
南開大學
Nankai University

计算机学院

# 并行程序设计实验报告

# 体系结构编程实验

姓名：唐明昊

学号：2113927

专业：计算机科学与技术

2023 年 3 月 12 日

# 目录

# 1 实验环境

操作系统版本：Windows 11 家庭中文版 64-bit (10.0, Build 22621)
系统类型：64 位操作系统, 基于 x64 的处理器
处理器：11th Gen Intel(R) Core(TM) i5-11300H @ 3.10GHz (8 CPUs)，3.1GHz
各级 cache 值：L1cache 320KB，L2cache 5MB，L3cache 8MB
内存容量：16384MB RAM
编译器版本：gcc (x86/64-posix-sjlj-rev0, Built by MinGW-W64 project) 8.1.0
IDE：Code::Blocks 20.03

# 2 基础要求

## 2.1 算法设计

### 2.1.1 n*n 矩阵与向量内积

- 平凡算法：逐列访问矩阵元素，外层循环执行一次计算出一个内积的结果，存入 sum 数组。
- 优化算法：逐行访问矩阵元素，外层循环执行一次计算不出任何一个内积，只向每个内积添加一个乘法结果。

### 2.1.2 n 个数求和

- 平凡算法：将给定元素依次累加到结果变量中。
- 多路链式：循环展开，调整循环步长，减少循环次数。
- 二重循环：相邻元素两两相加，连续存储到数组最前面。
- 递归函数：每次将后一半的元素加到前一半元素上，得到 n/2 个中间结果，递归调用直到最终结果。

## 2.2 编程实现

完整代码见github仓库

## 2.3 性能测试

使用 Windows.h 头文件下的 QueryPerformance 系列函数测量程序执行时间。

### 2.3.1 n*n 矩阵与向量内积

渐进地增大矩阵规模 n。研究不同规模下的算法性能表现，如图2.1所示。

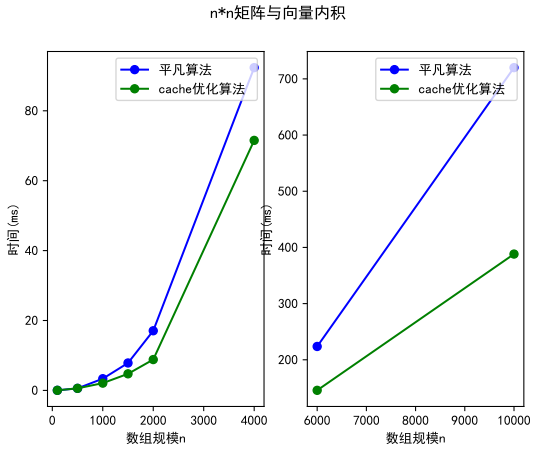


图 2.1: 两算法性能比较

由图中数据可以看到，同等的矩阵规模下，cache 优化算法明显快于平凡算法。随着矩阵规模不断增大，由于 cache 优化算法有很好的 cache 命中，时间曲线增长更为缓慢，而平凡算法由于 cache 缺失，需要频繁地访问主存，特别是随着矩阵规模增大至 4000 以上，平凡算法访问主存次数激增，cache 优化算法的优势更为突出。

### 2.3.2 n 个数求和

为方便递归算法设计，数据规模设置为 2 的幂。数据规模 n 每次增大 2 倍，对比在不同规模下几个算法的性能表现。

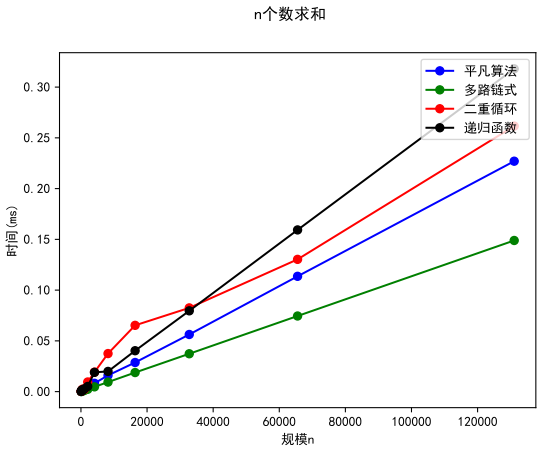


图 2.2: 四算法性能比较

由图中数据可以看到，多路链式算法用到循环展开技术后，提高步长，降低了循环次数；并且在减少指令依赖，运用到指令级并行后，时间开销有明显下降。

而递归算法和多重循环算法用到了指令级并行，理应更快，实验结果却是较平凡算法时间反而增加，原因可能是：

- 算法多层嵌套，指令执行条数更多。两个算法增加了更多的操作，如额外的循环判断条件，递归函数返回和调用等。

- 程序没有利用空间局部性，cache 未命中次数增多。二重循环在访问了 a[i] 后立马访问 a[2*i]，递归访问 a[i] 后访问 a[n-i-1]，两个算法访问数组元素时没有连续访问，而平凡算法则是链式的顺序访问，更多的 cache 未命中导致算法变慢。
- 递归需要使用额外的栈空间去完成函数调用，导致程序运行速度变慢。这也解释了二重循环和递归算法在算法上属于同一种思想，结果上递归慢于二重循环。

## 2.4 profiling

### 2.4.1 n*n 矩阵与向量内积

使用 Vtune 分别对矩阵规模 1000 和 10000 下两个算法进行比较，结果如表1所示。

| Events | 平凡算法规模 1000 | cache 优化规模 1000 | 平凡算法规模 10000 | cache 优化规模 10000 |
|---|---|---|---|---|
| L1 cache 命中 | 225,228,195 | 9,008,382 | 225,224,586 | 648,579,849 |
| L1 cache 未命中 | 100,252,503 | 0 | 99,053,013 | 600,324 |
| L2 cache 命中 | 27,010,332 | 0 | 27,011,022 | 0 |
| L2 cache 未命中 | 71,738,469 | 0 | 72,037,062 | 0 |

表 1: n*n 矩阵与向量内积 Vtune 分析结果

由 Vtune 分析结果看到，优化算法在 cache miss 次数上远远低于平凡算法，频繁的 cache miss 导致平凡算法频繁访问主存，故运行时间落后。

在规模提升至 10000 后，矩阵规模过大，cache 中无法完整地存储整个矩阵，即使是 cache 优化算法也需要频繁去访问内存，故时间效率下降也较为明显。

### 2.4.2 n 个数求和

使用 Vtune 对四个算法进行分析比较，四个算法主函数运行结果如表2所示。

| Events | 平凡算法 | 二路链式 | 二重循环 | 递归 |
|---|---|---|---|---|
| Clockticks | 13,800,000 | 9,200,000 | 16,100,000 | 11,500,000 |
| Instuctions Retired | 34,500,000 | 27,600,000 | 43,700,000 | 29,900,000 |
| CPI Rate | 0.4 | 0.333 | 0.368 | 0.385 |

表 2: n 个数求和 Vtune 分析结果

由 Vtune 分析结果可得，四个算法的代码主要执行部分，优化算法的 CPI 明显低于平凡算法，即 **IPC** 明显优于平凡的链式算法。

平凡算法和二路链式两算法执行指令条数大致相等，对二者进行 1000 次重复测试，去除偶然误差。由结果来看二路链式执行条数甚至更低，且算法的 CPI 也明显更优，故其运行时间更短。

| Events | 平凡算法 | 二路链式 |
|---|---|---|
| Clockticks | 6,410,100,000 | 3,889,300,000 |
| Instuctions Retired | 16,842,900,000 | 10,529,400,000 |
| CPI Rate | 0.381 | 0.369 |

表 3: 1000 次重复测试

## 2.5 结果分析

### 2.5.1 n*n 矩阵与向量内积

从图2.1来看，同等规模下，cache 优化算法比平凡算法有着更好的性能表现，从程序执行时间中可以体现。由表1可知，底层原因是由于 cache 优化算法更好地利用了空间局部性，大幅降低了 cache miss 次数，减少了访问主存的需要，故运行时间更短。

对于 cache 优化算法，不断增大矩阵规模也会导致程序性能大幅降低。因为矩阵规模过大，cache 无法一下存储整个矩阵，也需要频繁地从内存中导入块，进而运行时间变长。

### 2.5.2 n 个数求和

从表2可以看出，二路链式算法，二重循环算法以及递归算法都比平凡算法有着更高的 IPC，即更好地利用了指令级并行的特点。

二路链式速度明显快于平凡算法，因为其不仅指令条数更少，CPI 也更低，运行时间一定更短。

而二重循环算法和递归算法虽然有着更高的 IPC，运行速度却慢于平凡算法。考虑到可能是由于指令条数增多，没有利用空间局部性等原因，利用 Vtune 和 Perf 对其进行了性能分析。

| Events | 平凡算法 | 二重循环 | 递归 |
|---|---|---|---|
| Instuctions Retired | 50,600,000 | 78,200,000 | 73,600,000 |

表 4: 三个算法执行指令总数

| Events | 平凡算法 | 二重循环 | 递归 |
|---|---|---|---|
| cache load | 14833118 | 34215996 | 60439090 |
| cache load miss | 489646 | 1574174 | 1249333 |

表 5: 三个算法 cache 命中比较

由表4和表5可知，二重循环算法和递归算法相较于平凡算法需要执行更多的指令；且由于每次访问的变量有较大的跨越，尽管 cache 命中率大体相当，但 cache miss 的总次数远远大于平凡算法，故两个算法性能表现弱于平凡算法。

另外，虽然二重循环和递归算法属于同一种思想，但由图2.2可以看出，递归算法慢于二重循环。分析可能是由于函数递归需要栈空间等额外的消耗，所以导致程序性能下降。

# 3 进阶要求

## 3.1 Windows 与 Linux 性能对比

在 Windows 系统上安装了 WSL，模拟 Linux。由图3.3中数据可知，Linux 仅轻微慢于 Windows，可认为二者性能几乎一致。

虽然 WSL 和 Windows 为不同的系统，但由于 WSL 会共享 Windows 主机的资源，并且在内核与通信上进行了专门优化，故二者最终性能表现一致。

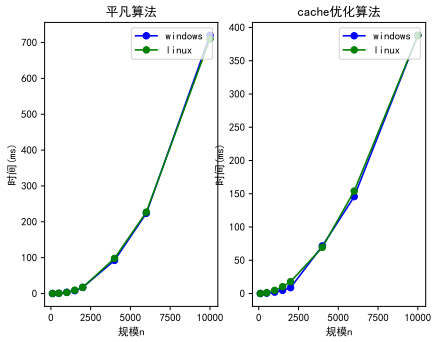


图 3.3: Windows 和 Linux 对比

## 3.2 cache 深度思考

### 3.2.1 问题描述

在进行 n*n 矩阵与向量内积实验时，为了渐进增长 n 的规模，每次预先设定好空间，做一次实验记录一次数据，而后再调整规模，即程序运行一次只做一种规模。

但实验过程中发现，如果预先将空间设定为一个最大值，而数组规模在一次程序执行过程中，用循环的方式逐渐增大，程序运行时间会大幅缩短。

### 3.2.2 实验方案设计

从直观上来看，造成问题的原因可能有预先开辟了较大的空间和循环增大 n 的规模两个。为了找出具体原因，做实验，对比性能具体差距。

为了考察是否是由于预先开辟空间的原因，控制变量，再增设一种情况。预先开辟大空间，程序一次仍然只做一种矩阵规模。

于是总共分成三种情况：

- 保持矩阵规模与开辟的空间一致，程序运行一次做一个规模下的实验，记录若干次运行结果。
- 预先开辟一个大的空间，数组规模从小到大逐渐增大，但程序执行一次只做一个规模，记录若干次运行结果。
- 预先开辟一个大的空间，用循环的方式让数组规模逐渐增大，程序执行一次输出若干结果。

### 3.2.3 实验结果

实验结果如图3.4所示。

由实验数据可知，第一种情况和第二种性能上没有明显的差异，即是否预先开辟一个大的空间或矩阵规模与空间是否保持一致，并不会影响最终程序运行结果。

而第一种（或第二种）情况和第三种情况数据对比可知，二者性能差距明显。具体表现为：平凡算法下，矩阵规模小时，第二种情况明显快于第一种情况，随着规模增大差距逐渐抹平；cache 优化算法下，第二种情况程序性能明显优于第一种，随着规模增大差距更为明显。

可以得出结论，在一次程序执行过程中，预先开辟空间后，渐进地增大矩阵规模而不断访问同一个数组，会提升程序运行效率。

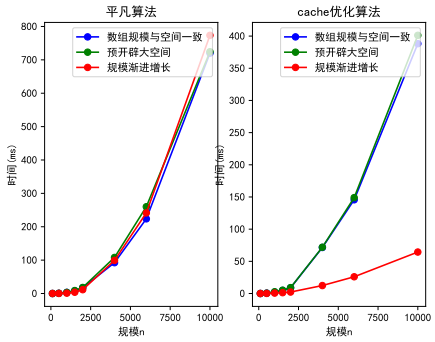


图 3.4: 三种情况性能对比

### 3.2.4 原因思考

经过调研 cache 运行机制 [1] 以及与同学讨论思考后，得出造成现象的原因可能是：

第三种情况用到了类似**分块**的处理手段，在不断增大矩阵规模时，利用到了**时间局部性**，同一块数据被频繁访问。根据 LRU 原理和重新组织储存器机制，小规模读取矩阵中数据时，会带出同一块的其他数据，没有用到的数据会被存放在 cache 中，而由于不断的频繁访问，他并不会被丢出 cache，如果 L1 装满，可能存放在 L2 中。这样，即使一次读取的行数可能变少了，但有可能矩阵某些数据从来没有被用到过而随着块已经被存放到了 cache 中，再随着规模增大时，块被频繁访问，一直保留在了 cache 中，进而在需要这些数据时就不用到内存中去访问。

| Events | 平凡算法 | cache 优化 |
|---|---|---|
| L1 cache 命中 | 563,756,016 | 1,128,995,634 |
| L1 cache 未命中 | 439,948,893 | 600,150 |
| L2 cache 命中 | 295,426,497 | 600,252 |
| L2 cache 未命中 | 0 | 0 |

表 6: 该情况下 Vtune 分析结果

由表6和表1中数据对比可以看到，第三种情况在规模总数更大的条件下，L2 的 cache 命中更多，未命中更少，这导致了该种情况运行时间明显更优。

特别的，由于该种情况下的 cache 算法既利用了**空间局部性**又利用了**时间局部性**，故其性能差距会表现得尤为突出。

而本身平凡算法就没有很好的**空间局部性**，在访问时常常就会导致 cache miss 的情况产生。而随着矩阵规模的不断增大，每次访问的数据相隔甚远，根据 LRU 原理，由于 cache 空间不足，以前存入的数据可能已被扔回内存，这就导致了平凡算法下，规模变大后差距逐渐抹平。

## 4 实验总结

本次实验针对 cache 优化和超标量两个问题，均证明了优化算法比平凡算法有着更好的性能表现。在实验过程中学会了新的工具使用，学会了自主思考问题并分析解决。也收获了一些教训：规划好再行动，在做实验前一定得有个清晰的思路，知道自己需要干什么，这样在查阅资料和使用工具时才不会迷茫；陌生知识并不可怕，在遇到陌生的问题时，很多完全可以通过查阅资料和请教他人解决，需要耐心钻研。

# 参考文献

[1] David A. Patterson and John L. Hennessy. 计算机组成与设计. 机械工业出版社, 2015.