并行与分布式计算作业

第四次作业

姓名：施天予

班级：计科2班

学号：19335174

一、问题描述

利用 Culler并行程序设计方法计算1000x1000的矩阵与1000x1的向量之间的乘积, 要求清晰地呈现 Culler 并行程序设计的四个步骤，并比较程序在不同阶段具有不同配置时如不同的子任务数量、不同的线程数量、不同的映射方案的性能差别。

二、解决方案

Culler并行程序设计方法

- Decomposition
- Assignment
- Orchestration
- Mapping

串行方法

为比较运行速度，我先实现了串行方法的矩阵乘法，代码如下：

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/time.h>

#define row 1000
#define col 1000
#define GET_TIME(now) { \
    struct timeval t; \
    gettimeofday(&t, NULL); \
    now = t.tv_sec + t.tv_usec/1000000.0; \
}
int matrix[row][col];
int vector[row];
int res[row];
double start, stop;

int main(){
    for (int i = 0; i < col; ++i)
```

```
        vector[i] = rand() % 10;
    for (int i = 0; i < row; ++i)
        for(int j = 0; j < col; ++j)
            matrix[i][j] = rand() % 10;
    GET_TIME(start);
    for (int i = 0;i < row;i++)
        for(int j = 0;j < col; ++j)
            res[i] += matrix[i][j] * vector[j];
    GET_TIME(stop);
    printf("Run time: %e\n", stop-start);
    return 0;
}
```

Decomposition

本次任务我使用的是 OpenMP 共享内存并行框架。对于一个并行程序设计，首先要对问题进行划分。因为原问题是要计算 1000×1000的矩阵与 1000×1的向量之间的乘积，一般来说我们可以将其划分为1000个子任务，每个任务对应结果向量的一个元素。具体代码如下：

```
#   pragma omp parallel for num_threads(thread_count) \
        default(none) private(i, j) shared(matrix, vector, res)
    for (i = 0; i < row; ++i)
        for(j = 0; j < col; ++j)
            res[i] += matrix[i][j] * vector[j];
```

也可以将10个相邻的子任务合成一个，这样共有100个任务。具体代码如下：

```
#   pragma omp parallel for num_threads(thread_count) \
        default(none) private(i, j, k) shared(matrix, vector, res)
    for (k = 0; k < row; k += 10)
        for (i = k; i < k + 10; ++i)
            for (j = 0; j < col; ++j)
                res[i] += matrix[i][j] * vector[j];
```

当然，划分任务的方法还有很多，我选择了这两种比较典型。

Assignment

第二步需要将任务分配给线程，目标是负载均衡，减少通信代价，可以静态或者动态调度。

在OpenMP中，有许多调度方法，如下表格所示：

类型	描述
static	迭代能够在循环执行前分配给线程
dynamic或guided	迭代在循环执行时被分配给线程，在一个线程完成了它的当前迭代集合后，它能从运行时系统中请求更多
auto	编译和运行时系统决定调度方式
runtime	调度在运行时决定

在static调度中，系统静态分配 chunksize 个子任务给每个线程，默认是块调度。

```
#   pragma omp parallel for num_threads(thread_count) \
        default(none) private(i, j) shared(matrix, vector, res, thread_count) \
        schedule(static)
    for (i = 0; i < row; ++i)
        for(j = 0; j < col; ++j)
            res[i] += matrix[i][j] * vector[j];
```

在dynamic调度中，迭代也被分成 chunksize 个连续迭代的块。每个线程执行一块，并且当一个线程完成一块时，它将从运行时系统请求另一块，直到所有的迭代完成。当它被忽略时， chunksize 为1。

```
#   pragma omp parallel for num_threads(thread_count) \
        default(none) private(i, j) shared(matrix, vector, res) \
        schedule(dynamic)
    for (i = 0; i < row; ++i)
        for(j = 0; j < col; ++j)
            res[i] += matrix[i][j] * vector[j];
    GET_TIME(stop);
```

在guided调度中，每个线程也执行一块，并且当一个线程完成一块时，将请求另一块。然而，在guided调度中，当块完成后，新块的大小会变小。

```
#   pragma omp parallel for num_threads(thread_count) \
        default(none) private(i, j) shared(matrix, vector, res) \
        schedule(guided)
    for (i = 0; i < row; ++i)
        for(j = 0; j < col; ++j)
            res[i] += matrix[i][j] * vector[j];
```

在 auto 调度中，系统自动分配子任务给线程。

```
#   pragma omp parallel for num_threads(thread_count) \
        default(none) private(i, j) shared(matrix, vector, res) \
        schedule(auto)
    for (i = 0; i < row; ++i)
        for(j = 0; j < col; ++j)
            res[i] += matrix[i][j] * vector[j];
```

当schedule（runtime）指定时，系统使用环境变量OMP_SCHEDULE在运行时来决定如何调度循环 。

```
#   pragma omp parallel for num_threads(thread_count) \
        default(none) private(i, j) shared(matrix, vector, res) \
        schedule(runtime)
    for (i = 0; i < row; ++i)
        for(j = 0; j < col; ++j)
            res[i] += matrix[i][j] * vector[j];
```

针对不同任务，需要选取合适的调度方法，才能提升性能。

Orchestration

第三步是编排。包括实现通信，必要时增加同步保持依赖性，优化内存数据结构，调度任务。本次我使用的 OpenMP 是共享内存的框架，其通信是隐式的，线程间的交流通过共享内存访问来完成。对于最后的结果 ，因为每个线程只会访问向量中不同的位置，所以也可以设置为共享内存访问，以简单地实现全局通信的功能。

```
#   pragma omp parallel for num_threads(thread_count) \
        default(none) private(i, j) shared(matrix, vector, res)
    for (i = 0; i < row; ++i)
        for(j = 0; j < col; ++j)
            res[i] += matrix[i][j] * vector[j];
```

如上代码所示，指定循环变量 i 和 j 为私有变量，防止各个线程之间的数据竞争，而将矩阵 matrix 和向量 vector 和 res 设为共享变量，使得各个线程之间可以简单地实现通信。 另外本次任务不需要特意添加同步，所以我也未添加过多代码。

Mapping

最后我们要将任务映射到资源上。在 OpenMP 的 static 调度中，有块调度和循环调度两种方法。比如：

- 循环调度 schedule（static，1）
    - Thread 0：0，3，6，9

    - Thread 1：1，4，7，10
    - Thread 2：2，5，8，11
- 块调度 schedule（static，4）
    - Thread 0：0，1，2，3
    - Thread 1：4，5，6，7
    - Thread 2：8，9，10，11

在块调度中，每个线程分配 total iteration / thread_count 个子任务

```
#   pragma omp parallel for num_threads(thread_count) \
        default(none) private(i, j) shared(matrix, vector, res, thread_count) \
        schedule(static, 1000/thread_count)
    for (i = 0; i < row; ++i)
        for(j = 0; j < col; ++j)
            res[i] += matrix[i][j] * vector[j];
```

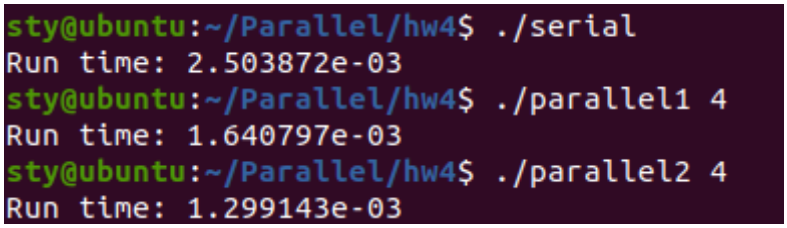
在循环调度中，每个线程完成所分配的子任务后继续分配给该线程相应的任务量，不断循环

```
#   pragma omp parallel for num_threads(thread_count) \
        default(none) private(i, j) shared(matrix, vector, res) \
        schedule(static, 1)
    for (i = 0; i < row; ++i)
        for(j = 0; j < col; ++j)
            res[i] += matrix[i][j] * vector[j];
```

三、实验结果

不同子任务数的性能比较

按照之前Decomposition提到的，我将本次矩阵和向量相乘分别划分成1000个子任务和100个子任务进行比较，以4个线程为例：



可以发现1000个子任务和100个子任务的并行方法都比串行要快，划分为100个子任务要稍快一些。

不同线程数的性能比较

以1000个子任务的版本为例，设置不同的线程数比较运行时间。可以发现开始随着线程数增多运行速度加快，但线程数过多开销也会过大，导致速度甚至比串行还慢。

```
sty@ubuntu:~/Parallel/hw4$ ./parallel1 2
Run time: 1.645088e-03
sty@ubuntu:~/Parallel/hw4$ ./parallel1 4
Run time: 1.543999e-03
sty@ubuntu:~/Parallel/hw4$ ./parallel1 8
Run time: 1.667023e-03
sty@ubuntu:~/Parallel/hw4$ ./parallel1 16
Run time: 2.840996e-03
sty@ubuntu:~/Parallel/hw4$ ./parallel1 32
Run time: 4.914045e-03
```

静态和动态调度的性能比较

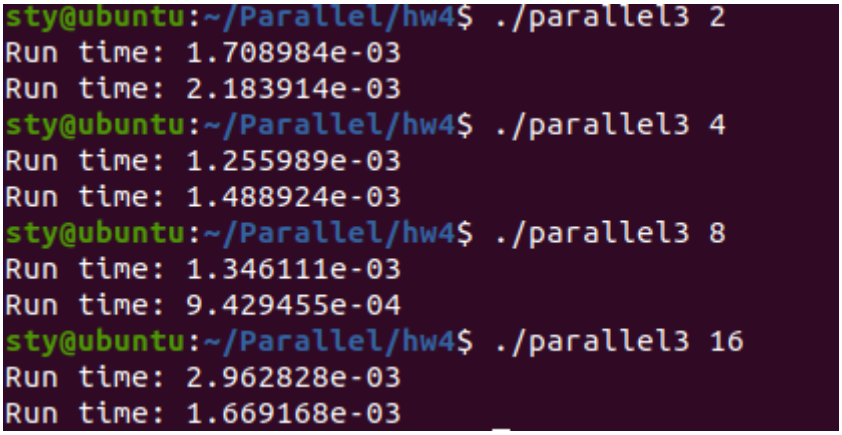
下图中的五种调度方法分别是：static、dynamic、guided、auto、runtime

```
sty@ubuntu:~/Parallel/hw4$ ./parallel4 2
Run time: 1.832008e-03
Run time: 2.661943e-03
Run time: 1.675129e-03
Run time: 2.101898e-03
Run time: 3.540993e-03
sty@ubuntu:~/Parallel/hw4$ ./parallel4 4
Run time: 9.229183e-04
Run time: 2.788782e-03
Run time: 1.088858e-03
Run time: 9.150505e-04
Run time: 2.584934e-03
sty@ubuntu:~/Parallel/hw4$ ./parallel4 8
Run time: 1.502991e-03
Run time: 2.754927e-03
Run time: 5.359650e-04
Run time: 6.580353e-04
Run time: 2.728939e-03
```

可以发现 guided 和 auto 的调度方法在各种线程数时都有不错的效果，static调度性能较好，但 dynamic 和 runtime 调度效果较差，甚至不如串行的速度。

块调度和循环调度的性能比较

块调度给每个线程分配 total iteration / thread_count 个子任务，循环调度中每个线程完成所分配的子任务后继续分配给该线程相应的任务量



可以发现在这次任务中，线程较少时块调度效果更好，线程较多时循环调度效果更好。

调度选择总结

- 如果循环的每次迭代需要几乎相同的计算量，那么可能默认的调度方式能提供最好的性能
- 如果随着循环的进行，迭代的计算量线性递增（或递减），那么采用比较小的chuncksize的static调度可能会提供最好的性能
- 如果每次迭代的开销事先不确定，那么就可能需要尝试使用多种不同的调度策略。这时应使用schedule（runtime），赋予环境变量OMP_SCHEDULE不同的值来比较不同调度策略下程序的性能

四、遇到的问题及解决方法

本次任务较为简单，需要按照 Culler 并行设计方法逐步设计一个并行程序，不过我觉得在 Assignment 和 Mapping 这两个步骤中似乎有一定重叠，可能在更复杂的问题上，处理的方式就不一定相同，这两个步骤就需要分开来设计了。另外，通过对不同子任务数、线程数、调度方法的比较和分析，让我对并行程序的性能有了更好的了解。总而言之，这次任务让我对 Culler 设计方法更加熟悉了，也对我的并行编程能力有很大的提高。