南開大學
Nankai University

计算机学院

并行程序设计报告

CPU 架构相关编程

姓名：章壹程

学号：2313469

专业：计算机科学与技术

2025 年 3 月 23 日

目录

1 前言

1.1 实验环境

- **CPU：** 12th Gen Intel(R) Core(TM) i7-12800HX 2.00 GHz ，16 内核，性能核主频 2.0GHz，最大频率 4.80 GHz，25MB Intel® Smart Cache。
- **操作系统：** Windows 11 家庭中文版 26100.3476。
- **编译器：** g++ 10.3.0 (tdm64-1) ，无额外编译选项。

1.2 开源代码

所有代码均可在https://github.com/u2003yuge/parallel-programming上下载，包括但不限于实验所需代码、可视化代码、实验图像。

2 基础要求

2.1 n*n 矩阵与向量内积

2.1.1 算法设计

平凡算法是按列进行，而 cache 优化改为逐行访问，具有很好的空间局限性，令 cache 作用得以发挥。

2.1.2 编程实现

逐列访问平凡算法

```
double cal1(int n){//平凡算法
    double head,tail,freq,head1,tail1,timess=0;
    QueryPerformanceFrequency((LARGE_INTEGER *)&freq );
    QueryPerformanceCounter((LARGE_INTEGER *)&head);
    for(int i=0;i<n;i++){
        sum[i]=0.0;
        for(int j=0;j<n;j++)
            sum[i]+=b[j][i]*a[j];
    }
    QueryPerformanceCounter ((LARGE_INTEGER *)& tail) ;
    return (tail-head)*1000.0 / freq;
}
```

逐行访问 cache 优化算法

```
double cal2(int n){//cache优化
    double head,tail,freq,head1,tail1,timess=0;
    QueryPerformanceFrequency((LARGE_INTEGER *)&freq );
    QueryPerformanceCounter((LARGE_INTEGER *)&head);

```

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

    QueryPerformanceCounter ((LARGE_INTEGER *)& tail) ;
    return (tail-head)*1000.0 / freq;
}
```

2.1.3 性能测试

实验设计

为了更好地测试 double 相乘的效果，设定 $a[i] = i/2.0$，$b[i][j] = 1.0 * i/(j+1)$。由于程序运行时间较短且波动很大，各实验均采用了舍弃第一次运行结果，除此之外**运行 100 次求平均**的方式以获得平均性能；同时所有实验均尽可能在同一段连续时间，环境不变的情况下进行，**以减少设备性能的波动对测试结果的影响**。对于部分不符预期的实验结果进行重跑以确保其结果并非受偶然因素干扰。

我实验了在不同内存空间设定下，不同问题规模下，使用平凡算法和优化算法各自所花费的时长。具体而言,我测试了当空间大小分别取 N=2500,2559,2560,2561,3000,3500,4000,4095,4096,4097,4500,5000，n 以 10 的间隔从 0 到 2500 取样时，平凡算法和优化算法的用时。

实验结果

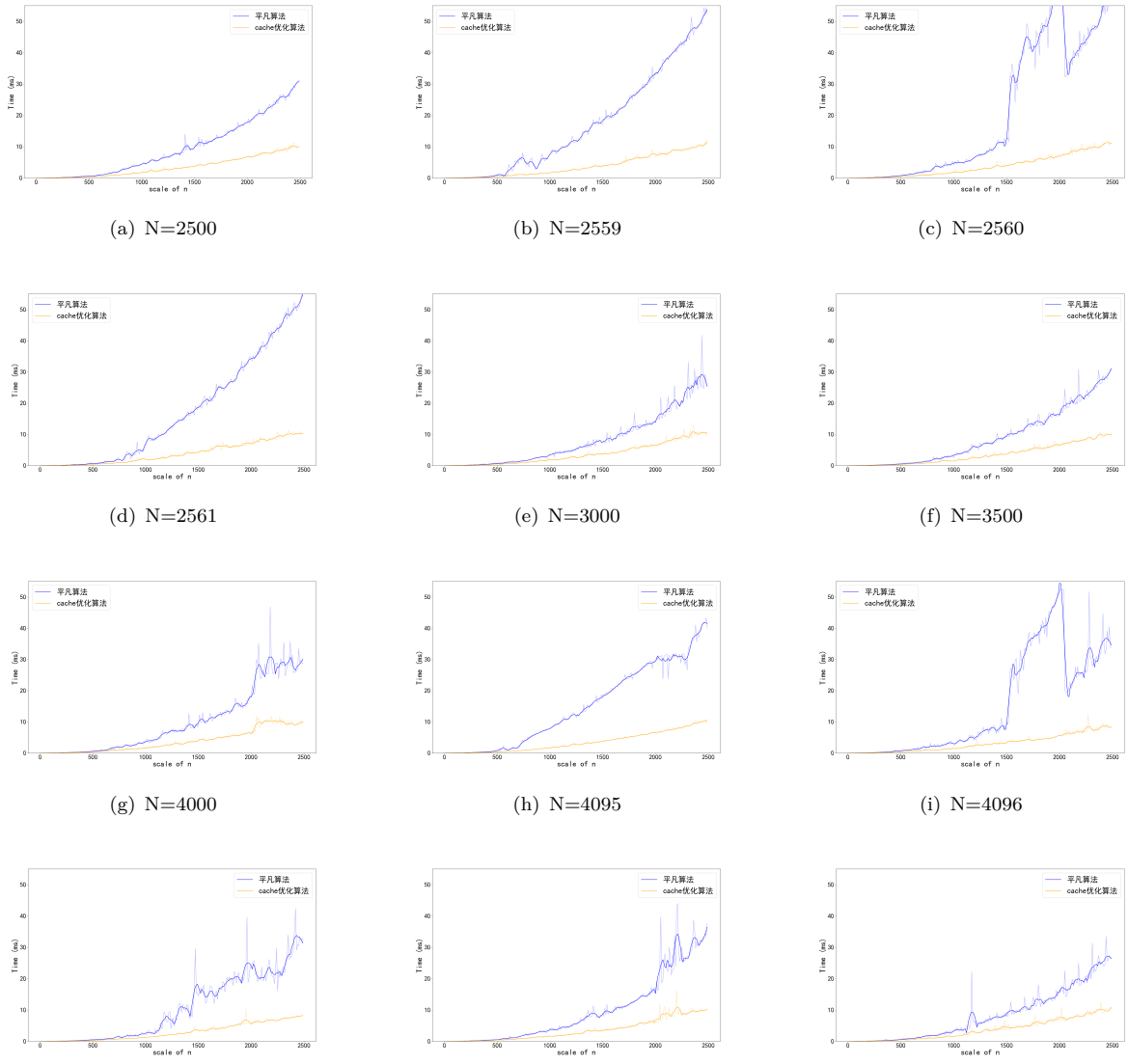
实验结果见图2.1。其中蓝色线代表了平凡算法，黄色线代表了 cache 优化算法，为了减少数据波动对趋势观测的影响加入了平滑化处理，透明度更高的折现为原始数据。

2.1.4 profiling

利用 Vtune 软件测试了当 N 和 n 取特定值时 Memory bound 的相关数值，用于间接反映缓存命中率，bound 的数值越高说明其缓存命中率越低。实验结果见表1以及表2。

	n=1000					n=2000				
N\Bound	Memory	L1	L2	L3	DRAM	Memory	L1	L2	L3	DRAM
2500	31.6%	3.9%	0.0%	0.0%	41.5%	44.7%	0.9%	6.2%	0.1%	42.1%
2560	60.8%	0.0%	0.0%	4.8%	51.5%	90.5%	0.0%	0.0%	0.0%	81.1%
3000	30.8%	13.6%	7.9%	0.0%	23.7%	31.0%	0.8%	0.0%	1.1%	23.4%
4000	41.1%	0.0%	0.0%	0.0%	53.1%	41.1%	0.0%	0.3%	3.4%	42.3%
4096	42.9%	0.0%	7.1%	6.2%	46.0%	73.3%	0.2%	0.8%	0.3%	72.0%
5000	25.6%	0.0%	6.2%	1.8%	13.2%	31.6%	4.8%	7.4%	0.0%	26.8%

表 1: 平凡算法



(a) N=2500 (b) N=2559 (c) N=2560

(d) N=2561 (e) N=3000 (f) N=3500

(g) N=4000 (h) N=4095 (i) N=4096

(j) N=4097 (k) N=4500 (l) N=5000

图 2.1: 不同设定参数下平凡算法和优化算法的执行时间对比

	n=1000					n=2000				
N\Bound	Memory	L1	L2	L3	DRAM	Memory	L1	L2	L3	DRAM
2500	18.6%	0.0%	5.9%	5.9%	1.5%	8.4%	0.0%	2.3%	3.0%	6.1%
2560	3.7%	6.8%	0.0%	5.5%	8.2%	8.7%	1.8%	2.2%	0.7%	5.4%
3000	13.5%	0.0%	1.5%	4.4%	5.9%	6.7%	6.7%	0.0%	3.7%	7.0%
4000	3.8%	0.0%	3.1%	3.1%	0.0%	9.3%	3.1%	0.0%	1.1%	7.6%
4096	11.9%	6.1%	0.0%	12.2%	7.6%	6.8%	0.0%	3.9%	2.3%	4.7%
5000	11.2%	1.9%	0.0%	4.1%	8.3%	7.3%	0.0%	1.6%	4.8%	0.0%

表 2: cache 优化算法

2.1.5 结果分析

根据图像可以看出，在相同 N 的情况下，平凡算法和 cache 优化算法的整体趋势都为线性上升，但 cache 优化算法的上涨速率要低于平凡算法。根据 profiling 分析我们能得其背后的原因，cache 优化后的缓存命中率大大提高，并且未命中的情况更多集中在 L1/L2/L3 中而非像平凡算法一样集中在 DRAM，这说明连续的地址访问更有助于 CPU 进行预测分配缓存内容从而优化性能。

但是令人感到奇怪的是，在 N=2560 和 N=4096 时，**从 1500 至 2200 整段使用平凡算法时运行时长出现了诡异的大幅上升，并在之后回归正常**；同时在其附近（即 N=2559,2561,4095,4097）时平凡算法运行速度的整体上升速率远高于其它情况。通过 profiling 分析可以发现，当 N=2560 时，整体的 Memory Bound 都异常的高；当 N=4096，n=2000 时结果也反常的高；而 N=3000，5000 时虽然分配的内存比 N=2560，4096 大，但是结果却比他更优异。2560 和 4096 两个数字十分的齐整，结合我的 CPU 的缓存总共是 25MB，2560×2560 的 double 数组所占用空间刚好是其两倍，可以合理推测此时恰为 CPU 预测算法设计的分割点，当应用占用内存更小时会倾向于利用缓存，而占用内存更大时会更倾向于利用内存，而在 N=2560 和 4096 时，算法难以抉择使得**数据频繁的在缓存和内存之间传输**，存储位置不固定，导致缓存命中率严重下降，效率严重下滑。

2.2 n 个数求和

2.2.1 算法设计

平凡算法使用普通的链式实现，优化算法分为两种，一种是使用 unroll 展开，另一种是使用递归实现。递归使用伪递归实现，即用循环模拟而非使用函数递归的方式，因为后者的效率实在太低了。

2.2.2 编程实现

平凡算法

```
double cal1(int n){//平凡算法
    double head,tail,freq,head1,tail1,timess=0;
    double sum=0.0;
    QueryPerformanceFrequency((LARGE_INTEGER *)&freq );
    QueryPerformanceCounter((LARGE_INTEGER *)&head);
    for(int i=0;i<n;i++)
        sum+=a[i];
    QueryPerformanceCounter ((LARGE_INTEGER *)& tail) ;
    return (tail-head)*1000.0 / freq;
}
```

两路优化

```
double cal2_2(int n){//两路优化
    double head,tail,freq,head1,tail1,timess=0;
    double sum,sum1=0.0,sum2=0.0;
    QueryPerformanceFrequency((LARGE_INTEGER *)&freq );
    QueryPerformanceCounter((LARGE_INTEGER *)&head);
    for(int i=0;i<n;i+=2){
        sum1+=a[i];
        sum2+=a[i+1];
    }
    sum=sum1+sum2;
    QueryPerformanceCounter ((LARGE_INTEGER *)& tail) ;
    return (tail-head)*1000.0 / freq;
}
```

递归优化

```
double cal3(int n){//递归优化
    double head,tail,freq,head1,tail1,timess=0;
    double sum=0.0;
    for(int i=0;i<n;i++)
        b[i]=a[i];
    QueryPerformanceFrequency((LARGE_INTEGER *)&freq );
    QueryPerformanceCounter((LARGE_INTEGER *)&head);
    for(int m=n;m>1;m/=2)
        for(int i=0;i<m/2;i++)
            b[i]=b[i<<1]+b[i<<1|1];
    sum=b[0];
    QueryPerformanceCounter ((LARGE_INTEGER *)& tail) ;
    return (tail-head)*1000.0 / freq;
}
```

2.2.3 性能测试

实验设计

设定 a[i]=a[i]/3.0, 由于程序运行时间较短且波动很大，各实验均采用了舍弃第一次运行结果，除此之外**运行 100 次求平均**的方式以获得平均性能；同时所有实验均尽可能在同一段连续时间，环境不变的情况下进行，**以减少设备性能的波动对测试结果的影响**。对于实验结果重跑了三次以确保其结果并非受偶然因素干扰。

测试了使用平凡算法，多种多路优化算法，和递归优化算法在不同规模下的运行时间，内存空间固定为 16777216。为了让测试的问题规模范围比较大，选择使用从 16 到 4000 步长为 16 的所有数的平方作为 n 的取值。**实验结果**

2.2.4 profiling

利用 Vtune 软件测量了各种算法下的 CPI 数值，来体现各种超标量优化算法的效果。

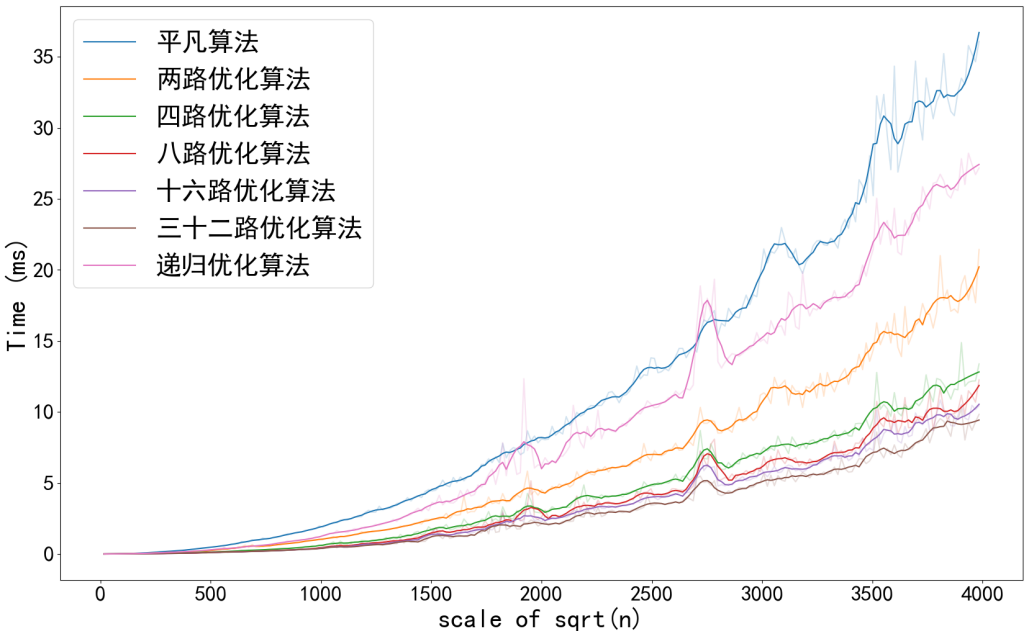


图 2.2: 不同问题规模下各种算法运行时长

算法\$\sqrt{n}$	1000	2000	4000
平凡	0.637	0.647	0.639
递归	0.229	0.271	0.308
两路	0.408	0.420	0.418
四路	0.276	0.327	0.334
八路	0.260	0.302	0.303
十六路	0.218	0.287	0.298
三十二路	**0.214**	**0.258**	**0.266**

表 3: 使用不同算法在不同任务规模下的 CPI 值

2.3 结果分析

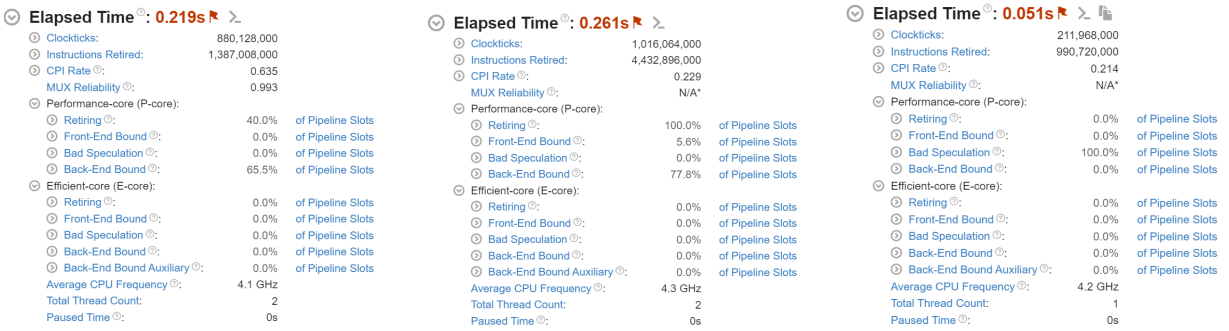
从图2.2中可以看到，递归优化算法的优化效果最差，三十二路优化算法优化最好。对于所有的 unroll 优化，从平凡到两路的优化最明显，十六路到三十二路的优化效果最低，这点通过2也能反映出来，其 CPI 的优化量越来越小，这说明多路并行优化是存在上限的。

除此之外，递归算法的 CPI 也很低，但它最后的用时却不低。结合图2.3可以发现，虽然相比之下递归优化算法的 Retiring 值为 100%，相比之下三十二路的 Bad Speculation 为 100%——即递归优化算法流水线槽的浪费几乎为 0，但是递归优化算法有更高的 Back-End Bound，也就是其 cache 命中率更低，导致了其速度更慢。

3 进阶要求

所有的进阶内容均被融入到了各项实验当中，在这里仅进行统一地重新介绍。

第一问中研究了分配内存空间大小对于运行速度的影响，进行了多组实验，并利用 VTune 分析了



(a) 平凡算法 (b) 递归优化算法 (c) 三十二路优化算法

图 2.3: n=1e6 时不同算法下 VTune 分析结果

其背后的原因。

第二问中对 unroll 技术进行了测试，测量了多组 CPI 数值以体现 unroll 的效果，并利用 VTune 分析了递归优化运行时间慢于多路优化的原因。

4 实验总结和思考

- 了解了 cache 优化技术和 unroll 展开技术，对于并行优化有了初步的了解：优化时间复杂度不是唯一优化的途径，在此基础上利用并行相关知识，适当地调整代码逻辑顺序和简化操作也可以有效地提高效率。
- 初步学习了 Vtune 软件的使用，了解比较性能的指标，并学会通过指标分析问题，利用指标优化算法。