南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

并行程序设计实验报告

---

n*n 矩阵与向量内积以及 n 个数求和

蒋薇

年级：2021 级

专业：计算机科学与技术

指导教师：王刚

2023 年 5 月 12 日

摘要

本次实验分两大部分，n 阶方阵和向量内积，求和；各部分分别从算法实现、编程设计、性能测试、profiling 及分析展开，各部分具体从平凡算法逐渐优化，主要利用超标量、Cache 的性质进行优化，提高并行度，从而提高性能。理解掌握 Cache 作用，流水线、超标量等概念，学习提高性能的方法。

关键字：Parallel,Cache,Profiling

目录

一、 基础要求

(一) 算法设计

1. n*n 矩阵与向量内积

要求：给定一个 n × n 矩阵，计算每一列与给定向量的内积。

平凡算法 定义一个 n×n 的矩阵，使用二维数组来实现，double matrix[n][n]; 向量的维度可以由用户输入或者预先定义。这里我们假设向量维度为 n，double vector[n]; 然后，我们需要对矩阵每一列与向量进行内积计算。内积是两个向量对应项相乘后相加的结果，可以使用如下公式计算：

$$sum_{i=1}^{n} matrix_{i,j} \times vector_i$$

其中，i 表示向量的第 i 个元素，j 表示矩阵的第 j 列.

Cache 优化 1. 首先，从矩阵中取出每一列和给定向量进行内积计算。2. 将每一列与向量的内积结果存储在 cache 中。3. 下次计算时，直接从 cache 中调用结果。4. 重复以上步骤，直到计算完所有列与向量的内积。用 cache 优化能够加速矩阵内积的计算过程，提升算法的效率。

2. n 个数求和

平凡算法 先定义 int n, 定义输入数的个数, 输入一个新建的动态数组, 输入数字存入动态数组中, 函数转换并求和, 最后输出.

Cache 优化 多链路式、部长可设置为 2，执行 n/2 次循环；可设置递归、将给定元素两两相加，得到 n/2 个中间结果；将上一步得到的中间结果两两相加，得到 n/4 个中间结果；依次类推，log(n) 步骤得到一个值为最终结果。

二重循环，相邻元素相邻相加连续存储到数组最前面，a[0] 为最终结果。

(二) 编程实现

1. n*n 矩阵与向量内积

平凡算法

关键代码：逐列访问矩阵元素：一步外层循环（内存循环一次完整执行）计算出一个内积结果

逐列访问矩阵元素

```
for(i = 0; i < n; i++)
{
 sum[i] = 0.0;
 for(j = 0; j < n; j++)
 sum[i] += b[j][i] * a[j ];
}
```

Cache 优化

关键代码：改为逐行访问矩阵元素：一步外层循环计算不出任何一个内积，只是向每个内积累加一个乘法结果

逐行访问矩阵元素

```
for(i = 0; i < n; i++)
{
    sum[i] = 0.0;
    for(j = 0; j < n; j++)
    for(i = 0; i < n; i++)
    sum[i] += b[j][i] * a[j];
}
```

后者的访存模式与行主存储匹配，具有很好空间局部性，令 cache 作用得以发挥。

n*n 矩阵与向量内积 github 地址 请点击这里

2. n 个数求和

平凡算法 关键代码：将给定元素依次累加到结果变量即可

链式求和

```
for(i = 0; i < n; i++)
{
    sum[i] = 0.0;
    for(j = 0; j < n; j++)
    for(i = 0; i < n; i++)
    sum[i] += b[j][i] * a[j];
}
```

Cache 优化 关键代码：

优化加速

```
sum1 = 0; sum2 = 0
for (i = 0; i < n; i += 2) {
    sum1 += a[i];
    sum2 += a[i + 1];
}
sum = sum1 + sum2;
```

实现

递归

```
void recursion(int n,int*a)
{
    if (n == 1)
        return;
    else {
        for (int i = 0; i < n / 2; i++)
        a[i] += a[n - i - 1];
        n /= 2;
        recursion(n,a);
}
```

```
}
```

优点是简单，缺点是递归函数调用开销较大。

二重循环

```
for (int m = n; m > 1; m /= 2) {//log n
    for (int i = 0; i < m / 2; i++)
    {
        array[i] = array[i * 2] + array[i * 2 + 1];//相邻元素相加存储到前面
}
```

n 个数求和 github 地址 请点击这里

(三) 设计测试

方便程序正确性检查，测试数据人为设定固定值给定问题规模 n; 从最坏的情况为在测试时 n 需要渐进的取到，同时当 n 取值较小时，同一个 n 需要测试多次来取到精确的结果。

一种可能的规模设计为：对于 n = 0 − 100，精细测试 0, 10, 50；对 n>1000，稀疏测试 5000,10000。

测试数据的设计需要仔细思考, 如果测试数据过于特别, 将无法得到程序真正的性能。以输入排序为例，最坏以及最好的情况下测试数据分别为递减序列和递增序列。但是如果我们需要测试程序的平均情况，这时候需要随机产生大量的测试数据。windows 下精确计时可使用 QueryPerformance 系列函数进行计时, 秒为单位。

测量运行时间

```
LARGE_INTEGER t1, t2, tc;
QueryPerformanceFrequency(&tc);
QueryPerformanceCounter(&t1);
f(n, sum, matrix, vector);
QueryPerformanceCounter(&t2);
cout << "time : " << (double)(t2.QuadPart - t1.QuadPart) /(double)tc.QuadPart
    << endl;
```

1. n*n 矩阵与向量内积

平凡算法 测试规模我们取 0，10，20，50，100，500，1000，8000，当 n 取值较小时，重复多次来获得更加精确的结果; 重复次数采用随机数种子生成；当矩阵规模较小时，程序运行时间很短，采用重复运行待测函数、延长计时间隔的方法，来解决计时函数精度不够、影响测量精度的问题。测试结果如下:

表

规模 N	重复次数	总时间/s	平均时间/s
0	30	5e-07	1.66667e-08
10	49	1.91e-05	3.89796e-07
20	76	9e-05	1.18421e-6
50	29	0.0002872	9.90345e-06
100	38	0.0014692	3.86632e-05
500	51	0.0636297	0.00124764
1000	22	0.119064	0.00541199
8000	11	8.27971	0.752701

表 1: n*n 矩阵与向量内积平凡算法

Cache 优化 设置对应各级 cache 大小的问题规模来研究 cache 对性能的影响观察问题规模与 cache 大小不同关系时的程序性能变化, 且设置一系列问题规模，研究程序性能随问题规模变化的趋势。

如果矩阵和向量的规模较小，且能够完全存放在缓存中，则计算速度会非常快。但是，如果矩阵和向量的规模很大，超出了缓存的容量，则会频繁地进行主存读写操作，导致计算时间增加。

举例：假设有一个 1000*1000 的矩阵和一个 1000 维的向量，如果缓存大小为 4MB，每个元素占用 4 字节，则矩阵需要占用 4MB * 1000 = 4GB 的内存，超出了缓存大小，需要频繁进行主存读写操作，导致计算时间增加。而如果缓存大小为 16MB，则可以完全存放矩阵和向量，计算速度会非常快。

一些现代的处理器采用了多级 Cache 结构，可以提高 Cache 的命中率和数据访问速度，从而减少从主存中读取数据的次数

2. n 个数求和

同理，多次实验拉长时间缓解实验误差, 测试不同问题规模，cache 大小相对关系对性能的影响, 多次测量取平均值以得到更好的测试数据结果。

平凡算法

表

规模 N	重复次数	总时间/s	平均时间/s			
0	59	0.000268	4.54237e-06			
10	34	0.000146	4.29412e-06			
20	23	0.0001898	8.25217e-06			
50	48	0.0003428	9.06875e-06			
80	51	0.0005159	1.01157e-05 100	62	0.0005443	8.77903e-06
500	35	0.0058661	0.00016703			
1000	27	0.0225454	0.000835015			

表 2: n 个数求和平凡算法

Cache 优化

n 个数求和 Cache 优化

```
start_time = get_current_time()  // 记录开始时间
sum = 0
for i = 0 to n-1
    sum = sum + data[i]  // 缓存行优化和数据对齐优化
    if (i+1) % 8 == 0  // 缓存预取优化
        prefetch(data[i+8])  // 预取下一个缓存行的数据
end
end_time = get_current_time()  // 记录结束时间
elapsed_time = end_time - start_time  // 计算运行时间
print("Sum: ", sum)
print("Elapsed time: ", elapsed_time)
```

缓存预取优化：在循环中，可以预取下一个缓存行的数据到 Cache 中，减少 Cache 的失效次数，提高数据访问速度。

(四) Profiling

用 CPU 的各级 cache 大小对应的规模，多次实验拉长时间缓解实验误差，测试不同问题规模，分析 cache 大小相对关系对性能的影响，设置的参数 Cache 的容量、块大小;

模拟结果 a) 访问总次数，总的不命中次数，总的不命中率 b) 读指令操作次数，不命中次数及其不命中率 c) 读数据操作次数，不命中次数及其不命中率 d) 写数据操作次数，不命中次数及不命中率

Cache 容量对不命中率的影响，选择文件，选择不同容量 Cache。

表

Cache(KB)	2	4	6	8	16	32	64	128
不命中率 (%)	9.87	7.19	4.48	2.65	1.42	0.89	0.42	0.40

表 3: Cache 容量对不命中率影响

根据实验结果，Cache 不命中率随着 Cache 容量的增大而降低，容量增大到一定程度后，再增大 Cache 容量变化不明显。

Cache 块大小对不命中率的影响，选择文件，选择不同 Cache 块大小。表

Cache 容量 (KB)	2	8	32	128	512
Cache 块大小、不命中率 (	%	)			
16	12.02	5.79	1.86	0.95	0.71
32	9.38	4.48	1.42	0.60	0.42
64	9.36	4.03	1.20	0.43	0.27
128	10.49	4.60	1.08	0.35	0.20
256	13.45	5.35	1.19	0.34	0.16

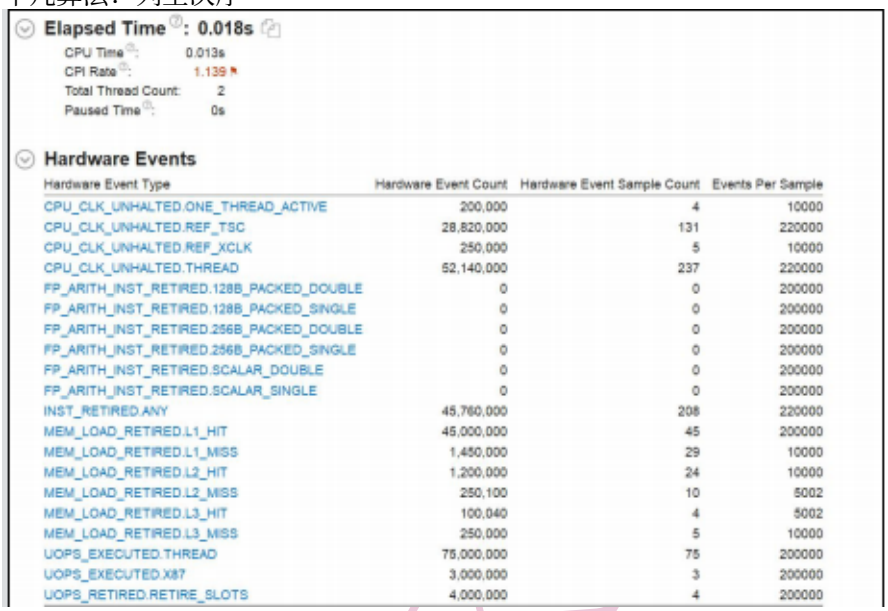
表 4: n 个数求和 Cache 优化

当 Cache 容量一定的时候，若增大 Cache 块大小，Cache 不命中率先降后升。

原因:1）增加空间局部性，减少强制性不命中；2）减少 Cache 中块的数目，增加冲突不命中增加冲突不命中块较小时，1）作用超过 2），命中率下降；块较大时，2）作用超过 1），不命中率下降。

1. n*n 矩阵与向量内积

平凡算法：列主次序



Elapsed Time: 0.018s

CPU Time: 0.013s
CPI Rate: 1.139
Total Thread Count: 2
Paused Time: 0s

Hardware Events

Hardware Event Type	Hardware Event Count	Hardware Event Sample Count	Events Per Sample
CPU_CLK_UNHALTED.ONE_THREAD_ACTIVE	200,000	4	10000
CPU_CLK_UNHALTED.REF_TSC	28,820,000	131	220000
CPU_CLK_UNHALTED.REF_XCLK	250,000	5	10000
CPU_CLK_UNHALTED.THREAD	52,140,000	237	220000
FP_ARITH_INST_RETIRED.128B_PACKED_DOUBLE	0	0	200000
FP_ARITH_INST_RETIRED.128B_PACKED_SINGLE	0	0	200000
FP_ARITH_INST_RETIRED.256B_PACKED_DOUBLE	0	0	200000
FP_ARITH_INST_RETIRED.256B_PACKED_SINGLE	0	0	200000
FP_ARITH_INST_RETIRED.SCALAR_DOUBLE	0	0	200000
FP_ARITH_INST_RETIRED.SCALAR_SINGLE	0	0	200000
INST_RETIRED.ANY	45,760,000	208	220000
MEM_LOAD_RETIRED.L1_HIT	45,000,000	45	200000
MEM_LOAD_RETIRED.L1_MISS	1,450,000	29	10000
MEM_LOAD_RETIRED.L2_HIT	1,200,000	24	10000
MEM_LOAD_RETIRED.L2_MISS	250,100	10	5002
MEM_LOAD_RETIRED.L3_HIT	100,040	4	5002
MEM_LOAD_RETIRED.L3_MISS	250,000	5	10000
UOPS_EXECUTED.THREAD	75,000,000	75	200000
UOPS_EXECUTED.X87	3,000,000	3	200000
UOPS_RETIRED.RETIRE_SLOTS	4,000,000	4	200000

平凡算法：行主次序

Elapsed Time: 0.018s

CPU Time: 0.015s
CPI Rate: 1.164
Total Thread Count: 2
Paused Time: 0s

Hardware Events

Hardware Event Type	Hardware Event Count	Hardware Event Sample Count	Events Per Sample
CPU_CLK_UNHALTED.ONE_THREAD_ACTIVE	100,000	2	10000
CPU_CLK_UNHALTED.REF_TSC	32,340,000	147	220000
CPU_CLK_UNHALTED.REF_XCLK	800,000	16	10000
CPU_CLK_UNHALTED.THREAD	54,780,000	249	220000
FP_ARITH_INST_RETIRED.128B_PACKED_DOUBLE	0	0	200000
FP_ARITH_INST_RETIRED.128B_PACKED_SINGLE	0	0	200000
FP_ARITH_INST_RETIRED.256B_PACKED_DOUBLE	0	0	200000
FP_ARITH_INST_RETIRED.256B_PACKED_SINGLE	0	0	200000
FP_ARITH_INST_RETIRED.SCALAR_DOUBLE	0	0	200000
FP_ARITH_INST_RETIRED.SCALAR_SINGLE	0	0	200000
INST_RETIRED.ANY	47,080,000	214	220000
MEM_LOAD_RETIRED.L1_HIT	1,000,000	1	200000
MEM_LOAD_RETIRED.L1_MISS	0	0	10000
MEM_LOAD_RETIRED.L2_HIT	50,000	1	10000
MEM_LOAD_RETIRED.L2_MISS	25,010	1	5002
MEM_LOAD_RETIRED.L3_HIT	25,010	1	5002
MEM_LOAD_RETIRED.L3_MISS	0	0	10000
UOPS_EXECUTED.THREAD	2,000,000	2	200000
UOPS_EXECUTED.X87	0	0	200000
UOPS_RETIRED.RETIRE_SLOTS	40,000,000	40	200000

从图中我们知道，行列次序 CPU Time，CPI Rate, 总体而言，Cache 列主次序优于行主次序。当问题规模较小时执行时间可能很短，可将核心计算重复多次，提高 profiling 的精度。

2. n 个数求和

平凡算法

Function / Call Stack	Clockticks ▼	Instructions Retired	CPI Rate	INST_RETIRED.ANY	CPU_CLK_UNHALTED.THREAD
func@0x18001c690	2,990,000	2,340,000	1.278	2,340,000	2,990,000
chain_unroll	2,080,000	3,640,000	0.571	3,640,000	2,080,000
func@0x1401d0376	1,820,000	130,000	14.000	130,000	1,820,000
func@0x180020340	650,000	260,000	2.500	260,000	650,000
ExpInterlockedPopEntrySList	650,000	130,000	5.000	130,000	650,000
func@0x18001c330	650,000	0		0	650,000
func@0x140112f10	520,000	520,000	1.000	520,000	520,000
func@0x1400e5b10	520,000	1,040,000	0.500	1,040,000	520,000
wcsrchr	520,000	650,000	0.800	650,000	520,000
func@0x1405d7e80	520,000	260,000	2.000	260,000	520,000
func@0x1c0004760	520,000	520,000	1.000	520,000	520,000
ExAcquirePushLockSharedEx	520,000	390,000	1.333	390,000	520,000
func@0x18001bbf0	390,000	390,000	1.000	390,000	390,000
func@0x1400c80a0	390,000	260,000	1.500	260,000	390,000
func@0x1406b3120	390,000	0		0	390,000
func@0x1401d39f8	390,000	260,000	1.500	260,000	390,000
func@0x14011e450	390,000	0		0	390,000
func@0x1400afac0	390,000	260,000	1.500	260,000	390,000
func@0x1400c9e30	390,000	260,000	1.500	260,000	390,000
func@0x1800231f0	390,000	130,000	3.000	130,000	390,000

超标量

Function / Call Stack	CPU Time ▼	Clockticks	Instructions Retired	CPI Rate	Retiring	Front-End Bound	Bad Speculation
func@0x18001c690	2.183ms	2,730,000	2,210,000	1.235	0.0%	0.0%	0.0
chain_2	1.132ms	1,300,000	3,640,000	0.357	0.0%	0.0%	0.0
ExAcquirePushLockSharedEx	0.728ms	130,000	520,000	0.250	0.0%	0.0%	0.0
func@0x1401d0376	0.647ms	1,820,000	0		0.0%	0.0%	0.0
func@0x18000bdb0	0.566ms	130,000	650,000	0.200	0.0%	0.0%	0.0
func@0x1400e3c29	0.566ms	130,000	260,000	0.500	0.0%	0.0%	0.0
func@0x1406b5de0	0.485ms	130,000	0		0.0%	0.0%	0.0
func@0x14011e450	0.485ms	260,000	390,000	0.667	0.0%	100.0%	0.0
func@0x14010c200	0.485ms	520,000	260,000	2.000	0.0%	0.0%	0.0
func@0x1400afac0	0.485ms	390,000	260,000	1.500	0.0%	0.0%	100.0
func@0x180023ff0	0.404ms	260,000	260,000	1.000	0.0%	0.0%	0.0
func@0x1400ceb90	0.404ms	0	0	0.000	0.0%	0.0%	
func@0x1400c65b0	0.404ms	130,000	0		0.0%	0.0%	0.0
func@0x1400cff23	0.404ms	130,000	0		0.0%	0.0%	0.0
ExpInterlockedPopEntrySList	0.323ms	130,000	260,000	0.500	0.0%	0.0%	0.0
func@0x1400c94b0	0.323ms	0	130,000	0.000	0.0%	0.0%	0.0
func@0x1c00398a5	0.323ms	390,000	390,000	1.000	0.0%	0.0%	0.0
memset	0.323ms	260,000	650,000	0.400	0.0%	0.0%	100.0
func@0x1c0007010	0.323ms	130,000	130,000	1.000	0.0%	0.0%	0.0
func@0x1406b7a50	0.323ms	130,000	390,000	0.333	0.0%	0.0%	0.0
func@0x18004552c	0.323ms	130,000	130,000	1.000	0.0%	0.0%	0.0

从图中可知，两路链式算法执行周期数为 1,300,000，CPI 为 0.357，优于链式算法的 CPI，同理据 Summary 主要分析的执行时间，高热点部分，CPU 使用直方图，可以看到总开销时间，程序中最耗时的部分，总体二路链式优于链式。

（五） 结果分析

算法角度分析：

Cache 是计算机系统中的一种高速缓存，它可以存储最近被访问过的数据和指令。因为内存访问速度较慢，所以我们可以将常用的数据存储在 cache 中，以减少内存访问的时间。在计算每一列与给定向量的内积时，我们可以将内积结果存储在 cache 中，以便下次计算时直接调用。当 Cache 容量一定的时候，若增大 Cache 块大小，Cache 不命中率先降后升。

profiling 分析硬件、系统软件性能的底层原因:

二维数组在 C/C++ 中为行主存储方式，这样按列访问数组可能会造成 cache 频繁 miss 的从而影响到程序执行的时间。

1. n 个数求和

数据生成人为指定，元素个数 n 取 2 的幂，程序的运行存在时间和空间上的局部性，前者是指只要内存中的值被换入缓存，今后一段时间内会被多次引用，后者是指该内存附近的值也被换入缓存。如果在编程中特别注意运用局部性原理，就会获得性能上的回报。关于 CPU 的流水

线并发性，Intel Pentium 处理器有两条流水线 U 和 V，每条流水线可各自独立地读写缓存，所以可以在一个时钟周期内同时执行两条指令。但这两条流水线不是对等的，U 流水线可以处理所有指令集，V 流水线只能处理简单指令。

进行浮点运算时，指令级并行算法与串行算法中元素累加顺序是不同的，由于计算机表示浮点数精度有限，计算次序的改变可能会导致结果变化。

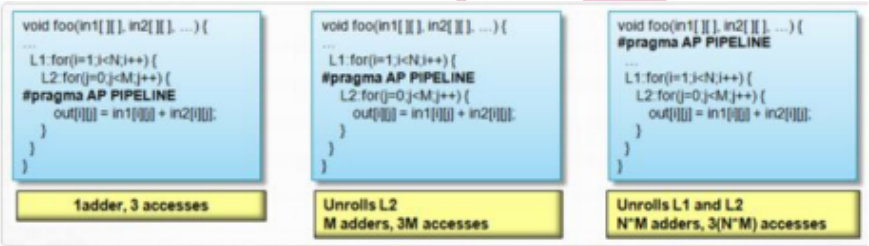
二、 进阶要求

(一) 循环策略

Unroll 和 Pipeline 指令的关系，Pipeline 指令放置的循环层次越高，循环展开的层次也越高，最终会导致使用更大面积的资源去实现，同时并行性也更高。

Unroll 指令在 for 循环的代码区域进行优化，这个指令不包含流水线执行的概念，单纯地将循环体展开使用更多地硬件资源实现，保证并行循环体在调度地过程中是彼此独立的。

Pipeline 指令在循环和函数两个层级都可以使用，通过增加重复的操作指令（如增加资源使用量等等）来减小初始化间隔 unroll 和 Pipeline 指令相互重合的关系在于，当对函数进行流水线处理时，以下层次结构中的所有循环都会自动展开，而使用展开指令的循环并没有给定对 II 的约束。



每个循环步进行多次加法运算，相当于将多个循环步的工作展开到一个循环步，从而大幅度降低簿记操作的比例。甚至可以采用宏/模板将循环完全去掉。不同算法尽量保持相同的展开比例，保证性能对比的公平性。循环展开可结合指令级并行，即合并到一个循环步中的多个计算通过合理设计令它们相互不依赖，可同时由多条流水线处理。

(二) X86ARM 对比

Dhrystone：测试文本处理能力

Whetstone：测试浮点运算效率和速度 Execl Throughput：execl 吞吐，这里的 execl 是类 unix 系统非常重要的函数，非办公软件的 execl 测试

File Copy：文件的读、写、复制测试

Pipe Throughput：进程之间的通讯

Pipe-based Context Switching：两个进程通过管道交换

Process Creation：进程创建

System Call Overhead：测试衡量进入和离开系统内核的消耗

Shell Scripts：一个进程可以启动并停止 shell 脚本的次数

表

5200	5200IPC	2711	2711IPC	IPCRate	Rate	
Dhrystone	2763.9	1063.0	1248	831.73	1.2781	2.2
Execl Put	1033.3	397.4	384.5	256.33	1.55042	2.7
Copy 256	1084.7	417.2	287	191.33	2.18045	3.8
Pipe put	779.8	299.9	214.3	142.87	2.09932	3.6
System Call	483.3	185.9	160.1	106.73	1.74158	3

表 5: x86 与 ARM 比较

像 Copy 和 Pipe put 分数可以通过提高缓存来解决，像 Dhrystone、Execl Put 这样的文本处理和计算型功能较好。树莓派的性能比 X86 落后很多，A72 的同频性能也跟 X86 差了不少，在 AVX512 这样的指令集面前，arm 的 NEON(32 位相当于 AVX64、64 位相当于 AVX12