**深 圳 大 学 实 验 报 告**

|  |  |
| --- | --- |
| **课程名称** | **：并行计算** |
| **实验项目名称** | **：矩阵乘法的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（创新班）** |
| **指导教师** | **：陆克中** |
| **报告人** | **：何泽锋** |
| **学号** | **：2022150221** |
| **实验时间** | **：2025年03月05日** |
| **实验报告提交时间** | **：2025年03月09日** |

**教务部制**

## 一、实验目的

1. 学会编写简单的OpenMP程序；

2. 掌握for编译制导语句；

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编写两个*n*阶方阵*a*和*b*的并行相乘程序，结果存放在方阵*c*中，其中矩阵乘法部分用for编译制导语句实现并行化操作。为了验证结果正确性，将矩阵乘法的串行计算结果存放在方阵*d*中，并比较是否与*c*相等。可利用缓存命中率和程序的局部性原理来优化矩阵乘法。

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1000，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 <stdio.h>  #include <math.h>  #include <iostream>  using namespace std;  const int n = 1000;  double a[n][n], b[n][n], c[n][n], d[n][n];  double tread1\_time = 0;  void test(int tread\_num) { // 线程控制函数，用于区分不同数量的线程  cout << "Thread count:" << tread\_num << endl;  int t = 5;  double t0, t1;  double T[5];  for (int x = 0; x < t; x++) {  int i, j, k;  for (i = 0; i < n; i++)  for (j = 0; j < n; j++)  {  a[i][j] = rand() \* 1.0 / RAND\_MAX; // 初始化随机数  b[i][j] = rand() \* 1.0 / RAND\_MAX;  c[i][j] = d[i][j] = 0;  }  t0 = omp\_get\_wtime(); // 记录并行开始时间  omp\_set\_num\_threads(tread\_num); // 设置线程数  //调整循环顺序，计算矩阵乘法  #pragma omp parallel for private(i, j, k) shared(a, b, c)  for (i = 0; i < n; i++) {  for (k = 0; k < n; k++) {  double a0 = a[i][k]; // 减少缓存未命中  for (j = 0; j < n; j++) {  c[i][j] += a0 \* b[k][j]; // 连续访问b和c的内存  }  }  }  t1 = omp\_get\_wtime(); // 记录并行结束时间  T[x] = t1 - t0;  cout << "Time: " << t1 - t0 << endl;  //验证代码  for (i = 0; i < n; i++) // 串行计算矩阵乘法  for (j = 0; j < n; j++)  for (k = 0; k < n; k++)  d[i][j] += a[i][k] \* b[k][j];  for (i = 0; i < n; i++) // 比较串行与并行的计算结果  for (j = 0; j < n; j++)  if (fabs(c[i][j] - d[i][j]) > 1e-6) { // 数据是浮点数不能直接采用等号  cout << "ERROR" << endl;  break;  }  cout << "PASS" << endl;  }  double sum\_t = 0;  for (int x = 0; x < t; x++) {  sum\_t += T[x]; //累加时间  }  sum\_t /= t; //计算时间均值  cout << "Average time: " << sum\_t << endl;  if (tread\_num == 1) { // 记录单线程情况下的运行时间  tread1\_time = sum\_t;  cout << endl;  }  else cout << "Acceleration ratio: " << tread1\_time/sum\_t << endl << endl; // 计算加速比  }  int main() {  test(1);  test(2);  test(4);  test(8);  test(16);  test(32);  test(64);  } |

使用分块优化矩阵乘法，将大矩阵分成小块进行计算，提高缓存利用率和计算效率

|  |
| --- |
| int block\_size = 32;  for (int ii = 0; ii < n; ii += block\_size) { // 按块大小遍历矩阵a的行  for (int kk = 0; kk < n; kk += block\_size) { // 按块大小遍历矩阵a的列和矩阵b的行  for (i = ii; i < min(n,ii+block\_size); i++) {  for (k = kk; k < min(n,kk+block\_size); k++) {  double a0 = a[i][k]; // 减少缓存未命中  for (j = 0; j < n; j++) {  c[i][j] += a0 \* b[k][j]; // 连续访问b和c的内存  }  }  }  }  } |

## 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=1000）

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 线程数  执行时间 | 1 | 2 | 4 | 8 | 16 | 32 | 64 |
| 第1次 | 0.441311 | 0.271175 | 0.123078 | 0.0710606 | 0.0619337 | 0.0614512 | 0.0391916 |
| 第2次 | 0.422868 | 0.231426 | 0.123232 | 0.0760164 | 0.0523515 | 0.0405915 | 0.033794 |
| 第3次 | 0.422564 | 0.231592 | 0.123199 | 0.0839909 | 0.0690198 | 0.0360399 | 0.0335545 |
| 第4次 | 0.422658 | 0.239442 | 0.126813 | 0.0881968 | 0.0683067 | 0.0374341 | 0.0366976 |
| 第5次 | 0.422804 | 0.23217 | 0.125352 | 0.0890195 | 0.0688638 | 0.0369814 | 0.0428664 |
| 平均值 | 0.426441 | 0.241161 | 0.124335 | 0.0816569 | 0.0640951 | 0.0424996 | 0.0372208 |
| 加速比 | 1 | 1.76828 | 3.42979 | 5.22235 | 6.65325 | 10.034 | 11.4571 |

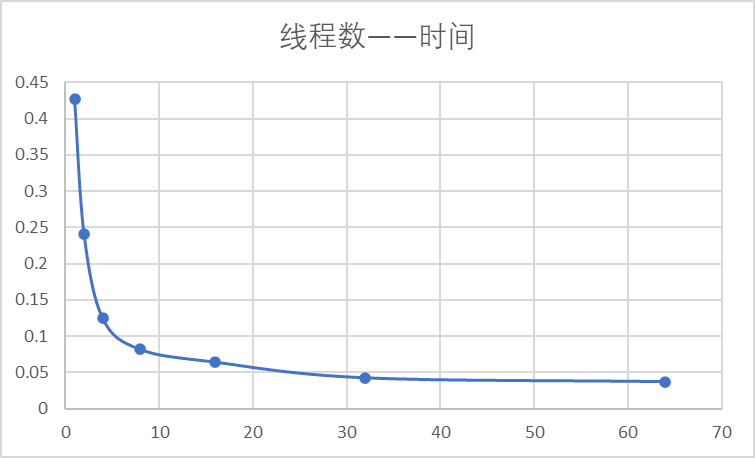
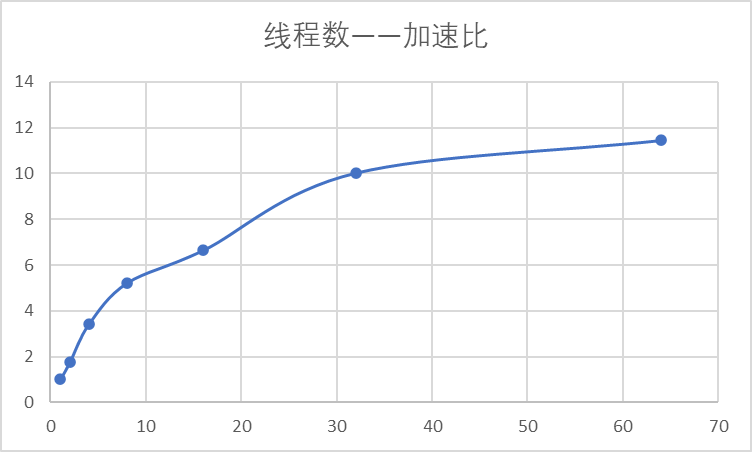
 

图1 线程数——时间折线图 图2 线程数——加速比折线图

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，并行矩阵乘法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。

表2 矩阵乘法优化比较

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 线程  优化 | 1 | 2 | 4 | 8 | 16 | 32 | 64 |
| 未优化 | 0.353109 | 0.198008 | 0.103788 | 0.0616719 | 0.0353018 | 0.0311098 | 0.0307965 |
| 优化循环 | 0.304583 | 0.160016 | 0.0878543 | 0.049653 | 0.0323455 | 0.0236944 | 0.0219726 |
| 循环分块 | 0.132089 | 0.062546 | 0.0321066 | 0.0232649 | 0.017273 | 0.0119807 | 0.0191317 |

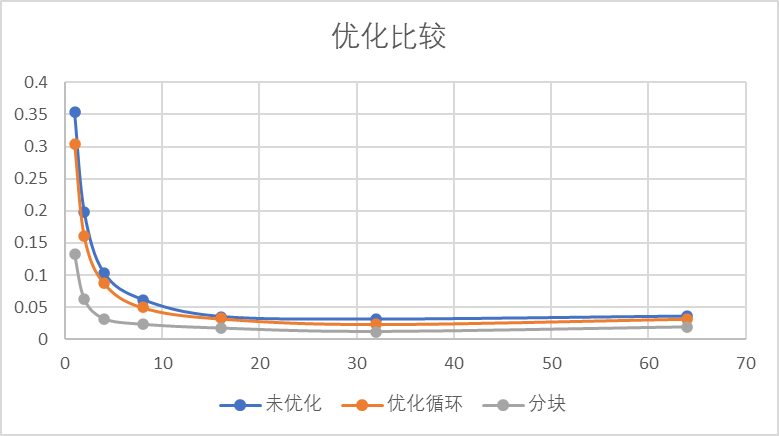


图3 优化比较折线图

实验分析：实验比较了定义法计算矩阵乘法、优化循环以及矩阵分块的矩阵乘法，比较定义法和优化循环，可以看到在不同数量的线程下优化结果都比未优化所需时间短，这是因为相乘的时候，数组b内存空间中相连的数据（空间局部性原理）会一次加载到缓存中，此时按照行遍历即可充分利用数据，增加cache的命中率，从而减少内存调用的时间，优化了程序的性能。在优化循环的前提下再使用矩阵分块，当矩阵块的大小合适时，可以完全放入缓存中，减少内存访问次数，加快程序运行。

## 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。*

本次实验学习使用了OpenMP的并行循环方式，通过并行的方式计算矩阵乘法，能够有效的减少程序运行时间，通过数据分析可知，随着线程数的增加，程序所需时间下降的速率逐渐变缓（加速比上升的速度也逐渐变缓），说明并行技术并不能无限制的优化程序，只能在有限范围内加速。因此本次实验也尝试使用了其他优化方式，通过使用行优先的循环遍历方式，增加cache的命中率，进而减少内存调用的频率，加速了矩阵计算。使用矩阵分块技术，在本地运行时有明显的提升，但使用服务器运行时基本没有提升，修改块的大小也基本没有优化，分析原因应该是服务器的缓存资源不足，无法充分利用分块。

实验问题，初次使用服务器时，尝试直接运行a.cpp文件但是编译过程中报错没有链接OpenMP库，使用实验文档中的指令并开启O3优化后得到a.out文件，再运行即可解决。实验在服务器上进行测试时发现，使用A0和A1服务器的时间有一定区别，A1的速度更快。在测试数据的时候，发现每次运行的首组数据与其他数据有一定的差别，时间会更长，分析是程序未预热，cpu缓存为空，需要时间分配资源等，因此为了使测试更准确，可以将串行计算矩阵乘法的代码调整到并行前，使得并行测试时数据较为稳定。