11. R中的并行计算III

罗翔宇
中国人民大学统计与大数据研究院

此课件内容基于Norman Matloff编著的《数据科学中的并行计算》（CRC Press，汪磊、寇强译）

上周复习

- 基于共享内存范式的Rdsm包
- myinfo$nwrkrs 线程个数，myinfo$id 当前线程id
- mgrinit()初始化Rdsm包系统
- mgrmakevar()在共享内存中创建变量
- 用snow包中的clusterEvalQ()来开始线程
- 锁：通过锁变量，将读和写看成一个不可分割的组合，即原子化
- 屏障：用来同步所有线程

例子1：时间序列中的最大脉冲

- 考虑一个长度为n的时间序列，我们可能对脉冲感(一个时间段)兴趣，也就是一段时间内保持较高的平均值。我们希望找到长度为k的具有最大平均值的周期
- 时间序列zoo包中有一个函数rollmean(w,k),它返回所有长度为k的块的各个平均值

例子1：时间序列中的最大脉冲

```
maxburst <- function(x, k, max, rslts){
        require(Rdsm)
        require(zoo)

        n <- length(x)
        myidxs <- getidxs(n-k+1)
        myfirst <- myidxs[1]
        mylast <- myidxs[length(myidxs)]
        mas[1, myfirst:mylast] <- rollmean(x[myfirst:(mylast+k-1)], k)

        barr()

        if(myinfo$id == 1){
                rslts[1,1] <- which.max(mas[1, ])
                rslts[1,2] <- mas[1, rslts[1,1]]
        }
}
```

- getidxs(n-k+1)将1到n-k+1根据线程个数分为若干段
- 通过rollmean(x[myfirst:(mylast+k-1)], k)将此段中的周期k的脉冲的平均值计算出来（即长度为k的脉冲的开头分别从myfirst到mylast）
- barr()确保所有线程都执行完了，指派一个线程合并各个进程的结果

例子1：时间序列中的最大脉冲

- 运行所得结果

```
test <- function(cls){
        require(Rdsm)
        mgrinit(cls)
        mgrmakevar(cls, "mas", 1, 9)
        mgrmakevar(cls, "rslts", 1, 2)
        x <<- c(5, 7, 6, 20, 4, 14, 11, 12, 15, 17)
        clusterExport(cls, "maxburst")
        clusterExport(cls, "x")
        clusterEvalQ(cls, maxburst(x, 2, mas, rslts))
        print(rslts[,])
}

c2 <- makeCluster(2)
test(c2)
```

```
> c2 <- makeCluster(2)
> test(c2)
[1]   9 16
```

例子2：变换邻接矩阵

- 假设我们有一个图，它的邻接矩阵是

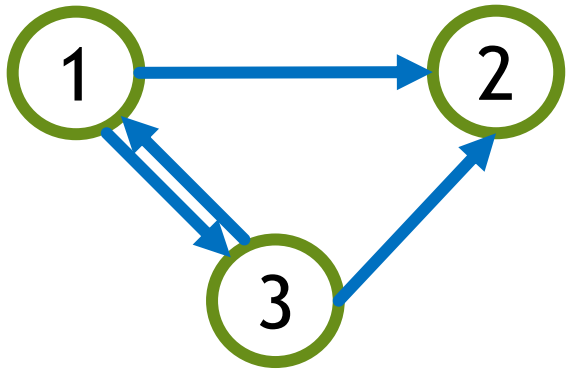
  0 1 1
- 0 0 0
  1 1 0
- 其中，第1行第2列为1，表示顶点1到顶点2有边；第1行第3列为1，表示顶点1到顶点3有边。
- 所对应的图为
- 我们的目标为将其变成一个两列矩阵

  1,2

  1,3

  3,1

  3,2



例子2：变换邻接矩阵

- 代码：

```
convert1row <- function(rownum, colswith1s){
        if(is.null(colswith1s)) return(NULL)
        cbind(rownum, colswith1s)
}
```

- 将某一行的所得1的位置，转化为矩阵形式
- 比如，第3行，在(2, 5, 9)的位置为1，那么返回的结果为

  3,2

  3,5

  3,9

例子2：变换邻接矩阵

```
findlinks <- function(adj, lnks, counts){
        nr <- nrow(adj)

        myidxs <- getidxs(nr)

        myout <- apply(adj[myidxs, ], 1, function(onerow) which(onerow==1))

        tmp <- matrix(nrow=0,ncol=2)
        my1strow <- myidxs[1]

        for(idx in myidxs)
                tmp <- rbind(tmp, convert1row(idx, myout[[idx-my1strow+1]]))

        nmyedges <- Reduce(sum, lapply(myout, length))
        me <- myinfo$id
        counts[1, me] <- nmyedges
        barr()

        if(me == 1) counts[1, ] <- cumsum(counts[1, ])
        barr()

        mystart <- if(me == 1) 1 else counts[1, me-1]+1
        myend <- mystart + nmyedges - 1
        lnks[mystart:myend, ] <- tmp

        0
}
```

- 输入中，adj表示邻接矩阵(n by n)，lnks表示存储的矩阵(n^2 by 2)，counts表示每个线程得到的边的数目

例子2：变换邻接矩阵

```
findlinks <- function(adj, lnks, counts){
        nr <- nrow(adj)

        myidxs <- getidxs(nr)

        myout <- apply(adj[myidxs, ], 1, function(onerow) which(onerow==1))

        tmp <- matrix(nrow=0,ncol=2)
        my1strow <- myidxs[1]

        for(idx in myidxs)
                tmp <- rbind(tmp, convert1row(idx, myout[[idx-my1strow+1]]))

        nmyedges <- Reduce(sum, lapply(myout, length))
        me <- myinfo$id
        counts[1, me] <- nmyedges
        barr()

        if(me == 1) counts[1, ] <- cumsum(counts[1, ])
        barr()

        mystart <- if(me == 1) 1 else counts[1, me-1]+1
        myend <- mystart + nmyedges - 1
        lnks[mystart:myend, ] <- tmp

        0
}
```

- getidxs将邻接矩阵的行分为若干块，每个线程通过apply()处理对应块中边的统计

例子2：变换邻接矩阵

```
findlinks <- function(adj, lnks, counts){
        nr <- nrow(adj)

        myidxs <- getidxs(nr)

        myout <- apply(adj[myidxs, ], 1, function(onerow) which(onerow==1))

        tmp <- matrix(nrow=0,ncol=2)
        my1strow <- myidxs[1]

        for(idx in myidxs)
                tmp <- rbind(tmp, convert1row(idx, myout[[idx-my1strow+1]]))

        nmyedges <- Reduce(sum, lapply(myout, length))
        me <- myinfo$id
        counts[1, me] <- nmyedges
        barr()

        if(me == 1) counts[1, ] <- cumsum(counts[1, ])
        barr()

        mystart <- if(me == 1) 1 else counts[1, me-1]+1
        myend <- mystart + nmyedges - 1
        lnks[mystart:myend, ] <- tmp

        0
}
```

- 在for循环中，每个线程遍历对应块中的每一行，利用convert1row将边的位置转化为一个矩阵，最后组成一个总的矩阵

例子2：变换邻接矩阵

```
findlinks <- function(adj, lnks, counts){
        nr <- nrow(adj)

        myidxs <- getidxs(nr)

        myout <- apply(adj[myidxs, ], 1, function(onerow) which(onerow==1))

        tmp <- matrix(nrow=0,ncol=2)
        my1strow <- myidxs[1]

        for(idx in myidxs)
                tmp <- rbind(tmp, convert1row(idx, myout[[idx-my1strow+1]]))

        nmyedges <- Reduce(sum, lapply(myout, length))
        me <- myinfo$id
        counts[1, me] <- nmyedges
        barr()

        if(me == 1) counts[1, ] <- cumsum(counts[1, ])
        barr()

        mystart <- if(me == 1) 1 else counts[1, me-1]+1
        myend <- mystart + nmyedges - 1
        lnks[mystart:myend, ] <- tmp

        0
}
```

- nmyedges为每个线程所找到的边的个数，将此信息放入counts中。为了将结果汇总在lnks中，通过在线程1中对counts累加，得到mystart

例子2：变换邻接矩阵

```
findlinks <- function(adj, lnks, counts){
        nr <- nrow(adj)

        myidxs <- getidxs(nr)

        myout <- apply(adj[myidxs, ], 1, function(onerow) which(onerow==1))

        tmp <- matrix(nrow=0,ncol=2)
        my1strow <- myidxs[1]

        for(idx in myidxs)
                tmp <- rbind(tmp, convert1row(idx, myout[[idx-my1strow+1]]))

        nmyedges <- Reduce(sum, lapply(myout, length))
        me <- myinfo$id
        counts[1, me] <- nmyedges
        barr()

        if(me == 1) counts[1, ] <- cumsum(counts[1, ])
        barr()

        mystart <- if(me == 1) 1 else counts[1, me-1]+1
        myend <- mystart + nmyedges - 1
        lnks[mystart:myend, ] <- tmp

        0
}
```

- 通过mystart和myend信息，将每个线程对应的结果存入lnks的对应列中

例子2：变换邻接矩阵

```
test <- function(adj, n, cls){
        mgrinit(cls)
        mgrmakevar(cls, "x", n, n)
        mgrmakevar(cls, "lnks", n^2, 2)
        mgrmakevar(cls, "counts", 1, length(cls))

        x[, ] <- adj
        clusterExport(cls, "findlinks")
        clusterExport(cls, "convert1row")
        clusterEvalQ(cls, findlinks(x, lnks, counts))
        print(lnks [1:counts[1, length(cls)], ])
}
```

- 做一个测试，adj为邻接矩阵，n为顶点个数，cls为分配的snow集群

例子2：变换邻接矩阵

- 结果

```
library(parallel)
library(Rdsm)

n <- 5
adj <- matrix(sample(0:1, n^2, replace = TRUE), ncol = n)
lnks <- matrix(NA, n^2, 2)
cls <- makeCluster(2)
test(adj, n, cls)
```

```
> adj
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    0    0    0    1
[2,]    1    0    0    1    1
[3,]    1    0    1    1    1
[4,]    1    0    0    1    1
[5,]    0    0    1    0    0
```

```
> test(adj, n, cls)
      [,1] [,2]
 [1,]    1    5
 [2,]    2    1
 [3,]    2    4
 [4,]    2    5
 [5,]    3    1
 [6,]    3    3
 [7,]    3    4
 [8,]    3    5
 [9,]    4    1
[10,]    4    4
[11,]    4    5
[12,]    5    3
```

例子2：变换邻接矩阵

- 串行代码

```
getlinksnonpar <- function(adj, lnks){
        nr <- nrow(adj)
        myout <- apply(adj, 1, function(rw) which(rw==1))
        nmyedges <- Reduce(sum, lapply(myout, length))
        lnksidx <- 1

        for(idx in 1:nr){
                jdx <- idx
                myoj <- myout[[jdx]]
                endwrite <- lnksidx + length(myoj) - 1
                if(!is.null(myoj)){
                        lnks[lnksidx:endwrite, ] <- cbind(idx, myoj)
                }
                lnksidx <- endwrite + 1
        }
        0
}
```

例子2：变换邻接矩阵

- 串行代码

```
n <- 10000
adj <- matrix(sample(0:1, n^2, replace = TRUE), ncol = n)
lnks <- matrix(NA, n^2, 2)
system.time(getlinksnonpar(adj, lnks))
```

```
> system.time(getlinksnonpar(adj, lnks))
   user  system elapsed
  8.373   0.492   8.867
```

例子2：变换邻接矩阵

- 并行代码

```
cls <- makeCluster(8)
mgrinit(cls)
mgrmakevar(cls, "x", n, n)
mgrmakevar(cls, "lnks", n^2, 2)
mgrmakevar(cls, "counts", 1, length(cls))

x[, ] <- adj
clusterExport(cls, "findlinks")
clusterExport(cls, "convert1row")
system.time(clusterEvalQ(cls, findlinks(x, lnks, counts)))
```

```
   user  system elapsed
  0.002   0.000  53.710
```

- 可能原因为某个线程耽误了太多时间，导致其它线程的长时间等候

例子2：变换邻接矩阵

- 为了给矩阵lnks分配内存来处理最坏的情形，我们浪费了空间和执行时间。问题在于我们并不能提前知道“输出”的大小，也就是lnks的大小
- 在测试代码中，用户通过检查counts[1, length(cls)]来确定lnks中“真实”的行数，把这些“真实”的行复制给另一个矩阵，然后释放大矩阵的内存

例子3：kmeans聚类

- kmeans将数据聚成k个组，现在我们通过Rdsm包进行实现
- 不断迭代以下步骤
- 1.对每个数据点，即数据矩阵中的每一行，确定这个点离哪个中心点最近
- 2.把这个数据点添加到那个中心所在的组内
- 3.更新中心

例子3：kmeans聚类

- 代码

```
kmeans_rdsm <- function(x, k, ni, cntrds, sums, lck, cinit = NULL){
        require(parallel)
        require(pdist)
        nx <- nrow(x)

        myidxs <- getidxs(nx)
        myx <- x[myidxs, ]

        if(is.null(cinit)){
                if(myinfo$id == 1)
                        cntrds[, ] <- x[sample(1:nx, k, replace = F), ]
                barr()
        }else{
                cntrds[, ] <- cinit
        }

        mysum <- function(idxs, myx){
                c(length(idxs), colSums(myx[idxs, , drop=F]))
        }

        for(i in 1:ni){
                if(myinfo$id == 1){
                        sums[] <- 0
                }

                barr()

                #calculate distances
                dsts <- matrix(pdist(myx, cntrds[, ])@dist, ncol = nrow(myx))

                nrst <- apply(dsts, 2, which.min)
```

例子3：kmeans聚类

- x表示数据矩阵，行为样本，列为维度；k表示聚类数目；ni表示迭代次数；cntrds表示中心点矩阵，行为类别，列为维度；sums表示一个矩阵，行为类别，第一列表示对应类别有多少样本量，其余列表示在不同维度上相同类别值的和；lck表示锁变量；cinit表示是否初始化，默认为空
- getidxs每个线程分配到对应块；myx表示线程的对应块
- 若cinit为空，就用随机获得cntrds初始值，由于是共享变量，只需要在线程1中赋值即可。
- 利用barr()让所有线程等待线程1的赋值
- mysum是一个自定义函数。其最终目的是利用tapply函数返回线程所负责那块数据的结果总结。该函数返回为一个向量，第一个元表示对应指标的个数，其余元表示不同维度上值的求和

例子3：kmeans聚类

```
                #update centers
                tmp <- tapply(1:nrow(myx), nrst, mysum, myx)

                realrdsmlock(lck)
                for(j in as.integer(names(tmp))){
                        sums[j, ] <- sums[j, ] + tmp[[j]]
                }
                realrdsmunlock(lck)
                barr()

                if(myinfo$id == 1){
                        for(j in 1:k){
                                if(sums[j, 1] > 0){
                                        cntrds[j, ] <- sums[j, -1] / sums[j, 1]
                                }else{
                                        cntrds[j, ] <<- x[sample(1:nx, 1), ]
                                }
                        }

                }
        }
        0
}
```

- tapply返回的是一个列表，列表的部分1表示线程负责的此块数据中属于类别1的信息，第一元表示该类别中的数据个数，其它元表示该类别的数据在不同维度上的求和
- realrdsmlock()通过锁操作将不同线程的结果汇总在sums，可以避免线程之间的竞争导致错误的结果

例子3：kmeans聚类

- 一个测试函数

```
test <- function(cls){
        mgrinit(cls)
        mgrmakevar(cls, "x", 6, 2)
        mgrmakevar(cls, "cntrds", 2, 2)
        mgrmakevar(cls, "sms", 2, 3)
        mgrmakelock(cls, "lck")

        x[, ] <- matrix(sample(1:20, 12), ncol=2)
        clusterExport(cls, "kmeans_rdsm")
        clusterExport(cls, "pdist")
        clusterEvalQ(cls, kmeans_rdsm(x, 2, 1, cntrds, sms,
                                      "lck", cinit=rbind(c(5, 5), c(15, 15))))
        cntrds[,]
}
```

例子3：kmeans聚类

- 另一个测试函数

```
test1 <- function(cls){
        mgrinit(cls)
        mgrmakevar(cls, "x", 10000, 3)
        mgrmakevar(cls, "cntrds", 3, 3)
        mgrmakevar(cls, "sms", 3, 4)
        mgrmakelock(cls, "lck")

        x[, ] <- matrix(rnorm(30000), ncol=3)
        ri <- sample(1:10000, 3000)
        x[ri, 1] <- x[ri, 1] + 5
        ri <- sample(1:10000, 3000)
        x[ri, 2] <- x[ri, 2] + 5

        clusterExport(cls, "kmeans_rdsm")
        clusterExport(cls, "pdist")
        clusterEvalQ(cls, kmeans_rdsm(x, 3, 50, cntrds, sms, "lck"))
        cntrds[,]
}
```

例子3：kmeans聚类

- 结果

```
library(parallel)
library(Rdsm)
library(pdist)

cls <- makeCluster(2)

test(cls)

system.time(test1(cls))

stopcluster(cls)
```

```
> test(cls)
       [,1] [,2]
[1,]   7.50  5.5
[2,] 12.75 10.5
```

```
> system.time(test1(cls))
   user  system elapsed
  0.049   0.002   1.600
```

共享内存范式：C语言层面

- 直接在多核机器上编程的标准方法是使用线程库，例如在Unix家族的系统上，pthreads库是相当流行的
- 人们开发了专门考虑并行计算的高级库，包括OpenMP, Intel的Threads Building Blocks 以及 Cilk++
- 在这里我们简单介绍下OpenMP，以下内容根据 https://helloacm.com/simple-tutorial-with-openmp-how-to-use-parallel-block-in-cc-using-openmp/

Hello OpenMp

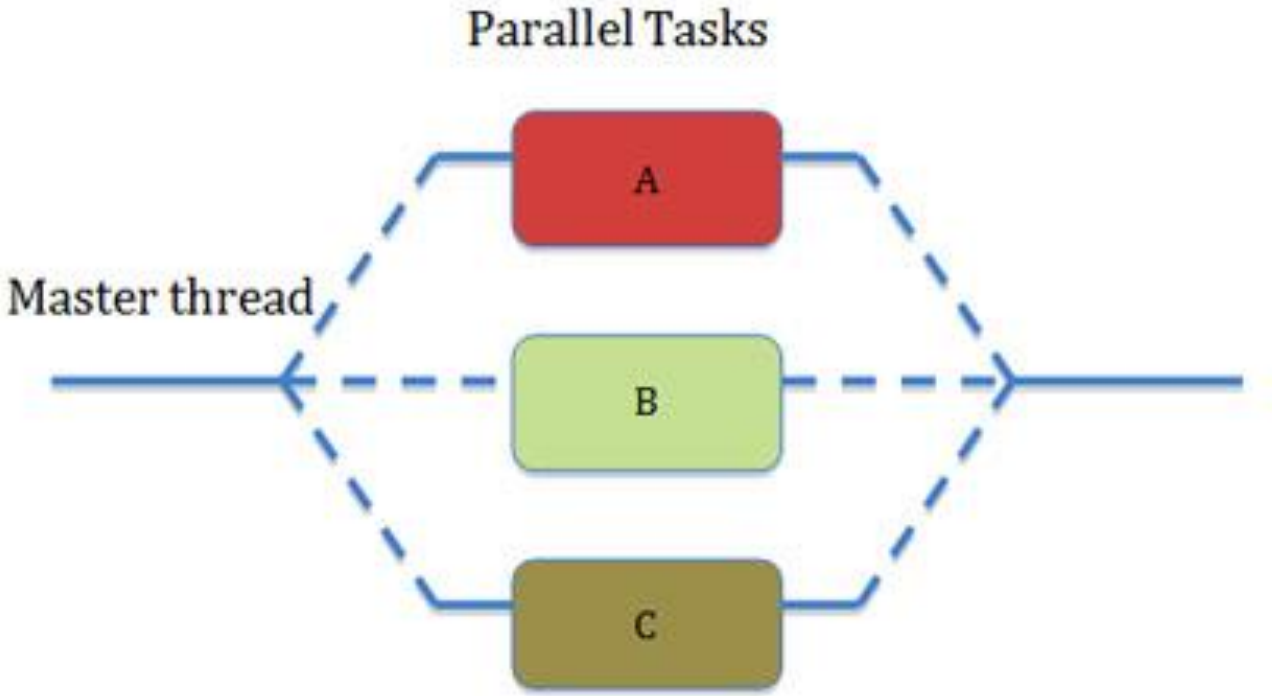
```c
#include <stdio.h>

int main(){
        #pragma omp parallel
        {
                printf("hellow openmp!\n");
        }
        return 0;
}
```

- 编译命令为 gcc -o file file.c -fopenmp

```
[luoxiangyu@rmdx-cluster openmp_files]$ gcc -o hello_openmp hello_openmp.c -fopenmp
[luoxiangyu@rmdx-cluster openmp_files]$ ./hello_openmp
hellow openmp!
hellow openmp!
hellow openmp!
hellow openmp!
hellow openmp!
hellow openmp!
hellow openmp!
```

Hello OpenMp



- #pragma omp parallel用来声明“从下面开始，我要用多个线程同时执行任务了~”
- 根据CPU的核数，一些线程将会生成用来执行任务
- 其实我们也可以设置线程个数

设置线程个数 num_threads()

```c
#include <stdio.h>

int main(){
        int x = 1;
        int y = x + 1;
        #pragma omp parallel num_threads(y*3)
        {
                printf("hellow openmp!\n");
        }
        return 0;
}
```

```
[luoxiangyu@rmdx-cluster openmp_files]$ ./hello_openmp_num_threads
hellow openmp!
hellow openmp!
hellow openmp!
hellow openmp!
hellow openmp!
hellow openmp!
```

获得每个线程的信息

```
#include <stdio.h>
#include <omp.h>

int main(){
        #pragma omp parallel num_threads(3)
        {
                int id = omp_get_thread_num();
                int data = id;
                int total = omp_get_num_threads();
                printf("Greetings from process %d out of %d with Data %d\n",
                        id, total, data);
        }
        printf("parallel for ends.\n");
        return 0;
}
```

- 需要用到头文件 omp.h
- omp_get_thread_num()得到每个线程的id
- omp_get_num_threads()得到线程的总个数

获得每个线程的信息

```
#include <stdio.h>
#include <omp.h>

int main(){
        #pragma omp parallel num_threads(3)
        {
                int id = omp_get_thread_num();
                int data = id;
                int total = omp_get_num_threads();
                printf("Greetings from process %d out of %d with Data %d\n",
                        id, total, data);
        }
        printf("parallel for ends.\n");
        return 0;
}
```

```
[luoxiangyu@rmdx-cluster openmp_files]$ ./num_threads
Greetings from process 2 out of 3 with Data 2
Greetings from process 0 out of 3 with Data 0
Greetings from process 1 out of 3 with Data 1
parallel for ends.
```

- 请注意，id, data, total是在每个线程中定义，不为共享变量

设置共享变量以及私有(private)变量

```
#include <stdio.h>
#include <omp.h>

int main(){
        int data;
        #pragma omp parallel num_threads(3)
        {
                int id = omp_get_thread_num();
                data = id;
                int total = omp_get_num_threads();
                printf("Greetings from process %d out of %d with Data %d\n",
                        id, total, data);
        }
        printf("parallel for ends.\n");
        return 0;
}
```

```
Greetings from process 2 out of 3 with Data 0
Greetings from process 1 out of 3 with Data 1
Greetings from process 0 out of 3 with Data 0
parallel for ends.
```

- 在这里data为共享变量。是否可以将共享变量转为私有变量呢

设置共享变量以及私有(private)变量

```
#include <stdio.h>
#include <omp.h>

int main(){
        int data, id, total;
        //each thread has its own copy of data, id and total
        #pragma omp parallel private(data, id, total) num_threads(6)
        {
                id = omp_get_thread_num();
                data = id;
                total = omp_get_num_threads();
                printf("Greetings from process %d out of %d with Data %d\n",
                        id, total, data);
        }
        printf("parallel for ends.\n");
        return 0;
}
```

- 通过private()的形式，设置已声明好的变量为私有变量

```
Greetings from process 2 out of 6 with Data 2
Greetings from process 5 out of 6 with Data 5
Greetings from process 0 out of 6 with Data 0
Greetings from process 3 out of 6 with Data 3
Greetings from process 1 out of 6 with Data 1
Greetings from process 4 out of 6 with Data 4
parallel for ends.
```

设置临界区域

```
#include <stdio.h>
#include <omp.h>

int main(){
        int data=0;

        #pragma omp parallel num_threads(20)
        {
                data = data + 1;
        }
        printf("data = %d.\n", data);
        return 0;
}
```

```
[luoxiangyu@rmdx-cluster openmp_files]$ ./crtical_section
data = 7.
```

得到的结果并不为20

设置临界区域

```
#include <stdio.h>
#include <omp.h>

int main(){
        int data=0;

        #pragma omp parallel num_threads(20)
        {
                #pragma omp critical
                {
                        data = data + 1;
                }
        }
        printf("data = %d.\n", data);
        return 0;
}
```

```
[luoxiangyu@rmdx-cluster openmp_files]$ ./crtical_section
data = 20.
```

设置临界区域，使得openmp在区域前后自动设置锁变量，因此得到的结果为20