第一题

课件中对各种调度方式的介绍如下：

schedule (static [, chunk])

- Blocks of iterations of size "chunk" to threads
- Round robin distribution
- Low overhead, may cause load imbalance

Best used for predictable and similar work per iteration

schedule (dynamic [, chunk])

- Threads grab "chunk" iterations
- When done with iterations, thread requests next set
- Higher threading overhead, can reduce load imbalance

Best used for unpredictable or highly variable work

schedule (guided [, chunk])

- Dynamic schedule starting with large block
- Size of the blocks shrink; no smaller than "chunk"
- The initial block is proportional to
  - number_of_iterations / number_of_threads
- Subsequent blocks are proportional to
  - number_of_iterations_remaining / number_of_threads

Best used as a special case of dynamic to reduce scheduling overhead when the computation gets progressively more time consuming

编写代码如下：

```
#include<iostream>
#include<unistd.h>
#include<omp.h>
#include<chrono>
using namespace std;
using namespace std::chrono;

#define size 20
#define ThreadNumber 4
```

```
void dummy(){
    sleep(1);
}

int main(){
    auto t1 = system_clock::now();
    #pragma omp parallel for num_threads(ThreadNumber) \
    schedule(dynamic, 2)
    for( int i = 0; i < size; i++ ){
        dummy();
    }
    auto t2 = system_clock::now();
    cout << "Time taken: " << duration_cast<milliseconds>(t2-t1).count() << " ms" <<
endl;
}
```

运行结果：

	Static, 1	Static, 2	Guided, 1	Dynamic, 1	Dynamic, 2
8 loops	2 s	2 s	2 s	2 s	2 s
20 loops	5 s	6 s	5 s	5 s	6 s

如表所示，在循环数量增加时，每次分配一块的调度方案似乎影响不大，但是可以看到如果每次分配的块数大于一个，则可能会出现线程之间任务分配不均衡的问题，导致计算资源的浪费。

第二题

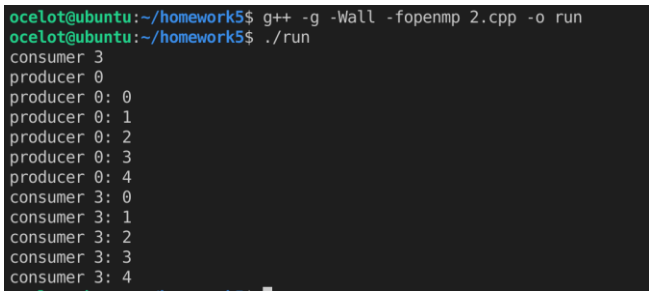
利用 C++标准库中的 queue，实现生产者-消费者模型，代码如下

```
#include <stdio.h>
#include <queue>
#include <omp.h>
#include <unistd.h>
using namespace std;

int producer_num = 2;
int consumer_num = 2;
```

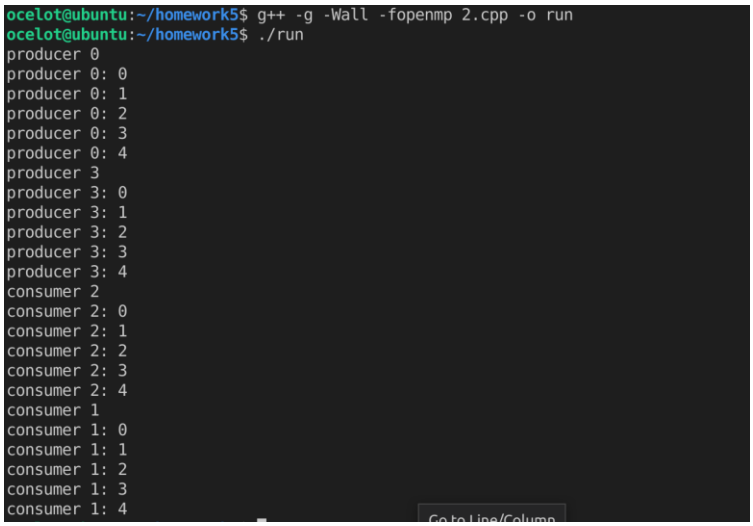
```
omp_lock_t pro_lock, con_lock;

void producer(queue<int> &q, int id){
    int i = 0;
    while ( i < 5 ){
        omp_set_lock(&pro_lock);
        q.push(i);
        printf("producer %d: %d\n", id, i);
        i++;
        omp_unset_lock(&pro_lock);
    }
}

void consumer(queue<int> &q, int id)
{
    int i = 0;
    while ( i < 5 ){
        omp_set_lock(&con_lock);
        while ( q.empty() );
        printf("consumer %d: %d\n", id, q.front());
        q.pop();
        omp_unset_lock(&con_lock);
        i++;
    }
}

int main(){
    queue<int> q;
    omp_init_lock(&pro_lock);
    omp_init_lock(&con_lock);
    omp_set_num_threads(producer_num + consumer_num);
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            sleep(1);
            printf("producer %d\n", omp_get_thread_num());
            producer(q, omp_get_thread_num());
        }
        #pragma omp section
        {
            sleep(1);
            printf("producer %d\n", omp_get_thread_num());
```

```
            producer(q, omp_get_thread_num());
        }
        #pragma omp section
        {
            sleep(1);
            printf("consumer %d\n", omp_get_thread_num());
            consumer(q, omp_get_thread_num());
        }
        #pragma omp section
        {
            sleep(1);
            printf("consumer %d\n", omp_get_thread_num());
            consumer(q, omp_get_thread_num());
        }
    }
    omp_destroy_lock(&pro_lock);
    omp_destroy_lock(&con_lock);
    return 0;
}
```

其中生产者将生产的数据放到队列后面，消费者从队列前面取数据。每个生产者/消费者为一个 section。一个生产者一个消费者输出结果如下：



两个生产者两个消费者输出结果如下：



第三题

先读入矩阵，生成向量再进行并行计算，代码如下：

```
#include<iostream>
#include<fstream>
#include<vector>
#include<time.h>
#include<omp.h>
using namespace std;

int row_size, col_size, num;
//用行压缩方式储存矩阵：
vector<double> val;//大小为 num，存值
vector<int> col_idx;//大小为 num，存对应值的列值
vector<int> row_ptr;//大小为行数，存每一行第一个非零元素在 val 中的索引
int thread_num;

vector<double> mul(vector<double>& vec)
{
    vector<double> result(row_ptr.size(),0);
   clock_t start = clock();
#pragma omp parallel for num_threads(thread_num)//并行,设置了并行线程数
    for (int i = 0; i < row_ptr.size(); i++)
    {
        int end;
        end = i == row_ptr.size() - 1 ? num - 1 : row_ptr[i + 1] - 1;
        for (int m = row_ptr[i]; m <= end; m++)
        {
            result[i] += val[m] * vec[col_idx[m]];//该行的非零值和 vec 的对应单元值相乘，
累加
        }
    }
   clock_t end = clock();
    cout << thread_num<<"级线程并行时间(s)：" << (double)(end - start) / CLOCKS_PER_SEC
<< endl;
    return result;
}

int main()
{
    ifstream f("s3dkt3m2.mtx");
    if (!f)
    {
```

```
        cout << "打开文件失败！" << endl;
        exit(1);
    }
    while (f.peek() == '%')
        while (f.get() != '\n') ;
    f >> row_size >> col_size >> num;
    val.resize(num);
    col_idx.resize(num);
    row_ptr.resize(row_size,0);

    vector<double> vec(col_size);
    int x, y;//x=行,y=列
    double t;//元素值
    int former = -1;
    for (int i = 0; i < num; i++)
    {
        f >> y >> x >> t;//读取每一个单元，且第一个数看成列，第二个数看成行
        if ((x - 1) != former)
        {
            row_ptr[x - 1] = i;//第 x 行的非零元素是 i 开始的
            former = x - 1;
        }
        val[i] = t;
        col_idx[i] = y-1;
    }
    for (int i = 0; i < col_size; i++)
    {
        vec[i] = rand()%100+1;
    }
    cout << "请输入并行的线程数：";
    cin >> thread_num;
    vector<double> result;
   result = mul(vec);
    return 0;
}
```

运行结果：

行/列数	1 thread	2 threads	4 thread	8 thread	16 thread
1138	0.000133 s	0.00864 s	0.003491 s	0.009572 s	0.010152 s
90449	0.030162 s	0.035858 s	0.033172 s	0.041715 s	0.02731 s

如表所示，在线程数量增加时，程序的运行时间波动较大而且并没有明显减少，当矩阵的行数增加到 90449 时依然如此。这可能是由于矩阵依然太小，多线程减少的计算时间还不足以抵消并行的开销导致的。

实验中遇到的问题和解决方法

第二题中使用 section 创建并行线程时，创建出的几个线程 id 相同

请教了老师和助教之后，发现是先创建的线程运行太快导致的，在各个线程运行的 block 最前面加上 sleep，就可以创建出 id 不同的线程了。