南開大學
Nankai University

计算机学院

并行程序设计第二次报告

CPU 架构相关编程

姓名：宋卓伦

学号：2311095

专业：计算机科学与技术

2025 年 3 月 29 日，南开大学计算机学院，天津

目录

1 放在前面

1.1 实验目的

经过这几周周的并行程序设计学习，我们对于在内存上编程、提升性能这件事情上有了非常多的体会和感悟。本次实验将从两个方向进行，从中我们可以看到充分利用内存进行编程，这个过程中，我们会看到面向 cache 编程在性能方面产生什么样的影响。

1.2 实验硬件和环境

本次实验选用远程 WSL 连接 Ubuntu24.04 进行编程（表1）

属性	相关内容
内核版本	5.15.167.4-microsoft-standard-WSL2
硬件架构	x86_64（64 位处理器）

表 1: 硬件与环境信息

此外我们在主机上面也进行了测试（系统类型为 x64-based PC，处理器架构为 Intel64 Family 6 Model 186 Stepping 2 GenuineIntel 2400 Mhz）

2 实验一：矩阵与向量内积

2.1 实验要求

实现两种计算 $n \times n$ 矩阵与向量内积的算法：

1. **平凡算法**：逐列访问矩阵元素，单次外层循环完成一列内积计算。
2. **Cache 优化算法**：逐行访问矩阵元素，通过空间局部性优化 Cache 利用率。

使用高精度计时工具测试算法执行时间，对比性能差异。

2.2 算法设计

- 平凡算法按列访问导致低空间局部性，Cache 命中率低。
- 优化算法按行访问提升空间局部性，充分发挥 Cache 性能。
- 测试数据采用 $A[i][j] = i + j$ 的固定矩阵和标准向量，便于结果验证。
- 通过量化执行时间差异，验证 Cache 优化的实际效果。

我们尽可能设计覆盖 CPU 各级 Cache 容量的测试用例，分析程序性能随矩阵规模变化的趋势及 Cache 临界点性能突变；构造特定大小数组覆盖不同内存空间，结合时空复杂度与访存开销揭示性能规律。

后面的分析，我们使用 VTune 等工具获取 Cache 命中率、缺页次数等系统级指标，验证理论分析结果；最后对小规模矩阵采用**重复执行待测函数**的方法延长计时周期，提升测量精度。详见 Profiling 部分。

2.3 编程实现

我们按照课上/讲义中的算法内容进行编程即可，比较简单。算法部分的代码详见我们的附录 A。通识经过检验，各个算法的代码计算结果是正确的。

2.4 性能测试

优化算法的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。我们继续插入执行的代码，将我们的代码实现，此处不再赘述，后面我会给出我的 GitHub 源码。

在完成我们的程序编写之后，我们先进行运行生成任务，编译之后得到一个文件（Linux 系统下是没有后缀的，Windows 系统下是.exe 文件），之后利用这个文件我们执行代码获取实验结果。

规模	轮数	平凡算法（μs）	优化算法（μs）
8	51200	0	0
16	25600	0	1
32	12800	4	6
64	6400	24	12
128	3200	82	50
256	1600	246	249
512	800	1026	937
1024	400	5122	3882
2048	200	49597	17987
4096	100	174762	63943

表 2: 矩阵向量运算结果：平凡算法和优化算法（使用 chrono 库计时）

规模	轮数	平凡算法（μs）	优化算法（μs）
8	51200	0.39	0.37
16	25600	1.53	1.49
32	12800	6.13	5.93
64	6400	24.55	24.40
128	3200	99.96	98.50
256	1600	401.30	393.30
512	800	1871.26	1603.36
1024	400	8099.00	6408.70
2048	200	59 361.00	26 233.00
4096	100	199 949.00	105 685.00

表 3: 矩阵向量运算结果：平凡算法和优化算法（使用 windows.h 库精确计时）

显而易见，数据量小的时候，扩大规模可以放大实验效应；数据量增大的时候，我们的实验自然耗时相差显著。所以规模增大、轮数增加都对于实验有显著影响，容易看出来平凡算法优势较多，所以性能相较于优化算法极差。

但是要注意的是，实验轮数增加可能导致时间溢出（如果采用 C++ 语言的计时库 chrono 而非 Windows.h 下的精确计时函数的话），这与我们的内存相关，后续设计测试数据的时候应当注意。

2.5 Profiling

如下表4所示，再次进行实验，优化算法比平凡算法运行时间和 CPU 时间都缩短了。CPI 率优化了 50% 左右。具有显著的优势。

指标	平凡算法	优化算法
运行时间 (s)	41.362	**21.888**
CPU 时间 (s)	37.479	**20.744**
CPI 率	0.425	**0.235**

表 4: 平凡算法与优化算法性能对比

如下表5所示，平凡算法的 Cache Miss 问题较为严重，尤其是 L1、L2 未命中次数过高，L3 承担压力过大。如果改进平凡算法应当优先优化数据访问模式，同时检查 L3 容量是否匹配工作集大小。优化算法（行主次序）的缓存效率明显更高，这正是采用优化算法、更优的内存布局或预取策略的结果。

Hardware Event Type	平凡算法	优化算法
CPU_CLK_UNHALTED.DISTRIBUTED	51,647,460,642	28,417,035,742
CPU_CLK_UNHALTED.REF_TSC	108,104,162,156	59,822,089,733
CPU_CLK_UNHALTED.THREAD	51,632,541,523	28,463,693,800
INST_RETIRED.ANY	122,036,183,054	122,048,183,072
MEM_LOAD_RETIRED.L1_HIT	49,693,835,576	52,680,825,552
MEM_LOAD_RETIRED.L1_MISS	3,543,914,638	**390,036,134**
MEM_LOAD_RETIRED.L2_HIT	2,582,891,386	**374,427,442**
MEM_LOAD_RETIRED.L2_MISS	961,026,618	**15,610,696**
MEM_LOAD_RETIRED.L3_HIT	734,476,068	**7,973,870**
MEM_LOAD_RETIRED.L3_MISS	221,536,704	**7,189,512**

表 5: 两组硬件事件数据对比

3 实验二：n 个数的加法

3.1 实验要求

考虑两种算法设计思路：**逐个累加**的平凡算法（链式）和**超标量优化**算法**（指令级并行）**，例如两路链式累加；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。

我们仍要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。

3.2 算法设计

本题需关注以下要点：

1. **中间结果处理** 利用输入数组或辅助数组保存中间结果，设计访问顺序优化空间局部性。
2. **问题规模设置** 结合流水线条数、Cache 大小等系统参数设置实验规模，便于利用性能。

3. **小规模问题处理** 通过重复执行核心计算提升性能测试精度，这是放大法的使用。

测试数据生成生成 2 的幂次规模数据，适配递归算法。**循环优化**采用循环展开减少簿记开销，结合指令级并行提升计算效率，保持展开比例一致性确保公平对比。

3.3 编程实现

我们按照课上/讲义中的算法内容进行编程即可，比较简单：仍然是将代码放置在附录部分。

3.4 性能测试

在完成我们的程序编写之后，我们运行生成任务，执行获取实验结果。

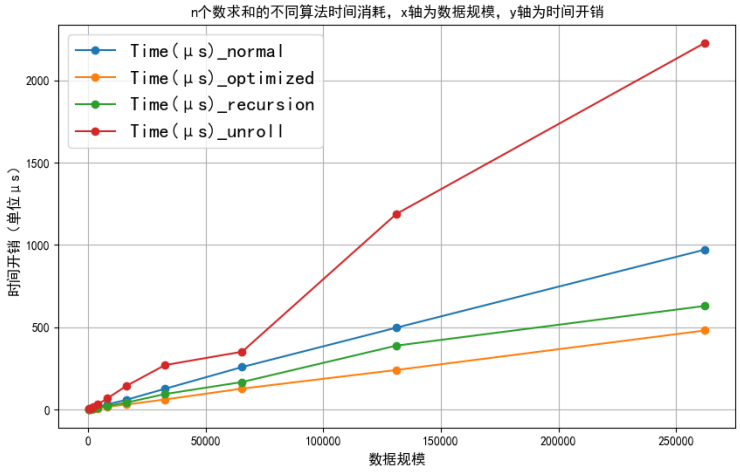

图 3.1: n 个数的加法：chrono 库下的时间记录

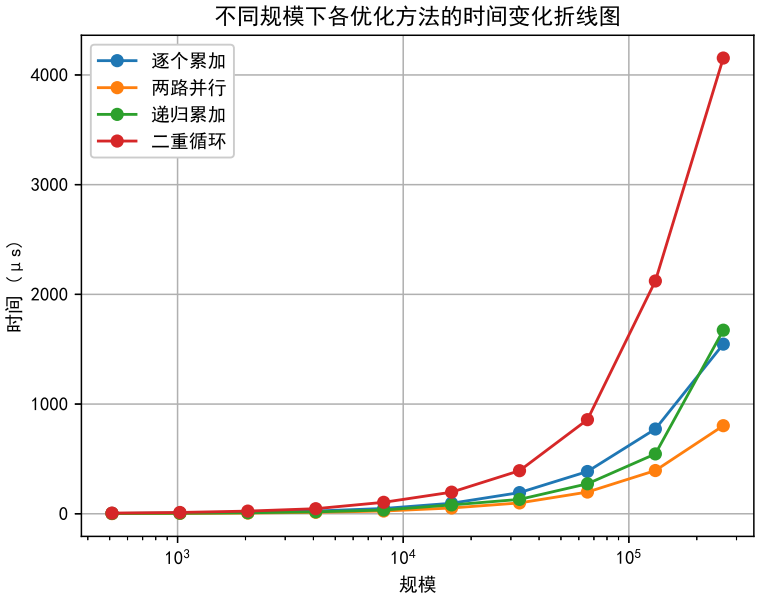

图 3.2: n 个数的加法：Windows.h 库高精度下的时间记录

可以看到，程序里面每段算法的运行时间的排序并不是稳定的。二重循环累加算法的时间开销最大，多路并行累加的时间优化程度最高。充分利用了在 cache 上面编程并多路实现的特点。

3.5 Profiling

如下表6所示，在硬件方面，两路并行算法运行时间和 CPU 时间最少，但是 CPI 率上却没有循环算法表现优异。基于这个原因，可能在于没有使用内联函数。

为了消除函数调用的时空开销，C++ 提供一种在编译时将函数调用处用函数体替换的提高效率的方法，类似于 C 语言中的宏展开。这种在函数调用处直接嵌入函数体的函数称为**内联函数（Inline Function）**。但也存在缺点：每一调用处均会展开，增加了重复的代码量。

指标	逐个累加	两路并行	递归累加	二重循环
运行时间 (s)	17.362	**10.076**	16.293	33.894
CPU 时间 (s)	15.191	**7.961**	12.176	33.109
CPI 率	0.574	0.297	**0.196**	0.232
Total Thread Count	2	2	2	1

表 6: 平凡算法与优化算法性能对比

如表7所示，递归累加和二重循环的 Cache Miss 问题较为严重，尤其是 L1、L2 未命中次数过高，我们可以优化数据访问模式，同时检查 L3 容量是否匹配工作集大小。两路并行和逐个累加的缓存效率明显更高，这正是采用优化算法、更优的内存布局或预取策略的结果。

但是逐个累加的效率竟然超过了一些函数，这可能与我们的函数设计有很大关系。正如上面所述，我们如果采取适当的内联函数并选择合适的编译选项进行编译，结果可能会好很多。

Hardware Event Type	Hardware Event Count			
	逐个累加	两路并行	递归累加	二重循环
CPU_CLK_UD	20,741,946,036	10,763,994,336	14,240,458,666	41,336,345,844
CPU_CLK_URT	43,898,065,847	23,232,034,848	29,934,044,901	87,472,131,208
CPU_CLK_UT	20,983,460,525	10,937,433,949	14,226,345,437	41,654,061,904
INST_R_ANY	36,790,055,185	36,854,055,281	79,152,118,728	189,910,284,865
MEM_LR_L1_HIT	17,802,692,458	16,296,110,306	27,791,359,604	68,132,756,410
MEM_LR_L1_MISS	116,384,696	129,173,788	229,192,916	316,595,448
MEM_LR_L2_HIT	114,671,170	126,711,036	225,695,942	305,780,506
MEM_LR_L2_MISS	1,714,826	2,463,968	3,499,046	10,819,624
MEM_LR_L3_HIT	1,660,704	2,125,700	3,368,008	8,470,378
MEM_LR_L3_MISS	113,528	138,720	118,980	1,654,824

表 7: 硬件事件数据汇总

4 实验总结

4.1 本实验的概括总结

在优化算法中，我们一开始遇到了性能远没有平凡算法好的情况，这个时候我们应当继续采取优化措施（采用内联函数或者选择合适的编译方式），结构调整（减少 cache_miss 的数值），实现更好的

功能和性能的优化。

在实验过程中，我们可以通过微小放大的方法将实验数据进行清晰的描述，避免测量过小而丢失差异。此外，我在一开始没有采用高精度计时的方法，后续实验当中也适当添加了相关内容，并记录了数据。高精度时间相较于 chrono 库的计时准确很多，也保留了足够多的精度可以看出细小变化之间的差别。

4.2 实验以外的总结

这次实验，我对于并行程序设计有了一个初步的深入体会。我们为了节约内存空间，可以对代码进行一定程度的优化。所谓优化不只是对冗余代码的删减，更是面向底层，面向 CPU 甚至以后的 GPU 进行编程，深刻了解而里面的架构之后才可以更好地实现性能的大幅提升。

最后，由衷感谢助教学长们的辛勤付出，他们对于我们实验的配置和实施给予了很大帮助。尤其当我看到助教转发了他和他的助教请教问题的聊天记录的时候，我内心一阵莫名的感动，希望未来我能够好好学习这门课程，将我所学的知识传递下去帮助更多小同学们！再次衷心感谢我们的老师和助教们的辛勤付出！

附录 A 实验一的全部代码

A.1 平凡算法

平凡算法——逐列访问

```
// 矩阵向量内积的平凡算法（列优先访问）
void mat_vec(double** matrix, double* vec, double* result, int n) {
    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];
        }
    }
}
```

A.2 优化算法

优化算法——逐行访问

```
// 矩阵向量内积的优化算法（行优先访问）
// 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
void optimized_mat_vec(double** matrix, double* vec, double* result, int n) {
    memset(result, 0, n * sizeof(double));
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            result[i] += matrix[i][j] * vec[j];
        }
    }
}
```

附录 B 实验二的全部代码

B.1 平凡算法

平凡算法——逐个累加

```
// 求和的平凡累加
double normal_sum(double* arr, int n, double sum) {
    for (int i = 0; i < n; i += 1) {sum += arr[i];}
    return sum;
}
```

B.2 多路（二路）算法

优化算法——两路并行累加

```
// 求和的两路并行累加
double optimized_sum(double* arr, int n, double sum) {
    double sum1 = 0.0, sum2 = 0.0;
    for (int i = 0; i < n; i += 2) {
        sum1 += arr[i];
        if (i + 1 < n) sum2 += arr[i + 1];
    }
    sum = sum1 + sum2;
    return sum;
}
```

B.3 循环算法

优化算法——递归累加

```
// 求和的递归累加
double recursion_sum(double* arr, int n, double sum) {
    if(n == 0) return 0.0;  // 处理边界情况
    if(n == 1) return sum + arr[0];
    double temp = 0.0;
    for(int i = 0; i < n/2; i++){temp += arr[i] + arr[n - i - 1];}
    n = n/2;
    return recursion_sum(arr, n, sum + temp);  //   累加中间结果
}
```

B.4 二重累加算法

优化算法——二重循环累加

```
// 求和的二重循环累加
double sec_roll_sum(double* arr, int n, double sum) {
    vector<double> summ(n);
    copy(arr, arr + n, summ.begin());
    for(int m = n; m > 1; m /= 2){
        for(int i = 0; i < m/2; i++){summ[i] = summ[2*i] + summ[2*i+1];}
    }
    sum = summ[0];
    return sum;
}
```