南開大學
Nankai University

计算机学院

并行程序设计试验报告

体系结构编程

姓名：潘涛

学号：2314033

专业：计算机科学与技术

2025 年 3 月 29 日

目录

1 实验相关说明 **2**

2 实验一：n*n 矩阵与向量内积 **2**

2.1 问题描述 . . . 2
2.2 算法设计 . . . 2
2.2.1 平凡算法 . . . 2
2.2.2 cache 优化算法 . . . 2
2.2.3 循环展开算法 . . . 2
2.2.4 高精度计时 . . . 2
2.3 编程实现 . . . 3
2.3.1 高精度计时 . . . 3
2.3.2 平凡算法 . . . 3
2.3.3 cache 优化算法 . . . 4
2.3.4 循环展开算法 . . . 4
2.4 性能测试 . . . 4
2.5 profiling . . . 5
2.5.1 cache 命中率和超标量 . . . 5

3 实验二：n 个数求和 **6**

3.1 问题描述 . . . 6
3.2 算法设计 . . . 6
3.2.1 平凡算法设计思路 . . . 6
3.2.2 优化算法设计思路 . . . 6
3.3 编程实现 . . . 6
3.3.1 平凡算法 . . . 6
3.3.2 优化算法 . . . 7
3.4 性能测试 . . . 8
3.5 profiling . . . 8
3.5.1 cache 命中率 . . . 8
3.5.2 超标量 . . . 9

4 实验总结和思考 **9**

4.1 对比两个实验的异同 . . . 9
4.2 总结 . . . 10

1 实验相关说明

实验平台	X86
系统	windows11
cpu	13th Gen Intel(R) Core(TM) i5-13500H 2.60 GHz
编译器	TDM-GCC
集成开发环境	Code::Blocks
测试工具	VTune

实验代码仓库地址：https://github.com/pandar1223333666/parallel.git

2 实验一：n*n 矩阵与向量内积

2.1 问题描述

给定一个 n × n 矩阵，计算每一列与给定向量的内积，考虑两种算法设计思路：逐列访问元素的平凡算法和 cache 优化算法，进行实验对比：1. 对两种思路的算法编程实现；2. 练习使用高精度计时测试程序执行时间，比较平凡算法和优化算法的性能。

2.2 算法设计

2.2.1 平凡算法

对于 n*n 矩阵与向量内积的计算，一种朴素的实现方法是每一步直接计算出完整的一个内积值，而不需要中间值。我们使用平凡算法设计实现方法，通过逐列访问矩阵元素，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。算法包含两层循环，时间复杂度为 $O(n^2)$。

2.2.2 cache 优化算法

考虑到计算矩阵向量内积的算法时间复杂度最低为 $O(n^2)$, 已经无法算法层面进行优化，所以我们通过体系结构层面优化代码。平凡算法是通过按列访问，属于跨行访问，而内存存储一般是通过行优先存储的，因此会导致缓存不命中率高，增加了访存延迟。所以我们可以采用逐行访问矩阵元素，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。后者的访存模式具有很好空间局部性，符合内存存储顺序，令 cache 的作用得以发挥，提高了效率。

2.2.3 循环展开算法

针对 cache 优化算法中的循环操作，我们可以进一步处理，利用循环展开的方式优化算法。循环展开是一种通过减少循环迭代次数来提高程序运行效率的优化技术。它的核心思想是用空间换时间，通过增加单次迭代的工作量来降低循环控制的开销。下面的方法将对 cache 中的第二重循环进行展开，我们将原本的需要 4 次循环实现的功能变为 1 次，将在一次循环中同时进行 4 个中间值的计算，同时考虑到可能不足 4 项的情况，单独进行计算。

2.2.4 高精度计时

为了比较两算法的性能差距，我们使用高精度的计时来测试程序执行时间。由于问题较为简单，当矩阵规模较小时，程序运行时间很短。可采用重复运行待测函数，来解决计时函数精度不够、影响测

量精度的问题。我们使用 windows 自带的程序计时器来进行程序计时。对于规模较小时，当运行时间过短且小于一个固定时间时，重复循环运行多次该程序，在循环结束时，在计算平均运行时间，减小误差。对于不同的规模，我们选择 2000 以内的数据，当 n<=100 时，步长为 20，当 n>1000 时步长为 2000。

2.3 编程实现

2.3.1 高精度计时

逐列访问平凡算法

```
void way_1(int m) {
    int n, step = 20;
    long long freq, start, end;
    long counter;
    double total, avg;
    QueryPerformanceFrequency((LARGE_INTEGER*)&freq);
    for (n = 20; n <= m; n += step) {
        init(n);
        counter = 0;
        QueryPerformanceCounter((LARGE_INTEGER*)&start);
        while ((end - start) * 1000.0 / freq < 10)
        {
            counter++;
            functions()//待测函数
            QueryPerformanceCounter((LARGE_INTEGER*)&end);
        }
        total = (end - start) / (double)freq;
        avg = total / counter;
        cout << "Size: " << n << "\tAvg: " << avg * 1000 << "ms" << endl;
        if (n == 100) step = 200;
    }
}
```

2.3.2 平凡算法

逐列访问平凡算法

```
void way_1{
    for (int i = 0; i < n; i++) sum[i] = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            sum[i] += b[j][i] * a[j];
        }
    }
}
```

2.3.3 cache 优化算法

逐行访问优化算法

```
void way_2{
    for (int i = 0; i < n; i++) sum[i] = 0;
    for (int j = 0; j < n; j++) {
        for (int i = 0; i < n; i++) {
            sum[i] += b[j][i] * a[j];
            }
    }
}
```

2.3.4 循环展开算法

循环展开优化算法

```
void way_2{
    for (int i = 0; i < n; i++) sum[i] = 0;
    for (int j = 0; j < n; j++) {
        double aj = a[j];
        int i = 0;
        //一次循环实现同时计算4个中间项的值。
        for (i; i <= n-4; i += 4) {
            sum[i]   += b[j][i]   * aj;
            sum[i+1] += b[j][i+1] * aj;
            sum[i+2] += b[j][i+2] * aj;
            sum[i+3] += b[j][i+3] * aj;
            }
            //剩余不足四项的单独计算
        for (i; i < n; i++) {
            sum[i] += b[j][i] * aj;
            }
    }
}
```

2.4 性能测试

如表3所示，我们得到了在不同规模 (n<2000) 的平凡算法和 cache 优化算法运行时间的对比表格。通过表格，可以发现两个算法的执行时间都随着问题规模的变大而不断增长，说明随着规模的不断增大运行的消耗越大。在相同的规模下，平凡算法的执行时间明显高于优化算法的执行时间，而且随着问题规模的变大这一差距明显变大了。证明了采用逐行访问的操作有利于提高运算效率，节省运算时间。对于循环展开算法，由于其在 cache 的基础上额外进行了循环展开操作，因此执行的算法效果是三个中最优的。

数据规模 n	平凡算法 (ms)	cache 优化 (ms)	循环展开 (ms)
20	0.00108	0.00076	0.00056
40	0.00296	0.00241	0.00209
60	0.00784	0.00657	0.00498
80	0.01351	0.00995	0.00769
100	0.02188	0.01951	0.01299
300	0.26465	0.12827	0.10436
500	1.07997	0.36835	0.28096
700	2.33032	0.72250	0.54217
900	3.96300	1.19649	0.90123
1100	6.60030	1.95080	1.58723
1300	8.08920	2.47400	2.17616
1500	12.22420	3.49973	3.03883
1700	15.37680	4.41240	3.67603
1900	21.61600	5.87560	4.92980

表 1: 性能测试结果 (单位:ms)

2.5 profiling

2.5.1 cache 命中率和超标量

平凡算法二维数组在 C/C++ 中为逐列访问，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，使用优化算法采用行访问，cache 缓存命中率高，使得连续数据可并行加载，从而提高运行效率。超标量则允许单个处理器核心在同一时钟周期内并行执行多条指令。所以影响的主要因素应该在于 cache, 超标量影响后两个算法的不同，我们可以用 cpi 来反映。理论上优化算法的命中率和 cpi 应该是优于平凡算法的。下面我们采用 vtune 来进一步探究在程序执行过程中的各级缓存的命中率如表3所示，我们得到了在不同规模下的平凡算法和 cache 优化算法的缓存相关情况。

参数	平凡算法	cache 优化算法	循环展开
Elaspsed Time	0.223	0.203	0.194
CPU Time	0.213s	0.184s	0.181
CPI Rate	0.375	0.218	0.203
L1_HIT	498,001,9140	1,180,003,220	1,008,003,024
L1_MISS	39,200,588	2,000,030	2,400,036
L2_HIT	19,600,294	2,400,024	1,200,018
L2_MISS	19,003,990	0	20,000
L3_HIT	17,003,570	0	0
L3_MISS	3,001,260	0	0

表 2: vtune 性能测试结果

根据下面 cache 缓存命中率的公式 (1)，我们可以计算出各级缓存的命中率如下表。

$$Li\_hit\_rate = \frac{Li\_hit}{Li\_hit + Li\_miss} \tag{1}$$

根据性能测试结果表格，可以清晰地看到平凡算法和缓存优化算法在 CPU 时间、CPI（每条指令周期数）以及各级缓存（L1/L2/L3）的命中/缺失情况对比。我们发现在使用 cache 优化算法之后，cpu time 和 cpi Rate 都有了很大提升，执行性效率提高。两个算法相比较，优化算法的 miss 率明显低于

缓存层级	平凡算法 (%)	Cache 优化算法 (%)	循环展开算法 (%)
L1 命中率	99.22	99.83	99.77
L2 命中率	50.8	100.0	98.36
L3 命中率	85.0	N/A	N/A

表 3: 缓存命中率

平凡算法，说明采用优化算法能够增加缓存的命中率，在优先率更高的缓存上的命中率更高。而循环展开算法由于实现一次循环同时独立进行 4 个中间量的计算，通过增加单次迭代的工作量来降低循环控制的开销，实现了超标量的优化，因而效率提高。

3 实验二：n 个数求和

3.1 问题描述

计算 n 个数的和，考虑两种算法设计思路：逐个累加的平凡算法（链式）；适合超标量架构的指令级并行算法（相邻指令无依赖），如最简单的两路链式累加，再如递归算法——两两相加、中间结果再两两相加，依次类推，直至只剩下最终结果。完成如下作业：1. 对两种算法思路编程实现；2. 练习使用高精度计时测试程序执行时间，比较平凡算法和优化算法的性能。

3.2 算法设计

3.2.1 平凡算法设计思路

平凡算法的求解思路在于使用一层循环遍历 n 个数的数组中的每个元素，然后逐个相加，等到最终结果, 其时间复杂度为 O(n)。

3.2.2 优化算法设计思路

超标量优化算法能够程序在一定时间内执行多条指令，实现指令级的并行操作，从而提高程序执行效率。为此，我们可以将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，从而大幅度降低操作的比例。对于多路链式的方法，我们将待加的数据分为两组，同时进行两路的加和运算，最后将两路的加和合并在一起，实现最终的加和。这种方式是并行的，两路运算独立进行，而每一步都同时进行中互不干扰的运算，计算效率显著提高。

对于递归算法实现，我们将给定元素两两相加，得到 n/2 个中间结果，再将上一步得到的中间结果两两相加，得到 n/4 个中间结果; 依此类推，log(n) 个步骤后得到一个值即为最终结果。为了实现这一目标，我们有两种实现方式。一种直接运用递归函数，优点是实现简单，缺点是递归函数调用开销较大。另一种是采用二重循环的方式实现。这两种方法都在实现了在一步对两个数据进行了累加计算，相当于降低了执行次数，因此算法效率有效提高。

3.3 编程实现

3.3.1 平凡算法

平凡算法求和

```
 void way1(int M) {
QueryPerformanceFrequency((LARGE_INTEGER*)&freq);
    for (int m = 1; m <= M; m++) {
        int n = pow(2,m);
        init(n);
        long counter = 0;
        QueryPerformanceCounter((LARGE_INTEGER*)&start);
        QueryPerformanceCounter((LARGE_INTEGER*)&end);
        while ((end - start) * 1000.0 / freq < 10) {
            sum = 0;
            for (int i = 0; i < n; i++) {
                sum += a[i];
            }
            counter++;
    QueryPerformanceCounter((LARGE_INTEGER*)&end);}
        total = (end - start) / (double)freq;
        avg = total / counter * 1000000;
        cout << "2^" << m << " (" << n << ")\t" << avg << " us" << endl;}
}
```

3.3.2 优化算法

由于高精度计时采用的方法与平凡算法类似，以下我们仅给出核心算法。

多链路式

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

递归

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

二重循环

```
for(int m=n;m>1;m/2){
    for(int i=0;i<m/2;i++){
        a[i] = a[i*2] + a[i*2+1];
    }
}
```

3.4 性能测试

如表6所示，我们得到了在不同规模时的平凡算法和不同方式实现的 cache 优化算法运行时间的对比表格。

规模 n	平凡算法	多路链式	递归	二重循环
2^1 (2)	0.022937	0.017579	0.018666	0.017865
2^2 (4)	0.018943	0.017888	0.021332	0.020763
2^3 (8)	0.024876	0.019609	0.026172	0.027432
2^4 (16)	0.038336	0.026493	0.039546	0.042314
2^5 (32)	0.067115	0.040503	0.065522	0.066240
2^6 (64)	0.124741	0.070012	0.137713	0.124299
2^7 (128)	0.239497	0.128005	0.246305	0.244248
2^8 (256)	0.467352	0.249767	0.420955	0.433069
2^9 (512)	0.935247	0.482160	0.727643	0.740865
2^{10} (1024)	1.846580	0.933106	1.329120	1.415330
2^{11} (2048)	5.247590	1.941580	2.577450	2.737300
2^{12} (4096)	7.576140	4.088030	4.915480	5.208430

表 4: 性能测试结果 (单位:us)

分析表中数据，我们可以明显的的发现使用 cache 优化后的数据明显优于平凡算法。综合来看，我们可以得知采用多路链式的优化方法是最优的，而递归和二重循环的实现优劣相差不大，平凡算法最差。考虑到递归的操作需要额外的开销，理论上应该差于二重循环，但实际相差不大，可能是由于规模太小，导致导致递归层数浅，开销可以忽略。另外，由于多路链式的方法通过独立的累加变量消除了数据依赖，允许超标量执行，并且采用了连续访问模式，要比后两个方法的跨步访问更加利于缓存预取。

3.5 profiling

3.5.1 cache 命中率

通过 vtune 探究 cache 命中率，更好的理解程序执行时变化。根据表格中的数据，我们可以发现后面三种方法的 cpi rate, 即表示每条指令消耗的平均时钟周期数，明显小于平凡算法，说明后面优化的三种方法的执行那个效率更高。另外，根据三个缓存的命中率，我们发现没有明显差别，可能是由于这四种方法都是按行依次访问数据不会造成过多的 miss 导致。有关 cache 方面没有很大差异，说明 cache 并不是造成不同执行效率的主要因素。

参数	平凡算法	多路链式	递归	二重循环
Elasped Time	0.140	0.147	0.151	0.137
CPU Time	0.129	0.134	0.132	0.137
CPI Rate	0.646	0.323	0.344	0.350
L1_HIT	342,001,026	480,001,890	554,001,662	416,001,248
L1_MISS	0	4,00,006	0	0
L2_HIT	0	400,006	0	0
L2_MISS	0	0	0	0
L3_HIT	0	0	0	0
L3_MISS	0	0	0	0
L1 命中率	1	0.998	1	1
L2 命中率	N/A	1	N/A	N/A
L3 命中率	N/A	N/A	N/A	N/A

表 5: 测试结果

3.5.2 超标量

下面，我们利用 vtune 进行基于超标量方面的测试和分析。超标量是一种现代 CPU 设计技术，允许单个处理器核心在同一时钟周期内并行执行多条指令，从而提升性能。它的核心思想是通过指令级并行挖掘代码中的并行性，让 CPU 更高效地利用硬件资源。我们可以利用 cpi 来衡量超标量的效果。考虑到 cpi 表示 CPU 执行一条指令平均需要的时钟周期数，用于衡量 CPU 执行指令的效率，CPI 越低，说明 CPU 执行指令的效率越高。优化算法可以令两条流水线可以充分的并发运行指令，从而提高计算效率，加快计算速度。对于平凡算法而言，其在测试中的 cpi 理论上应该高于优化算法。我们以 n=1024 为例，进行 500 次计算，得到下表结果。结果显示，多路链式的 cpi 最优，递归和二重循环的次之相差不大，平凡算法的 cpi 最大，说明执行一条指令平均需要的时钟周期数较长，计算效率较低。对于实现递归的两种算法，由于需要额外的开销，所以 cpi 略差于多路链式的方法。

参数	平凡算法	多路链式	递归	二重循环
Clockticks	44,800,000	41,600,000	48,000,000	44,800,000
Instructions Retired	83,200,000	86,400,000	92,800,000	92,800,000
CPI Rate	0.638	0.481	0.517	0.537

表 6: 测试结果

4 实验总结和思考

4.1 对比两个实验的异同

第一个矩阵向量积的实验主要通过探究了 cache 优化的影响，逐行访问的算法更加循计算机内部行主次序存储的本质，通过每次访问行中的元素使得计算机内部的缓存能够等到充分利用，不会因为 miss 而产生消耗；第二个实验这是采用各种方法通过一步操作执行更多指令来降低内部循环的次数，同时采用多路可以实现并行操作，在同一时钟周期执行多条命令，体现了超标量的影响。对于实验一追加的循环展开操作，则是综合了 cache 优化和超标量的影响，体现了将两者结合的优势。

4.2 总结

1. 程序运行的性能效果与多种因素相关,采用一些合理的算法可以有效提高算法运行的效率,cache优化和超标量是可以考虑的两个有效方法。

2. 在实验过程中，我加深了对计算机体系结构的认识，了解了高速缓存的三级结构，了解了 cpi 可以作为衡量程序执行效率的一个指标，能够利用 vtune 对程序进行简单的测试和分析。

3. 不足之处，对各种工具不够熟悉，对于实验的整体流程不够熟悉。