南開大學
Nankai University

计算机学院

并行程序设计实验报告

体系结构相关编程

马浩祎

学号：2213559

专业：计算机科学与技术

2024 年 3 月 25 日

目录

1 摘要

Abstract

本次实验分为两个部分：n*n 矩阵向量内积和 n 个数求和，分别测试并行体系结构中的 cache 命中原理和超标量架构处理相邻无依赖指令的性能。经过实验和 Vtune 分析，由于 cache 存储的空间相邻的数据是以行存储为主，所以行存储矩阵的算法效率要优于列存储矩阵。而且，多路求和算法可以更好的利用超标量架构，相对于串行单路链式求和有着更好的性能。除此之外，我还对 $0 \to n-1$ 求和算法的数组记录抑或是循环变量处理进行了汇编代码级别的分析，说明循环变量处理有更少的指令；在此探索上我还在矩阵内积的部分进行冗余化处理，观察内存冗余对于两种算法的影响，证明了列存储需要更多的访存从而受到内存冗余数组的影响会更大。此外实验还包括了 Vtune，Godbolt 等工具的使用。

2 实验环境

本实验的实验环境为:
Windows：

- 硬件:
  - CPU：Intel(R) Core(TM)i9-14900HX,24 核，基准速度 2.2GHz
  - 内存容量 16GB
  - Cache L1:2.1MB;L2:32.0MB;L3:36.0MB
- 软件:
  - IDE:CodeBlock 20.03
  - 编译设置:GNU GCC Compiler
  - 分析工具：Vtune 2024.0.1，Godbolt 网站

3 实验一：$n*n$ 矩阵与向量内积

3.1 算法设计

本实验要计算一个 $n*n$ 矩阵和给定向量的内积，所用数据：

- 向量：$A[i]=i$
- 矩阵：$D[i][j]=i+j$

一般计算机都以行存储为主，由于伪代码已经可以较好说明程序，故省略具体编程部分。两种算法设计如下：

3.1.1 平凡算法

本方法按列访问矩阵元素。伪代码如下：

Algorithm 1 列访问算法

Input: 矩阵规模 n

Output: 矩阵和向量的内积结果

```
1: function ColMultiple(n, sum[])
2:     for i = 0 → n − 1 do
3:         sum[i] ← 0.0
4:         for j = 0 → n − 1 do
5:             sum[i] ← sum[i] + D[j][i] ∗ A[j]
6:         end for
7:     end for
8: end function
```

3.1.2 优化算法

由于上述方法与正常计算机行访问存储的 cache 有所不匹配，不妨用行访问方式优化。伪代码如下；

Algorithm 2 行访问优化算法

Input: 矩阵规模 n

Output: 矩阵和向量的内积结果

```
1: function ColMultiple(n, sum[])
2:     for i = 0 → n − 1 do
3:         sum[i] ← 0.0
4:     end for
5:     for j = 0 → n − 1 do
6:         for i = 0 → n − 1 do
7:             sum[i] ← sum[i] + D[j][i] ∗ A[j]
8:         end for
9:     end for
10: end function
```

3.2 性能测试

这里主要测试不同算法的时间性能。分别记录不同规模下程序所用时间，同时为了保证精确和稳定且高效，$n <= 1024$ 时每个规模下采用重复 1000 次取平均的方法, 而过大的 n 直接计算，不再重复。部分用时数据如表 1，同时作出对比折线图，如图 3.1。

规模 n	平凡算法	优化算法	规模 n	平凡算法	优化算法
8	0.0001012	8.79E-05	776	1.2141	0.811513
24	0.0009617	0.0007866	904	1.66567	1.08534
56	0.0048343	0.0042232	2048	41.2268	5.8568
168	0.0526828	0.038467	4096	187.146	22.0598
392	0.305365	0.216658	6144	339.115	52.1238
520	0.518863	0.369107	7168	449.226	82.0269
648	0.836162	0.557746	8192	564.871	109.955

表 1: 矩阵向量内积算法时间性能表

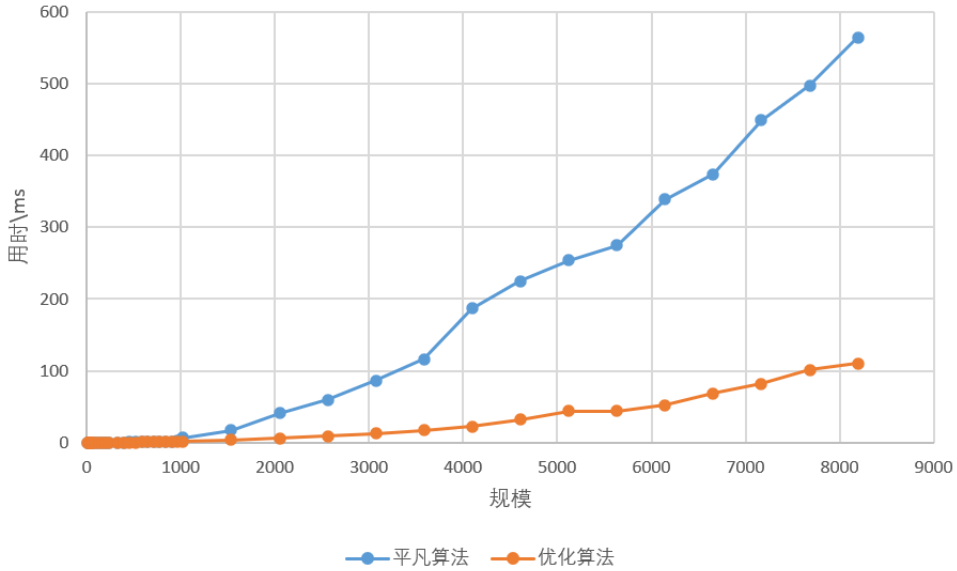

图 3.1: 矩阵向量内积算法时间对比图

3.3 profiling--进阶 Vtune

这里我采用 Vtune 进行了缓存命中的分析和指令条数的对比验证。注意，软件分析的是同一算法重复 10 次的程序。

项目	列访问算法	行访问算法
MEM_LOAD_RETIRED.L1_HIT	10,824,032,472	11,361,034,083
MEM_LOAD_RETIRED.L1_MISS	1,041,015,615	22,200,333
MEM_LOAD_RETIRED.L2_HIT	852,612,789	22,200,333
MEM_LOAD_RETIRED.L2_MISS	189,039,690	300,063
MEM_LOAD_RETIRED.L3_HIT	149,131,311	0
MEM_LOAD_RETIRED.L3_MISS	36,465,309	300,126
Instructions Retired	32,719,200,000	32,743,200,000

表 2: Vtune 分析 Cache 命中结果

如表 2，我们可以看到，两个算法的指令条数相差不大，这说明指令级上两个算法性能相近。但是行访问算法的 L1 缓存命中略微高一点，未命中数大大减小，L2 和 L3 的各项数据也比列访问大大减小。这应该是计算机在 L1 中 MISS 后才会访问 L2，然后是 L3（会是单纯的这样吗？后面还将分析），所以这证明了列访问算法中 Cache 利用效果不是很好，有很多数据未命中，在 L2，L3 中更是如此。行访问算法则有明显的利用效果，尤其是在 L2，L3 级缓存中的表现。

有意思的是，行访问算法的 L1.MISS 等于 L2.HIT，这意味着 L1 中没命中的在 L2 中全部命中，或许能说明 L1，L2 已经存了行访问过程中的所有相邻数据。但是 L3 中也有未命中数据，说明并不是 L1 未命中的都会去 L2 中访问，也会去访问 L3 的，这说明缓存并非简单的逐级命中，会有缓存替换策略等等因素的影响。但是 L3 中并没有命中，所以 L1,L2 缓存了大部分数据这个命题应该是正确的。

3.4 结果分析

本实验主要研究所取规模和 cache 大小与程序性能的关系，因为内存的频率比 CPU 要慢很多，所以一般的计算机中都会有高速缓存 cache 来把内存中常用的数据存起来，这样以后 CPU 再需要访问的时候会加速存取。通过图 3.1 的对比，不难看出随着规模增长，两个算法的效率差距越来越大，我认为是：

- 行访问算法利用了 cache 中存的是矩阵的行间相邻元素，这样可以有效命中并迅速取出，而规模的增大对时间的影响只在对于 cache 的存取，cache 是可以一次放下相邻内容的。
- 列访问算法则每一次几乎都用不上 cache 中存的相邻数据，都得再次从内存中取，效率必然很差。
- 上面提到本机 L1 大约 2MB，L2 和 L3 大约 32MB，L1 对应的矩阵规模大致为 750，L2 对应的规模大致为 3000，这在规模设计中有所考虑，而内存中其实也有数组 A 和数组 sum，故相对能利用的规模会更少，我们通过图 3.1 其实可以看出，上升趋势在 1000 左右开始变大，在 4000 左右更大一点。这确实可以印证缓存级数越大，规模越大但是存取效率越慢的理论认知。
- 经过 Vtune 量化结果，我们确实可以看到两种算法在缓存利用上的差距。而且还在一定程度上证明了缓存访问并非单纯的逐级访问，会有缓存替换等策略的存在。
- 总结分析一下，行访问算法有效利用了并行概念中很关键的空间局部性，列访问算法则没有，我们应该在算法优化的时候考虑到局部性原理。

4 实验二：计算 n 个数的和

本实验要求进行 n 个数的求和以分析性能影响, 所用数据是从 $0 \to n-1$ 的 n 个数。

4.1 算法设计

4.1.1 平凡累加

由于算法较简单，省略具体编程实现部分。

Algorithm 3 链式累加算法

Input: 数据个数 n

Output: $sum = n$ 个数据的和

1: **function** SUM(n, sum)
2: **for** $i = 0 \to n-1$ **do**
3: $sum \leftarrow sum + i$
4: $i \leftarrow i + 1$
5: **end for**
6: **end function**

4.1.2 优化算法

这里采用多路链式算法。由于伪代码能较好解释算法，省略具体编程实现部分。注意在设计四路的时候，规模的取值都是 100 的倍数，而 100 整除 4，故算法最后可以在 n 处停止，保证正确性。

Algorithm 4 多链算法

Input: 数据个数 n
Output: $sum = n$ 个数据的和

function MulSum($n, sum, sum1, sum2, sum3, sum4$)
  for $i = 0 \to n - 1$ **do**
    $sum1 \leftarrow sum1 + i$
    $sum2 \leftarrow sum2 + i + 1$
    $sum3 \leftarrow sum3 + i + 2$
    $sum4 \leftarrow sum4 + i + 3$
    $i \leftarrow i + 4$
  end for
  $sum \leftarrow sum1 + sum2 + sum3 + sum4$
end function

4.2 性能测试

这里主要测试不同算法的时间性能。分别记录不同规模下程序所用时间，同时为了保证精确和稳定，每个规模下采用重复 1000 次取平均的方法。部分数据如表 3，同时作出对比折线图，如图 4.2。

规模 n	平凡算法	多链算法	规模 n	平凡算法	多链算法
100	4.65E-05	4.65E-05	61000	0.021036	0.0178669
300	0.000124	0.0001451	81000	0.0291478	0.0205149
700	0.0002779	0.0003292	311000	0.0541878	0.0344088
2000	0.0013566	0.0007161	711000	0.213242	0.174238
6000	0.002328	0.0017375	2111000	0.361042	0.25554
10000	0.0039276	0.0030606	6111000	1.2446	0.86483
51000	0.0167091	0.0120012	10111000	2.81262	1.97484

表 3: n 个数求和算法时间性能表

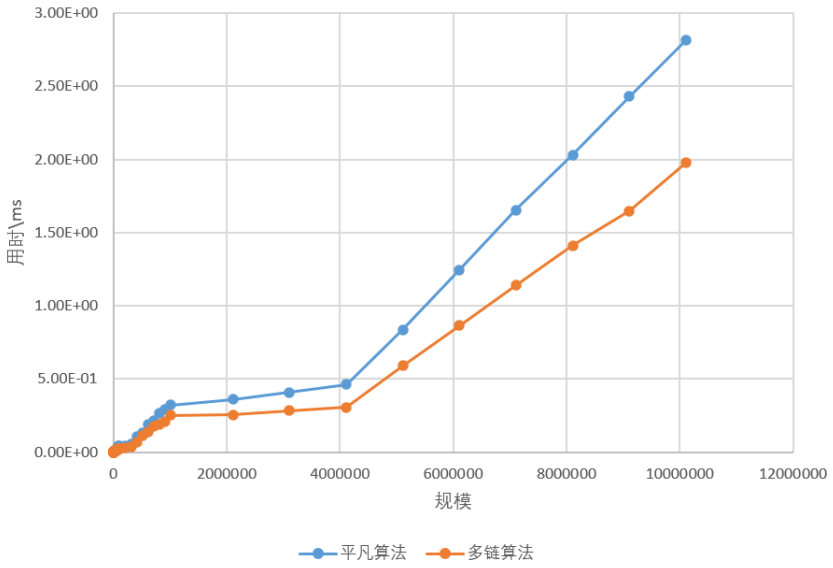

图 4.2: n 个数求和算法时间对比图

4.3 profiling--进阶 Vtune

这里我采用 Vtune 进行了平凡/优化算法指令条数的对比量化。注意，软件分析的是同一算法重复 100000 次的程序。

项目	单链算法	多路算法
main	80,032,800,000	50,018,400,000
func1	31,200,000	12,000,000
func2	7,200,000	2,400,000
func3	2,400,000	2,400,000

表 4: Vtune 分析指令数结果

如表 4，这里展示了指令条数最多的几个部分，其中我们可以看到多路算法在各个层级上都优于单链算法。尤其是 main 函数的指令数，可以看到有很大的优化力度。我目前不太清楚 Vtune 上展示的各个小 func 是什么，编写程序中只有 main 函数。但经过询问了解，这可能是编译器在优化的时候将一些语句变为部分函数的内联语句，将直接调用那些小函数，因此会有这些小函数的存在。而且我们也看到 func3 的指令数是固定的，结合重复次数，这个小函数只有 24 个指令，这应该可以验证上述结论。

4.4 结果分析

本实验旨在理解超标量架构计算机对于相邻无依赖指令的高效率原理。这种计算机会在处理程序的时候识别无依赖的指令并分发给多路指令流水线去执行，指令数会有所减少，用时也会有所缩减，据表 4 前者在比值上近乎 1.6 1，据表 3 和图 4.2，后者略微减小到 $1.42:1$，都并未达到预期的加速比。据刚才计算，实际用时比低于指令数比，这应该是因为分发过程的耗时和底层硬件有延迟所致。

还有值得思考的是，这里的数据采用循环变量记录而并未采用数组预先记录的方式，是否会导致空间局部性的未充分利用？这在后面会详细探索。

5 进阶要求

5.1 数组记录法

这里我对数组记录和循环变量求和的差别进行了探索和思考。

5.1.1 实验对比结果

如图 5.3，我进行了三组对照实验，分别是：

- 纯数字记录：$sum \leftarrow sum + i$
- 数组记录：$sum \leftarrow sum + A[i]$ 其中 $A[i] = i$
- 纯数字 + 数组内存：$sum \leftarrow sum + i$, 并保留 $A[i]$ 数组

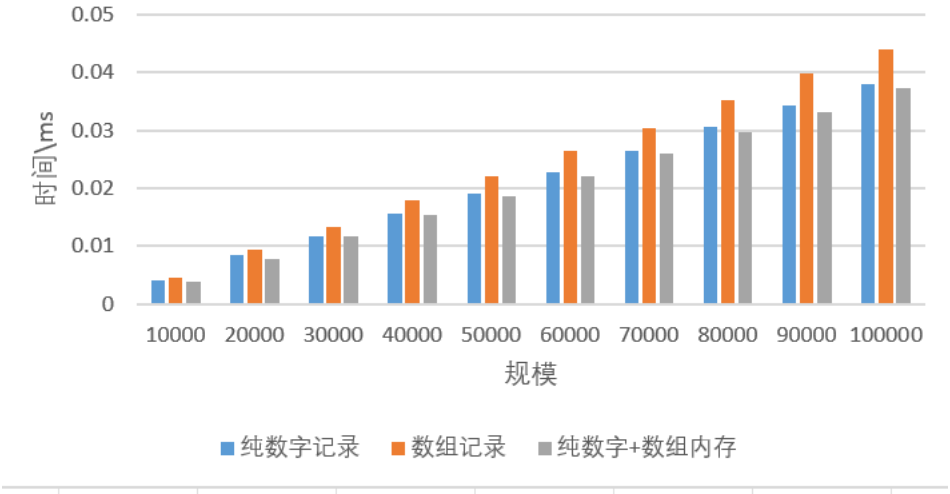

图 5.3: 数组与直接数据效率图

5.1.2 分析与汇编级解释

在设置了前两组并得到结果后，普遍地，并没有我想象的那样，数组记录因为具有空间局部性而更快，相反效率普遍慢了，多次不同规模实验验证了这不是偶然的，我怀疑是数组记录占用了额外的内存，故增设第三组，保留了同等大小的数组，但是结果也差不多。经过更细致的观察，我发现这个直接数据是 for 循环中的变量，汇编语言中循环计数变量是单独有一个寄存器 ecx 的，难道说这段代码编译后，是直接从寄存器取数，那比数组的空间局部性还快就可以解释了。我们看利用 Godbolt 工具生成的汇编代码。

```
        mov     DWORD PTR [rbp-4], 0
        mov     DWORD PTR [rbp-8], 0
        jmp     .L2
.L3:
        mov     eax, DWORD PTR [rbp-8]
        add     DWORD PTR [rbp-4], eax
        add     DWORD PTR [rbp-8], 1
.L2:
        mov     eax, DWORD PTR [rbp-8]
        cmp     eax, DWORD PTR [rbp-20]
        jl      .L3
        nop
        nop
        pop     rbp
        ret
```

(a) 循环变量求和汇编代码

```
.L5:
        cmp     DWORD PTR [rbp-4], 9999
        jle     .L6
        mov     DWORD PTR [rbp-8], 0
        mov     DWORD PTR [rbp-12], 0
        jmp     .L7
.L8:
        mov     eax, DWORD PTR [rbp-12]
        cdqe
        mov     eax, DWORD PTR [rbp-40016+rax*4]
        add     DWORD PTR [rbp-8], eax
        add     DWORD PTR [rbp-12], 1
.L7:
        mov     eax, DWORD PTR [rbp-12]
        cmp     eax, DWORD PTR [rbp-40020]
        jl      .L8
        nop
        nop
        leave
        ret
```

(b) 数组求和汇编代码

图 5.4: 不同算法的汇编代码

如图 5.4(a)，L3 代码段，实际证明是没有用到 ecx 的，但是我们依旧可以对比汇编代码来理解指令级上的差异，图 (a) 仅仅用 2 条指令实现循环内部的加法，以 eax 作为中介；反观图 (b),L8 代码段即为循环内部的加法操作，其有 4 条指令，除了第一行和第四行刚好是图 (a) 的两条外，又多访问了一次局部变量数组和 cdqe 扩展指令。

这样看，由于访问内容增多，指令条数增加，确实可以验证之前的实验结果。

5.2 冗余的影响

上面那节我们看到了冗余一个数组似乎对实验 2 的程序没有影响，那却否如此呢，由于实验 1 对访存的次数更高，接下来我们看矩阵向量积实验下冗余内存会有什么影响。

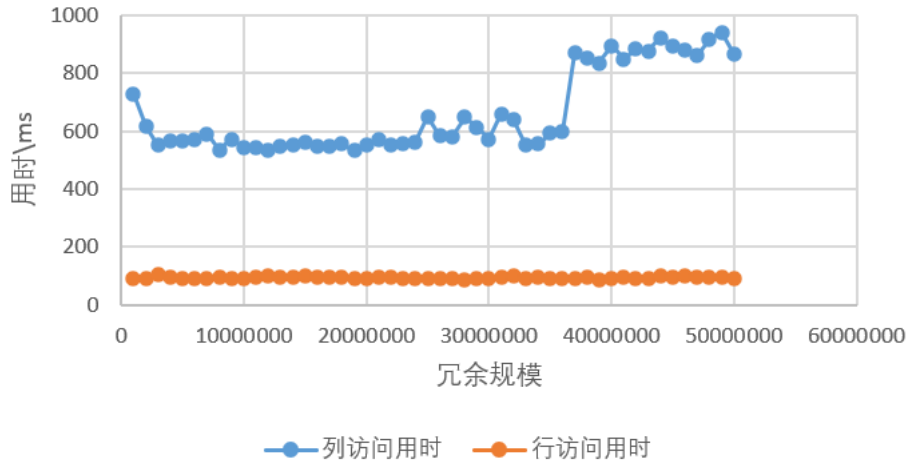

图 5.5: 冗余规模下性能对比图

如图 5.5, 这里为尽可能取大观察影响，规模 $n \leftarrow 8192$，然后根据图中趋势，行访问算法受冗余影响较少，甚至没有，结合之前的分析，这应该是因为其能较好利用 cache 的原因。而列访问算法受冗余影响较大，结合前面分析，这应该是因为其未能很好利用 cache，而需要更多次访问内存从而放大冗余的影响。

6 总结分析

通过本次实验，我掌握了：

- 对并行体系结构性能测试的基本方法（如测高精度时间）和进阶方法（如 Vtune 量化分析）。
- 缓存 Cache 的行存储工作机理，以及缓存替换策略等的存在。
- 超标量架构下相邻无依赖指令的高效率分发，以及编译器在优化时将代码内联函数化的操作。
- 尝试探索自己提出问题的解释，如汇编层面上理解数组记录和循环变量记录的指令级差别，和冗余内存对程序的影响。

总之，这次实验让我初步了解了并行体系结构中的问题细节，优化方法，增强了自己提出问题，分析问题并给出解释的能力，这对我未来的并行程序研究具有重要影响。