SONGSHAN LAKE
MATERIALS LABORATORY
松山湖材料实验室

OpenMP并行编程简介

Linux-C

魏祎雯

2020年2月28日

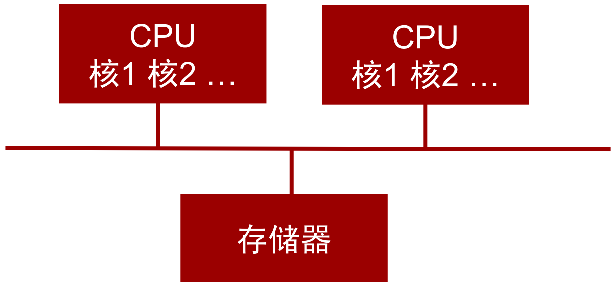
松山湖材料实验室 & 中科院物理研究所

目录

OpenMP
Enabling HPC since 1997

The OpenMP API specification for parallel programming
https://www.openmp.org/

共享内存



OpenMP(Open Multi-Processing) 用于共享内存系统

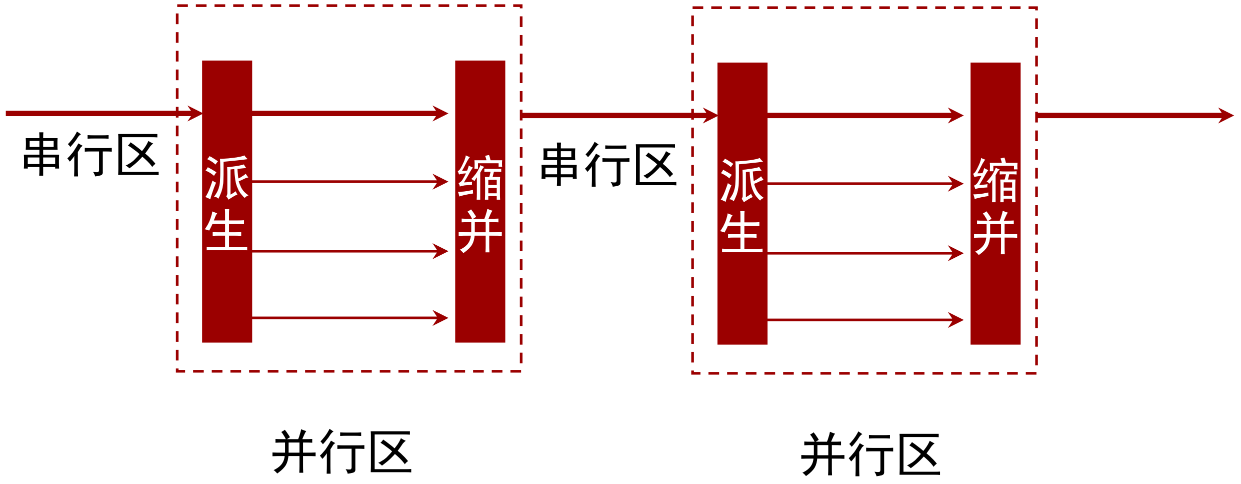
多颗CPU拥有物理上共享的内存

编程语言支持：C,C++,Fortran

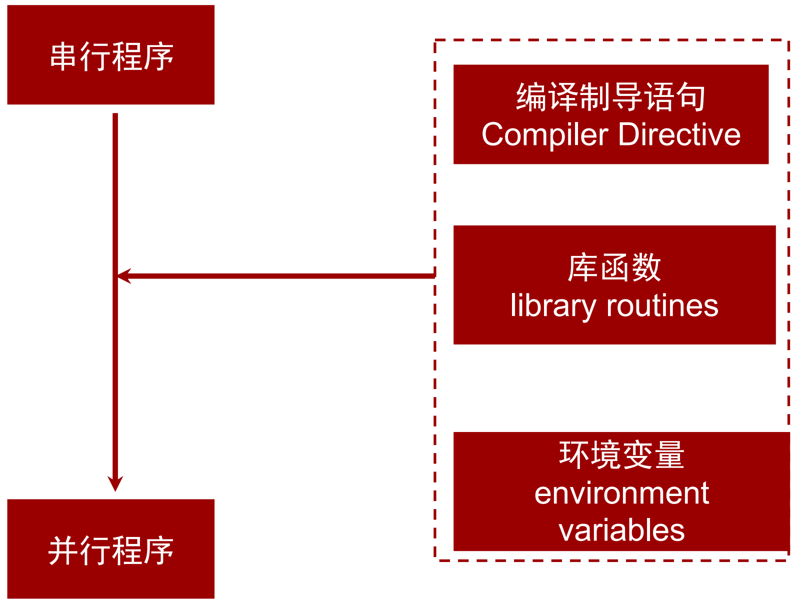
OpenMP 并行执行模式

串行-并行-串行-并行-……

派生/缩并(Fork-Join)模式



OpenMP 并行的实现



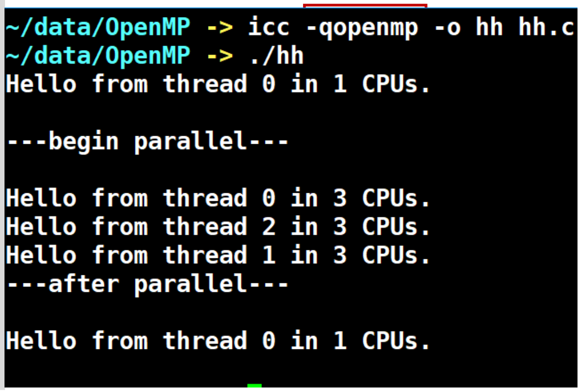
OpenMP简单实例

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

int main()
{
    int tid, mcpu;
    tid = omp_get_thread_num();
    mcpu = omp_get_num_threads();
    printf("Hello from thread %d in %d CPUs.\n\n", tid, mcpu);

    printf("---begin parallel---\n\n");
#pragma omp parallel num_threads(3) private(tid, mcpu)
    {
        tid = omp_get_thread_num();
        mcpu = omp_get_num_threads();
         printf("Hello from thread %d in %d CPUs.\n", tid, mcpu);
    }
    printf("---after parallel---\n\n");

    tid = omp_get_thread_num();
    mcpu = omp_get_num_threads();
    printf("Hello from thread %d in %d CPUs.\n\n", tid, mcpu);

    return 0;
}
```

所需头文件

制导语句

库函数



```
~/data/OpenMP -> icc -qopenmp -o hh hh.c
~/data/OpenMP -> ./hh
Hello from thread 0 in 1 CPUs.

---begin parallel---

Hello from thread 0 in 3 CPUs.
Hello from thread 2 in 3 CPUs.
Hello from thread 1 in 3 CPUs.
---after parallel---

Hello from thread 0 in 1 CPUs.
```

编译制导语句

编译制导语句由3部分构成： 标识符 制导名称 子句

标识符 C/C++:	#pragma omp
制导名称	parallel, for, sections, critical ,atomic ……
子句	private, shared, reduction……

```
#pragma omp parallel [clause]
{
….
}
```

常用指令

用来指导多个CPU共享任务或用来指导多个CPU同步。

parallel 放在一个代码段之前，表示这段代码段将分配给多个线程并行执行。

for   放在for循环之前，将循环分配给多个线程并行执行。

sections 在被分块并行执行的代码段之前。

critical 表明代码只能由一个线程执行，其它线程被阻塞在临界块开始的位置。

atomic 指特定一块内存区域被自动更新

single 用在一段只被单个线程执行的代码段之前，表示后面的代码段将被单线程执行。

……

常用子句

private() 表示变量列表中列出的变量，每个线程都有它自己的变量私有副本。

shared()表示变量列表中列出的变量被所有线程共享

firstprivate() 进入并行区域时，子线程继承主线程同名变量作为初始值

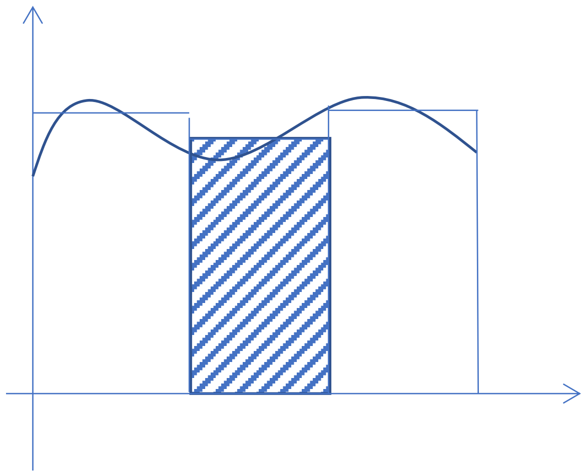
lastprivate() 退出并行区域后，最后一次迭代或最后一个线程中的私有变量复制给主线程中的同名变量

num_threads() 设置并行区域使用的线程数目

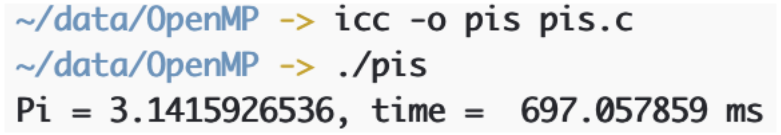
reduction(运算符：变量列表) 指定一个或多个变量是私有变量，并在并行结束后对响应变量执行指定的规约运算，并将结果返回给主线程的同名变量

实例：求Pi | 串行

$$\pi = \int_0^1 \frac{4}{x^2+1}dx$$



Pi= 3.14159265358...



```
~/data/OpenMP -> icc -o pis pis.c
~/data/OpenMP -> ./pis
Pi = 3.1415926536, time =  697.057859 ms
```

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

#define NSTEP 1000000000

int main()
{
    double pi = 0.0;
    int ix;
    double x, step = 1.0 / NSTEP;
    struct timespec tstart, tend;
    clock_gettime(CLOCK_MONOTONIC, &tstart);

    for (ix = 0; ix < NSTEP; ix++)
    {
        x = (ix + 0.5) * step;
        pi += 4.0 / (x * x + 1.0);
    }

    pi = step * pi;

    clock_gettime(CLOCK_MONOTONIC, &tend);
    printf("Pi = %.10lf, time = % f ms\n", pi, 1e3 *
(tend.tv_sec - tstart.tv_sec) + (tend.tv_nsec -
 tstart.tv_nsec) / 1e6);

    return 0;
}
```

指令parallel

并行控制指令

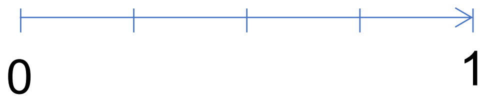
在C/C++中，指令parallel的语法格式如下

```
#pragma omp parallel private( )
                      shared( )
                      reduction( )
                      num_threads( )
                      ……
{
        代码块
}
```

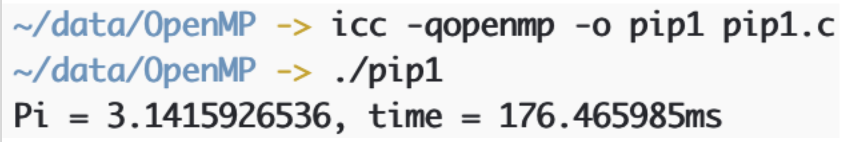
并行实例| parallel

```c
#include <stdio.h>
#include <time.h>
#include <omp.h>
#define NSTEP 1000000000
#define N_THREADS 4
int main()
{
    double pi = 0.0;
    int ix, id;
    double x, step = 1.0 / NSTEP, pithr[N_THREADS];
    struct timespec tstart, tend;
    clock_gettime(CLOCK_MONOTONIC, &tstart);
#pragma omp parallel num_threads(N_THREADS) private(i
x, id, x)
    {
        id = omp_get_thread_num();
        pithr[id] = 0;
        for (ix = id * (NSTEP / N_THREADS); ix < (id
+ 1) * (NSTEP / N_THREADS); ix++)
        {
            x = (ix + 0.5) * step;
            pithr[id] += 4.0 / (x * x + 1.0);
        }
    }
    for (id = 0; id < N_THREADS; id++)
        pi += step * pithr[id];
    clock_gettime(CLOCK_MONOTONIC, &tend);
    printf("Pi = %.16lf, time = %fms\n", pi, 1e3 * (t
end.tv_sec - tstart.tv_sec) + (tend.tv_nsec -
 tstart.tv_nsec) / 1e6);
    return 0;
}
```

$$\pi = \int_0^1 \frac{4}{x^2+1}dx$$



Pi= 3.14159265358...



```
~/data/OpenMP -> icc -qopenmp -o pip1 pip1.c
~/data/OpenMP -> ./pip1
Pi = 3.1415926536, time = 176.465985ms
```

指令critical & atomic

critical 临界块结构

对存在数据竞争的并行区域内的变量进行保护，在同一时间内只允许一个线程执行critical结构，其它线程必须排队等待执行。

```
#pragma omp critical (名称，可省略 )
{
            代码块
}
```

atomic 原子操作

指令atomic要求一个特定的内存地址必须自动地更新，而不让其它线程对此线程进行写操作，原子操作实际上是一个“微型”的critical指令，atomic只对一个表达式语句有效。

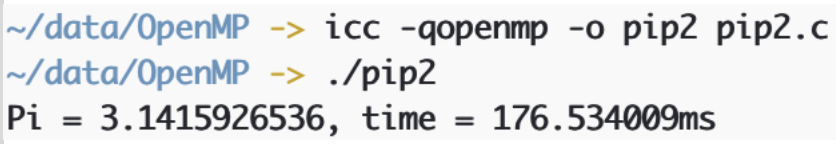
```
#pragma omp atomic
            表达式
```

并行实例| parallel & critical

```
#include <stdio.h>
#include <time.h>
#include <omp.h>
#define NSTEP 1000000000
#define N_THREADS 4
int main()
{
    double pi = 0.0;
    int ix, id;
    double x, step = 1.0 / NSTEP, pithr[N_THREADS];
    struct timespec tstart, tend;
    clock_gettime(CLOCK_MONOTONIC, &tstart);
#pragma omp parallel num_threads(N_THREADS) private(i
x, id, x)
    {
        id = omp_get_thread_num();
        pithr[id] = 0;
        for (ix = id * (NSTEP / N_THREADS); ix < (id
+ 1) * (NSTEP / N_THREADS); ix++)
        {
            x = (ix + 0.5) * step;
            pithr[id] += 4.0 / (x * x + 1.0);
        }
#pragma omp critical
        pi += step * pithr[id];
    }
    clock_gettime(CLOCK_MONOTONIC, &tend);
    printf("Pi = %.16lf, time = % f ms\n", pi, 1e3 *
(tend.tv_sec - tstart.tv_sec) + (tend.tv_nsec -
 tstart.tv_nsec) / 1e6);
    return 0;
}
```

$$\pi = \int_0^1 \frac{4}{x^2+1}\,dx$$

Pi= 3.14159265358...



```
~/data/OpenMP -> icc -qopenmp -o pip2 pip2.c
~/data/OpenMP -> ./pip2
Pi = 3.1415926536, time = 176.534009ms
```

指令for

用指令for并行的前提是所在区域已被parallel 初始化

在C/C++中，指令parallel的语法格式如下

```
 #pragma omp parallel
{
   #pragma omp for  private( )
                    shared( )
                    reduction( )
                    num_threads( )
                        ……
   {
          代码块
   }


}
```

可缩写为

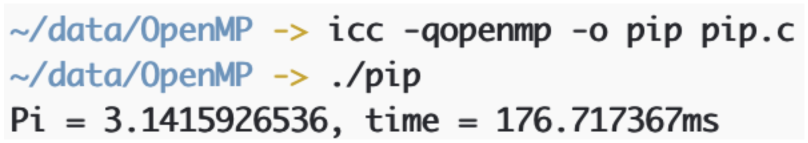
```
#pragma omp parallel for
                private( )
                shared( )
                reduction( )
                num_threads( )
                          ……
{
          代码块
}
```

并行制导| for

```c
#include <stdio.h>
#include <time.h>
#include <omp.h>
#define NSTEP 1000000000
#define N_THREADS 4
int main()
{
    double pi = 0.0;
    int ix, id;
    double x, step = 1.0 / NSTEP, pithr[N_THREADS];
    struct timespec tstart, tend;
    clock_gettime(CLOCK_MONOTONIC, &tstart);
#pragma omp parallel num_threads(N_THREADS) private(
ix,id,x)
    {
        id = omp_get_thread_num();
        pithr[id] = 0;
#pragma omp for
        for (ix = 0; ix < NSTEP; ix++)
        {
            x = (ix + 0.5) * step;
            pithr[id] += 4.0 / (x * x + 1.0);
        }
    }
    for (id = 0; id < N_THREADS; id++)
        pi += step * pithr[id];
    clock_gettime(CLOCK_MONOTONIC, &tend);
    printf("Pi = %.16lf, time = % f ms\n", pi, 1e3 *
 (tend.tv_sec - tstart.tv_sec) + (tend.tv_nsec -
 tstart.tv_nsec) / 1e6);
    return 0;
}
```

$$\pi = \int_0^1 \frac{4}{x^2+1}\,dx$$

Pi= 3.14159265358...



```
~/data/OpenMP -> icc -qopenmp -o pip pip.c
~/data/OpenMP -> ./pip
Pi = 3.1415926536, time = 176.717367ms
```

子句reduction

反复把运算符（累加，累减，累乘等操作）作用在一个变量，并保存在原变量中，称为规约操作。

reduction(运算符：变量列表)

运算类别	运算符
加	+
减	-
乘	*
逻辑与	&&
逻辑或	||
最大值	max
最小值	min
按位与	&
按位或	|
按位异或	^

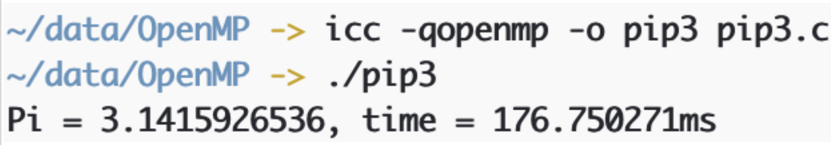
并行开始区域，私有化变量，并行结束区域，对所有副本进行规约计算。

并行制导| for & reduction

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

#define NSTEP 1000000000
#define N_THREADS 4

int main()
{
    double pi = 0.0;
    int ix;
    double x, step = 1.0 / NSTEP;
    struct timespec tstart, tend;
    clock_gettime(CLOCK_MONOTONIC, &tstart);
#pragma omp parallel for num_threads(N_THREADS) priva
te(ix, x) reduction(+:pi)
    for (ix = 0; ix < NSTEP; ix++)
    {
        x = (ix + 0.5) * step;
        pi += 4.0 / (x * x + 1.0);
    }
    pi = step * pi;
    clock_gettime(CLOCK_MONOTONIC, &tend);
    printf("Pi = %.16lf, time = % f ms\n", pi, 1e3 *
(tend.tv_sec - tstart.tv_sec) + (tend.tv_nsec -
 tstart.tv_nsec) / 1e6);
    return 0;
}
```

$$\pi = \int_0^1 \frac{4}{x^2+1}\,dx$$

Pi= 3.14159265358...



```
~/data/OpenMP -> icc -qopenmp -o pip3 pip3.c
~/data/OpenMP -> ./pip3
Pi = 3.1415926536, time = 176.750271ms
```

并行制导| for & critical & atomic

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

#define NSTEP 1000000000
#define N_THREADS 4

int main()
{
    double pi = 0.0;
    int ix;
    double x, step = 1.0 / NSTEP;
    struct timespec tstart, tend;

    clock_gettime(CLOCK_MONOTONIC, &tstart);
#pragma omp parallel for num_threads(N_THREADS)
private(ix, x)
    for (ix = 0; ix < NSTEP; ix++)
    {
        x = (ix + 0.5) * step;
        #pragma omp critical
        pi += 4.0 / (x * x + 1.0);
    }
    pi = step * pi;
    clock_gettime(CLOCK_MONOTONIC, &tend);
    printf("Pi = %.16lf, time = % f ms\n", pi,
1e3 * (tend.tv_sec - tstart.tv_sec) +
(tend.tv_nsec - tstart.tv_nsec) / 1e6);
    return 0;
}
```

$$\pi = \int_0^1 \frac{4}{x^2+1}\,dx$$

Pi= 3.14159265358...

```
~/data/OpenMP -> ./pip4
Pi = 3.1415926536, time = 438227.044148ms
```

$\rightarrow$ `#pragma omp atomic`

```
~/data/OpenMP -> ./pip4
Pi = 3.1415926536, time = 148876.406767ms
```

1. atomic比critical更高效
2. 要注意避免额外的并行开销

指令sections

用指令sections并行的前提是所在区域已被parallel 初始化

在C/C++中，指令parallel的语法格式如下

```
#pragma omp parallel
{
  #pragma omp sections  private( )
                        shared( )
                        reduction( )
                      num_threads( )
                       ……
  {
       #pragma omp section
             结构块
       #pragma omp section
             结构块
       ……
  }

}
```

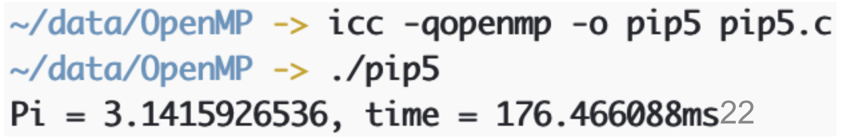
缩写为

```
#pragma omp parallel sections
                   private( )
                   shared( )
                   reduction( )
                   num_threads( )
                               ……
{
      #pragma omp section
            结构块
      #pragma omp section
            结构块
      ……

}
```

并行制导| sections

```
#include <stdio.h>
#include <time.h>
#include <omp.h>
#define NSTEP 1000000000
#define N_THREADS 4
int main()
{
    double pi = 0.0;
    int ix, id, istep = NSTEP / N_THREADS;
    double x, step = 1.0 / NSTEP;
    struct timespec tstart, tend;
    clock_gettime(CLOCK_MONOTONIC, &tstart);
#pragma omp parallel sections num_threads(N_THREADS)
private(id, ix, x) reduction(+: pi)
    {
#pragma omp section
        {
            id = omp_get_thread_num();
            for (ix = id * istep; ix < (id + 1) * istep;
ix++)
            {
                x = (ix + 0.5) * step;
                pi += 4.0 / (x * x + 1.0);
            }
        }
#pragma omp section
        {
            id = omp_get_thread_num();
            for (ix = id * istep; ix < (id + 1) * istep;
ix++)
            {
                x = (ix + 0.5) * step;
                pi += 4.0 / (x * x + 1.0);
            }
        }
#pragma omp section
        {
            id = omp_get_thread_num();
            for (ix = id * istep; ix < (id + 1) * istep;
ix++)
            {
                x = (ix + 0.5) * step;
                pi += 4.0 / (x * x + 1.0);
            }
        }
#pragma omp section
        {
            id = omp_get_thread_num();
            for (ix = id * istep; ix < (id + 1) * istep;
ix++)
            {
                x = (ix + 0.5) * step;
                pi += 4.0 / (x * x + 1.0);
            }
        }
    }
    pi = step * pi;

    clock_gettime(CLOCK_MONOTONIC, &tend);
    printf("Pi = %.10lf, time = %fms\n", pi, 1e3 *
(tend.tv_sec - tstart.tv_sec) + (tend.tv_nsec -
tstart.tv_nsec) / 1e6);

    return 0;
}
```

Pi= 3.14159265358...



```
~/data/OpenMP -> icc -qopenmp -o pip5 pip5.c
~/data/OpenMP -> ./pip5
Pi = 3.1415926536, time = 176.466088ms
```

库函数

OpenMP标准定义了一个应用程序编程接口来调用库中的多个函数。

包含头文件 : #include<omp.h>

常用库函数：

omp_get_num_procs　返回系统处理器数量

omp_set_num_threads 设置并行执行代码的线程个数

omp_get_max_threads 返回当前并行区域内可用的最大线程数量

omp_get_num_threads 确定当前活动线程数量

omp_get_thread_num 返回线程号，0-主线程

omp_get_dynamic 判断是否支持动态改变线程数量

omp_set_dynamic 启用或关闭线程数的动态改变

……

环境变量

通过环境变量控制程序的运行

OMP_DYNAMIC

用来启用或禁用并行区域线程数的动态调整，缺省值true。

```
export OMP_DYNAMIC = false
```

OMP_NUM_THREADS

设置并行区域使用的线程数目

可以使用num_threads子句或调用omp_set_num_threads()覆盖此值。

```
export OMP_NUM_THREADS=4
```

并行效率

并行开销：

- 线程的建立与销毁，线程间的通信，线程间同步造成的开销
- 为争夺共享资源而竞争引起的开销
- 由于各个CPU工作负载分配不均衡等因素，一个或多个线程由于缺少工作任务或因为等待特定事件发生造成的开销

谢谢！