**深 圳 大 学 实 验 报 告**

|  |  |
| --- | --- |
| **课程名称** | **：并行计算** |
| **实验项目名称** | **：归并的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16日** |
| **实验报告提交时间** | **：2025年04月16日** |

**教务部制**

## 一、实验目的

1. 设计归并的并行算法；

2. 掌握数据划分方法；

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，将有序数组*a*和*b*归并到数组*c*中。将数组*a*等分为*p*段，并选前*p*-1段的最后一个元素作为划分元，将数组*b*也分为*p*段；线程*i*归并数组*a*和*b*的第*i*段。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。*

|  |
| --- |
| #include <omp.h>  #include <iostream>  #include <algorithm>  #include <iomanip>  using namespace std;  double table[70][10];  double serial\_time = 0; // 存储串行版本的运行时间  // 串行归并函数：合并两个已排序数组a和b到c中  void serial\_merge(double\* a, double\* b, double\* c, int n) {  int i\_a = 0, i\_b = 0, i\_c = 0;  // 比较两个数组当前元素，取较小者  while (i\_a < n && i\_b < n) {  if (a[i\_a] < b[i\_b]) c[i\_c++] = a[i\_a++];  else c[i\_c++] = b[i\_b++];  }  // 处理剩余元素  while (i\_a < n) c[i\_c++] = a[i\_a++];  while (i\_b < n) c[i\_c++] = b[i\_b++];  }  // 并行归并函数  void parallel\_merge(double\* a, double\* b, int n, int p, double\* c\_serial) {  int k = n / p; // 每个线程处理的a数组基本块大小  bool validation = false; // 验证标志  double total\_time = 0; // 累计5次运行的总时间  for (int run = 0; run < 5; run++) { // 运行5次取平均值  double\* c = new double[2 \* n] {0}; // 结果数组  double t0 = omp\_get\_wtime(); // 开始计时  // 确定a数组的分割点（每个分区的最后一个元素）  double\* split\_points = new double[p - 1];  for (int i = 0; i < p - 1; i++)  split\_points[i] = a[(i + 1) \* k - 1];  // 在b数组中查找对应的分割点  int\* b\_splits = new int[p + 1];  b\_splits[0] = 0; // 起始位置  b\_splits[p] = n; // 结束位置  for (int i = 0; i < p - 1; i++) {  // 使用二分查找确定分割位置  double\* pos = lower\_bound(b + b\_splits[i], b + n, split\_points[i]);  b\_splits[i + 1] = pos - b;  }  // 计算每个线程在结果数组c中的写入偏移量  int\* c\_offsets = new int[p + 1] {0};  for (int i = 0; i < p; i++)  c\_offsets[i + 1] = c\_offsets[i] + k + (b\_splits[i + 1] - b\_splits[i]);  // 并行归并 - 每个线程处理自己的分区  #pragma omp parallel for  for (int i = 0; i < p; i++) {  // 确定当前线程处理的区间  int a\_beg = i \* k;  int a\_end = (i + 1) \* k;  int b\_beg = b\_splits[i];  int b\_end = b\_splits[i + 1];  int c\_beg = c\_offsets[i];  // 标准归并操作（与串行版本相同）  while (a\_beg < a\_end && b\_beg < b\_end) {  if (a[a\_beg] < b[b\_beg]) c[c\_beg++] = a[a\_beg++];  else c[c\_beg++] = b[b\_beg++];  }  while (a\_beg < a\_end) c[c\_beg++] = a[a\_beg++];  while (b\_beg < b\_end) c[c\_beg++] = b[b\_beg++];  }  // 记录本次运行时间  double t1 = omp\_get\_wtime();  table[p][run] = t1 - t0;  total\_time += t1 - t0;  // 第一次运行结果与串行版本比较  if (run == 0) {  validation = true;  for (int i = 0; i < 2 \* n; i++) {  if (c[i] != c\_serial[i]) {  validation = false;  break;  }  }  }  // 释放内存  delete[] c;  delete[] split\_points;  delete[] b\_splits;  delete[] c\_offsets;  }  // 计算并记录平均时间和加速比  double avg\_time = total\_time / 5;  table[p][5] = avg\_time;  table[p][6] = serial\_time / avg\_time;  // 输出结果  cout << "p = " << p << "\tAvg Time: " << avg\_time << "s\t";  cout << "Validation: " << (validation ? "Passed" : "Failed") << endl;  }  int main() {  const int n = 1e8; // 数组大小  int p\_values[] = { 1, 2, 4, 8, 16, 32, 64 }; // 测试的线程数  int num\_p = sizeof(p\_values) / sizeof(p\_values[0]);  double\* a = new double[n];  double\* b = new double[n];  // 生成有序随机数组  for (int i = 0; i < n; i++) {  a[i] = (i ? a[i - 1] : 0) + rand() / (double)RAND\_MAX;  b[i] = (i ? b[i - 1] : 0) + rand() / (double)RAND\_MAX;  }  // 串行归并结果用于验证  double\* c\_serial = new double[2 \* n];  double t0 = omp\_get\_wtime();  serial\_merge(a, b, c\_serial, n);  double t1 = omp\_get\_wtime();  serial\_time = t1 - t0;  cout << "Serial Time: " << serial\_time << "s" << endl;  // 测试不同线程数下的并行性能  for (int pi = 0; pi < num\_p; pi++) {  int p = p\_values[pi];  parallel\_merge(a, b, n, p, c\_serial);  }  // 打印结果表格  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 : p\_values) {  cout << fixed << setprecision(6) << table[j][i] << " ";  }  cout << endl;  }  // 释放内存  delete[] a;  delete[] b;  delete[] c\_serial;  return 0;  } |

## 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 并行程序在不同线程数下的执行时间（秒）和加速比（*n*=1e8）

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 线程数  执行时间 | 1 | 2 | 4 | 8 | 16 | 32 | 64 |
| 第1次 | 1.173870 | 0.715047 | 0.441447 | 0.320070 | 0.336050 | 0.328868 | 0.335014 |
| 第2次 | 1.273474 | 1.191981 | 0.521149 | 0.316137 | 0.318555 | 0.339533 | 0.337332 |
| 第3次 | 1.172402 | 1.328525 | 0.442877 | 0.309179 | 0.317823 | 0.350207 | 0.335141 |
| 第4次 | 1.224453 | 0.632799 | 0.386543 | 0.305282 | 0.330946 | 0.354838 | 0.332525 |
| 第5次 | 1.174134 | 1.748520 | 0.617497 | 0.332262 | 0.318650 | 0.341820 | 0.337475 |
| 平均值 | 1.203667 | 1.123375 | 0.481903 | 0.316586 | 0.324405 | 0.343053 | 0.335497 |
| 加速比 | 1.000000 | 1.871637 | 4.363018 | 6.641325 | 6.481248 | 6.128935 | 6.266962 |

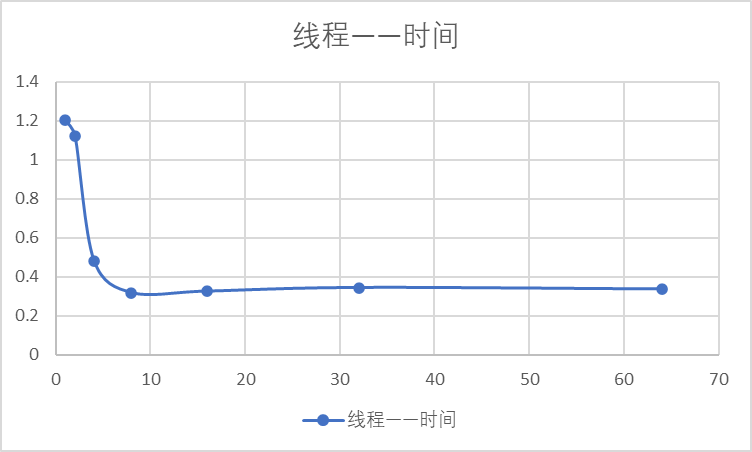
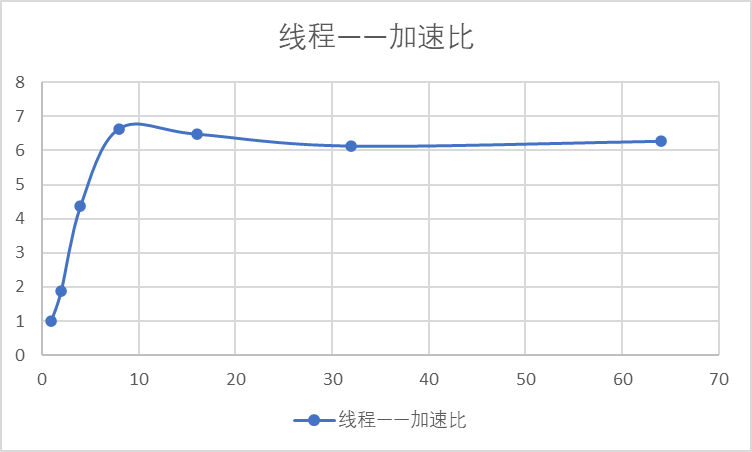
 

图1 线程——时间图 图2 线程——加速比

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。

分析时间复杂度计算理论加速比：串行归并的时间为，并行归并的时间为：

①a数组分割：取a的每块的最后一个元素作为分割点，共执行p次，

②b数组分割：对a的分割点，在b中使用二分寻找大于等于分割点的第一个数据，共执行p-1次，每次的时间为*logn*，时间复杂度为

③计算c的偏移，计算每个块的大小作为c的偏移量，时间为

④并行归并：每个线程归并块内的a和b，假设a、b分块均匀，每个块的大小约为，因此归并时间为

因此总时间复杂度为，。根据得到复杂度结合数据分析可知，当*p*较小时，耗时较长，因此时间呈现反比例函数的形式减少，当*p*逐渐增大，最终会趋于对数平缓。

## 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。*

本实验通过OpenMP实现了归并排序的并行化。分析数据结果可知，合理控制线程数、优化内存管理是提升并行性能的关键。实际应用中需结合硬件环境调整并行策略，避免过度并行化带来的额外开销。此外，并行算法并非在所有场景下优于串行算法。

具体结论如下：

①线程数与性能关系：

当线程数从1增至8时，执行时间显著降低，并行效果明显。当线程数超过16后，加速比趋于平缓，甚至因线程调度开销和内存带宽限制略有下降，过度并行化会引入额外开销。

②时间复杂度：

理论复杂度为。当较小时主导时间，加速效果明显；当较大时，项逐渐成为瓶颈，限制加速比提升。

③并行适用性：

实际加速比与理论值存在偏差，受到内存访问延迟、线程调度开销等影响。