南開大學
Nankai University

计算机学院

并行程序设计报告

并行体系结构相关实验及性能测试

姓名：张铭

学号：2211289

专业：计算机科学与技术

2024 年 3 月 16 日

目录

1 实验环境

本实验程序使用 c++ 语言编写，并分别在本机 x86 平台 windows 操作系统以及华为鲲鹏 arm 平台 Linux 操作系统服务器上进行运行。编译器方面 x86 平台使用 Visual Studio2022 进行编译，arm 平台使用 gcc 编译器进行编译，性能剖析工具方面分别使用 VTune 和 Perf 分析 cache 未命中率。

实验源代码仓库：https://github.com/DianGun-otto/Parallel-Project/tree/main/Lab1

2 矩阵与向量内积

2.1 实验内容

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

2.2 算法设计与编程

2.2.1 设计思路

平凡算法逐列访问矩阵元素，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；cache 优化算法则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。后者的访存模式具有很好空间局部性，令 cache 的作用得以发挥。

测试数据的生成: 对于本问题，人为设定 a[i][j]=i+j，b[i]=i, 方便程序正确性检查；由于每次程序运行时间较短，容易产生较大误差，因此使用多次循环取平均值的方法计算单次运行时间。

2.2.2 编程实现

平凡算法

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    ord_sum[i] += a[j] * b[j][i];
```

cache 优化算法

```
for (int j = 0; j < N; j++)
  for (int i = 0; i < N; i++)
    cache_sum[i] += a[j] * b[j][i];
```

2.3 结果分析

2.3.1 平凡算法和优化算法性能对比

在本机 x86 平台下运行程序得到的运行时间结果如表 1 所示

N	ord	cache	N	ord	cache	N	ord	cache	N	ord	cache
10	0.0003	0.0004	80	0.0122	0.012	600	1.443	0.8117	4000	129.626	33.319
20	0.0009	0.001	90	0.015	0.0153	700	2.0796	1.1184	5000	110.721	48.1789
30	0.0026	0.0025	100	0.0243	0.0237	800	2.5775	1.217	6000	220.04	74.2657
40	0.0068	0.0068	200	0.0839	0.085	900	3.6246	1.5501	7000	269.428	101.584
50	0.0069	0.0072	300	0.2103	0.2043	1000	3.5505	1.8547	8000	664.048	150.948
60	0.0068	0.007	400	0.4982	0.3192	2000	17.6497	8.6737	9000	590.581	197.253
70	0.0112	0.0121	500	0.959	0.6298	3000	38.0255	20.3741	10000	758.071	241.993

表 1: x86 平台运行结果（单位：ms）

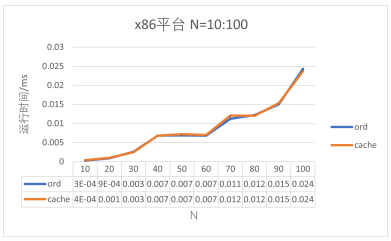
在华为鲲鹏服务器 arm 平台下运行程序得到的运行时间结果如表 2 所示

N	ord	cache	N	ord	cache	N	ord	cache	N	ord	cache
10	0	0	80	0.034	0.028	600	1.843	1.594	4000	210.237	80.192
20	0.002	0.002	90	0.045	0.035	700	2.463	2.148	5000	348.479	146.896
30	0.004	0.004	100	0.048	0.043	800	3.478	2.813	6000	519.083	208.061
40	0.008	0.007	200	0.189	0.177	900	4.404	3.583	7000	850.715	283.490
50	0.014	0.012	300	0.426	0.404	1000	5.514	4.434	8000	1299.901	370.141
60	0.020	0.016	400	0.788	1.108	2000	23.939	18.786	9000	1589.27	471.228
70	0.029	0.022	500	1.279	1.101	3000	56.497	44.230	10000	1930.99	578.769

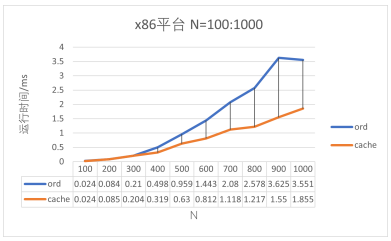
表 2: arm 平台运行结果（单位：ms）

绘制折线图：x86 平台如图2.1所示，arm 平台如图2.2所示。

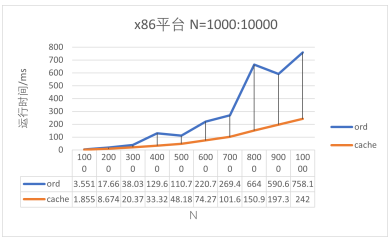
通过折线图可以看出，当 N=10:100 时，无论在哪个平台上运行，cache 算法与平凡算法的运行时间基本一致；当 N=100：1000 时，x86 平台 cache 算法相较于平凡算法有了很大提升，而 arm 平台的提升效果并不显著；当 N=1000:10000 时，x86 平台与 arm 平台 cache 算法均有较大提升。从程序的角度分析，当采用平凡算法时逐列访问矩阵元素，每次访问元素的内存位置并不相邻，并且随着数据规模的变大相隔更远，不会同时在 L1 级 Cache 中，导致访问下一个元素时无法快速从 Cache 中取出下一个操作对象，发生 Cache 缺失，又要花费更多时间从内存中去读数，增加了时间开销，效率降低；Cache 优化算法的访存模式具有很好空间局部性，令 cache 的作用得以发挥，减少 Cache 缺失的次数，提高工作效率。



(a) N=10:100

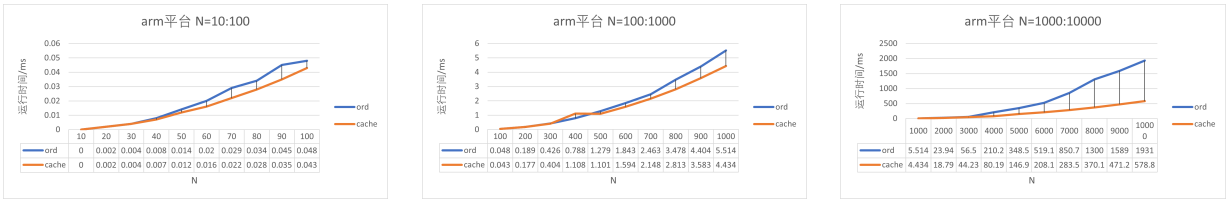


(b) N=100:1000



(c) N=1000:10000

图 2.1: x86 平台 ord 算法与 cache 算法运行效果对比



(a) N=10:100 (b) N=100:1000 (c) N=1000:10000

图 2.2: arm 平台 ord 算法与 cache 算法运行效果对比

2.3.2 CPU 系统参数对 cache 算法优化力度的影响

	x86 本机	arm 服务器
L1 缓存	1,229KB	64KB
L2 缓存	11776KB	512KB
L3 缓存	24576KB	49512KB

表 3: x86 平台与 arm 平台 cache 各级缓存对比

如表3所示, 粗略估算后得出:

x86 平台填满各级 cache 的 N 取值大约为: 400,900,1800

arm 平台填满各级 cache 的 N 取值大约为: 90,250,2500

因此我们使用 VTune 测试 x86 平台下 N=400,900,2000 时的 cache 未命中率, 如表4所示, N=300 时, x86 系统上运行 cache 算法的 cache 未命中率与平凡算法差距不大, 并未产生较大运行时间差异; N=900 时, 此时平凡算法的未命中率远高于 cache 算法, 由于 L1 被填满, 被迫在 L2 进行访存操作, 造成了时间差异; N=4000 时, L2 被填满, 数据进入 L3 进行访存, 进一步造成 L3 未命中率远高于 cache 算法, 运行时间的差异被进一步放大。

		L1_MissRate	L2_MissRate	L3_MissRate
N=300	ord	1.7%	0.0%	0.0%
	cache	0.7%	0.0%	0.0%
N=900	ord	5.1%	11.1%	0.0%
	cache	1.5%	0.0%	0.0%
N=4000	ord	1.6%	13.0%	39%
	cache	0.2%	0.0%	0.0%

表 4: x86 系统下的缓存未命中率

使用 perf 测试 arm 平台下 N=90,300,3000 时的 cache 未命中率, 如表5所示, N=90 和 N=300 时, 两种的 cache 未命中率差距极其小, 基本可以忽略不计, 而当 N 提升至 3000 时, 由于 L1 与 L2 容量远小于 L3, 此时数据进入 L3 级 cache 中, 平凡算法的 cache 未命中率远高于 cache 算法, 多次访存操作致使平凡算法的性能大幅低于 cache 算法。

		cache-misses	cache-references	MissRate
N=90	ord	7,455	608,558	1.22%
	cache	7,366	608,395	1.21%
N=300	ord	16,404	2,256,447	0.72%
	cache	12,119	2,256,892	0.53%
N=3000	ord	6,438,568	180,619,325	3.56%
	cache	547,943	180,611,901	0.30%

表 5: arm 系统下的缓存未命中率

综合看来，数据规模越大，cache 算法的优化力度越显著，对运行时间的降低效果越好，此时 cache 算法相较于平凡算法，降低了 cache 的未命中率，大幅提高了 cache 的利用率，减少了数据向更高层缓存的迁移，从而更好的减少了对内存的访问与储存操作，从而降低了运行时间，达到优化程序性能的效果。

3 n 个数求和

3.1 实验内容

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，再如递归算法等。

3.2 算法设计与编程

3.2.1 设计思路

链式算法通过将首先给定元素两两相加，得到 n/2 个中间结果，之后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。

递归算法可通过函数递归的方式实现，虽然该思路比较简单容易实现，但缺点是递归函数调用开销较大因此考虑第二种思路，通过相邻元素相加连续存储到数组最前面，中间结果重复上述操作，依次类推，总共执行 log(n) 个步骤，直至只剩下最终结果 a[0]。

测试数据的生成: 对于本问题，认为设定问题规模取 2N，方便递归算法设计。

3.2.2 编程实现

平凡算法

```
for(int i=0;i<N;i++)
sum+=a[i];
```

链式算法

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

函数递归

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

3.3 结果分析

3.3.1 平凡算法与优化算法性能对比

如表6, 表7所示，我们可以看出，而两种递归算法相较于平凡算法反而运行时间更长了，这是因为递归算法对内存的调用开销较大，因而造成 cpu 对内存空间的访存十分频繁，即使降低了算法的时间复杂度，也并没有带来性能的提示，而且两种递归算法的每次操作对内存的访问方式与矩阵与向量内积问题相似，即每次访问元素的内存位置并不相邻，因此有所性能降低。

N	ord	chain	recursion	cir	N	ord	chain	recursion	cir
10	9.43E-04	7.74E-04	2.90E-03	1.40E-03	19	4.18E-01	3.73E-01	1.65E+00	6.11E-01
11	2.06E-03	1.74E-03	5.70E-03	2.70E-03	20	8.57E-01	7.46E-01	3.37E+00	1.72E+00
12	4.31E-03	3.29E-03	1.10E-02	5.90E-03	21	1.86E+00	1.61E+00	6.67E+00	2.86E+00
13	8.78E-03	5.25E-03	2.40E-02	1.13E-02	22	3.94E+00	3.43E+00	1.33E+01	5.43E+00
14	1.64E-02	1.23E-02	5.30E-02	1.85E-02	23	7.40E+00	6.83E+00	2.69E+01	9.20E+00
15	3.34E-02	2.46E-02	9.70E-02	3.69E-02	24	1.51E+01	1.35E+01	5.31E+01	1.81E+01
16	6.68E-02	4.86E-02	1.70E-01	7.33E-02	25	3.32E+01	2.75E+01	1.18E+02	3.64E+01
17	1.33E-01	9.32E-02	3.40E-01	1.70E-01	26	7.19E+01	5.56E+01	2.58E+02	7.31E+01
18	2.55E-01	1.85E-01	7.30E-01	3.69E-01	27	1.25E+02	1.15E+02	4.42E+02	1.76E+02

表 6: x86 平台下不同算法的运行时间（单位：ms）

N	ord	chain	recursion	cir	N	ord	chain	recursion	cir
10	3.45E-03	1.96E-03	5.14E-03	3.92E-03	19	1.81E+00	1.05E+00	2.70E+00	2.15E+00
11	6.83E-03	3.87E-03	9.66E-03	7.74E-03	20	3.68E+00	2.16E+00	5.65E+00	4.35E+00
12	1.36E-02	7.70E-03	1.87E-02	1.54E-02	21	7.63E+00	4.70E+00	1.49E+01	9.22E+00
13	2.72E-02	1.54E-02	3.62E-02	3.07E-02	22	1.65E+01	1.05E+01	2.87E+01	1.99E+01
14	5.50E-02	3.16E-02	7.54E-02	6.30E-02	23	3.76E+01	2.10E+01	7.32E+01	4.00E+01
15	1.10E-01	6.32E-02	1.56E-01	1.26E-01	24	6.67E+01	4.20E+01	1.52E+02	7.98E+01
16	2.20E-01	1.27E-01	3.23E-01	2.54E-01	25	1.32E+02	8.34E+01	3.10E+02	1.58E+02
17	4.52E-01	2.57E-01	6.63E-01	5.14E-01	26	2.67E+02	1.69E+02	6.43E+02	3.19E+02
18	9.06E-01	5.31E-01	1.34E+00	1.07E+00	27	5.71E+02	3.40E+02	1.30E+03	7.21E+02

表 7: arm 平台下不同算法的运行时间（单位：ms）

3.3.2 x86 平台与 arm 平台性能对比

(a) x86 平台 (b) arm 平台

图 3.3: ord 算法与超标量算法运行效果对比

	ratio
x86	13.52%
arm	39.02%

表 8: x86 平台与 arm 平台平均加速比

3.3.3 加速比与问题规模的关系

从图3.4可以看出，arm 平台下的性能加速比随问题规模的变化并不显著，基本趋于平稳；x86 平台下的性能加速比起伏较大，问题规模较小（210 212）时，性能加速比随着问题规模的增大而提升，当问题规模在 213 218 之间时，加速比基本维持稳定，而当问题规模进一步提升时，加速比有所下降，推测此时由于问题规模的提高，L2 和 L3 不可避免的进行访存操作，cache 命中率降低，造成加速比的降低。

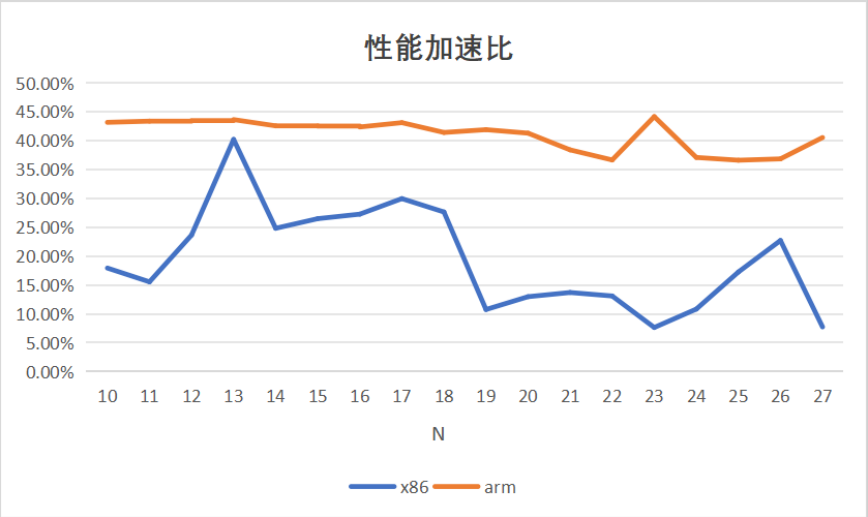


图 3.4: 性能加速比随问题规模的变化

4 总结与体会

4.1 实验总结

在矩阵计算实验中，我们使用了 cache 优化算法提高了矩阵运算的性能，通过逐行访问数组元素，向每个内积累加乘法结果，这样的模式可以减少访存操作，降低 cache 未命中率，具有很好的空间局部性；累加实验中我们使用了超标量优化算法，通过两条 CPU 流水线同时进行累加操作，将时间复杂度从 n 降低到了 log（n）量级，提高了程序性能，使用递归算法时进行优化虽然也能降低时间复杂度，但由于对内存的较大开销，反而使得性能有所降低。因此在进行程序性能优化时，我们必须同时考虑时间复杂度和空间复杂度对性能的影响。

4.2 心得与体会

通过本实验，我对 CPU 的缓存机制有了初步的了解与体会，学会了相关的 cache 优化算法，对超标量的概念有了初步的了解，也在编程与结果剖析等过程增加了对并行体系结构的理解，提高了学习并行程序的兴趣；此外，我还能更加熟练地掌握使用 Latex 撰写实验报告和绘制图表进行实验结论的阐述，学习使用 VTune 和 Perf 进行性能剖析。