**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** | **：何泽锋** |
| **学号** | **：2022150221** |
| **实验时间** | **：2025年03月19日** |
| **实验报告提交时间** | **：2025年03月19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.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。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 <math.h>  using namespace std;  const int MAX = 100;  const int N = 5000000;// 查找完数的上限  int a[MAX];// 存储并行计算找到的完数  int s[MAX];// 存储串行计算找到的完数  int pn = 0;// 完数的数量  double baseline = 0;// 串行执行的基准时间  inline bool perfect(int num) {// 检查一个数是否为完数  if (num < 2) return false;  int sum = 1;  for (int i = 2; i<= sqrt(num); i++) {  if (num % i == 0) {  sum += i;  if (i != num / i) {  sum += num / i;  if (sum > num)return 0;  }  }  }  return sum == num;  }  void Serial() {// 串行查找完数  pn = 0;  double t0 = omp\_get\_wtime();  for (int i = 2; i <= N; i++) {  if (perfect(i)) {  if (pn < MAX) {  s[pn++] = i;  }  }  }  double t1 = omp\_get\_wtime();  baseline = t1 - t0;// 串行执行时间  printf("Serial time: %lf s\n", t1 - t0);  printf("Total pefect number: %d\n", pn);  for (int i = 0; i < pn; i++) {// 记录答案  printf("%d ", s[i]);  }  printf("\n");  }  bool is\_prime(int num) {  if (num <= 1) return 0;  for (int i = 2; i <= sqrt(num); i++) {  if (num % i == 0) return 0;  }  return 1;  }  void Parallel(int tread\_num) {// 并行查找完数  cout << endl << "Thread count:" << tread\_num << endl;  int t = 5;  double t0, t1;  double T[5];// 存储每次实验的执行时间  for (int x = 0; x < t; x++) {  pn = 0;  omp\_set\_num\_threads(tread\_num);  t0 = omp\_get\_wtime();  #pragma omp parallel for schedule(static) shared(a,pn)  for (int i = 2; i <= N; i++) {  if (perfect(i)) {  #pragma omp critical// 确保以下代码块不会并行执行  {  if (pn < MAX) {  a[pn++] = i;  }  }  }  }  t1 = omp\_get\_wtime();  T[x] = t1 - t0;// 并行执行时间  printf("Parallel time: %lf s \n", t1 - t0);  sort(a, a + pn);  for (int i = 0; i < pn; i++) {// 测试结果  if (a[i] != s[i]) {  printf("ERROR\n");  exit(0);  }  }  printf("PASS CHECK\n");  }  double at = 0;  for (int i = 0; i < t; i++) {  at += T[i];  }  at /= t;  printf("Average Parallel time: %lf s \n", at);  printf("Acceleration rate: %lf\n", baseline / at);// 加速比  }  int main() {  Serial();  Parallel(1);  Parallel(2);  Parallel(4);  Parallel(8);  Parallel(16);  Parallel(32);  Parallel(64);  } |

*串行代码优化，通过欧几里得完全数公式寻找完美数，如果 是素数，则 是完全数。*

|  |
| --- |
| bool is\_prime(int num) {  if (num <= 1) return 0;  for (int i = 2; i <= sqrt(num); i++) {  if (num % i == 0) return 0;  }  return 1;  }  void optimize\_perfect() {  double t0 = omp\_get\_wtime();  pn = 0;  for (int p = 2; ; p++) {  int mersenne\_prime = pow(2, p) - 1; // 确保2^p-1是素数  if (!is\_prime(mersenne\_prime)) continue;  int perfect\_number = pow(2, p - 1) \* mersenne\_prime;  if (perfect\_number > N) break; // 超过n则停止搜索  s[pn++] = perfect\_number;  }  double t1 = omp\_get\_wtime();  double tmp = t1 - t0;  printf("Optimized Serial time: %lf s\n", t1 - t0);  printf("\n");  } |

## 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=5000000）

（schedule采用dynamic）

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 线程数  执行时间 | 1 | 2 | 4 | 8 | 16 | 32 | 64 |
| 第1次 | 24.562098 | 12.345081 | 6.234764 | 3.124087 | 1.659812 | 0.959508 | 0.967051 |
| 第2次 | 24.564014 | 12.358426 | 6.274594 | 3.155504 | 1.663743 | 0.956750 | 0.963495 |
| 第3次 | 32.616068 | 12.579086 | 6.267970 | 3.137858 | 1.664267 | 0.963873 | 0.962948 |
| 第4次 | 24.562374 | 12.509148 | 6.255590 | 3.138491 | 1.666439 | 0.958208 | 0.967333 |
| 第5次 | 24.563270 | 12.508991 | 6.255770 | 3.137660 | 1.666515 | 0.958567 | 0.961377 |
| 平均值 | 24.562909 | 12.460146 | 6.257738 | 3.138720 | 1.664155 | 0.959381 | 0.964441 |
| 加速比 | 1 | 2.003768 | 3.989819 | 7.954593 | 15.002953 | 26.024322 | 25.887787 |

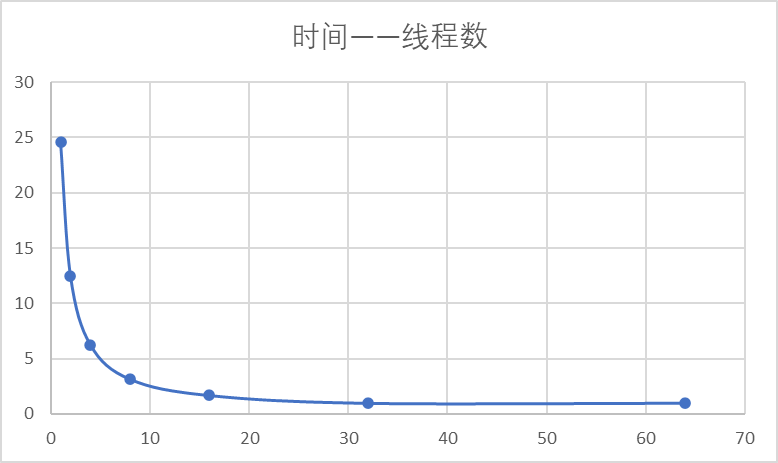
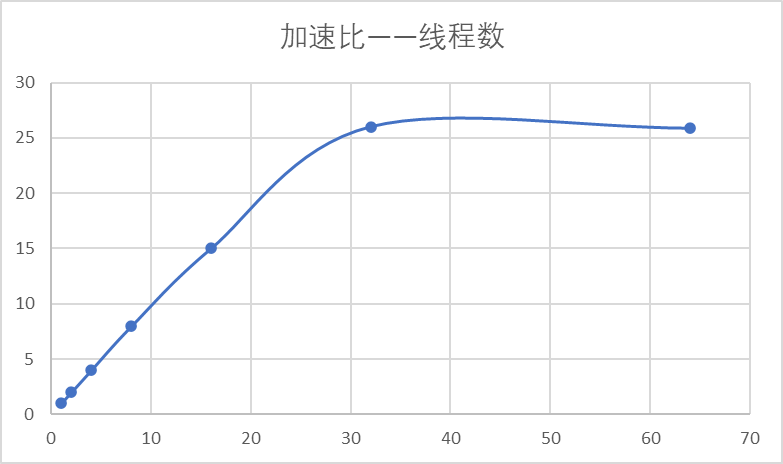
 

图1 时间——线程折线图（dynamic） 图2 加速比——线程折线图（dynamic）

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，并行计算完全数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。代码中还使用到了critical，这是为了确保当找到完全数时记录数量时不会出现冲突，在这个问题下，发生错误的概率非常低，在N=5000000的范围内仅有4个完全数，因此多个线程同时写入的情况基本不会出现，但为了代码的严谨性，此处仍然使用。需要注意用数组存储并行结果后需要排序，防止进程运行速度不同导致写入顺序不同。

表2 并行程序在不同线程数下的执行时间（秒）和加速比（n=5000000）

（schedule采用static）

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 线程数  执行时间 | 1 | 2 | 4 | 8 | 16 | 32 | 64 |
| 第1次 | 24.488356 | 16.084175 | 8.861386 | 4.520359 | 2.379474 | 1.192073 | 1.003759 |
| 第2次 | 24.488697 | 15.903841 | 8.838975 | 4.472906 | 2.285246 | 1.188653 | 0.946010 |
| 第3次 | 24.488729 | 15.904285 | 8.716618 | 4.635392 | 2.273850 | 1.185770 | 0.995569 |
| 第4次 | 24.488890 | 15.904097 | 8.627437 | 4.473512 | 2.273819 | 1.182082 | 0.979902 |
| 第5次 | 24.489559 | 15.904128 | 8.627596 | 4.473477 | 2.273848 | 1.185959 | 0.913747 |
| 平均值 | 24.488846 | 15.940105 | 8.734402 | 4.515129 | 2.297248 | 1.186907 | 0.967797 |
| 加速比 | 1 | 1.548568 | 2.826104 | 5.467027 | 10.745177 | 20.797186 | 25.505681 |

表3 并行程序在不同线程数下的执行时间（秒）和加速比（n=5000000）

（schedule采用guided）

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 线程数  执行时间 | 1 | 2 | 4 | 8 | 16 | 32 | 64 |
| 第1次 | 24.490681 | 12.246865 | 6.123371 | 3.062203 | 1.544518 | 0.783826 | 0.786014 |
| 第2次 | 24.491656 | 12.246592 | 6.165726 | 3.061869 | 1.531269 | 0.765891 | 0.802863 |
| 第3次 | 24.495166 | 12.246569 | 6.123791 | 3.061923 | 1.531334 | 0.765913 | 0.789920 |
| 第4次 | 24.530069 | 12.246603 | 6.123604 | 3.061871 | 1.531327 | 0.818097 | 0.786828 |
| 第5次 | 24.655071 | 12.246625 | 6.123678 | 3.079410 | 1.531261 | 0.781293 | 0.788165 |
| 平均值 | 24.532529 | 12.246651 | 6.132034 | 3.065455 | 1.533942 | 0.783004 | 0.790758 |
| 加速比 | 1 | 2.015600 | 4.025474 | 8.052425 | 16.092099 | 31.525183 | 31.216050 |

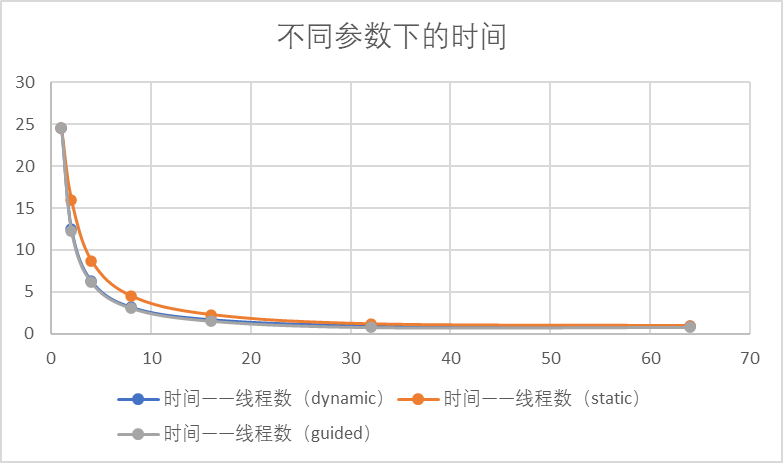
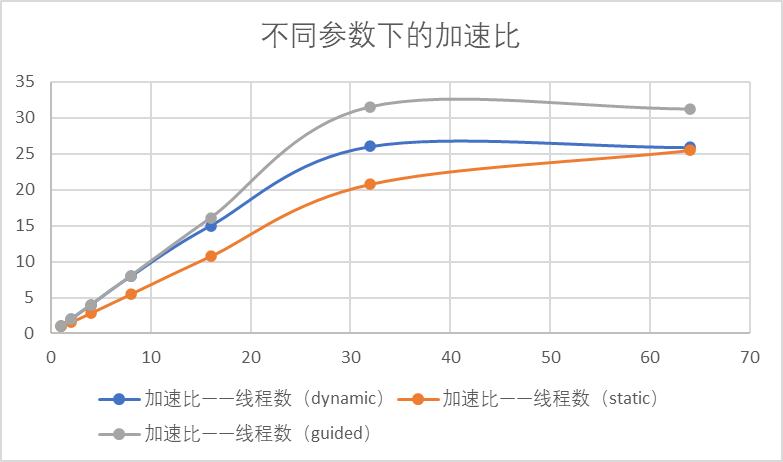
 

图3 不同参数下的时间——线程数对比 图4 不同参数下的加速比——线程数对比

实验分析：测试了schedule不同的参数下的运行时间，通过上述数据对比dynamic、static和guided的加速比，可以看到在线程数未超过32前guided的加速比基本上接近线程的数量，而static的加速比明显低于线程数量，这是因为static是预先将固定数量的循环分配给每个线程，而在完全数问题下越靠后的循环所需时间越长，因此按照循环数量提前划分会导致部分线程运行的时间较长，造成程序整体时间变长；guided是分配的循环数量是随代码运行逐渐减少的，初始时每个循环所需时间较少，分配更多的循环可以减少分配时的消耗，而循环大时每个循环结束时间不一致，分配更少的循环可以避免，将大量循环分配给同一个线程；dynamic是根据线程的实际运行情况分配循环，当线程空闲时才将循环分配下去，避免了部分进程结束较早的等待问题。

总结来说，guided可以防止static情况下一个线程运行较长时间的循环，也可以减少dynamic动态分配的进程切换时间。

表4 并行程序在不同size下的执行时间（秒）

（schedule采用dynamic，16线程）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Size  执行时间 | 1 | 10 | 100 | 1000 | 10000 |
| 平均执行时间 | 1.612497 | 1.535006 | 1.531689 | 1.544027 | 1.589486 |

实验分析：经过测试可以发现在dynamic情况下size对执行时间影响不大，代码运行时间也存在误差。

## 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OpenMP并行循环中schedule的使用，schedule主要用于循环不是均等的情况。schedule有两个参数分别是kind和size，本次实验主要分析了kind的区别，用到了dynamic、static以及guided，经过数据测试发现guided所需运行时常最短，分析发现这是因为static固定了每个线程循环分配的大小，这会导致较长的循环会集中在其中一个线程中，而dynamic因为每次分配的线程数是固定的，但循环早期运行较快时会有频繁的分配，导致部分时间花费在线程分配上。而guided分配的循环数量是逐渐减少的，更符合完全数计算问题。除此之外也用到了critical，可以让指定的代码段不并行计算，防止同时写入的数据冲突问题。

除了kind参数外，也测试了size，但在当前问题下size对运行时间影响不大，不能很好的得出对循环分配的影响，因此不做过多分析

实验中用到了欧几里得完全数公式寻找完美数，这是一种数学方法，时间复杂度约为因此运行速度比多线程还快。该公式证明了偶完数都遵循 （如果是素数），而对于奇完数，目前在有效的数据范围内都没能找到，因此将该公式用于数据范围为5000000的问题上能有效减少串行运行时间。

对代码进行了一部分优化，首先在求解素数上，通过开根号的方式减少了因数的求解时间，而在求解完数上，若当前因数的和已经大于原数的那么就可以提前退出，可以减少不必要的因数求解。