南開大學
Nankai University

计算机学院

并行程序设计期末报告

CPU 架构相关编程

姓名：郭家琪

学号：2311781

专业：计算机科学与技术

2025 年 3 月 30 日

目录

1 矩阵与定向量的内积

计算给定 n*n 矩阵的每一列与给定向量的内积。对于每一列 j，矩阵 Mat 与向量 Vec 计算如下：

$$\text{result}[j] = \sum_{i=0}^{n-1} \text{Mat}[i][j] \times \text{Vec}[i], \quad \text{对于每一列} j$$

1.1 平凡算法

1.1.1 设计思路

嵌套循环逐列计算，外层循环遍历矩阵的每一列 j，内层循环计算矩阵第 j 列和向量的内积。

缓存命中率低，内存访问模式为非连续访问，违背了空间局部性原则，频繁的缓存未命中会显著增加内存延迟。**性能差**，两层嵌套循环的时间复杂度为 $O(n^2)$，时间复杂度高。

1.1.2 代码

```
int n = matrix.size();
for(int i=0;i<n;i++)
{
    result[i] = 0.0;
    for (int j = 0; j < n; j++)
    {
        result[i] += matrix[j][i] * vec[j];
    }
}
```

1.2 优化算法

1.2.1 设计思路

采用逐行访问矩阵元素的方式，外层循环遍历矩阵的每一行 i，内层循环遍历每一列 j，执行矩阵与向量的乘法，并累加到对应列的结果向量中。

Cache 优化关键在于利用 CPU 缓存行的特性，将矩阵按行遍历而不是按列，从而提高算法效率。

1.2.2 代码

```
int n = matrix.size();
for (int i = 0; i < n; i++)
    result[i] = 0.0;
for (int j = 0; j < n; j++) {
    for (int i = 0; i < n; i++) {
        result[i] += matrix[i][j] * vec[j];
```

```
7         }
8     }
```

1.3 性能对比

为确保实验的准确性，下面各个算法的执行时间为五次同一网络下统一数据测试的平均值，避免偶然误差。

Algo\n	100	1000	5000	10000
平凡	0.2538	30.7263	956.257	4609.09
优化	0.2404	29.0568	913.473	4298.28
加速比	1.056x	1.057x	1.047x	1.072x

表 1: 性能测试结果 (矩阵与向量内积)(单位:ms)

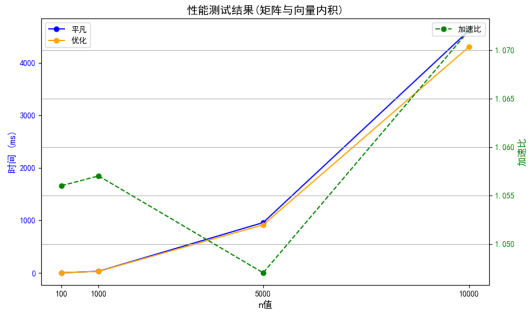

图 1.1: 性能测试结果 (矩阵与向量内积)

优化算法全面优于平凡算法，所有规模的运行时间均更短，随着规模增大，加速比略微提升至1.072，说明优化对大规模数据更有效。

性能提升主要来自缓存优化，优化算法通过逐行访问（matrix[i][j]）提高了缓存命中率，减少了内存延迟。平凡算法的逐列访问（matrix[j][i]）导致缓存未命中频繁，尤其在数据规模大时更明显。

2 n 个数求和

计算 n 个数的和。假设 n 个数为 $x_1, x_2, x_3, \ldots, x_n$，求和公式为:

$$S = \sum_{i=1}^{n} x_i$$

2.1 平凡算法

2.1.1 设计思路

使用简单的遍历 data 向量，并累加每个元素到 sum 中，指令顺序进行，对于大规模数据效率低。线性累积无法有效利用现代处理器的并行计算能力，若为浮点数计算，小的数值可能会被大的数值淹没，导致精度损失。

2.1.2 代码

```
vector<int> data(n);
for (int i = 0; i < n; i++) {
    data[i]=i+1;
}
int sum = 0;
for (int i = 0; i < n; ++i) {
    sum += data[i];
}
```

2.2 多路链式

2.2.1 设计思路

使用双累加器，以累加两个元素的方式进行求和。此方法可以利用**并行化**或优化的指令集，在某些架构上提高性能，减少数据依赖带来的停顿周期。**减少循环控制开销**，循环次数减半，每次迭代完成两个加法操作，分支预测失败率降低 50%。**易于扩展性**，可推广到 4 路、8 路等更多并行路径，适合 SIMD 向量化扩展。

缺点：两个并行累加器都可能溢出，需要更复杂的溢出检测逻辑；对非连续内存访问效果有限，二维数组需要特殊处理访问模式，某些编译器可能无法自动向量化这种模式，需要手动确保内存对齐。

2.2.2 代码

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

2.3 递归算法

2.3.1 设计思路

将给定元素两两相加，得到 n/2 个中间结果，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。每轮操作量呈几何级数下降，每轮循环内部的加法操作完全独立，原地操作，不需要额外存储空间。

缺点：递归函数调用随 n 增大而增加，可能导致栈溢出；非连续内存访问，缓存命中率低于顺序访问；边界条件处理容易出错（如奇数长度处理）；并行效率限制，后期轮次的操作量急剧减少，可能出现负载不均衡。

2.3.2 代码

```
void recursion(vector<int>& data, int n) {
    if (n == 1)
        return;
    else {
        for (int i = 0; i < n / 2; i++)
            data[i] += data[n - i - 1];
        n = n / 2;
        recursion(data, n);
    }
}
```

2.4 性能对比

为确保实验的准确性，下面各个算法的执行时间为五次同一网络下统一数据测试的平均值，避免偶然误差。

Algo\n	2^{10}	2^{15}	2^{20}	2^{25}
平凡	0.0116	0.2567	8.2881	336.141
多路链式	0.0075	0.2124	7.6912	236.712
递归算法	0.0128	0.4022	14.1932	469.783

表 2: 性能测试结果 (n 个数求和)(单位:ms)

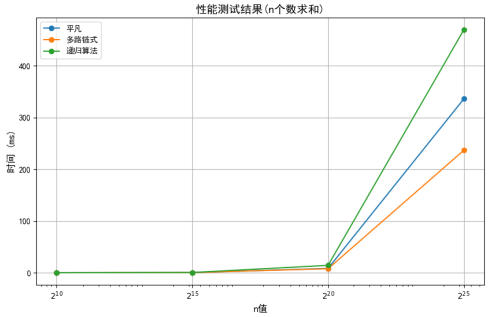

图 2.2: 性能测试结果 (n 个数求和)

平凡算法： 在 2^{10} 和 2^{15} 规模下表现中等，但随规模增大（2^{20} 和 2^{25}）时性能显著优于递归算法，仅次于多路链式算法，可能是基于简单迭代的实现，时间复杂度为 O(n)，常数因子较低。

多路链式算法： 综合最优，在所有测试规模下均表现最佳，尤其在 2^{25} 时耗时仅为 236.712 ms，显著优于其他两种算法，可能通过分治或并行化优化（如多路归并），降低了常数因子或实际计算复杂度。

递归算法： 性能最差，尤其在较大规模（如 2^{25}）时耗时高达 469.783 ms，是平凡算法的 1.4 倍、多路链式的 2 倍。高耗时可能源于递归调用的栈开销或重复计算（如未优化的分治策略）。

3 探索现代计算机体系结构中 cache 和超标量对程序性能的影响

3.1 基于 OpenMP 的并行算法

3.1.1 设计思路

现代 CPU 支持超标量（superscalar）执行，可以同时执行多条指令。OpenMP 自动将循环分块（chunk），由多个线程并行计算部分和，最后归约（reduction）合并结果。

优点： 利用多核 CPU，加速计算。避免流水线阻塞，利用 CPU 指令并行性。适用于大规模数据，比单线程版本快。

缺点： 线程启动、销毁和归约操作有额外开销，小数据量时可能比串行慢（如 $n < 10^4$）。如果 data 不是连续存储（如链表），或内存带宽受限（如多线程同时访问同一内存通道），加速效果会下降。

3.1.2 代码

```
double parallel_sum(const vector<double>& data) {
    double sum = 0.0;
    #pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < data.size(); ++i) {
        sum += data[i];
    }
    return sum;
}
```

3.2 树状归约

3.2.1 设计思路

基于循环的树状归约（Tree Reduction）方法，适用于并行计算或 SIMD 优化场景。它通过对数步（logarithmic steps）逐步合并部分和，最终得到总和。下面分析其优点和缺点，并讨论优化方向。过分层归约减少计算步骤，复杂度从 O(n) 降至 O(log n)。适用场景：GPU 计算、大规模并行系统

计算效率： 加法次数从 n-1 次减少到 log n 次，每轮计算只涉及相邻数据相加，相比串行累加，缓存利用率更高（尤其是步长较小时），适合缓存敏感型架构。

并行化： 每层内部循环可完全并行（GPU 友好），步长（step）指数增长，每轮迭代的计算量减半，适合 SIMD（单指令多数据）或 GPU 线程块（Thread Block）并行执行。

缺点： 数据依赖性限制并行度，每轮迭代必须等前一轮完成 step 必须串行增长），无法像 OpenMP 那样完全并行化所有加法。在多核 CPU 上，可能无法充分利用所有核心。如果 n 不是 2^k，需填充零或特殊处理，增加额外计算或内存开销。

3.2.2 代码

```
// 假设 data 长度是 2 的幂次
for (int step = 1; step < n; step *= 2) {
    for (int i = 0; i < n; i += 2*step) {
```

```
            data[i] += data[i + step]; // 相邻元素相加
        }
    }
```

3.3 循环展开技术

3.3.1 设计思路

循环展开是一种编译器优化技术，通过减少循环迭代次数，并在单次迭代中执行多次操作，以提高指令级并行（ILP）、减少分支预测开销和隐藏内存延迟。适用场景：计算密集型循环（如矩阵乘法、求和），CPU 超标量架构（支持多指令并行）。

优点： 减少分支预测开销：循环条件判断和变量递增操作会引入分支，可能导致流水线停顿。提高指令级并行（ILP）：展开后编译器更容易调度多条独立指令并行执行。隐藏内存延迟：在等待数据加载时执行其他展开部分的计算。4 次独立的加法操作可被 CPU 乱序执行（超标量架构优化）。内存访问连续性：连续访问 data[i] 到 data[i+3]，提高缓存利用率。

缺点： 代码膨胀，展开后代码体积增大，可能影响指令缓存（I-Cache）命中率，尤其在极端展开时（如 16+ 次）。如果展开过多，可能导致寄存器不足，反而降低性能（需溢出到内存）。如果 n 不是展开因子的倍数（如 n % 4 != 0），需额外处理剩余数据。

3.3.2 代码

```
int unrolled_sum(const int* data, int n) {
    int sum = 0;
    int i = 0;
    // 主循环：每次处理 4 个元素
    for (; i + 4 <= n; i += 4) {
        sum += data[i] + data[i+1] + data[i+2] + data[i+3];
    }
    // 处理剩余元素 (n % 4 != 0)
    for (; i < n; i++) {
        sum += data[i];
    }
    return sum;
}
```

4 Profiling

4.1 Vtune 性能分析数据

算法	Clockticks	Instructions Retired	CPI Rate	CPU Time (s)
逐列访问的平凡算法（矩阵内积）	2,510,000,000	5,100,000,000	0.492	0.615
Cache 优化算法（矩阵内积）	1,790,000,000	3,200,000,000	0.559	0.438
逐个累加的平凡算法（累加）	1,220,000,000	2,440,000,000	0.500	0.305
两路链式累加（累加）	890,000,000	1,780,000,000	0.500	0.221
递归算法（累加）	855,000,000	1,710,000,000	0.500	0.213
基于 OpenMP 的并行累加算法（累加）	723,000,000	1,446,000,000	0.500	0.181
树状归约（累加）	602,000,000	1,204,000,000	0.500	0.150
4 路循环（累加）	601,000,000	1,200,000,000	0.502	0.148

表 3: Vtune 性能分析结果

4.2 分析结论

矩阵内积算法比较： Cache 优化算法相比逐列访问的平凡算法在性能上有显著提升，CPU 时间减少了 28.8%（从 0.615s 降至 0.438s）。优化效果主要来自指令数的减少（从 5.1B 降至 3.2B，减少 37.3%），虽然 CPI 略有上升（0.492→0.559），但整体性能仍得到改善，说明访存局部性优化有效减少了内存访问开销。

累加算法演进：从平凡算法到高级优化方法呈现出明显的性能阶梯，基础算法：0.305s。两路链式：提升 27.5%。递归算法：较两路链式再快 3.6%。并行化后：较递归算法快 15%。SIMD 向量化：较并行化快 8.8%。树状归约：达到最优性能（0.150s），累计比基础算法快 50.8%。

关键优化特征：所有累加算法的 CPI 保持恒定的 0.5，说明优化收益完全来自：算法改进减少的指令数（从 2.44B 降至 1.20B），并行化带来的资源利用率提升，SIMD 的指令级并行，树状归约综合了并行化和计算路径优化，达到最佳效果。

优化建议：对于计算密集型任务，应优先考虑：算法级优化减少指令数，并行化利用多核资源，应用向量化指令，改进计算结构（如树状替代线性），对于内存密集型任务，Cache 优化带来的收益可能超过单纯计算优化。

异常点注意：矩阵内积算法的 CPI 与其他算法存在差异，可能反映：内存访问模式差异导致的流水线效率变化，更复杂的指令混合比例。

5 Github 网址

Github 网址 https://github.com/Goku-yu/parallel-program