并行作业——CPU 架构相关编程

作者：Yu XiangYou

学号：2312900

专业：计算机科学与技术

提交日期：2025 年 3 月 26 日

目录

1 编程一. 计算 N*N 矩阵每列与给定向量的内积

1.1 基础要求

实验环境:

实验设备: Macbook Air M1(8+256)

实验平台: Xcode 平台

1.1.1 逐列访问元素的平凡算法

函数:vector<int> column_major_calculate(vector<vector<int> > matrix,vector<int> vector):

该函数的返回类型为一个向量，向量第 i 项就存储的是第 i 列与向量 vector 运算后的结果，此时通过先遍历列，再遍历行的方式查找每列的元素。这种方式很符合直觉，即一列列实现计算，将结果逐个存进向量内。

```
// 逐列访问的计算方式 : 一列一列实现计算
vector<int> column_major_calculate(const vector<vector<int>>&
    matrix, const vector<int>& vec)
{
    int n = matrix.size();
    vector<int> result(n, 0);
    for (int j = 0; j < n; ++j)
    {
        for (int i = 0; i < n; ++i)
        {
            result[j] += matrix[i][j] * vec[i];
        }
    }
    return result;
}
```

1.1.2 cache 优化算法

函数: vector<int> row_major_calculate(vector<vector<int> > matrix,vector<int> vector):

该函数的返回类型为一个向量，向量第 i 项就存储的是第 i 列与向量 vector 运算后的结果，此时先遍历行，再遍历列，相当于每遍历一行时，结

果就完成一部分，遍历完就有了最终结果。这种方式可能不太符合直观感受，但更符合计算机逐行读取 matrix[i][j]，省去了多余的元素查找时间。

```
// 逐行访问的计算方式：相当于一行算一点结果，最后遍历完就有最后结果
vector<int> column_major_calculate(const vector<vector<int>>& matrix, const vector<int>& vec)
{
    int n = matrix.size();
    vector<int> result(n, 0);
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            result[j] += matrix[i][j] * vec[i];
        }
    }
    return result;
}
```

1.1.3 计时实现

调用 C++ 11 引入的时间库，调用该库中的三个函数共同作用即可实现计时：

函数 1：high_resolution_clock::now()：

此函数可以获取当时运行的具体时间并返回。在此处，我们只需要分别在函数调用前实现一次获取时间并且在调用后再获取一次时间，通过计算差值我们就能获取运行的具体时间，如同下方示例：

```
// 记录开始的时间
auto start = high_resolution_clock::now();
// 运行待测的函数
exampleFunction();
// 记录结束的时间
auto end = high_resolution_clock::now();
```

函数 2：duration_cast<>(end_time-start_time)：

该函数负责将时间转换为给定单位下的数值，单位由 <> 内给出，可以是秒（seconds）、毫秒（milliseconds）、微秒（microseconds）、纳秒（nanoseconds）等。在此处，我们可以将单位选择为毫秒，实现完整运行时

间的计算与转换。

```
// 计算运行时间（单位：ms）
auto duration = duration_cast<microseconds>(end - start);
```

函数 3：duration_count()：

该函数负责获取数值。在此处，我们可以使用其获取转换后的运行时间，并且输出出来。

```
 // 计算运行时间（单位：ms）
auto duration = duration_cast<microseconds>(end - start);
// 获取并输出出来
cout << "代码执行时间:" << duration.count() << "ms" << endl;
```

综合以上三个函数，我们就可以实现对代码运行时间的测定与获取。

1.1.4 逐列访问元素 VS cache 优化算法

表 1: 两种方式平均运行时间比较

矩阵规模	逐列访问/ms	cache 优化/ms
1000*1000	10.47	10.02
2000*2000	42.42	39.02
3000*3000	120.15	87.47
4000*4000	316.89	155.66
5000*5000	498.66	242.82
6000*6000	487.79	350.81
7000*7000	698.77	481.05
8000*8000	1136.41	627.47
9000*9000	1922.34	793.32
10000*10000	2373.28	981.35

ps: 用于计算的向量的大小等于矩阵的行数，且矩阵和向量所有元素都为 int 类型的 1

图表如下：

图 1: 逐列计算 VS cache 优化

1.2 进阶要求

1.2.1 块优化方法

又从网上了解得知了另一种优化方法：块优化。

该方法在处理大矩阵相乘计算时特别好用。主要想法就是将大矩阵拆分为一个个小块，计算的时候只访问相邻的数据，这样就能尽量避免从内存中加载数据，尽量从快速缓存中读取数据进行计算。此处我尝试实现了一下：

```
// block优化计算函数
vector<int> block_optimized_calculate(const vector<vector<int
    >>& matrix, const vector<int>& vec, int block_size) {
    int n = matrix.size();
    vector<int> result(n, 0);
    // 外层循环按照块的大小分块
    // 可视化的话：实际移动方向就是先向下 再往右挪 直到覆盖完
    for (int bi = 0; bi < n; bi += block_size) {
        for (int bj = 0; bj < n; bj += block_size) {
            // 对于每个小块，我们处理该块内的计算
            // 使用min()防止在处理最右边&最下边几个块越界
            for (int i = bi; i < min(bi + block_size, n); ++i)
```

```
                {
                for (int j = bj; j < min(bj + block_size, n);
                    ++j) {
                    result[j] += matrix[i][j] * vec[i];  //
                        matrix[i][j]实际上就等于matrix[b][j]
                }
            }
        }
    }
    return result;
}
}
```

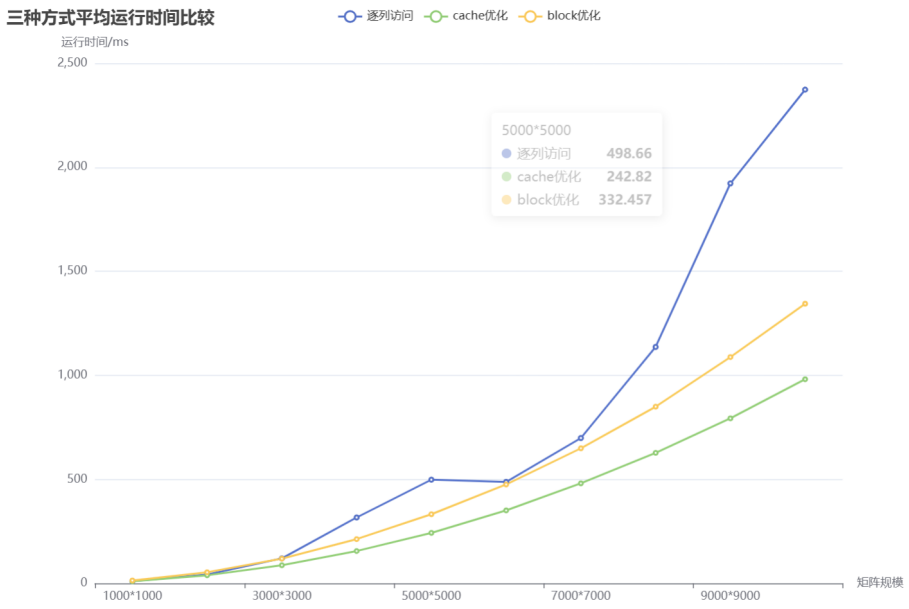

图 2: 逐列计算 VS cache 优化 VS 块优化

可以看出块优化的方式效果还是比较不错的，虽然在我所尝试的范围还不如 cache 优化方式，但也显著好于按列访问计算的方式。

1

[1]块优化参考来源：https://www.twisted-meadows.com/high-performance-gemm/

2 编程二. 计算 n 个数的和

2.1 基础要求

实验环境:

实验设备: Macbook Air M1(8+256)

实验平台: Xcode 平台

2.1.1 逐个累加的平凡算法

函数: ordinary_get_sum(vector<int>):

通过遍历 vector 类型变量，不断进行累加求和，从而实现最简单的单一链式求和，返回结果类型为 longlong。

```
// 单一链式的求和（最简单）
long long ordinary_get_sum(vector<int> v)
{
    long long result=0;
    for(int i=0;i<v.size();i++)
    {
        result = result+v[i];
    }
    return result;
}
```

2.1.2 超标量优化算法（指令集并行）

函数: two_ways_get_sum(vector<int>):

同上，通过遍历 vector 类型变量，分别对奇数索引和偶数索引分开求和，实现并行的求和运算，最后返回总和, 类型为 long long。

```
// 两路链式累加求和: 分成两路，分别对于奇数索引和偶数索引实现求和，最后加在一起
long long two_ways_get_sum(const vector<int>& v)
{
    // 用于存储奇数索引的和
    long long odd_sum = 0;
    // 用于存储偶数索引的和
    long long even_sum = 0;
```

```
8
9       // 遍历数组，分别累加奇数索引和偶数索引的元素
10      for (int i = 0; i < v.size(); ++i) {
11          if (i % 2 == 0) {  // 偶数索引
12              even_sum += v[i];
13          }
14          else {   // 奇数索引
15              odd_sum += v[i];
16          }
17      }
18      // 返回两路的总和
19      return odd_sum + even_sum;
20  }
```

2.2 进阶要求

2.2.1 OpenMP 并行计算

采用 OpenMP 实现并行求和：

```
1   // OpenMP 并行求和
2   long long parallel_get_sum(const vector<int>& v)
3   {
4       long long result = 0;
5       // pragma omp为OpenMP指令的起始标志 parallel for表示开始并
            行操作 reduction(+:result) 进行规约操作，防止数据竞争，
            +指的是此处为累加操作，result指的是赋值给result变量
6       #pragma omp parallel for reduction(+:result)
7       for (int i = 0; i < v.size(); i++)
8       {
9           result += v[i];
10      }
11      return result;
12  }
```

2

[2]OpenMP 参考来源：https://zhuanlan.zhihu.com/p/397670985

表 2: 三种方式平均运行时间比较

数组规模	单链/ms	双链/ms	OpenMP/ms
1000	11	5	42
10000	100	46	128
100000	970	459	195
1000000	9604	4429	592
10000000	96486	44095	4806

ps: 对于规模为 n 的数组，其内的值就为 0-n-1。

从表格就可知：OpenMP 在处理规模非常大的数组的时候，能十分有效地将运行时间节省下来。

3 结论

说实话，直到当自己真的亲自将代码写出来并且实现计时之后，才能深刻体会到一个小小的改变能对程序的性能有这么大的影响。由第一个实验，光是一个简单的矩阵与向量相乘，按照平常思路写出来的逐列访问的代码竟然性能会比换个角度思考之后得出来的逐行的代码性能相差这么多。这些也正印证了老师所说的“纸上得来终觉浅，绝知此事要躬行”。

这次实验也十分有收获，不仅仅学会了使用 chrono 库进行精细地计时，还学会了在 mac 内使用终端运行自己的程序（主要是 OpenMP 无法直接导入），更启发了我对于自己以后编程可能会潜意识设法从机器的角度出发来思考代码。收获满满！