并行与分布式计算作业

第五次作业

姓名：施天予

班级：计科2班

学号：19335174

一、问题描述

1. Consider a simple loop that calls a function `dummy` containing a programmable delay （sleep）. All invocations of the function are independent of the others. Partition this loop across four threads using `static`, `dynamic`, and `guided` scheduling. Use different parameters for static and guided scheduling. Document the result of this experiment as the delay within the dummy function becomes large.
2. Implement a producer-consumer framework in OpenMP using sections to create a single producer task and a single consumer task. Ensure appropriate synchronization using locks. Test your program for a varying number of producers and consumers.
3. Consider a sparse matrix stored in the compressed row format (you may find a description of this format on the web or any suitable text on sparse linear algebra). Write an OpenMP program for computing the product of this matrix with a vector. Download sample matrices from the Matrix Market (http://math.nist.gov/MatrixMarket/) and test the performance of your implementation as a function of matrix size and number of threads.

二、解决方案

不同的调度方法

在OpenMP中有许多调度方法，这次我们主要研究其中的三种。

类型	描述
static	迭代能够在循环执行前分配给线程
dynamic	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
guided	编译和运行时系统决定调度方式

首先我们设计dummy函数用于延迟，可以用一个简单的for循环，加大循环次数代表延迟增大

```
void dummy(){
    for (int j = 0; j < 10; ++j);
}
```

在并行循环的设计中，我设置一个100000次的循环，每次循环调用dummy函数，共有4个线程，计算不同调度方法的运行时间

static调度

在static调度中，系统静态分配 chunksize 个子任务给每个线程

设置chunksize为1（循环调度）

```
#   pragma omp parallel for num_threads(4) \
        schedule(static,1)
    for (i = 0; i < 100000; ++i)
        dummy();
```

设置chunksize为25000（块调度）

```
#   pragma omp parallel for num_threads(4) \
        schedule(static, 100000/4)
    for (i = 0; i < 100000; ++i)
        dummy();
```

dynamic调度

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(4) \
        schedule(dynamic)
    for (i = 0; i < 100000; ++i)
        dummy();
```

guided调度

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。

设置chunksize为1（最小块为1）

```
#   pragma omp parallel for num_threads(4) \
        schedule(guided, 1)
    for (i = 0; i < 100000; ++i)
        dummy();
```

设置chunksize为10000（最小块为10000）

```
#   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];
```

生产者消费者问题

生产者消费者是一个经典的同步问题，生产者不断将资源放入，消费者不断消耗。可以用一个队列来进行模拟，生产者放入队列，消费者取出队列。然而，如果多个生产者、消费者一起操作，就可能产生数据竞争，所以需要用信号量设置临界区保护队列的操作。另外，如果消费者试图从空队列取出资源，也会发生错误，所以需要另一个信号量判断队列是否为空。

首先建立队列，设置两个信号量。

```
queue<int> resources;
sem_t n ,s;
```

生产者不断产生资源，入队的代码需要写在临界区内

```
void producer(int id){
    int x;
    if (resources.empty())
        x = 1;
    else
        x = resources.back() + 1;
    if (x == MaxSize)
        return;
    sem_wait(&s);
    resources.push(x);
    sem_post(&s);
    sem_post(&n);
    printf("%d is produced by thread %d.\n", x, id);
}
```

消费者不断消耗资源，出队的代码也要写在临界区内

```
void consumer(int id){
    sem_wait(&n);
    sem_wait(&s);
    int x = resources.front();
    resources.pop();
    sem_post(&s);
    printf("%d is consumed by thread %d.\n", x, id);
}
```

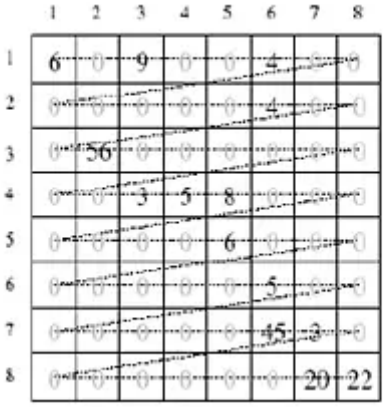
利用omp parallel sections创建section，先获取线程号，然后使用两个omp section分别对应生产者和消费者，实现任务并行。需要注意信号量s要设置为1，这样就能实现同步：使producer先进行，consumer不会从空队列取出。

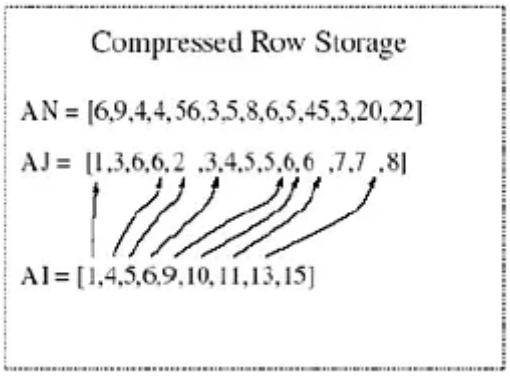
```
int main() {
    sem_init(&n, 0, 0);
    sem_init(&s, 0, 1);
    int producers = 4;
    int consumers = 2;

#   pragma omp parallel num_threads(producers+consumers)
    {
        while (true) {
            int id = omp_get_thread_num();
#           pragma omp parallel sections
            {
#               pragma omp section
                {
                    if (id < producers)
                        producer(id);
                }
#               pragma omp section
                {
                    if (id >= producers)
                        consumer(id);
                }
            }
        }
    }
}
```

稀疏矩阵乘法

压缩行储存模式(Compressed Sparse Row Matrix, CSR)，要求稀疏矩阵按照行进行储存（行主序），每一行的元素顺序可以打乱，仅记录每一行元素的起始位置，具有不错的压缩效率。具体而言，采用三个一维数组，分别存放行指针（各行非零元素的起始位置/列下标）、列下标以及非零元素。通过行指针指向每一行元素的列下标起始

位置，相邻行指标的元素（k和k+1）相减可得到该行（k）的非零元素数目，而自身的下标代表对应的行序号。





Matrix Market中的HB格式类似于CSR压缩行存储格式，开头是一些附加信息，第三行含有行数n、列数 m、非零元素个数total

```
Driven Cavity   5 x  5     Re = 500.                                    E05R0500
        2418           15          293         1952          158
RUA                   236          236         5856            1
(16I5)          (20I4)         (3D23.15)       (3D23.15)
FNX                      1             0
```

接下连续n个数就是行指针：记录每一行元素的起始位置，比如下面的11代表矩阵第2行的元素从第11个元素开始

```
    1   11   21   42   63   84  105  148  191  199  207  222  237  263  289  319
  349  401  453  465  477  489  504  519  545  571  601  631  683  735  747  759
  771  786  801  823  845  875  905  949  993 1005 1017 1029 1040 1051 1073 1095
 1103 1111 1119 1134 1149 1179 1209 1235 1261 1313 1365 1377 1389 1401 1422 1443
 1479 1515 1551 1587 1649 1711 1729 1747 1765 1786 1807 1843 1879 1915 1951 2013
 2075 2093 2111 2129 2150 2171 2201 2231 2267 2303 2355 2407 2425 2443 2461 2476
 2491 2517 2543 2555 2567 2579 2594 2609 2639 2669 2695 2721 2773 2825 2837 2849
 2861 2882 2903 2939 2975 3011 3047 3109 3171 3189 3207 3225 3246 3267 3303 3339
 3375 3411 3473 3535 3553 3571 3589 3610 3631 3661 3691 3727 3763 3815 3867 3885
 3903 3921 3936 3951 3977 4003 4015 4027 4039 4054 4069 4099 4129 4151 4173 4217
 4261 4273 4285 4297 4318 4339 4375 4411 4441 4471 4523 4575 4593 4611 4629 4650
 4671 4707 4743 4773 4803 4855 4907 4925 4943 4961 4982 5003 5033 5063 5093 5123
 5167 5211 5229 5247 5265 5280 5295 5317 5339 5351 5363 5375 5386 5397 5419 5441
 5449 5457 5465 5480 5495 5521 5547 5559 5571 5583 5598 5613 5639 5665 5677 5689
 5701 5716 5731 5753 5775 5787 5799 5811 5822 5833 5841 5849 5857
```

然后是total个列下标，结合前面的行指针可以得到矩阵中每个非零元素的位置

```
 7   5   3   1   8   6   4   2   9  10   1   7   5   3   8   6   4   2   9  10
 1   2   7   5   8   3   6   4   9  10  17  15  13  11  18  16  14  12  19  20
21   1   2   3   7   5   8   6   4   9  10  17  15  13  11  18  16  14  12  19
20  21   1   2   3   4   7   8   6   5   9  10  57  55  53  51  58  56  54  52
59  60  61   1   2   3   4   5   7   8   6   9  10  57  55  53  51  58  56  54
52  59  60  61   1   2   3   4   5   6   8   7  58   9  10  53  18  13  17  15
72  11  57  16  14  12  19  20  21  55  51  71  56  54  52  59  60  61  68  66
64  62  69  67  65  63  70   1   2   3   4   5   6   7  72  16  71  18   9   8
10  17  15  13  11  55  14  12  19  20  21  53  51  56  57  54  52  59  60  61
58  70  63  62  66  64  68  69  67  65   1   2   3   4   5   6   7   8   1   2
 3   4   5   6   7   8   3   4   7   8  15  17  18  16  13  14  11  12  19  20
21   3   4   7   8  11  15  17  18  16  13  14  12  19  20  21   3   4   7   8
17  15  11  16  18  12  14  13  27  19  20  21  26  28  25  24  22  23  30  31
32  29   3   4   7   8  11  17  15  18  12  16  13  26  14  19  20  21  28  24
22  27  29  25  23  30  31  32   3   4   7   8  72  17  71  70  13  18  11  12
```

最后就是所有非零元素的具体值了

```
-0.885491220781795D+00 -0.122926380312671D+01 -0.212596013798501D+01
 0.705873818047171D+01 -0.429509542366312D+00 -0.137072149098932D+00
 0.266457845993479D+00  0.560353309194897D+00  0.177777768505930D+00
-0.132454768507484D-08  0.169560436876827D+01 -0.327829223831255D+00
 0.113450235118379D-01  0.113272720184073D+01 -0.675165666943212D+00
 0.105756911879067D+01 -0.355544421738035D+01  0.100079284963375D+02
 0.132454758619560D-08  0.177777779102311D+00  0.355443920363766D+00
 0.266457845993478D+00 -0.105559720291029D+01 -0.411705060267517D+00
 0.121476637277097D+00  0.392412266711404D+01  0.459379346522556D+00
 0.138691951476501D+01  0.444444421264772D-01  0.132454768594220D-08
 0.388145190208244D+00 -0.118462688406886D+01  0.137657626224471D+01
-0.456934984895070D+01  0.103865395500474D+00 -0.383371542817753D+00
-0.370892756933356D+00  0.372501352107854D+00 -0.198682152674490D-08
 0.444444421264772D-01  0.132454768334012D-08  0.113272720184073D+01
-0.107404015903157D+01 -0.213987840788801D+01 -0.508565982708382D+00
 0.561059665057509D+00  0.933635201188266D+00 -0.201379466692504D+00
 0.780921084502725D+01  0.331136904788837D-09 -0.888888888888817D-01
 0.107249431688977D+00 -0.903150806926197D+00 -0.210140521585094D-02
```

对于稀疏矩阵乘法，我们先定义结构体代表矩阵元素

```c
typedef struct Element{
    int row, col;
    double val;
}Element;
```

首先从命令行读取线程数和文件名，指定缓冲区大小BUFSIZE为100，再过滤掉一些辅助信息后，读取行数n、列数m、非零元素个数total。然后读取行指针和列下标，分别存在两个vector里。

```c
int main(int argc, char *argv[]) {
    char buf[BUFLEN];
    int n, m, total, t;
    int thread_num = atoi(argv[1]);
    char* filename = argv[2];
    FILE* fp = fopen(filename,"r");
```

```
    fgets(buf, BUFLEN, fp);
    fgets(buf, BUFLEN, fp);
    fgets(buf, BUFLEN, fp);
    sscanf(buf, "%*s%d%d%d", &n, &m, &total);
    fgets(buf, BUFLEN, fp);
    fgets(buf, BUFLEN, fp);

    vector<int> rid;
    vector<int> cid;
    for (int i = 0; i < n; ++i) {
        fscanf(fp, "%d", &t);
        rid.push_back(t);
    }
    fscanf(fp, "%*d");
    for (int i = 0; i < total; ++i) {
        fscanf(fp, "%d", &t);
        cid.push_back(t);
    }
```

构建三个矩阵A、B、C，每个元素都是结构体Element类型。接下来读取每个非零元素的值value，并找出对应的行下表和列下标，组成三元组存进结构体内，这样矩阵A和B就成功构建完成了。

```
    vector<vector<Element>> A(m+1);
    vector<vector<Element>> B(n+1);
    vector<vector<double>> C(n+1);
    for (int i = 1; i <= n; ++i)
        C[i].resize(m+1);
    for (int i = 1; i <= n; ++i)
        for (int j = 1; j <= m; ++j)
            C[i][j] = 0.0;

    int col, row = 0;
    int rowptr = 1;
    for (int i = 0; i < 12; ++i) {
        char temp[30];
        double value;
        col = cid[i];
        fscanf(fp, "%s", temp);
        value = atof(temp);
        if (i >= rowptr-1) {
            row++;
            rowptr = rid[row];
        }
        Element e1 = {row, col, value};
        Element e2 = {col, row, value};
        A[col].push_back(e1);
        A[row].push_back(e2);
```

```
        B[row].push_back(e1);
        B[col].push_back(e2);
    }
```

运行矩阵运算的函数，最后输出运行时间

```
    clock_t begin = clock();
    compute(rows, cols, A, B, C, thread_num);
    clock_t end = clock();
    double runtime = (double)(end - begin) / CLOCKS_PER_SEC;
    printf("Runime: %lf\n", runtime);
    return 0;
}
```

矩阵运算的函数如下，使用parallel for语句并行化，指定线程数num_threads，对向量的下标进行遍历，计算结果矩阵。为了防止发生数据竞争，所以使用#pragma omp atomic进行赋值。

```
void compute(int row, int col, vector<vector<Element>>& A,
vector<vector<Element>>& B,
            vector<vector<double>>& C, int thread_num) {
#   pragma omp parallel for num_threads(thread_num)
    for (int k = 1; k <= col; ++k)
        for (int i = 0; i < (int)A[k].size(); ++i)
            for (int j = 0; j < (int)B[k].size(); ++j) {
#               pragma omp atomic
                C[A[k][i].row][B[k][j].col] += A[k][i].val * B[k]
[j].val;
            }
 }
```

三、实验结果

不同的调度方法

对于每种调度方法，为防止某些运行的随机性，我都使用100次循环计算运行时间的平均值，统计结果

其中纵坐标是不同的调度方法，横坐标是dummy函数中的循环的次数

	10	100	1000
static调度chunksize=1	438μs	4425μs	40652μs
static调度chunksize=25000	326μs	4432μs	40393μs
dynamic调度	2375μs	4649μs	39663μs
guided调度chunksize=1	288μs	4325μs	40126μs
guided调度chunksize=10000	320μs	4887μs	45936μs

1. static调度中，在延迟比较小时chunksize=25000的块调度相比chunksize=1的循环调度加速效果更明显，随着延迟的增大，运行时间增加，chunksize的影响不再那么大了。
2. dynamic调度，在延迟比较小时运行速度相比其他调度大很多，可能是因为其线程完成迭代再申请下一个，开销比较大，然而当延迟最大时它的运行速度是最快的，此时彰显了它的优势。
3. guided调度普遍效果比较好，在chunksize=1时任何延迟的加速效果都比较出色，可见这种从大块到小块的调度方法效果好、开销低；然而当chunksize=10000时它的效果就没有那么好了，因为本身迭代就有100000次，其最小块是10000的话就不能体现大块到小块的变化优势，当延迟较大时，运行速度甚至要比其它方法都慢许多。

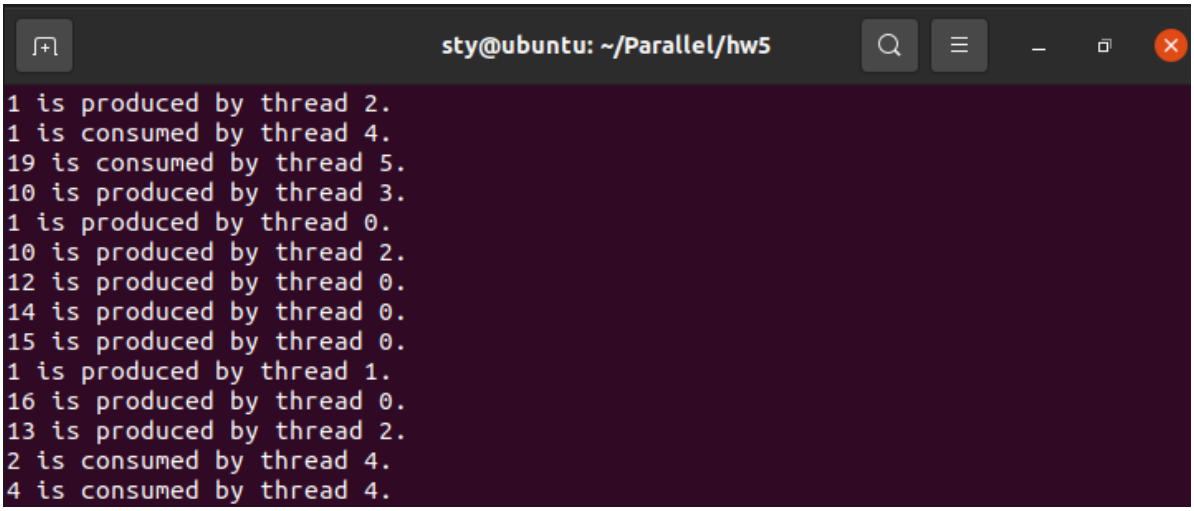
总而言之

- static调度适合迭代次数可预测和任务较均衡的问题
- dynamic调度最适合不可预测或高度可变的工作
- guided调度是动态调度的特殊情况，在计算逐渐消耗更多时间时减少调度开销

生产者消费者问题

令生产者数量为4，线程号是0-3；令消费者数量为2，线程号是4-5。

可以发现，消费者不断产生资源放入队列，消费者不断消耗资源取出队列。经过多次试验，改变生产者和消费者的数量，结果正常。



稀疏矩阵乘法

矩阵行数\线程数	1	2	4	8	16
236	252μs	236μs	228μs	209μs	3792μs
685	781μs	443μs	301μs	274μs	4382μs
1138	827μs	545μs	403μs	285μs	4267μs

我从Matrix Market上分别下载了三个维度为236、685、1138的矩阵，线程数设置1、2、4、8、16。可以发现当线程数设置为8时并行效果最佳，而当线程数超过CPU本身线程数12时运行时间就会大幅增长，线程开销过大，效果甚至比串行还差。另一方面，矩阵越大，加速效果越明显，说明问题规模扩大时，并行化效果越来越好。

四、遇到的问题及解决方法

- 在比较不同调度方法时每一次运行时间结果变化很大，难以比较分析

  直接重复100次实验，结果相对来说比较稳定。

- 生产者消费者问题运行过程中程序暂停

  后来发现是因为我为了设置最大资源数的数量，想让生产者生产的资源id小于20时先不继续生产，而我把这段代码写进了临界区内，导致其它代码一直无法进入临界区。将判断生产者生产的资源id小于20的代码写在临界区外后，问题解决。

- 稀疏矩阵格式错误

  开始我没有看清题目要用压缩行存储格式的矩阵，直接使用了Matrix Market format的三元组形式，好在群里有人提醒，我就修改过来了