1 并行举例

1.1 遍历像素

- 假设对于每个像素的处理与邻域无关
- 顺序执行代码每次只计算一个像素

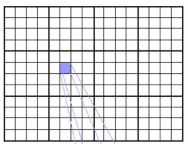

图 1: 遍历像素

代码如下：

```
for( int i=0; i<cols; ++i)
{
    for(int j=0; j<rows; ++j)
    {
        element(i,j) = function(i,j);
    }
}
```

并行改造方案1

- 将任务分块，如：子数组划分(如task1,task2...)
- 不同子数组的处理不需要通信

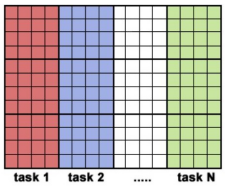

图 2: 并行改造方案1

代码(1)如下：

```
for( int i = task_start; i < task_end; ++i)
{
  for(int j=0; j<rows; ++j)
  {
    element(i,j) = function(i,j);
  }
}
```

如上例，在逻辑上将任务划分为多个小任务并行处理，这些小任务之间没有通信或通信较少，并且不需要任何调度，称之为"尬并行""易并行""尴尬并行"（**embarrasingly parrallel**）

优劣：

- 人工分配任务，操作简单
- 可迅速实现对现有顺序代码的并行改造
- 粒度较粗，容易引起负载不均衡

并行改造方案2

- 主线程分配任务
- 主线程收集子线程的处理结果
- 子线程从主线程领取任务
- 子线程运算，将结果返回给主线程

代码(2)举例：

```
int pid = fork() // 新建线程
if( pid != 0)    // 对于主线程
{
  SendTaskToThreads();
  ReceiverResultFromThreads();
}
else             // 对于子线程
{
```

```
    ReceiveTaskFromMaster();
    DoTask();
    SendResultToMaster();
}
```

优劣：

- 自动分配任务
- 操作较复杂，不适合对现有代码进行并行改造的情景。
- 粒度较细，负载较均衡
- 有额外的运算支出，如通信、任务分配等

1.2 计算PI

- 生成随机点
- 用落在圆内点和所有点的个数近似圆面积和方形面积
- 确定PI值

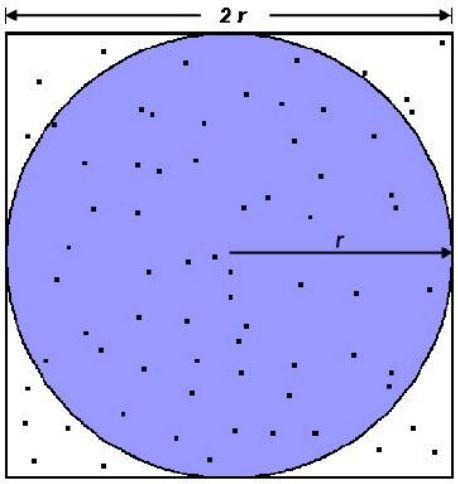

图 3: PI的计算

代码(3)举例：

```
int total_point_number = 10000;
int circle_point_number = 0;

for( int i=0; i < total_point_number; ++i)
{
  int x = random(0,1);
  int y = random(0,1);
  if( x*x + y*y < r*r)
  {
    circle_point_number++;
  }
}

double PI = 4.0*circle_point_number/total_point_number;
```

- 增加随机点个数，提高Pi的运算精度
- 循环导致计算量巨大

计算Pi并行改造1

- 如案例1，并行改造1的思路
- 将总任务分成多个子任务,每个子任务均执行上述操作
- 最后将所有子任务的结果将加

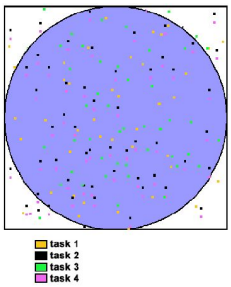

图 4: PI的并行改造

计算Pi并行改造2 代码(4)举例: 如代码2所示，可具体为：

```
#ChildThread:  // 子线程
  GenerateRandomNumber();
  if(InsideCircle())
  {
    circle_count++;
  }
  SendMasterCircleCountAndTotalNumber();

#MasterThread:  // 主线程
  ReceiveCircleCountAndTotalNumber();
  ComputePI();
```

- 基本思路和改造方案1一致
- 任务调度和任务分配交给操作系统
- 负载均衡，优化效果好，但增加额外计算支出