下图为举例的求解例子：

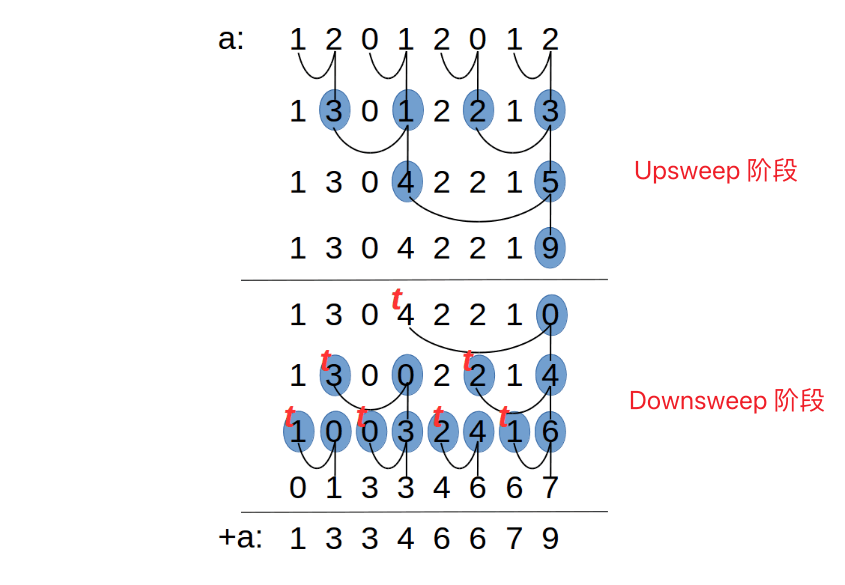


图5 Blelloch法例子

经过这种方法计算的时间复杂度为，但是实际情况下难以获得足够多的计算单元，要获得理论的时间复杂度需要用到 个核，因此实际的时间复杂度约为，得到的实际运行时间会比串行还要慢，这是因为出了并行数量不够外，cache的命中率也大幅下降，需要频繁调用内存，线程之间还可能存在总线竞争的问题。对比两种并行方法，前者所需的时间远小于后者，但趋势是现近的。

## 六、实验结论

*实验过程中遇到的问题及解决办法，运用了哪些技术方法以提高实验性能，从该实验得到的客观结论，等等。*

本次实验学习了如何利用并行算法计算前缀和，学习和分析了多种方法，实验发现分块法并行的时间是最短的，而Blelloch并行扫描的实际时间会比串行还要长。从实验数据可以看到，随着线程数的增加，求解长度前缀和所需的时间逐渐减少，开始时下降快后续趋于平缓，而加速比逐渐增加，但实际加速比值与分析的时间复杂度并不一致，还存在其他的制约因素。如下表是两种并行方法的简单总结：

表3 分块法与Blelloch法对比总结

| 算法 | 分块法 | Blelloch算法 |
| --- | --- | --- |
| 时间复杂度 |  |  |
| 并行度 | 依赖p，受限于串行累加 | 完全并行（无串行部分） |
| 适用场景 | p ≪ n | p ≈ n |
| 实现复杂度 | 逻辑分块 | 通过二叉树实现 |

实验遇到计算多组数据时同样线程数但每一轮的时间都在增加，分析发现是因为在第一次计算完前缀和后没有恢复数组，导致后续结果一直在累加，随着数据的增大，计算量也在不断增加，因此总时间增加，解决方式为添加一个temp临时数组，复制输入数据，每次计算得到结果并于串行比较完后即可释放。实验刚开始时没有考虑边界问题，最后一组数据数量相对其他组数据会少一些，如果按照一样的块大小进行计算，最后一组会出现数据溢出的问题，添加一个min来计算边界值即可。

实验发现并行算法有时并不一定会比串行算法更优，这是因为在设计算法时是理想情况，没有考虑实际的线程数量以及硬件上的影响，比如在本次使用到的Blelloch算法中，要达到最优的时间复杂度需要用到 个线程，在当前数据量（1e-8）的情况下显然是不现实的，并且通过频繁的数组间隔替换也会导致cache失效，需要频繁访问内存，一定程度上也增加了运行时间。