南開大學
Nankai University

计算机学院

并行实验报告

并行体系结构相关编程

姓名：杨涵

学号：2213739

专业：计算机科学与技术

2024 年 3 月 24 日

目录

1 实验环境

源码链接

1.1 硬件参数

CPU 型号	12th Gen Intel(R) Core(TM) i7-12700H
CPU 核数	20
CPU 主频	2.3GHz
CPU 缓存 (L1)	1.2MB
CPU 缓存 (L2)	11.5MB
CPU 缓存 (L3)	24.0MB
内存容量	16384MB RAM

表 1: 硬件参数

1.2 Windows 系统

为了更加直观地体现出优化算法和平凡算法的差距，编译选项设定为-O0, 即不进行任何优化。

系统版本	Windows 11 家庭中文版 (23H2)
操作系统版本	22631.3296
编译器版本	TDM-GCC 4.9.2 64-bit Release
编译选项	-std=c++11 -O0

表 2: Windows 环境

2 基础要求

2.1 实验内容

2.1.1 n*n 矩阵与向量内积

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。

2.1.2 n 个数求和

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。

2.2 算法设计与编程实现

2.2.1 n*n 矩阵与向量内积

- 平凡算法

逐列访问平凡算法

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

- 优化算法

逐行访问 Cache 优化算法

```
//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
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];
    }
}
return sum;
```

相比逐列访问平凡算法，逐行访问在第二层循环中的数据迭代均是在连续内存下进行的，其访存模式与行主存储匹配，具有很好空间局部性，令 Cache 得以发挥。

2.2.2 n 个数求和

- 平凡算法: 链式

平凡算法

```
//链式:将给定元素依次累加到结果变量
for(i = 0; i < n; i++)
{
    sum += a[i];
}
return sum;
```

- 优化算法 1: 多链路式

优化算法 1: 多链路式

```
//多链路式:
sum1 = 0;
sum2 = 0;
for(i = 0; i < n; i += 2)
{
        sum1 += a[i];
        sum2 += a[i + 1];
}
sum = sum1 + sum2;
return sum;
```

- 优化算法 2:(类) 递归调用

  1. 将给定元素两两相加, 得到 N/2 个中间结果;
  2. 将上一步得到的中间结果两两相加, 得到 N/4 个中间结果;
  3. 依此类推,log(N) 个步骤后得到一个值即为最终结果。

优化算法 2:(类) 递归调用

```
//优化算法2:将递归转换为循环直接调用
for(m = n; m > 1; m >>= 1)
{
    for(i = 0; i < m >> 1; i++)
    {
        a[i] = a[i << 1] + a[(i << 1) + 1];
    }
}
return a[0];
```

2.3 性能测试与结果分析

2.3.1 n*n 矩阵与向量内积

本实验中，由于每次计算时间过短，可以采用多次重复实验取平均值的方法来取得更为准确的测量数据。

具体采用的策略为: 提前设定好总运算次数并保持不变 (e.g. Total=10^9), 不同的数据规模 N 对应的总运算次数为 N*N, T=Total/(N*N) 即为重复实验次数。测得若干次实验的总时间即可得到单次实验的平均结果。代码逻辑如下。

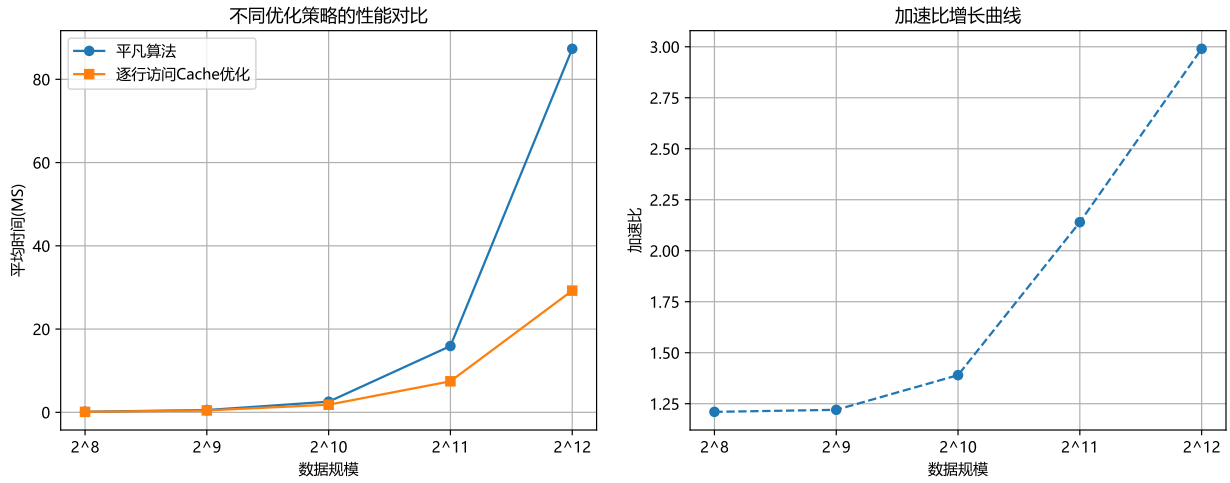
多次实验取平均值

```
//多次实验取平均值策略:维持总运算次数不变
  int n = 1 << 11; //此处数据规模为2^11
  int T = 100000000ll / (n * n);
  high_resolution_clock::time_point t1 = high_resolution_clock::now();
  for(int i = 1; i <= T; i++)
  {
```

```
    //选择不同的优化算法，分别记录平均用时
        mul_normal(n);
    //  mul_pro(n);
    //  mul_zkpro_2(n);
    //  mul_zkpro_4(n);
    //  mul_zkpro_8(n);
    }
    high_resolution_clock::time_point t2 = high_resolution_clock::now();
    duration<double,std::milli> time_span = t2 - t1;
    return time_span.count() / T;
```

优化策略\数据规模	2^8	2^9	2^{10}	2^{11}	2^{12}
平凡算法	0.1325	0.5490	2.5479	15.9271	87.3309
逐行访问 Cache 优化	0.1099	0.4497	1.8294	7.4483	29.2393
加速比	1.21	1.22	1.39	2.14	2.99

表 3: 平均时间 (MS)



(a) 不同优化策略的性能对比

(b) 加速比增长曲线

图 2.1: 算法性能测试结果

下面对结果进行分析。

- 横向分析: 对于两个算法，随着数据规模成倍增长，用时大致满足 4 倍的增长，这与运算次数的增长在直观上的是一致的。增长倍率大于 4 的原因可能在于数据增多后 cache 容量不足，访问变慢。
- 纵向分析: 在运算次数一致的情况下，可以明显看出优化算法用时更短，且随着数据规模增长二者差距越来越明显。同时，数据规模对优化算法的影响明显小于其对平凡算法的影响。原因应当也是行主存储模式对优化算法访存方式的利好。

对于测试结果背后的原因，我们将在 profiling 一节中进行进一步探讨。

2.3.2 n 个数求和

多次实验取平均值的策略与上一个实验类似，区别在于实际运算次数与数据规模相当。

优化策略/数据规模	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
平凡算法	0.1076	0.2486	0.4952	0.9727	1.9903
双链路式	0.0574	0.1306	0.2593	0.5042	1.0380
类递归调用	0.0823	0.1792	0.3514	0.7038	1.4001

表 4: 不同算法与数据规模下的性能对比

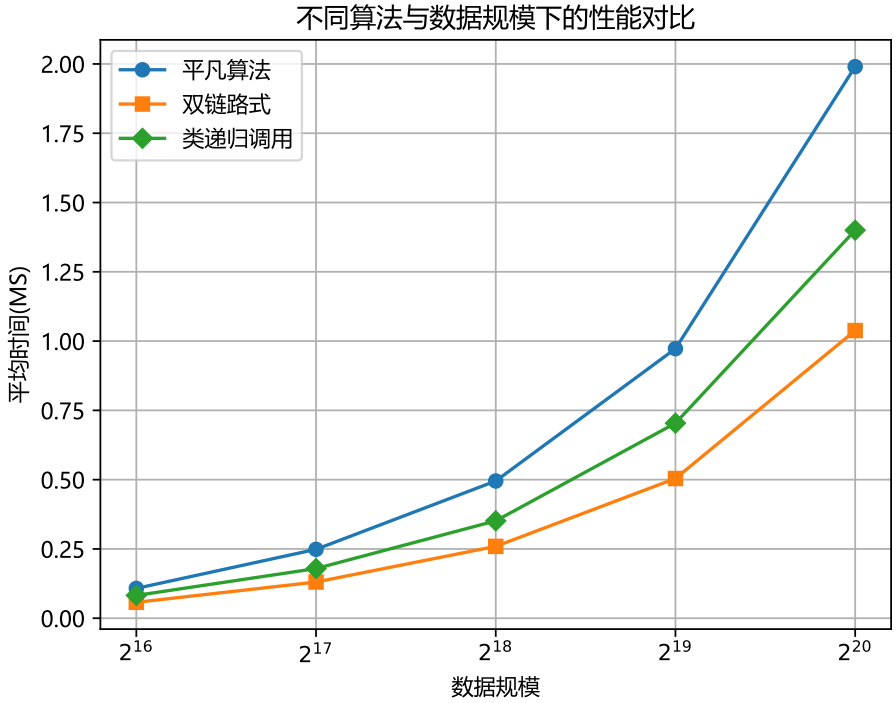


图 2.2: 不同算法与数据规模下的性能对比

下面对结果进行分析。

- 平凡算法: 排除掉实验和统计的随机性，平凡算法的运行时间对数据规模大致是满足线性增长的(注意折线图的 X 轴并不均匀)，与逻辑上的直观理解相符。
- 双链路式: 此算法利用了并行性，同时计算数组元素的相邻两个值，将计算任务分解成两个部分，从而利用了现代处理器的多核心特性或者乱序执行的能力。

  在循环中，每次迭代都会处理两个元素，减少了循环次数，降低了迭代次数，从而减少了指令级并行中的相关开销。

  在相同数据规模下，所用时间约为平凡算法的一半，并行效果良好。
- 类递归调用: 此算法实现了一种分治的思想，将相邻两个元素相加得到一个新的元素，然后不断地合并相邻的元素，最终只剩下一个元素，即数组的总和。

  通过不断地将相邻元素合并，减少了内存访问的随机性，提高了缓存的命中率，减少了内存访问的延迟，从而提高了效率。

3 进阶要求

3.1 利用 vtune 进行的 Profiling

本节的实验环境为 Windows 平台，具体配置见第一节表 2。

3.1.1 n*n 矩阵与向量内积

此实验中主要针对的分析指标为 cache 的命中率 (=Hit/(Hit+Miss)。由于绝大部分 Cache 访存都在 L1 和 L2 中进行，此处不对 L3 进行分析。使用策略仍是固定总运算次数。

优化策略\数据规模	2^8	2^9	2^{10}	2^{11}	2^{12}
Hit: 平凡算法	167	336	2360	2413	2373
Miss: 平凡算法	5	40	311	320	334
Hit: 优化算法	82	96	71	48	65
Miss: 优化算法	0	14	7	8	8

表 5: 不同算法与数据规模下的 L2:Hit/Miss 数

数据解释: 由于固定了总运算次数，即使在 L2 未得到充分利用的情况下，L2 的总访问次数仍没有太大变化。

可以看到，从2^{10} 开始，平凡算法中的 L2 开始大量应用，即使在最大测试数据规模下 L2 也仍未使用充分。

逐行访问 Cache 优化算法则始终对 L2 使用较少 (大部分在 L1 中即可完成)，可以明显看出优化算法对 Cache 利用更为充分。

优化策略\数据规模	2^8	2^9	2^{10}	2^{11}	2^{12}
L1: 平凡算法	96.91%	89.58%	66.07%	66.10%	66.42%
L1: 逐行访问 Cache 优化	98.51%	98.37%	98.78%	99.11%	98.53%
L2: 平凡算法	100%	89.36%	88.36%	88.29%	87.66%
L2: 逐行访问 Cache 优化	97.10%	87.27%	91.03%	85.71%	89.04%

表 6: 不同算法与数据规模下的 Cache 命中率

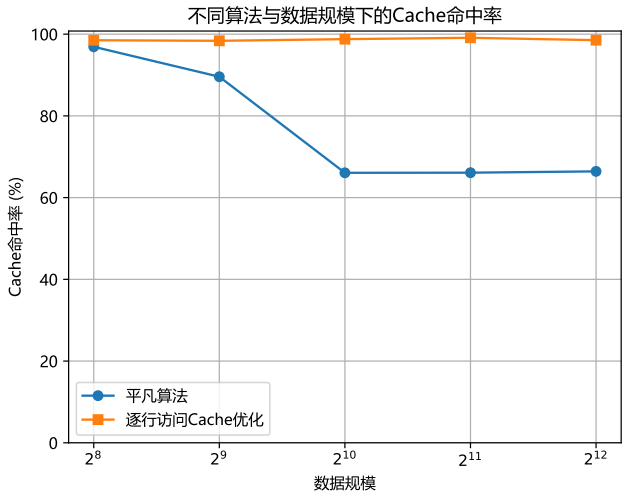


图 3.3: 不同算法与数据规模下的 Cache(L1) 命中率

数据解释: 由于 L3 始终未开始大量访存，两个算法 L2 的命中率都较高。从2^{10} 开始，平凡算法中随着 L2 开始大量应用，命中率大幅下降，此后达到稳定; 逐行访问 Cache 优化算法则始终对 L1 利用率极高。

联系前一节中，从程序运行时间增长速度来看，2^{10} 开始两个算法表现出极大差异，联系 Cache 相关指标，几乎可以认定是 L1 缓存利用率差异造成的影响。

3.1.2 n 个数求和

此实验中主要针对的分析指标为 IPC（Instruction Per Clock），即每个时钟周期执行的指令数。

优化策略/数据规模	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
平凡算法	1.4728	1.4409	1.4493	1.4368	1.4409
双链路式	2.6738	2.4876	2.5575	2.5907	2.5189
类递归调用	5.1813	5.0251	5.1282	5.1546	5.1020

表 7: 不同算法与数据规模下的 IPC 指标

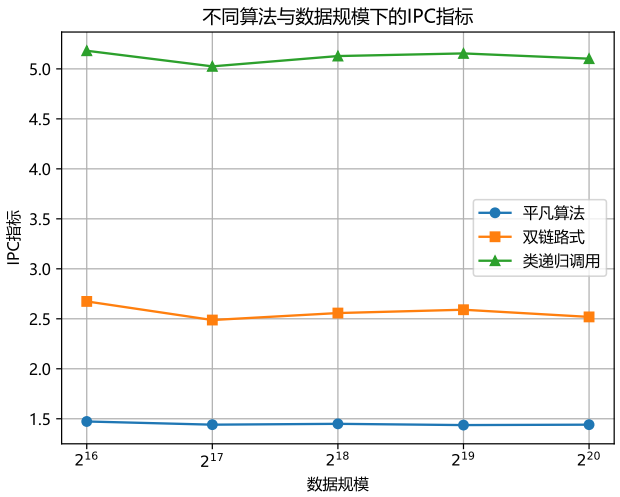


图 3.4: 不同算法与数据规模下的 IPC 指标

三个算法的 IPC 指标随着数据规模增长均相对稳定。

- 可以看到，双链路式的 IPC 明显高于平凡算法，在不同数据规模下均成 2 倍关系。这与前面算法运行效率的结果是一致的，因此可以认为双链路式的优化主要体现在指令级并行方面，即通过同时执行两条语句缩短了流水线长度。
- 类递归调用的方式中，由于内层循环不同的 i 之间的操作相互独立、可以同时计算，编译器会尝试识别和利用程序中的并行化机会，以实现同时执行不冲突的几条语句，从而提高程序的性能。从 IPC 指标也可以验证这一点。
- 类递归调用的 CPI 明显高于双链路式，优化效率却并没有双路链式高，一大可能的原因在于提高并行度的同时，频繁的访存改变了 cache 中数据的存储顺序，从而降低了 cache 利用率。

3.2 循环展开优化

我们以 n*n 矩阵与向量内积的实验为例，分析不同程度的循环展开优化对程序性能的影响。

统计策略仍为固定总运算次数。

循环展开优化

```
//以展开度k=2为例:
for(i = 0; i < n; i++)
{
    sum[i] = 0.0;
}
for(j = 0; j < n; j++)
{
        for(i = 0; i < n; i += 2)
        {
            sum[i] += b[j][i] * a[j];
            sum[i+1] += b[j][i+1] * a[j];
        }
}
return sum;
```

利用循环展开技术，可以减少循环语句执行次数，尽管无法改变总运算次数，但凭借减少循环控制开销，往往在数据规模较大时取得一定的优化效果。

展开度\数据规模	2^8	2^9	2^{10}	2^{11}	2^{12}
$k=1$ (普通优化)	0.1057	0.4291	1.6829	6.9997	28.2393
$k=2$	0.1043	0.4171	1.6840	6.7867	27.0294
$k=4$	0.1042	0.4276	1.6797	6.7600	27.1293
$k=8$	0.1066	0.4320	1.7078	6.8129	27.9052

表 8: 不同程度循环展开的平均用时

然而，不同展开度的循环展开算法比起普通优化的提升都微乎其微，甚至没有优化效果。

使用 vtune 对它们分别进行 profiling 之后，发现越高的展开度下 cache 的 hit 数越低 (考虑 i 贡献的 hit 数因循环次数减少而减少)，miss 数却未发生显著变化，命中率有不小幅度的降低。可能的原因是提高并行度的同时，过于频繁的数据访存改变了缓存记录的顺序，降低了 cache 的利用率。

4 实验总结

Martix 实验中，Cache 逐行访问优化算法相对平凡算法的优化力度 (从结果上) 与 Cache 命中率的差异一致，可以认为优化主要体现在对 Cache 的利用率上；循环展开算法并没有取得预期之中的效果，可能与提高并行度、减少了对 Cache 利用率有关。

求和运算实验中。双链路优化算法相对平凡算法的优化力度 (从结果上) 与 IPC 的差异一致，可以认为优化主要体现在并行度上；类递归优化算法的 IPC 指标更高、并行度更高，优化效率却不如双链路算法，同样可能与提高并行度、减少了对 Cache 利用率有关。

程序运行效率是多因素的综合体，当影响程序运行效率的因素出现冲突时要作出适当的取舍，对于不同的数据规模要选择合适的算法。