南開大學

并行程序设计实验一

学院　计算机学院
专业　计算机科学与技术
南开大学　程伟卿
学号　2311865

目录

1 概述

1.1 源码链接

实验一的 github 链接（点击跳转）

1.2 配置信息

本地 arm 平台。

2 Part1: 矩阵乘法

2.1 问题描述

计算给定 n*n 矩阵的每一列与给定向量的内积。

2.2 算法设计

初始化如下：

```
#define SIZE 4000
int matrix[SIZE][SIZE];
int b[SIZE];
int sum[SIZE];
void init() {
    for (int i = 0; i < SIZE; i++) {
        b[i] = i;
        for (int j = 0; j < SIZE; j++) {
            matrix[i][j] = i + j;
        }
    }
}
```

2.2.1 平凡算法

思想即逐列计算。代码如下：

```
void p1() {
```

```
    for (int i = 0; i < SIZE; i++) {
        sum[i] = 0;
        for (int j = 0; j < SIZE; j++) {
            sum[i] += matrix[j][i] * b[j];
        }
    }
}
```

2.2.2 Cache 优化算法

改为行主遍历矩阵，代码如下：

```
void p2() {
    for (int i = 0; i < SIZE; i++) sum[i] = 0;
    for (int j = 0; j < SIZE; j++) {
        for (int i = 0; i < SIZE; i++) {
            sum[i] += matrix[j][i] * b[j];
        }
    }
}
```

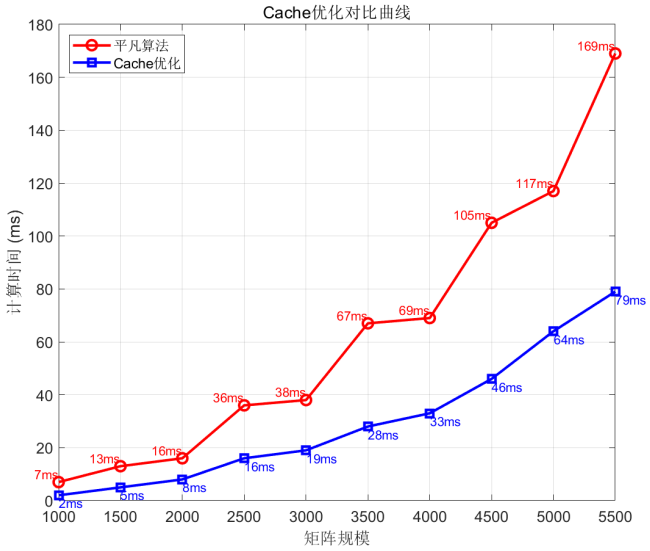
缓存优化（Cache Optimization）主要通过提高缓存命中率来减少内存访问延迟，提升程序性能。对于大规模矩阵运算，采用按行访问而非按列访问，可以利用 CPU 的缓存局部性（特别是空间局部性），使数据加载后尽可能长时间保留在缓存中，从而减少缓存未命中和降低数据访问开销。这种优化方法特别适用于矩阵运算、图像处理、数值计算等高性能计算场景，能够显著提高计算效率。

2.3 实验结果与分析

2.3.1 结果统计

统计如下：

矩阵规模	平凡算法	cache 优化
1000×1000	7ms	2ms
1500×1500	13ms	5ms
2000×2000	16ms	8ms
2500×2500	36ms	16ms
3000×3000	38ms	19ms
3500×3500	67ms	28ms
4000×4000	69ms	33ms
4500×4500	105ms	46ms
5000×5000	117ms	64ms
5500×5500	169ms	79ms



2.3.2 数据分析

这组数据表明，计算时间随矩阵规模增长呈非线性上升，但缓存优化算法始终优于普通算法，尤其在小规模矩阵（1000×1000）时加速比最高，达到 3.5 倍。随着矩阵增大，加速比逐渐趋于 2.0 左右，但仍显著降低计算时间。这说明缓存优化能有效减少缓存 miss，提高数据局部性，但在大规模

矩阵下，缓存替换增多会削弱优化效果。整体来看，缓存优化提升了计算效率，在矩阵计算中具有显著优势。

缓存优化的原理是通过提高数据访问的局部性来减少缓存未命中的次数。在矩阵计算中，通过改变数据访问顺序，使得内存中的数据按顺序加载到缓存中，从而最大化利用缓存的空间，减少了频繁的内存访问。例如，在行优先存取的算法中，内层循环遍历矩阵的列，使得每行数据连续地存入缓存，提高了缓存命中率，相较于列优先存取方法，减少了缓存 miss，从而提高了计算效率。

2.4 心得体会

通过使用缓存优化方法，我明白计算机性能的提升不仅依赖于硬件的提升，更多的是通过合理的软件优化来最大化利用现有硬件资源。尤其在大规模计算中，缓存的作用至关重要，优化数据访问顺序可以显著减少内存访问延迟，从而提升程序执行效率。

3 Part2:n 数之和

3.1 问题描述

用平凡算法和优化方法实现 n 数累加。

3.2 算法设计

3.2.1 平凡算法

```
1 void p1() {
2     for (int i = 0; i < SIZE; i++) sum += a[i];
3 }
```

直接累加。

3.2.2 超标量优化

```
1 void p2() {
2     double sum1 = 0, sum2 = 0;
```

```
    for (int i = 0; i < SIZE - 1; i += 2) {
        sum1 += a[i];
        sum2 += a[i + 1];
    }
    if (SIZE % 2 != 0) sum1 += a[SIZE - 1];
    sum = sum1 + sum2;
}
```

按奇偶分组求和，利用 cpu 的并行性提高算法效率。

3.2.3 递归优化

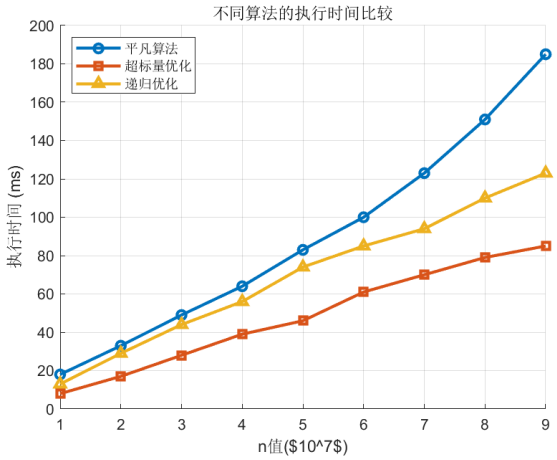
```
void p3(int n) {
    if (n <= 1) return;

    int m = n / 2;
    for (int i = 0; i < m; i++)
        a[i] += a[n - i - 1];

    p3(m); // 递归调用
}
```

将数组均分，将后半部分的值依次对应加到前半部分对应项，然后递归处理直至数组规模小于等于 1。

3.3 实验结果与分析

3.3.1 结果统计

n 值 (10^7)	平凡算法	超标量优化	递归优化
1	18ms	8ms	13ms
2	33ms	17ms	29ms
3	49ms	28ms	44ms
4	64ms	39ms	56ms
5	83ms	46ms	74ms
6	100ms	61ms	85ms
7	123ms	70ms	94ms
8	151ms	79ms	110ms
9	185ms	85ms	123ms



3.3.2 数据分析

可见超标量优化是效果最好的，其次是递归方法，在数据量较大时超标量优化的优势更明显。

3.4 心得体会

并行的优化利用了计算机硬件设计的原理，而递归调用则只是在算法层面。算法的设计应当兼顾软硬件。