南开大学
Nankai University

南　　开　　大　　学

计　算　机　学　院

并行体系结构编程实验及其性能测试

---

n*n 矩阵向量内积、n 数求和并行算法探究

姓名：李威远

年级：2021 级

专业：计算机科学与技术

指导教师：王刚

2024 年 3 月 24 日

目录

一、 实验环境与背景

(一) 实验前言

作为一名大三下的同学，此时我已经完整学习了**体系结构、计算机组成、编译原理、操作系统**等 system 方向的重要课程。因此，我希望自己能结合这几门课程所学，能够对并行课程的这些课题有更好的理解，更好地学习并行这门课程。

在以后的实验中，我希望我能够结合自己所学的内容，**以一个更全面的、对计算机组成更了解的视角**，尝试做一些有意义的并行程序探究，结合这些课程的视角来分析，我觉得这会让我的实验报告充满乐趣与意义，一方面能感受到自己的成长，另一方面，这让我能够以一种时刻思考的角度来完成实验，并非无意义的卷工作量，而是做实实在在的自己感兴趣的研究。

(二) 主要实验环境

在我们基础部分中，两个代码的完成依托于如下的实验环境。

- **实验平台：** Windows X86、Ubuntu、鲲鹏服务器
- **CPU 配置：** AMD Ryzen 9 5900HX with Radeon Graphics 3.30 GHz(本机)
- **各级 cache：** L1 512KB L2 4.0MB L3 16.0MB(本机)
- **编译器：** TDM-GCC
- **开发环境：** vscode

代码在github中给出。

二、 编程实验 1：计算 n*n 矩阵每一列与给定向量的内积

(一) 程序要求

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：

1. 对两种思路的算法编程实现；
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。

(二) 程序设计思路与代码展示

对于本题而言，平凡算法即我们平时所使用的自然思路，对于内积后生成的向量的每一个元素，我们通过对每列的遍历计算来得到其值即可。

给出实现的平凡算法代码如下所示：

逐列访问平凡算法

```
for (int col = 0; col < n; col++) {
    dotProduct[col] = 0;
    for (int row = 0; row < n; row++) {
        dotProduct[col] += matrix[row][col] * vector[row];
    }
}
```

但是实际上，这种做法尽管符合我们的直观逻辑，但是没有充分利用 cache 的空间局部性。

我们知道，cache 在设计之初就充分利用了空间局部性和时间局部性，最基本的 cache 在取数据时，往往不会只取一个地址的数据，而是将这个地址所在的一片区域的数据都取进 cache，我们称之为一个“cache 块”。

但是，cache 的空间局部性仅仅是这些吗？本题还可能受到什么因素影响呢？

- **体系结构：** cache 的预取策略，strip 策略、next-line 策略等等。
- **操作系统：** 页管理系统也会对空间局部性进行利用。
- **编译原理：** 几种循环优化算法，在这里显然也是可以应用的。

由此一联想，不禁感叹 system 方向的知识确切的是紧密联系在一切，自底向上梳理一遍，顿时觉得宫老师那句“牵一发而动全身”的意义所在了。

这里我不去过多探究。假设**刚刚回忆的几种可能与本题相关的优化都不存在**，那么这时候 cache 是简单地取一块空间来利用空间局部性，我们如何给出 cache 优化的 n*n 矩阵每一列与给定向量的内积的工作呢？

很显然，由于二维数组在内存中的组织方式实际上是连续的一块空间，按照行去读取，能够保证读的一块可能在一个块里，可以较好的利用空间局部性，我们给出按照行去读取二维数组的 cache 优化算法：

cache 优化算法

```
for (int col = 0; col < n; col++) dotProduct[col] = 0;
for (int row = 0; row < n; row++) {
    for (int col = 0; col < n; col++) {
        dotProduct[col] += matrix[row][col] * vector[row];
    }
}
```

(三) 实验设置

基础部分实验的设置，我们只在 windows 系统下，基于 vscode 的 ide，使用 gcc 编译器来编译两个程序，进行计时和测试，我们从矩阵大小为 50*50 开始，依次提高矩阵大小，比较两个算法的速度差异（起初采用 O0 级别的优化）。

怎么给出精准的时间测量呢？实验指导书已经给出，在 windows 系统下，我们可以使用如下的方法来获取微秒级的时间测量：

时间计算

```
QueryPerformanceFrequency(&frequency);
QueryPerformanceCounter(&start);
……
QueryPerformanceCounter(&end);
double time_taken = (double)(end.QuadPart - start.QuadPart) / frequency.
    QuadPart;
```

但是，以防时间太小，我们进行多次计算，以时间总和来计算平均值，从而保证实验的准确性

cache 优化算法

```
    double sum = 0;
    int iters = 30;
    for (int i = 0; i < iters; i++) {
        sum += calculateDotProduct(matrix, vector);
    }
    printf("All Time taken: %f seconds\n", sum / iters);
```

对于数据的定义，由于担心我自行定义的数据存在歧义，我还是选择了随机生成矩阵和向量的数据，其中每个元素为 **10000 以内的随机数**，此外，由于测试很麻烦，我还提供了自动测试的 makefile 脚本，针对不同大小的 n 输入，进行测速和统计，如下图所示：

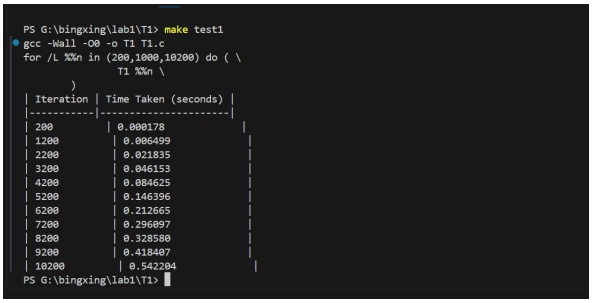


图 1：自动化测试脚本输出一定范围内的测速结果

迫于篇幅所限，这里就不再贴出随机生成矩阵和自动化测试脚本的代码，都可以在github找到。

（四） 实验结果展示与分析

矩阵大小	cache 优化算法时间开销（s）	平凡算法时间开销（s）
200	0.000178	0.000182
2200	0.021835	0.028784
4200	0.084625	0.108765
6200	0.212665	0.285144
8200	0.328580	0.428612
10200	0.542204	0.697431

如上表格所示，为不同矩阵大小下，经过各 30 次测试平均计算出 cache 优化算法与平凡算法开销的对比，这里**迫于篇幅，展示的跨度很大**，实际上在自动测试脚本的帮助下，我们测量了从 200 到 10000、每次跨度为 200 的所有矩阵大小，具体的数据，我们也已经给到github中了。最终，我们统计得到了如下的对比图像：

可以看到，cache 性能优化算法的开销实际上一直要比平凡算法开销小，表明了其可以充分利用 cache 的空间局部性这一优点。

其中，我们可以注意到如下的几个有趣现象，并给出相应的原因分析：

- **矩阵大小越大，二者的性能差异越大**。这是因为矩阵大小比较小的时候，很可能 cache 取的一个块就包含了矩阵的大量数据，哪怕是逐列读取也会造成 cache 命中，因此二者在矩阵小的时候性能差别小。而矩阵大小增大会加深空间局部性的利用，性能差距会变大。

- **当矩阵大小超过 8000 的时候，会发生性能的突变，且之后的性能抖动性大**。这是为什么呢? 其实**和系统 cache 的大小相关**，我们可以做一个简单的计算，矩阵大小如果超过 8024，一行就会占据 8*8024 个字节，也就是 64Kb 的大小，这很大可能就是我们的一级 cache 的块大小，如果一行占据空间大于块大小，哪怕是逐行访问的 cache 优化算法，也会出现较为频繁的 cache 不命中，大大提高性能差。
- **实际上两个算法的性能差异并不巨大**。这是因为如今的计算机中，空间局部性的利用远远不止 cache 块粒度这一方面，很可能我们的 cpu 中存在某种预取策略，如 strip 策略就能很好地捕捉逐列的访问模式，并基于此提高 cache 命中率。因此，平凡算法和 cache 性能优化算法的性能差异不会很大。

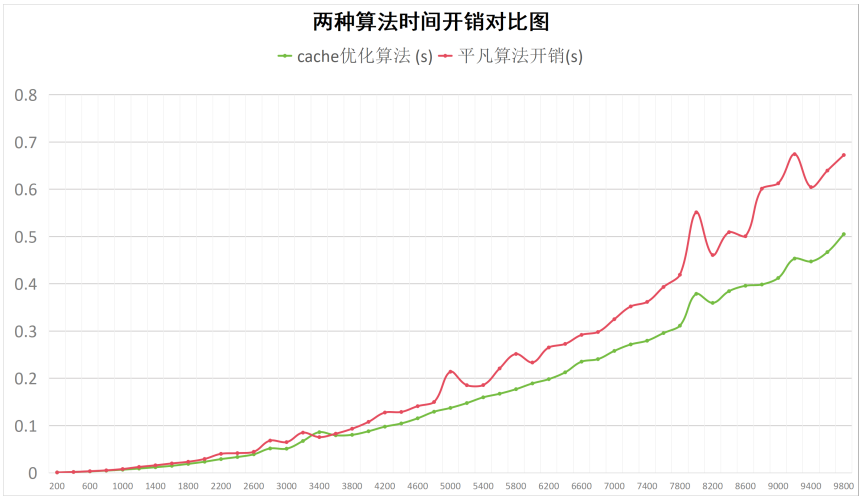


图 2: 两种算法效果对比图

三、 编程实验 2：计算 n 个数求和

(一) 程序要求

计算 n 个数的和，考虑两种算法设计思路：

- a) 逐个累加的平凡算法 (链式)
- b) 超标量优化算法 (指令级并行)，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。

(二) 程序设计思路与代码展示

对于本题，平凡算法我们不再赘述，非常简单。

有关超标量优化算法，我们会想起体系结构中所学，最经典的莫过于去年考试前一直复习的**托马斯洛算法**，那么我们对本题的分析和实验设置，实际上可以以托马斯罗算法为底层原理进行思考：

考虑两路链式相加算法，实际上就是一个循环跑两条求和指令，而且这两条指令不存在写后读的依赖，因此结合托马斯洛算法来看，两条指令都能顺利进入保留站，且没有数据依赖的标记，能够完成“指令级并行”。

其代码非常简单，即 for 中计算两次：

两路链式相加算法

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

而递归算法在 c++ 环境下却非常难以实现，我们先给出如下的直观思考得来算法，该算法实际上**以托马斯洛算法的视角来看是完全不能达成指令级并行**的，我们在结构分析部分给出解释。

递归算法

```
while (n > 1) {
    for (int i = 0; i < n - 1; i += 2) tempArray[i / 2] = tempArray[i] +
        tempArray[i + 1];
    n = (n + 1) / 2;
}
```

(三) 实验设置

实验设置与编程实验 1 几乎一样，同样通过自动化测试脚本完成了测试和统计的工作，记录了 200 到 10000 大小数组求和的性能。

(四) 实验结果展示与分析

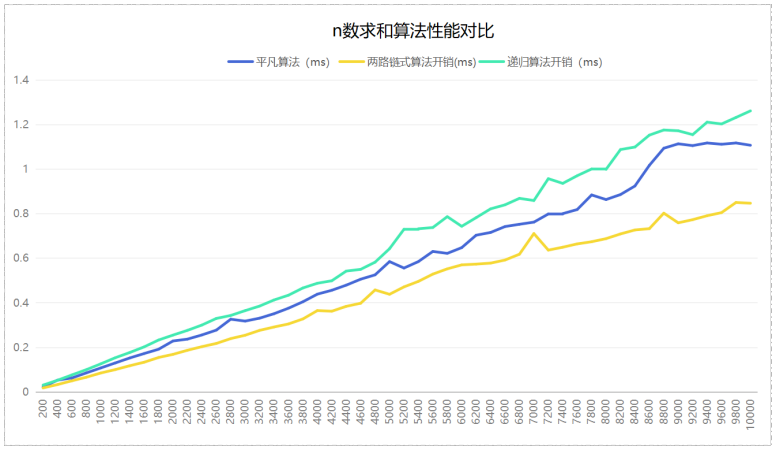


图 3: n 数求和性能对比图

具体的数据由于篇幅，这里我们不再作出展示，同样可以在github中找到具体测量的数据，这里我们给出如下的图像分析

从图中，我们可以分析得到如下信息：

1. **递归算法居然性能比平凡算法还要差**。事实上，这是完全正确的现象，结合托马斯洛算法所学，在我们的递归中，不可避免的存在 i++ 操作，而取存对该指令具有严格的写后读数据依赖，基于此，递归算法很难做到真正的指令集并行。

2. **两路算法对平凡算法优化明显**事实上，我们对这个循环有太多的优化空间了，稍微增加一路就会优化很多，我们放到后面提高部分进一步探究

四、 提高部分

(一) 循环优化性能探究 (unroll)

我们之前实现的两路链式相加实际上已经是循环展开的一种了，接下来，我们进一步的进行循环展开，尝试进行 4、8、16 路的相加，得到如下的结果，具有如下特性：

- **8 路展开和 4 路展开效果相近**。我们发现 8 路展开和 4 路展开效果相加，可能指令级并行的上限为 8-4 之间，因此此时达到了瓶颈。
- **16 路展开反而造成效果很差**。可能具有如下原因：
  1. 缓存溢出：如果展开的循环体太大，可能会超出 CPU 的缓存大小，导致缓存命中率下降，从而降低性能。
  2. 指令调度：过大的循环体可能会超出硬件的指令调度能力，导致 CPU 无法充分利用其执行单元，从而降低性能。

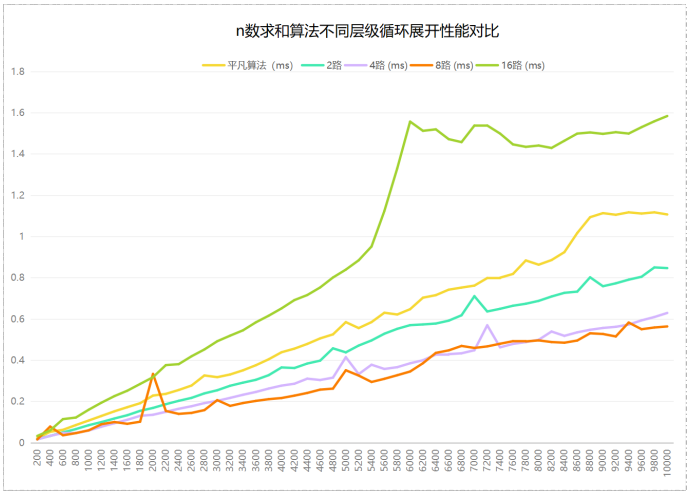


图 4: n 数求和不同层级循环展开性能对比图

(二) 现代计算机 cache 和超标量影响探究

考虑到我们所学的托马斯洛算法原理，我认为通过**写后读而非写后写**的数据依赖更能够体现超标量的影响，我们可以多设计一些存在写后读的数据依赖和不存在写后读的数据依赖的算法进行对比。

基于托马斯洛算法实际上可以通过**寄存器重命名**一定程度绕开我们刚刚给出的写后写依赖的示例，因此最好采用写后读的算法进行分析，这里受限于篇幅，就不再贴出写后读的代码，在 github 中已经给出，经测试要劣于写后写的实例。

写后读下的算法

```
1    for (int i = 0; i < n; i++) {
2        temp += array[i];
```

```
        sum += temp;
    }
```

(三) 更多的对比实验和分析

1. 编译器不同优化粒度对性能的影响

我们知道，gcc 具有 O0、O1、O2、O3 多种优化层级。我们调整 makefile 中的编译选项，使用-O0、-O1、-O2 的编译选项选择优化力度，我们以第一个实验中的代码为例，测试不同优化层级下性能的影响：

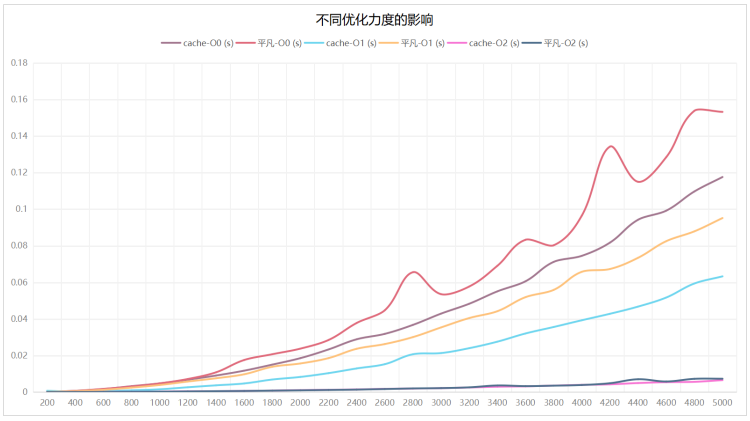


图 5: 不同优化层级的性能对比

结果非常显然，**当优化层级不断提高，性能也是不断提高的，当优化层级到 O2 的时候达到最高**。

2. Windows 和 linux 平台的性能对比

对于 windows 和 linux 平台性能的对比，具体测试通过 ubuntu 虚拟机进行，我们需要做一些代码的修改，以适应 linux 系统，这里我们不再展示代码，具体代码也在 github 中给出，我们给出 linux 和 windows 下性能的对比，发现相差无几。

总体上，**windows 平台要略优于 linux，可能是由于虚拟机本身逻辑 cpu 性能的问题**:

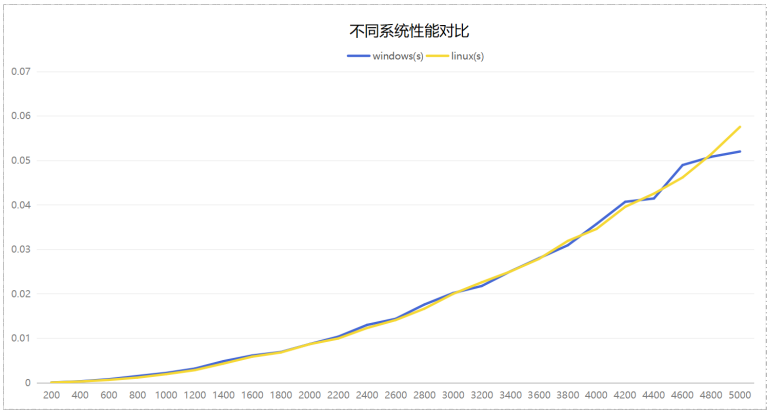


图 6: 不同优化层级的性能对比

3. x86 平台和 arm 平台的对比

我们使用给出的鲲鹏服务器，用 bisheng 编译器与本机 gcc 的测试进行对比（由于本次使用使用鲲鹏服务器人数不多，服务器压力很小，经助教允许后直接 make 编译运行了）。

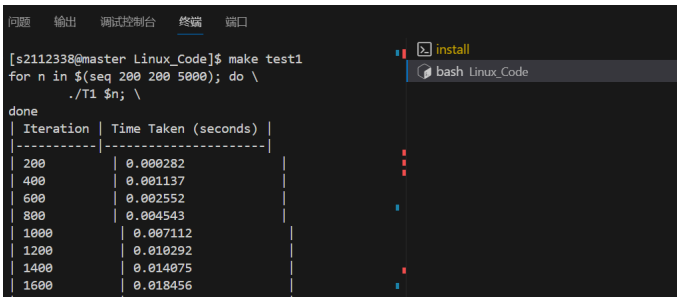
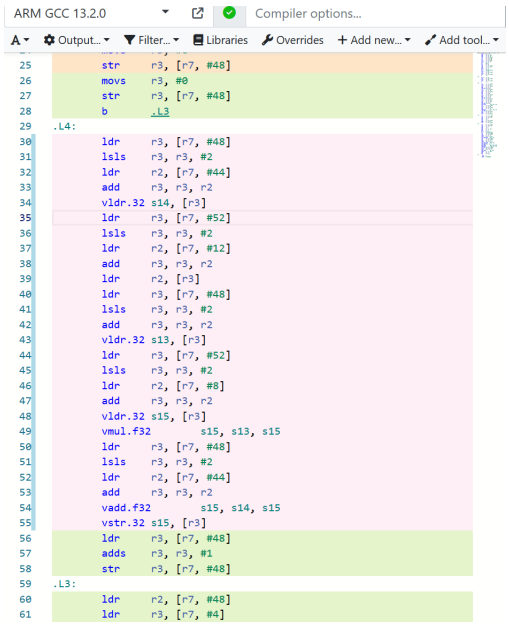


图 7: 鲲鹏服务器下成功执行

可以看到，鲲鹏服务器下，arm 架构与 x86 架构的执行速率还是有一定的差距的，这可能出于如下几点原因：

1. **流水线设计**：ARM 和 x86 处理器的流水线设计可能不同，影响指令的延迟和吞吐量。
2. **内存访问性能**：ARM 和 x86 处理器对内存访问的优化不同，影响代码执行速度。
3. **体系结构特性**：ARM 和 x86 在体系结构特性上有差异，如乱序执行、分支预测等。

(四) 研究不同编译器表现



(a) arm 编译结果

(b) x86 编译结果

我们对比 arm 和 x86 的 gcc 的编译器差别，能清晰的看到，x86 下指令的数量有明显下降。尽管 x86 为复杂指令集，但是我们的问题 1 中的很多指令，经查询后并非特别复杂，可以看出 x86 下 gcc 编译器的优化要远远好于 arm 下的优化。

因此，我们能够在上一部分的提高内容中看到，arm 平台下程序的执行性能要弱于 x86。