并 行 与 分 布 式 计 算

homework 3

院 (系) 名 称：　数据科学与计算机学院

专 业 名 称：　计算机科学与技术

学 生 姓 名：　王永锋

学 生 学 号：　16337237

指 导 教 师：　陈鹏飞

二〇一八年五月七日

目　录

1 第一题

第一题

Consider a simple loop that calls a function dummy containing a programmable delay.All invocations of the function are independent of the others. Partition this loop across four threads using static, dynamic, and guided scheduling. scheduling. Use different parameters for static and guided scheduling.Document the result of this experiment as the delay within the dummy function becomes large.

1.1 题目分析

这一道题，主要需要我们探讨这样的几个问题。

- 对于 `dynamic` 调度方法，选取怎样的 `chunk size` 能够得到更高的加速比。
- 对于 `guided` 调度方法，选取怎样的 `chunk size` 能够得到更高的加速比
- 对不同的调度方法的加速比进行横向的比较。

1.2 实验前提

1.2.1 延迟函数设计

本次实验所使用的延迟主要来自于这样的一个双重循环：

```
void parallel_delay(int times){
    # pragma omp parallel
    {
        num_threads = omp_get_num_threads();
        # pragma omp for schedule(runtime)
        for (int i = 0; i < times; i++){
            for (int j = 0; j < times; j++){
                sum += i*j;
        }
    }
}
```

其中循环的次数为$times*times$, 该 times 的值在后面的图中表示为$delay_times$。

因此，可用 $delay\_times$ 的值表示延迟的大小。

1.2.2 实验环境

本次实验在一台 CPU 为 `i3-3110M`，系统为 ubuntu 的笔记本上测试，该笔记本放在宿舍里挂机，没有运行其他的程序。我通过 ssh 连接到该电脑中跑代码记录时间，因此可保证运行环境的稳定。

该笔记本 CPU 为物理双核，因此理论上能够达到的加速比最高为 2。

1.3 dynamic 调度方法

本节探讨不同的 `chunk size` 与 `dynamic` 调度方法的加速比之间的关系。

在实验前，我先在网上查了相关资料，了解到在 dynamic 调度方法中，`chunk size` 的意义为每一次分配给线程的循环次数。又了解到 dynamic 默认的参数为 1，在该参数下，使用该调度方法会极其缓慢，因此在后面，我根据该参数的意义，将研究的 `chunk size` 范围大致定在 100-4000 左右。

实验中，我编写了脚本在电脑中跑程序并记录时间，然后使用 python 画图以直观的展现加速比与 chunk size 之间的关系。可见图 1.1。

图 1.1　加速比与 chunk size 之间的关系

为了更精确的展现结果，我将横坐标的范围缩小，得到新的图，可见图 1.2。

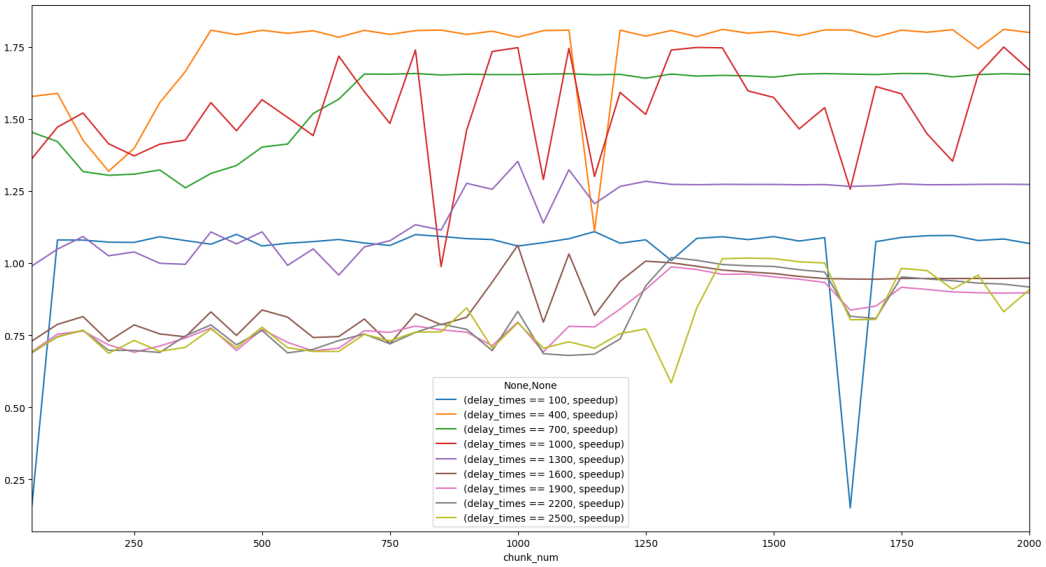

图 1.2　加速比与 chunk size 之间的关系

1.3.1　实验结果

- 在延时比较小的时候，（如图 1.2中橙线），使用适当的 chunk size 可以显著的提高运行速度。
- 通过图 1.2中绿线和橙线的比较，可以知道，当延时变大的时候，要使用更大的 chunk size 才可以达到更好的加速效果。
- 当延时越来越大的时候，并行程序的加速效果渐渐弱化，甚至不如串行程序。

1.4　guided 调度方法

本节研究不同的 `chunk size` 对 guided 调度方法效果的影响。

需要注意到，在 `guided` 调度方法中，`chunk size` 参数的意义是最终分配的迭代数收敛到的值。guided 调度方法对每次分配的迭代数的处理方式为指数级收敛，收敛速度很快，若很快就收敛到 1，而剩余未分配的迭代数仍有许多，这时候便会如先前提到的默认 dynamic 一样，导致运行时间变长。

因此，我根据参数的意义，选取了一个合适的范围进行研究。结果如图 1.3所示。

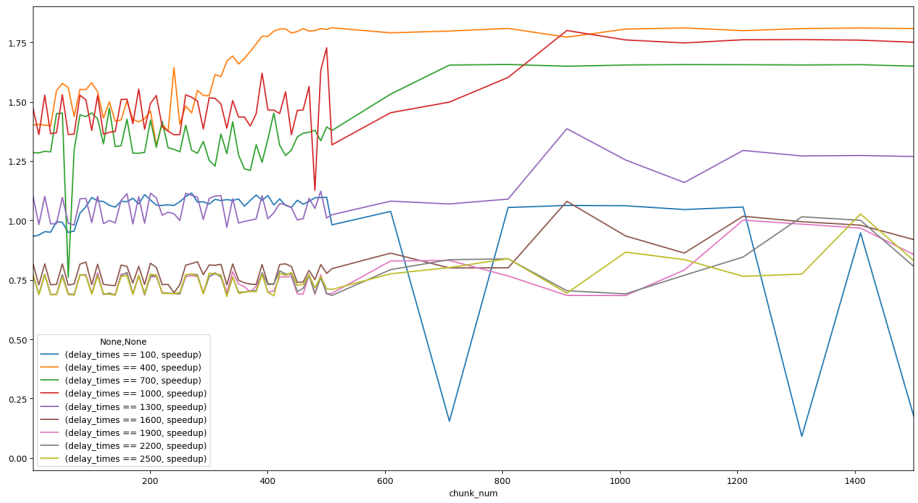

图 1.3　guided 调度方法中，加速比与 chunk size 之间的关系

1.4.1　实验结果

- 在延迟较小的情况下 (delay_times 在 100 到 1000 之间), 多线程程序的加速效果是比较明显的。
- 在一定范围内，随着延迟的增大，需要更大的 `chunk size` 才能够达到理想的加速效果。
- 当延迟过大时，并行程序的加速效果渐渐消失，甚至不如串行程序。

1.5　不同调度方法之间的对比

`OpenMP` 中，主要由这三种调度方法：

- `static`
- `dynamic`
- `guided`

其中，我需要说明一下上面没有研究 static 调度方法的原因。static 调度方法在默认情况下是将循环的迭代数平均分配给每一个线程，而我认为这已经做到足够好，在此基础上，无论是分配的迭代数再增加，还是减少，都会带来额外的分配的开销。相比起 static，其他两种调度方法对参数的选择就要灵活很多，加速效果也不清晰，因此需要研究。

1.5.1 作图

从上面对两个调度方法的研究来看，我在这一次研究中，对 `dynamic` 调度方法，选取的 `chunk size` 为 1000，而在 `guided` 调度方法中则选取 910。

下面做出程序加速比随 `delay_times` 增加而变化的趋势图，如图 1.4。

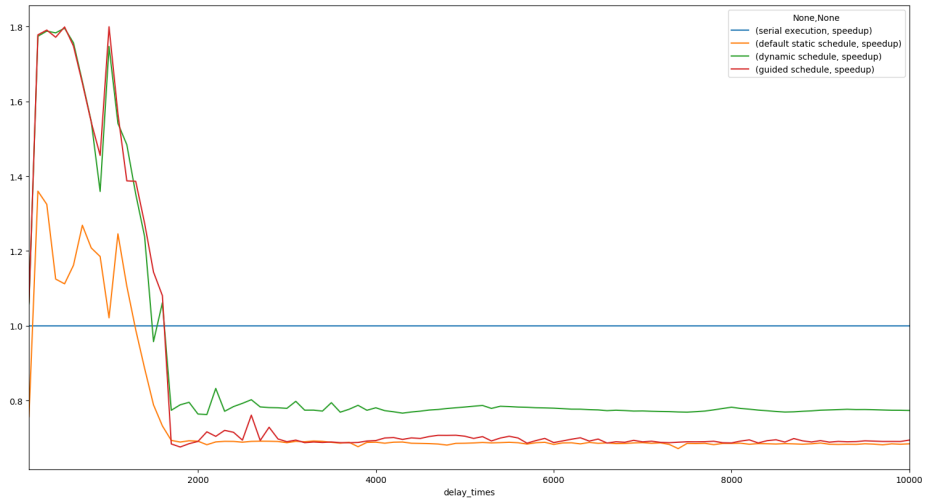

图 1.4　程序加速比与运行延时之间的关系。

放大前面一段更有意义的数据，如图 1.5所示。

1.5.2 实验结果

- 在一定范围的延时内，三种调度方法均达到了不错的加速效果，其中 dynamic 和 guided 都达到了加速比为 1.8 的程度，接近极限值 2。
- static 调度方法表现不如 dynamic 与 guided 调度方法，dynamiac 调度方法与 guided 调度方法表现类似。
- 这些调度方法在延时超过一定大小时，会渐渐失去加速效果，运行时间甚至不如串行程序。

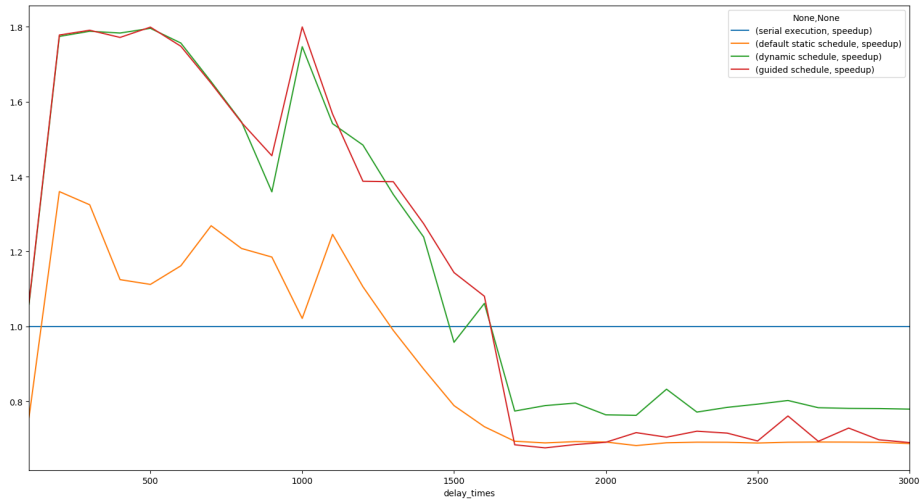

图 1.5　程序加速比与运行延时之间的关系。

2 第二题

第二题

Implement and test the OpenMP program for computing a matrix-matrix (50 * 50) product. Use the OMP_NUM_THREADS environment veriable to control the number of threads and plot the performance with varying numbers of threads. Consider three cases in which

- only the outermost loop is parallelized.
- the outer two loops are parallelized.
- all three loops are parallelized.

What is the observed result from these three cases?

2.1 实验前提

2.1.1 矩阵规模

在测试过程中发现，50*50 的矩阵乘法规模太小，导致在输出时间的时候在大多数情况下只会输出 0。出于便于研究，易于对比数据的目的，此后研究的矩阵乘法均是 1200 维下的矩阵乘法。

2.1.2 测试环境

在我自己电脑进行测试的时候，由于我一边用我自己的电脑完成作业，一边跑程序，跑出来的时间波动过大，看不出规律，因此，对于测试环境，要求要尽可能少的活动进程，本次作业的数据，我是在宿舍的一台没有人用的电脑跑的，保证了运行环境没有过多的其他进程的干扰。

2.1.3 运行性能的评价

本次实验在评价矩阵乘法的性能时，由于并没有会改变矩阵乘法串行时间的因素，因此主要以运行时间来进行评价。

2.2 矩阵乘法的实现

可见代码文件中的 ‘t1.cpp‘。除去其他的辅助函数，矩阵乘法的核心部分如此实现。

```
void product(){
    # pragma omp parallel for collapse(3)
    // 该参数用于设置并行到哪一个级别的循环
    for (int i = 0; i < MATRIX_SIZE; i++){
        for (int j = 0; j < MATRIX_SIZE; j++){
            for (int k = 0; k < MATRIX_SIZE; k++){
                ans[i][j] += lhs[i][k]*rhs[k][j];
            }
        }
    }
}
```

在 OpenMP 编译指令中，可以见到 ‘collapse‘，其后的数字表明并行化的循环层数，根据这个层数，我分为三种情况来研究，这三种情况分别对应题目的三小问。

- 情况一：最外层循环并行化
- 情况二：两层循环并行化
- 情况三：三层循环并行化

2.2.1 对 cache 优化的简单矩阵乘法

考虑到计时不准的缘故，尽量的减少 I/O 时间能够提高实验结果的准确度。因此我对该矩阵乘法做了一些不影响并行度的对 cache 友好的优化，再重复进行了一次实验。

优化后的代码如此实现, 通过将第二层循环和第三层循环交换位置，提高该矩阵乘法的空间局部性，从而减少程序 cache 不命中的情况。

```
void product(){
    # pragma omp parallel for collapse(3)
    // 该参数用于设置并行到哪一个级别的循环
    for (int i = 0; i < MATRIX_SIZE; i++){
        for (int k = 0; k < MATRIX_SIZE; k++){
            for (int j = 0; j < MATRIX_SIZE; j++){
                ans[i][j] += lhs[i][k]*rhs[k][j];
            }
        }
    }
}
```

2.3 实验过程

当运行线程数控制在 1-850 范围中时,运行时间随线程数量变化的图如图 2.1所示。

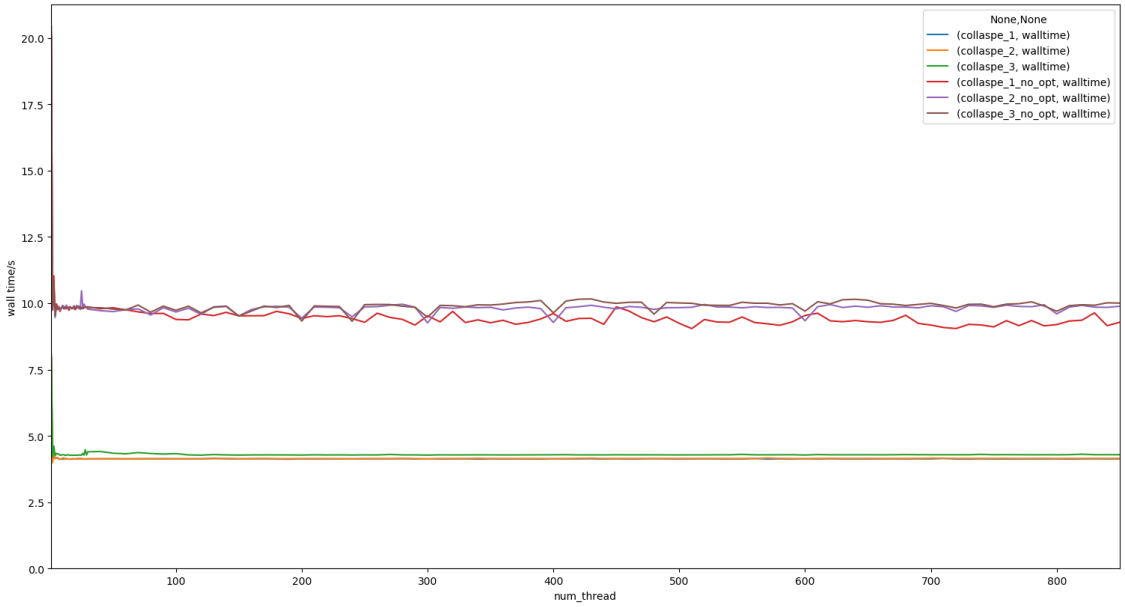

图 2.1 运行时间与线程数量之间的关系

为了更清晰的展现实验结果，当运行线程数控制在 1-30 范围中时，运行时间随线程数量变化的图如图 2.2所示。

2.4 实验结果

- 在对 cache 没有优化的矩阵乘法中，运行时间波动大，从中反映的运行时间规律不明确。相对而言，对 cache 优化后的矩阵乘法总体运行时间波动较小，反映的规律较为可信。
- 从对 cache 优化过的矩阵乘法得出的结果可知，三层循环都并行化时，运行时间总是会比其他两种情况长约 3.5

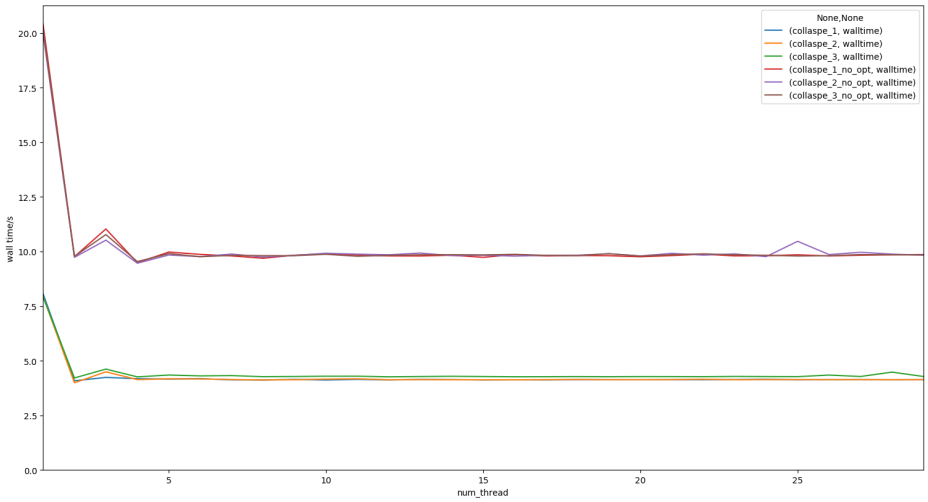

图 2.2　运行时间与线程数量的关系

附录 A　文件的组织

```
.
    figure
        2018-05-06-16-50-54.png
        2018-05-06-16-51-58.png
        collapse_1-3_1200_all_to-850.eps
        collapse_1-3_1200_no_opt.eps
        collapse_1-3_1200_no_opt_to-850.eps
        collapse_1-3_1200_opt.eps
        collapse_1-3_1200_opt_to-850.eps
    HW3-1
        a.exe
        all_time.csv
        a.out
        b.exe
        figure
            all_2.png
            all.png
            all_to10000.png
            dynamic_chunk_num100-2000.png
            dynamic_chunk_num.png
            gueded_chunk_num1-500.png
            guided_chunk_num1-1500.png
            guided.png
        loop-parallel-schedule.cpp
        read_data.py
        readme.md
        run.sh
        save.cpp
    HW3-2
        all_time.csv
        matrix_product.cpp
        read_data.py
        readme.md
        run_all.sh
        run.sh
    report.pdf
```