计算机学院

并行程序设计实验报告

体系结构相关实验分析

姓名：刘家骥

学号：2211437

专业：计算机科学与技术

2024 年 3 月 27 日

目录

Abstract

实验一：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。

实验二：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。

1 实验环境

- CPU：12th Gen Intel(R)Core(TM)i7-12700H
- 指令集架构：x86 架构
- GPU：NVIDIA GeForce RTX 3060 Laptop GPU GDDR6 @ 6GB (192 bits)
- 内存容量：16GB
- 操作系统及版本：Windows 11 家庭中文版
- 编译器及版本：GNU GCC Compiler
- 编译选项：Have g++ follow the C++17 GNU C++ language standard(ISO)

  Optimize even more(for speed) [-O2]

2 实验一：n*n 矩阵与向量内积

2.1 算法设计

2.1.1 平凡算法

矩阵的每一列依次与向量相乘，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。

由于当 n 较小时，程序运行速度过快，导致时间测量不精准。所以将代码关键部分循环 times 次，最后算出平均时间，以减小误差。

逐列访问平凡算法

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

2.1.2 cache 优化算法

相较于平凡算法，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。与平凡算法相比，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。

除算法部分代码外，其余部分与上述代码基本相同，cache 优化算法代码部分如下

cache 优化算法

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

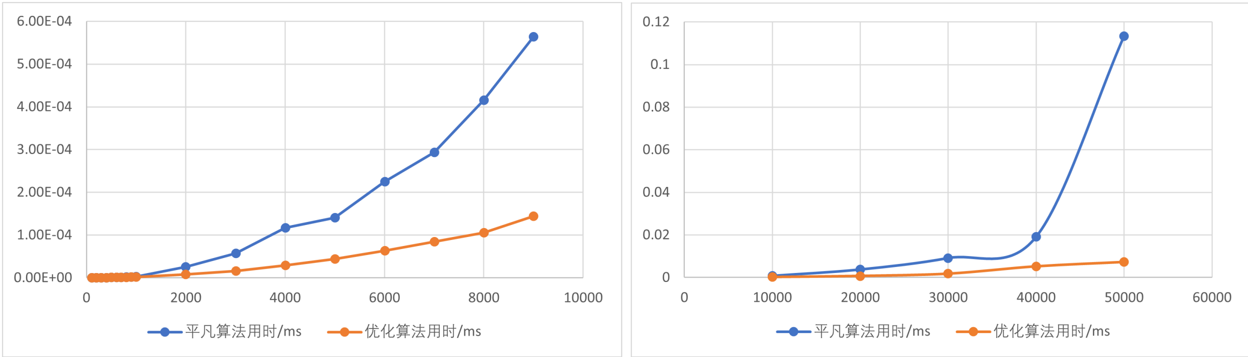
2.2 算法的性能比较和分析

通过测量算法核心步骤的运行时间，来比较两种算法的性能，运行时间越短说明算法性能越好

问题规模 n	平凡算法用时/ms	优化算法用时/ms
100	1.93E-08	1.93E-08
200	8.85E-08	7.88E-08
300	1.79E-07	1.71E-07
400	3.85E-07	3.64E-07
500	5.59E-07	4.31E-07
600	7.42E-07	6.49E-07
700	1.08E-06	8.46E-07
800	1.44E-06	1.11E-06
900	1.85E-06	1.39E-06
1000	2.31E-06	1.70E-06
2000	2.51E-05	7.65E-06
3000	5.71E-05	1.61E-05
4000	0.000116531	2.89E-05
5000	0.000140168	4.38E-05
6000	0.000225069	6.32E-05
7000	0.000292939	8.45E-05
8000	0.000415705	0.000105711
9000	0.000564145	0.000143588
10000	0.000731075	0.00017372
20000	0.00372609	0.000685627
30000	0.00910936	0.00176622
40000	0.019055	0.00526996
50000	0.113282	0.00734839

如图2.1所示，通过 excel 画出折线图，我们可以更加直观的看出两种算法运行的时间差异，以及随着问题规模的变化，从两张图我们可以看出，随着问题规模 n 增大，两种算法运行时间差异越来越大。

如图2.2所示，由于时间差异的数量级过大，所以引入优化力度这一参数，其中优化力度 = 平凡算法执行时间/优化算法执行时间，显然优化力度越大，算法优化效果越好，随着问题规模增加，优化力度有明显的增大趋势。



(a) 小规模矩阵内积运算时间对比

(b) 大规模矩阵内积运算时间对比

图 2.1: 不同并行优化算法的执行时间与准确率对比

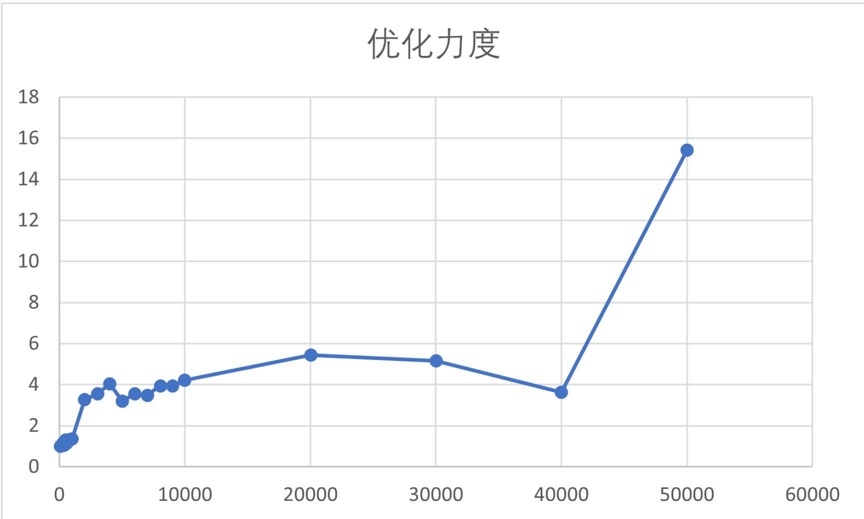


图 2.2: 优化力度

2.3 汇编分析

c++ 代码中算法关键部分的汇编代码如下

逐列访问平凡算法

```
.L23:
        mov     eax, DWORD PTR [rbp-48]
        cdqe
        lea     rdx, [0+rax*4]
        mov     rax, QWORD PTR [rbp-72]
        add     rax, rdx
        mov     ecx, DWORD PTR [rax]
        mov     eax, DWORD PTR [rbp-44]
        cdqe
        lea     rdx, [0+rax*8]
        mov     rax, QWORD PTR [rbp-56]
        add     rax, rdx
```

```
        mov     rax, QWORD PTR [rax]
        mov     edx, DWORD PTR [rbp-48]
        movsx   rdx, edx
        sal     rdx, 2
        add     rax, rdx
        mov     edx, DWORD PTR [rax]
        mov     eax, DWORD PTR [rbp-44]
        cdqe
        lea     rsi, [0+rax*4]
        mov     rax, QWORD PTR [rbp-64]
        add     rax, rsi
        mov     eax, DWORD PTR [rax]
        imul    edx, eax
        mov     eax, DWORD PTR [rbp-48]
        cdqe
        lea     rsi, [0+rax*4]
        mov     rax, QWORD PTR [rbp-72]
        add     rax, rsi
        add     edx, ecx
        mov     DWORD PTR [rax], edx
        add     DWORD PTR [rbp-48], 1
```

在指令 mov eax, DWORD PTR [rbp-48] 和 mov eax, DWORD PTR [rbp-44] 中，从内存中加载数据到 eax 寄存器。这里的 [rbp-48] 和 [rbp-44] 分别表示矩阵和向量中的某个元素的内存位置。

矩阵的每一行都是连续存储的，而向量也是连续存储的。这样的数据布局有助于利用缓存的空间局部性，此时相邻的数据项会被同时加载到缓存中。因此，cache 优化代码的数据访问模式是按行访问，能够充分利用缓存的空间局部性，在数据读取上可以节省很多时间。

矩阵 b 的每一行和向量 a 都是连续存储的。在乘积运算过程中，连续访问矩阵 b 的每一行和向量 a 的元素，使得它们很可能在同一个缓存行中，从而提高了缓存的利用率，充分利用缓存，加快了运算速度。

3 实验二：n 个数求和

3.1 算法设计

3.1.1 平凡算法

由于递归算法的问题规模 n 均需要为 2 的指数幂，所以在这几种算法的比较中，问题规模 n 都取 2 的指数幂，且当 n 较小时程序运行过快导致时间难以测量，所以我们从 n=1024 开始测试。

平凡算法

```
for(n=1024;n<=536870912;n*=2)
{
    int* a=new int[n];
    for(int i=0;i<n;i++)
    {
        a[i]=i;
```

```
        }
        long long total_time=0;
        for(int num=0;num<times;num++)
        {
            QueryPerformanceCounter((LARGE_INTEGER *)&head);
            for(int i=0;i<n;i++)
            {
                sum+=a[i];
            }
            QueryPerformanceCounter((LARGE_INTEGER *)&tail);
            total_time+=(tail-head);
        }
        cout<< (total_time)/times * 1000.0 / freq<<endl;
    }
```

3.1.2 多链路式算法

按照奇偶索引分成两部分，并分别计算这两部分的和。通过在循环中每次处理两个连续的元素，其中一个是偶数索引处的元素，另一个是奇数索引处的元素，然后将它们分别加到两个变量 sum1 和 sum2 中。

这样能够有效地利用循环，并且将数组元素的访问与加法操作结合在一起，每次迭代都同时处理了两个元素，减少了迭代次数。

多链路式算法

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

3.1.3 递归算法

通过递归算法，每次递归都将数组的规模减半，直到数组只有一项时停止递归。

递归算法

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

```
    }
}
```

3.1.4 双重循环算法

通过不断地合并相邻的元素，将数组中的元素逐步减半，最终 a[0] 存储整个数组的和。

双重循环算法

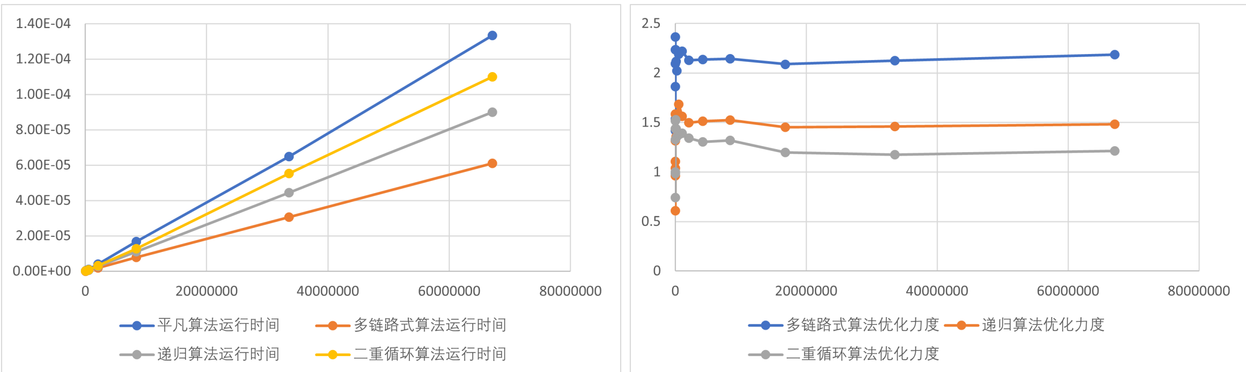
```
    for(int m=n;m>1;m/=2)
        for(int i=0;i<m/2;i++)
            a[i]=a[i*2]+a[i*2+1];
```

3.2 算法的性能比较和分析

从问题规模 n=32768 开始记录，得到如表??所示数据，并绘制折线图，观察时间变化趋势，以及各算法运行时间的差异。

问题规模 n	32768	131072	524288	2097152	8388608	33554432	67108864
平凡算法运行时间/ms	6.62E-08	2.69E-07	1.06E-06	4.05E-06	1.68E-05	6.48E-05	0.0001333
多链路式算法运行时间/ms	2.96E-08	1.27E-07	4.85E-07	1.90E-06	7.82E-06	3.05E-05	6.10E-05
递归算法运行时间/ms	4.18E-08	1.98E-07	6.31E-07	2.70E-06	1.10E-05	4.45E-05	8.99E-05
二重循环算法运行时间/ms	4.60E-08	1.87E-07	7.73E-07	3.02E-06	1.27E-05	5.52E-05	0.0001099
多链路式算法优化力度	2.234158	2.11611	2.187485	2.128292	2.143662	2.1260339	2.1844239
递归算法优化力度	1.585196	1.356067	1.682215	1.498416	1.525231	1.4580083	1.4828213
递归算法优化力度	1.438564	1.437637	1.374487	1.340518	1.317925	1.1748833	1.2122681

如图3(a)所示，三种算法在计算上速度都有所增加，运行速度最快的算法是多链路式算法，其次是递归算法，最后是二重循环算法。从优化力度上来看，三种算法均随着 n 增大而趋于稳定。



(a) 求和各算法运行时间比较

(b) 求和各算法优化力度

图 3.3: 不同算法的性能对比

3.3 汇编分析

循环部分涉及到频繁的内存访问。然而，由于数组 a 中的元素存储模式是连续的内存块，所以访问也是连续的，利用缓存机制来提高运算速度。缓存机制会将最近访问的内存块保存在高速缓存中，以

便下一次访问时可以更快地获取数据，依次提高运算速度。

多链路式算法

```
mov     eax, DWORD PTR [rbp-16]  ;sum1+=a[i]
        cdqe
        lea     rdx, [0+rax*4]
        mov     rax, QWORD PTR [rbp-32]
        add     rax, rdx
        mov     eax, DWORD PTR [rax]
        add     DWORD PTR [rbp-8], eax

        mov     eax, DWORD PTR [rbp-16]  ;sum2+=a[i+1]
        cdqe
        add     rax, 1
        lea     rdx, [0+rax*4]
        mov     rax, QWORD PTR [rbp-32]
        add     rax, rdx
        mov     eax, DWORD PTR [rax]
        add     DWORD PTR [rbp-12], eax
        add     DWORD PTR [rbp-16], 2
```

双重循环算法

```
    mov     eax, DWORD PTR [rbp-16]
        add     eax, eax
        cdqe
        lea     rdx, [0+rax*4]
        mov     rax, QWORD PTR [rbp-32]
        add     rax, rdx
        mov     ecx, DWORD PTR [rax]
        mov     eax, DWORD PTR [rbp-16]
        add     eax, eax
        cdqe
        add     rax, 1
        lea     rdx, [0+rax*4]
        mov     rax, QWORD PTR [rbp-32]
        add     rax, rdx
        mov     edx, DWORD PTR [rax]
        mov     eax, DWORD PTR [rbp-16]
        cdqe
        lea     rsi, [0+rax*4]
        mov     rax, QWORD PTR [rbp-32]
        add     rax, rsi
        add     edx, ecx
        mov     DWORD PTR [rax], edx
        add     DWORD PTR [rbp-16], 1
```

源代码 github： https://github.com/RRRRReus/Parallel-Programming/tree/main/实验一