

# 《并行程序设计》实验手册

（2019年4月版）

**郑重声明：**

**1、实验手册中的所有实验均有本人独立编码、调试和测试。**

**2、实验手册中给出的实验数据和结果完全由本人所完成的程序给出。**

**3、本人了解：不按照前两条要求所完成的实验报告已经构成了抄袭或造假行为，本人将承担相应的不良后果。**

姓名： （签名） 学号：

提交日期： 总成绩：

本课程以设计竞赛方式评分，执行速度排名与分数的比例如下表。

|  |  |  |  |
| --- | --- | --- | --- |
| 执行速度排名 | 分数 | 执行速度排名 | 分数 |
| 前10% | 100% | 前10～30% | 85% |
| 前40～60% | 70% | 前60～85% | 60% |
| 最后15%或程序执行崩溃，或结果不正确 | | | 50% |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 题号 | 总分 | 程序执行时间(s) | 输入参数 | 运行结果 | 排名 | 得分 |
| 1 | 35 |  |  |  |  |  |
| 2 | 35 |  |  |  |  |  |
| 报告 | 30 |  |  |  |  |  |
| 总分 | | | | | |  |

测试员： （签名）指导教师： （签名）

本实验所使用的CPU平台为：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| CPU | | | Cache | | | 操作系统 | 编译器 | |
| 型号 | 核数 | 主频 | L1 | L2 | L3 | 版本 | 优化参数 |
| XEON E2620 | 12 | 1.2GHZ | 32KB | 256KB | 15MB | CentOS6.4 | gcc 4.4.7 |  |

使用GPU平台的型号是Tesla K10，核数为3072，主频为745MHZ。

## 1. 矩阵乘法（50分）

实现单精度浮点的矩阵乘法：，并求矩阵C的迹Trace。

**程序输入输出要求**

Matrix\_mul *N seed*

参数：

*N:* 矩阵大小

*seed*：输入的种子浮点数

输出格式：

Trace

Trace: 结果矩阵C的迹（），用于验证矩阵乘法的正确性。

矩阵产生函数

/\*

Input: a, b are the N\*N float matrix, 0<seed<1, float

This function should initialize two matrixs with rand\_float()

\*/

float rand\_float(float s){

return 4\*s\*(1-s);

}

void matrix\_gen(float \*a,float \*b,int N,float seed){

float s=seed;

for(int i=0;i<N\*N;i++){

s=rand\_float(s);

a[i]=s;

s=rand\_float(s);

b[i]=s;

}

}

**1. 【基准程序】（7分）**

输入seed=0.3，调整矩阵规模，观察不同矩阵规模下串行程序执行时间的变化。

**1.1（3分）不同规模矩阵的串行计算时间**

表2-1 不同规模矩阵的串行计算时间——（3分）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *N* | 512 | 1024 | 2048 | 4096 | 8192 |
| Trace |  |  |  |  |  |
| 计算时间(s) |  |  |  |  |  |

**1.2（4分）对串行计算时间与矩阵规模关系的分析**

根据表2-1绘制图2-1，其中X轴为N，Y轴为矩阵规模为*N*时的计算时间。

图2-1 不同矩阵规模的计算时间

从图2-1中，可以得到的结论是 。

**2. 【多线程并行化】（12分）**

使用多线程并行方法计算矩阵乘法（*N*=8192）。

**2.1（5分）请简述你使用的多线程并行化方法**

**2.2（5分）多线程并行化的计算时间和加速比**

测试不同线程数下，矩阵计算的执行时间，得到表2-2。

表2-2 不同线程数的计算时间和加速比（*N*=8192）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 线程数(T) | 1 | 2 | 4 | 8 | 16 |
| 计算时间(s) |  |  |  |  |  |
| 加速比 | 1.00 |  |  |  |  |

根据表2-2绘制图2-2，其中X轴为T，Y轴为计算时间。

图2-2 不同线程数的计算时间（*N*=8192）

**2.2 （2分）多线程并行化的实验结论**

处理器核数是 。当线程数(T)为 时，性能最好，加速比达到 。

**3.【矩阵分块计算】（10分）**

可以将矩阵分解为*m*×*m*的矩阵小块，每次完成一对小块的计算，以提高Cache的命中率。如下式所示。

，其中*n=N/m*

**3.1（3分）矩阵分块大小的计算**

L1和L2 Cache容量分别是 KB和 KB。你认为*m*设置为 比较合适（记为*m*0），原因是 。

**3.2（5分）最优分块大小实验**

在最优的线程数下，调整*m*的大小，得到表2-3。

表2-3 不同分块大小的计算时间和加速比（*N*=8192）——（2分）

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 分块大小(*m*) | 不分块 | *m*0/4 | *m*0/2 | *m*0 | 2*m*0 | 4*m*0 |
| 计算时间(s) | (第2节中的最优时间) |  |  |  |  |  |
| 加速比 | 1.00 |  |  |  |  |  |

**3.3（2分）对最优分块大小的分析**

实验表明，分块大小为 时性能最好。这个结果和你的预期一致吗？ 。如果不一致，其原因在于 。（2分）

**4.【使用SIMD指令加速】（10分）**

使用SSE指令可以一次完成4个浮点乘法操作和加法操作。下图给出了4×4的矩阵乘法操作示意图。可以使用<https://github.com/tommychen74/SIMDLib>提供的内嵌原语实现SSE或者AVX指令。



根据同样的方法，使用AVX指令可以完成8路矩阵的乘法。

**4.1（6分）SIMD指令的优化**

结合上述矩阵分块策略，在最优的线程数下，最优的分块数下，使用SSE/AVX指令的性能如表2-4所示。

表2-4 使用SIMD指令的计算时间和加速比

|  |  |  |  |
| --- | --- | --- | --- |
| SIMD指令 | 不使用 | 使用SSE | 使用AVX |
| 计算时间(s) | (第3节中的最优时间) |  |  |
| 加速比 | 1.00 |  |  |

**4.2（4分）SIMD指令优化的分析**

当前最优的执行时间是 s，与第1节中的串行计算相比，加速比达到 。

**5.【性能对比】（6分）**

**5.1（3分）总体性能对比**

画出以上串行、并行、分块+并行、分块+并行+SSE、分块+并行+AVX等5种实现策略在N=8192的情况下的程序执行时间对比图。

**5.2（3分）GFlops的计算**

从理论上，矩阵乘法中总共需要 次浮点乘法和加法操作。比较上述6种算法的计算性能（单位GFlops）。

表2-5 6种矩阵乘法的性能比较

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 执行方法 | 串行 | 多线程并行 | 并行+分块 | 并行+分块+SSE | 并行+分块+AVX | GPU |
| 执行时间(s) |  |  |  |  |  |  |
| 加速比 | 1.00 |  |  |  |  |  |
| 性能(GFlops) |  |  |  |  |  |  |

从表2-4中，你认为哪种并行化策略最为有效？ 。原因是 。

**6. 【Strassen方法】（5分）**

Strassen给出了一个减少矩阵乘法计算量的方法。



按照以下方式计算：

*P1=*(*A11+A22*)(*B11+B22*)

*P2=*(*A21+A22*)*B11*

*P3=A11* (*B12-B22*)

*P4=A22*(*B21-B11*)

*P5=*(*A11+A12*) *B22*

*P6=*(*A21-A11*)(*B11+B12*)

*P7=*(*A12-A22*)(*B21+B22*)

*C11=P1+P4-P5+P7*

*C12=P3+P5*

*C21=P2+P4*

*C22=P1+P3-P2+P6*

可否使用这个方法进一步提高矩阵乘法的性能？如果可以，请说明你的方案以及实现效果。

## 3. 整数排序（50分）

仔细阅读论文：

[1] Jatin Chhugani, William Macy, Efficient Implementation of Sorting on MultiCore SIMD CPU ArchitecturePVLDB '08, August 23-28, 2008, Auckland, New Zealand

[2] Nadathur Satish,etc., Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort, SIGMOD’10, 2010

对N个整数排序，并且输出排序后中位数（结果数组中第N/2个元素的值）。

**程序输入输出要求**

sort *N seed*

参数：

*N:* 待排序整数，本实验固定为1,000,000,000

*seed*：输入的种子浮点数

输出格式：

XXXX

XXXX: N个整数排序后第N/2个元素的值，用于验证排序的正确性；

整数数组初始化程序

/\*

Initialize integer array d[N] with seed

\*/

void sort\_gen(int \*d,int N,int seed){

srand(seed);

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

d[i]=rand();

}

}

**1. 【quicksort】（5分）**

使用C语言中quicksort()函数进行排序，排序时间为 ms。

**2. 【基数排序】（17分）**

**2.1（5分）请简述你的并行基数排序方法，主要包括多线程并行化方法和每个线程内的程序优化方法**

**2.2（7分）针对Cache的Hash桶插入操作优化**

针对所用处理器的参数，你认为缓冲大小*K*0= 和基数位数*D*0= 时，性能最优。原因是 。调整不同的参数组合，得到表3-1。

表3-1 不同参数组合下的排序时间（单位：ms）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 线程数(*T*) | 基数位数*D* | 缓冲大小*K* | | |
| *K*0/2 | *K*0 | 2 *K*0 |
| 1 | *D*0-1 |  |  |  |
| *D*0 |  |  |  |
| *D*0+1 |  |  |  |
| 2 | *D*0-1 |  |  |  |
| *D*0 |  |  |  |
| *D*0+1 |  |  |  |
| 4 | *D*0-1 |  |  |  |
| *D*0 |  |  |  |
| *D*0+1 |  |  |  |
| 8 | *D*0-1 |  |  |  |
| *D*0 |  |  |  |
| *D*0+1 |  |  |  |
| 16 | *D*0-1 |  |  |  |
| *D*0 |  |  |  |
| *D*0+1 |  |  |  |

**2.3（5分）多线程优化实验结果**

在*T*= , *K*= , *D*= 时，性能最佳，排序时间为 ms。相对于quicksort而言，加速比达到 。

**3. 【并行归并排序】（19分）**

**3.1（5分）请简述你的并行归并排序方法，包括多线程优化方法和基于SIMD的双调排序方法。**

**3.2（7分）多线程的归并排序**

请测试线程数发生变化时并行归并排序的时间变化情况。

表3-2 多线程的归并排序

|  |  |
| --- | --- |
| 线程数 | 执行时间（ms） |
| 1 |  |
| 4 |  |
| 8 |  |
| 12 |  |
| 16 |  |
| 24 |  |

多线程归并过程中，线程数为 时，性能最佳，相对于Quicksort的排序方法加速比为 。

**3.3（7分）基于SIMD指令的归并**

选择最优的多线程数，使用SSE或者AVX指令实现双调排序，并应用于归并过程中。此时排序的时间为 ms，相对于未使用SIMD方法加速比为 ，相对于Quicksort方法加速比为 。

**5. 【比较】（3分）**

用表3-2和柱状图3-1比较上述四种排序方法的执行时间，

表3-1 排序性能比较（*N*=1G）

|  |  |  |  |
| --- | --- | --- | --- |
| 方法 | quicksort | 优化的基数排序 | 多路归并排序 |
| 执行时间(s) |  |  |  |
| 加速比 | 1.00 |  |  |

图3-1 排序性能比较

**6. 【分析】（6分）**

请根据上述实验结果，分析哪种排序方法性能最好？是否与论文提出的观点一致？如果不一致，请说明原因。