南開大學
Nankai University

计算机学院

并行程序设计实验报告

并行体系结构编程实验报告

姓名：文昱韦

学号：2213125

专业：计算机科学与技术

2024 年 3 月 25 日

目录

1 算法设计

1.1 n*n 矩阵与向量内积

平凡算法是遍历 n*n 矩阵的每一列，当遍历第 i 列时，将该列与给定向量的第 i 个数进行内积运算，得到的结果即是结果向量中的第 i 个数，每一次循环即可得到一个内积结果。

cache 优化算法改变了循环的顺序，使得内存访问更加连续。具体来说，我们可以将针对矩阵列的循环改为遍历矩阵的每一行，因为矩阵中的数据是一行一行连续地储存在内存中的，当遍历第 i 行时，将该行与给定向量进行内积运算，虽然一次计算得不到任何一个最终结果，但有利于缓存的有效利用。

1.2 n 个数累加求和

对于给定的 n 个数，平凡算法就是采用简单的累加方式求和。cache 优化算法主要是采用最简单的两路链式累加，也成循环展开的技术，将多个数相加的操作合并到一次循环中，这样可以减少循环迭代次数，从而减少了缓存未命中的次数。还有一种方法是通过双重循环将给定的元素两两相加，得到 n/2 个数，再两两相加，直到最后只剩 1 个数。

2 程序设计

以下列出每个算法的核心代码,完整的代码可以见 github 仓库https://github.com/YuweiWen1217/2024--/tree/main/ 1。

2.1 n*n 矩阵与向量内积：平凡算法

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

2.2 n*n 矩阵与向量内积：cache 优化算法

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

2.3 n 个数累加求和：平凡算法

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

2.4 n 个数累加求和：优化算法 1

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

2.5 n 个数累加求和：优化算法 2

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

2.6 测试方法

测试主要方法是测量核心运算程序的执行时间，由于这几个运算程序相对简单，特别是在文体规模 n 较小的时候，因此我们将这几段程序重复运行 **1000 遍**，然后计算总时间。此外，为了结果的普遍性，每个测试都应该做 10 遍，然后取均值。

```
long long head, tail, freq;
//第一个问题 n 的规模可以取 10, 20, 30……100, 200; 第二个问题 n 的规模取 2 的幂。
cin >> n;
//矩阵和向量的初始化 (a[i] = 2 * i; b[i][j] = i + j;)
//n 个数的初始化 (a[i] = 2 * i)
//以下为测试 1 遍的过程。
QueryPerformanceFrequency((LARGE_INTEGER *)&freq);
QueryPerformanceCounter((LARGE_INTEGER *)&head);
for(int count = 0; count < 1000; count++)
{
    //program
}
QueryPerformanceCounter ((LARGE_INTEGER *)& tail);
cout << " 测试时间是" <<(tail-head)*1000.0 / freq<< "ms" << endl;
```

3 测试结果与分析

3.1 n*n 矩阵与向量内积

本实验是在 x86 平台、Windows 系统、GCC 8.1.0、Code::Blocks 环境下进行的，本台计算设备 CPU12 核心 16 线程，缓存大小 18M。

对于每个问题规模，都进行了 10 次实验，取平均值后的结果如表1所示。图3.1显示了在不同的问题规模下，平凡算法和 cache 优化算法的运行时间差异。图3.2显示了在不同的问题规模下，优化算法相对于平凡算法提升的时间百分比，计算公式是 $\frac{cache优化算法所用时间-平凡算法所用时间}{平凡算法所用时间} \times 100\%$。

结果分析

- 通过表1和图3.1可以看出，随着问题规模 n 的增长，所用时间成 n^2 规模增长，因此 n*n 矩阵与向量内积这个运算的时间复杂度是 $O(n^2)$。
- 通过表1和图3.1可以看出，cache 优化算法与平凡算法相比，运行时间缩短，速度大幅提升，说明矩阵中的数据一行一行地连续储存在内存中，且由于缓存空间局部性的特点，相邻数据会被一起读取到缓存中，从缓存中读取数据快于从内存中读取，因此，优化算法的按行遍历要快于平凡算法的按列遍历。
- 通过表1和图3.2可以看出，cache 优化算法所缩短的时间，在问题规模足够大后，大致在 26% 左右。
- 在表1和图3.2中，有一个特殊情况，那就是当问题规模 n=40 时，平凡算法的时间短于优化算法的时间，我认为原因可能有两个：一是偶然因素，虽然最后得到的结果是运行 10 次后取得平均值，但 10 依旧是一个比较小的数，因此可以通过增大运行次数来判断这个原因的正确与否；二是在较小的数据集上，缓存的影响可能不如在大型数据集上显著。当问题规模较小时，缓存优化的好处可能会被掩盖，甚至由于额外的计算而导致性能下降。

问题规模 n	平凡算法	cache 优化算法	提升时间比/%
1	0.00383	0.00382	0.26
10	0.30018	0.27731	7.62
20	1.15513	1.10612	4.24
30	2.33963	1.78236	23.82
40	2.73763	2.96266	-8.22
50	4.69616	3.78156	19.48
60	6.5795	5.29337	19.55
70	8.26944	6.66288	19.43
80	10.98628	8.40937	23.46
90	14.38529	11.34216	21.15
100	17.18519	12.9497	24.65
200	70.41087	51.7175	26.55
300	147.9124	113.8159	23.05
400	270.6061	198.9808	26.47
500	418.1363	311.7628	25.44
600	611.4401	451.1777	26.21
700	835.3408	610.581	26.91

表 1: n*n 矩阵与向量内积性能测试结果 (单位:ms)

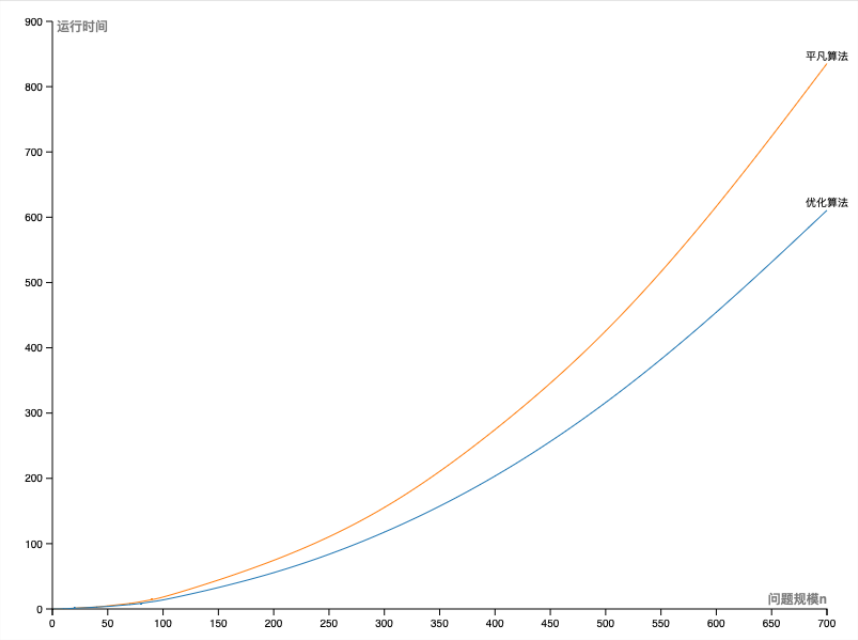

图 3.1: n*n 矩阵与向量内积实验平凡算法与优化算法性能对比

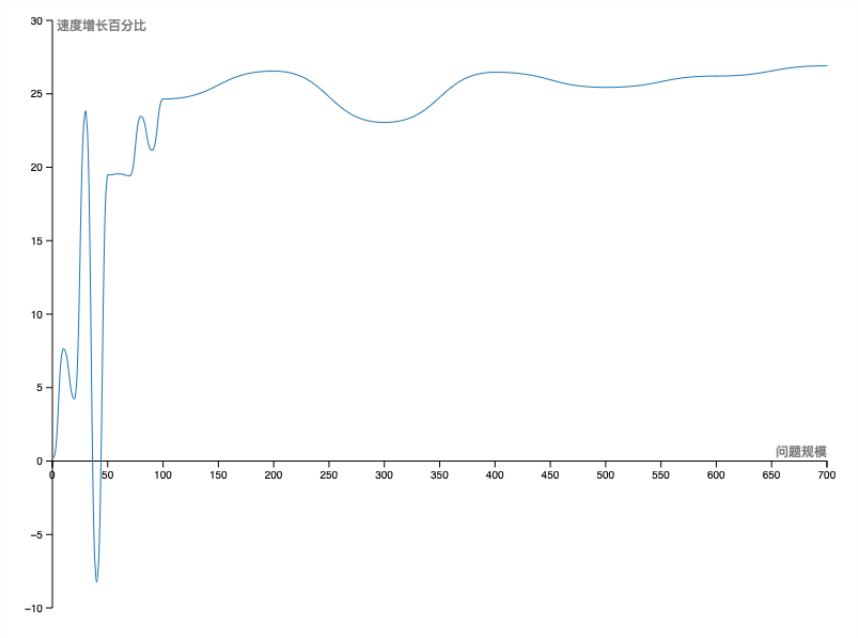

图 3.2: 优化算法时间提升时间比

3.2 n 个数累加求和

本实验是在 x86 平台、Windows 系统、GCC 8.1.0、Code::Blocks 环境下进行的，本台计算设备 CPU12 核心 16 线程，缓存大小 18M。

问题规模按照 2 的幂增长，对于每个问题规模，都进行了 10 次实验，然后取平均值，结果填在了表2中。图3.3是对三种算法的可视化分析。

结果分析

- 通过表2和图3.3可以看出，随着问题规模 n 的增长，平凡算法所用时间成 n 规模增长，此时 n 个数累加求和时间复杂度是 $O(n)$。结合代码可知，二重循环递归算法外层循环需要进行 $\log_2 n$ 次，内层循环最多进行 $\frac{n}{2}$ 次，时间复杂度是 $O(n\log n)$，两路链式算法只有一层循环，会进行 $\frac{n}{2}$ 次，时间复杂度是 $O(n)$。

- 通过表2和图3.3可以看出，两种优化算法与平凡算法相比，运行时间均有所缩短，速度提升，说明减少循环次数有利于程序进行。

- 对于二层循环递归算法，内层循环中的 a[2 * i] 和 a[2 * i + 1] 对内存的访问是连续的，这有利于缓存的命中率。在现代计算机体系结构中，局部性对于性能至关重要。由于连续的内存访问模式，缓存能更有效地预取和重用数据，从而提高了算法的速度。

- 二路链式算法中的循环每次处理两个元素，而不是一个。这允许并行处理，因为每次迭代都是独立的，不需要等待上一次迭代的结果。现代计算机通常具有多核心处理器，可以同时处理多个任务。因此，通过并行处理，可以在相同的时间内处理更多的数据。

- **减少指令级别并发冲突**，二路链式算法中的循环结构允许更多的指令级并发，因为每次迭代都有两个独立的内存访问（a[i] 和 a[i + 1]）。这意味着在现代超标量处理器上，可以更有效地利用指令级并行性，从而提高了性能。

- 经计算，二路链式算法性能提高了大约 60%，且规模较小时，优势更明显（达到了 70% 以上)。二重循环递归算法大约提高了 45%。

问题规模 n	平凡算法	两路链式算法	二重循环递归算法
2	0.00535	0.0016	0.00279
4	0.01069	0.00367	0.00636
8	0.02307	0.00632	0.03306
16	0.04896	0.01141	0.02636
32	0.09792	0.03562	0.04808
64	0.19478	0.07396	0.09628
128	0.38227	0.11339	0.19946
256	0.77463	0.2321	0.44898
512	1.41657	0.48672	0.63483
1024	2.26661	0.88361	1.20803
2048	5.72154	1.71818	2.24488
4096	7.48125	3.40559	4.49192
8192	14.59605	7.93437	8.73163
16384	29.44366	14.19552	17.27545
32768	57.90445	27.64619	34.79778
65536	115.3954	55.19948	68.44821
131072	230.2534	108.8375	135.0837

表 2: n 个数累加求和测试结果 (单位:ms)

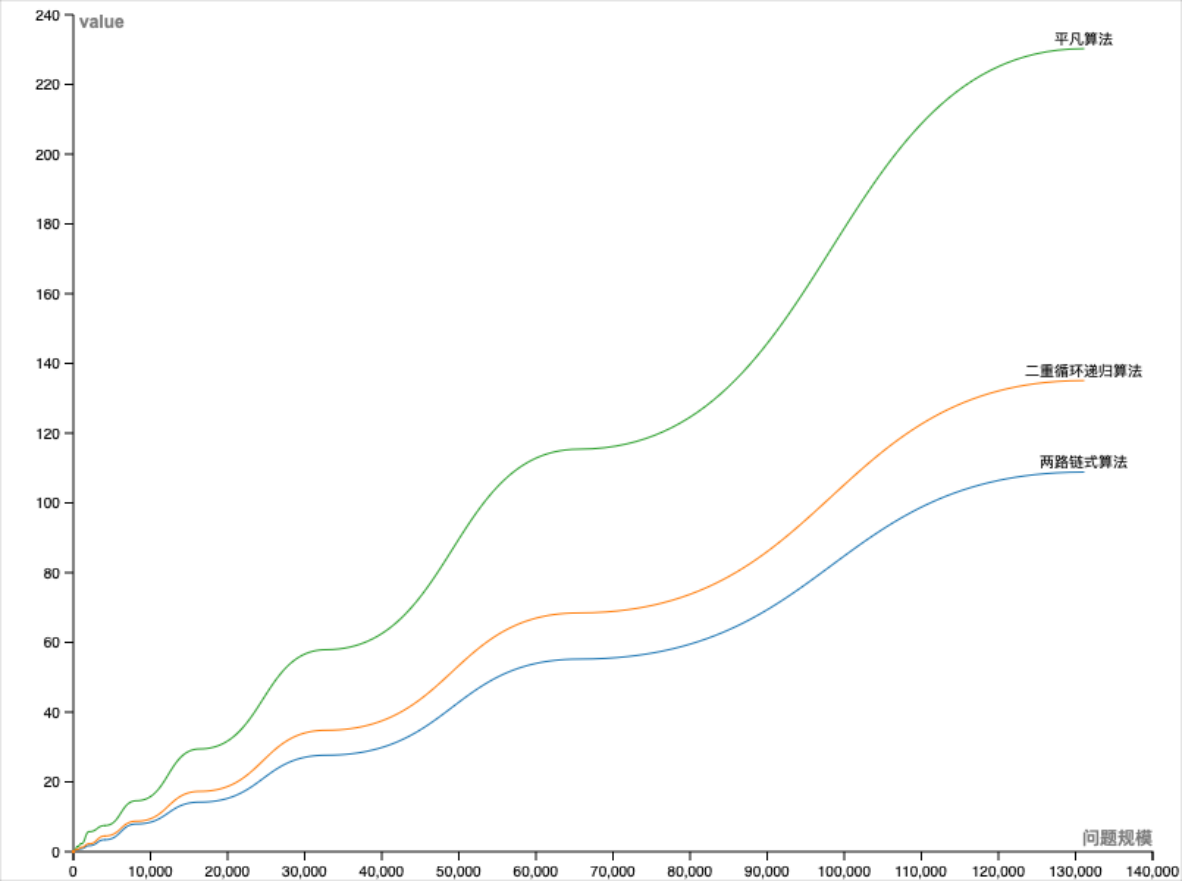

图 3.3: 三种算法性能比较

4 进阶：循环展开技术（以 n 个数累加求和为例）

循环展开（Loop Unrolling）是一种优化技术，用于减少循环的迭代次数，从而减少循环的开销，提高程序的性能。循环展开的基本思想是将循环体中的多次迭代展开成多个重复的代码块，使得每次迭代执行的指令数增加，从而减少了循环控制和分支操作的开销。

通过循环展开，可以实现减少循环开销、利用指令级并行（循环展开增加了每次迭代执行的指令数，可以更好地利用处理器的指令级并行性，提高计算效率）、增加局部性等优点。

之前算法中两路链式算法就是一种循环展开，他把 n 次循环减少为了 $\frac{n}{2}$ 次。另一种更灵活的方法是使用宏/模板技术来彻底消除循环。这种方法利用了元编程的特性，在编译时生成多次展开的代码，从而实现了循环展开的效果，而不需要显式地编写多个循环体。通过使用宏/模板技术，我们可以在编写代码时指定展开的次数，使得代码更加灵活和可维护。

这里，我们将之前运用到的二路链式展开与四路链式展开、八路链式展开的结果作对比和分析，结果如表3。下面给出的代码是四路链式展开。

```
1  for (i = 0; i < n - 3; i += 4)
2  {
3      result += a[i] + a[i + 1] + a[i + 2] + a[i + 3];
4  }
```

或

```
for (int i = 0; i < n; i += 2)
{
    sum1 += a[i];
    sum2 += a[i + 1];
    sum3 += a[i + 2];
    sum4 += a[i + 3];
}
sum = sum1 + sum2 + sum3 + sum4;
```

问题规模 n	二路链式展开	四路链式展开	八路链式展开
16	0.01141	0.00995	0.00919
32	0.03562	0.01729	0.01606
64	0.07396	0.03507	0.0269
128	0.11339	0.06171	0.05269
256	0.2321	0.12045	0.10197
512	0.48672	0.23689	0.19332
1024	0.88361	0.46509	0.4381
2048	1.71818	0.95781	0.76484
4096	3.40559	2.04754	1.72892
8192	7.93437	3.6762	3.15795
16384	14.19552	7.47502	6.17486
32768	27.64619	16.2076	13.9072
65536	55.19948	31.5434	26.5607
131072	108.8375	64.359	55.5784

表 3: 性能测试结果 (单位:ms)

由以上结果可知，程序运行的性能随着路数的增多而提高，说明减少循环操作次数、即减少判定归纳变量是否满足条件、变量递增/递减等操作，可以提升性能。多路链式展开减少了循环开销，并在一定程度上利用并行性，使得处理器能够更好地利用其多个执行单元进行并行计算。这有助于提高程序的并行度和整体性能。同时还增加了循环体内的指令数量，有利于提高指令级并行度。通过连续地访问数组元素，可以利用处理器缓存系统的预取机制，减少缓存未命中率，提高内存访问效率。但我们需要注意的是，多路链式展开并不适用于所有情况，比如对于循环体内的代码量较小或者存在复杂的数据依赖关系的情况下，多路链式展开可能并不能带来明显的性能提升，甚至可能会降低程序的性能。

5 进阶：Linux 环境下的程序性能（以 n*n 矩阵与向量内积运算为例）

在 x86 平台、Windows 系统、GCC 编译器环境下，运用 WSL 提供的 Linux 环境，运行 n*n 矩阵与向量内积运算的两种算法的代码，并作对比和分析。

在此之前，我们需要对之前程序所使用的计时工具进行修改，因为 WSL 里无法编译 windows.h 头文件，就不能使用 QueryPerformanceCounter 来获取时间。这里我们改用 sys/time.h 中的 gettimeofday 功能，修改后的部分程序如下：

数据规模	平凡算法		优化算法	
	Windows	Linux(WSL)	Windows	Linux(WSL)
10	0.30018	0.3	0.27731	0.3
20	1.15513	1.3	1.10612	1.1
30	2.33963	2.2	1.78236	1.8
40	2.73763	3.5	2.96266	2.5
50	4.69616	5.1	3.78156	4
60	6.5795	6.7	5.29337	5.3
70	8.26944	8.7	6.66288	7
80	10.98628	11.2	8.40937	8.8
90	14.38529	14.7	11.34216	11.4
100	17.18519	17.4	12.9497	13.4
200	70.41087	66.6	51.7175	51.3
300	147.9124	148.8	113.8159	112.2
400	270.6061	268.4	198.9808	198.6
500	418.1363	422.1	311.7628	311.7
600	611.4401	617.8	451.1777	443.6
700	835.3408	838.4	610.581	607.7

表 4: 性能测试结果 (单位:ms)

```cpp
#include <sys/time.h>
int main()
{
    int n;
    cin >> n;
    int sum[n], b[n][n], a[n];
    auto sumtim = 0.0;
    timeval currentTime;
    //以下为进行 1 次实验的过程，矩阵和向量的初始化过程已略
    gettimeofday(&currentTime, NULL);
    //将时间单位统一为毫秒
    long long startTime = currentTime.tv_sec * 1000 + currentTime.tv_usec / 1000;
    //此处为需要计时的程序
    gettimeofday(&currentTime, NULL);
    long long endTime = currentTime.tv_sec * 1000 + currentTime.tv_usec / 1000;
    sumtim = endTime - startTime;
    cout << sumtim << endl;
}
```

结果如表4所示，虽然在时间测量方面，gettimeofday 的精度没有 QueryPerformanceCounter 高，但我们依旧可以看出 Linux 系统下的性能与在 Windows 系统下的性能差异不大，算法的优化效果在两个系统下也表现出了大致相同的性能。