参考文献：[《考虑释放时间和调度顺序的可分任务调度模型》]

题一

题目

一、若考虑处理机的释放时间，即，设从处理机 P_i 的释放时间为r_i（即从开始到时刻 r_i 从处理机 P_i 是非空闲的，从时刻r_i开始空闲，可以给其安排任务）。请：

1. 叙述带有释放时间的同构网络可分任务调度问题；
2. 建立带有释放时间的同构网络可分任务调度问题的数学模型。

带有释放时间的同构网络可分任务调度问题描述

在并行与分布式计算中，**同构网络**是指具有相同计算能力和通信能力的处理机所组成的网络。**可分任务调度问题**指的是将一个可分任务划分为多个子任务并分配到多台处理机，以使得总完成时间最短。

考虑释放时间时，假设每个处理机在某一特定时间点后（释放时间）才可以开始接收任务。这种限制导致任务调度的难度增加，因为需要额外考虑任务开始时间的约束。

问题关键点：

- 系统由一台主处理机和多台从处理机构成。
- 主处理机负责分配任务，从处理机接收并处理任务。
- 每个从处理机具有不同的释放时间，表示从该时刻起处理机变为空闲。
- 目标是设计任务分配策略，确保总完成时间最短。

带释放时间的同构网络可分任务调度问题数学模型

参数定义

- P_0：主处理机。
- $P_i(i=1,2,...,N)$：从处理机。
- W_{total}：总任务量。
- α_i：分配给 PiP_iPi 的任务量。
- r_i：从处理机 PiP_iPi 的释放时间。
- z：链路单位任务传输时间。
- w：处理机单位任务处理时间。
- E、F：分别为通信启动开销和计算启动开销。
- s_i：从处理机 PiP_iPi 开始接收任务的时刻。
- T_f：任务完成时间。

目标

最小化任务完成时间 T_f

约束条件

任务划分约束:

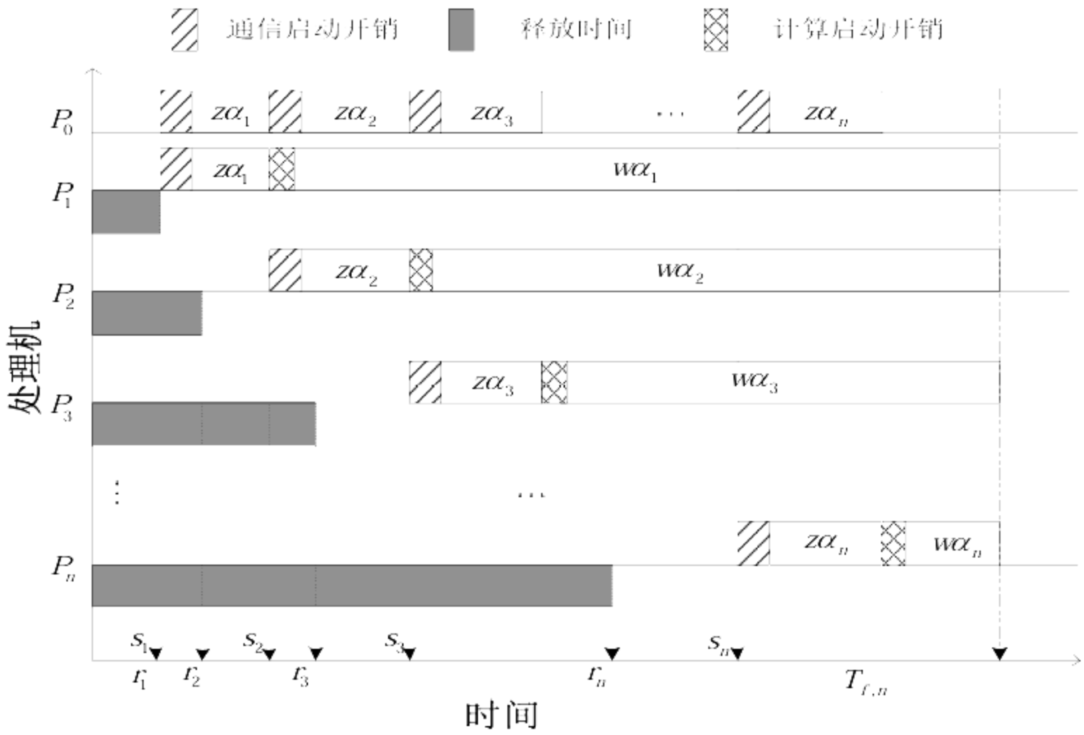
$\sum_{i=1}^{N} \alpha_i = W_{\text{total}}, \quad \alpha_i \geq 0$

时间约束:

对于 P_i 的任务开始时刻 s_i，存在两种可能约束:

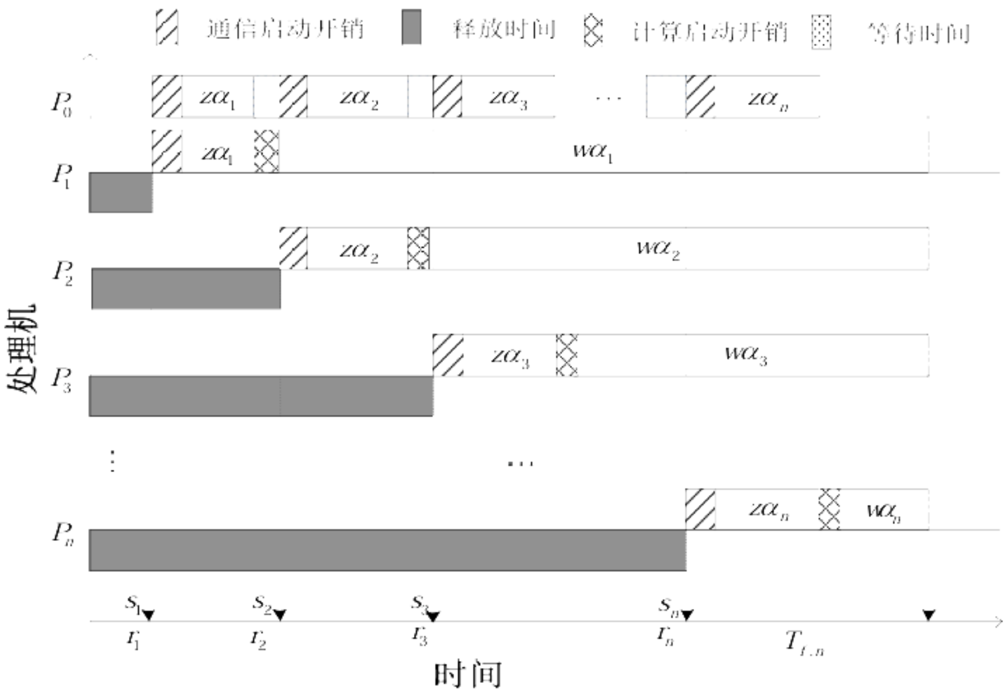
约束条件 I:

如果 $s_{i+1} \leq r_{i+1}$: $s_{i+1} = s_i + E + z\alpha_i$



约束条件 II:

如果 ($s_{i+1} > r_{i+1}$): $s_{i+1} = r_{i+1}$



完成时间约束：

每个从处理机完成任务的时间为：$f_i = s_i + E + z\alpha_i + F + w\alpha_i$

任务完成时间 T_f 为：$T_f = \max\{f_i\}$

优化问题形式： $minT_f$

题二

题目

二、对函数$f(x)$=$\sum_{i=1}^{5}\left[(1-x_i)^2+100(x_{i+1}-x_i^2)^2\right]$，初始点取为$x^0=(0,0,\ldots,0)\in R^6$，分别用最速下降法和牛顿法编程迭代10次，把结果总结在如下形式的一张表里，记录各次迭代的函数值最终 CPU。比较两个方法所得到的结果，并分析结果。

算法实现

最速下降法:
在最速下降法中，搜索方向为 $p_k = -\nabla f(x_k)$，步长通过线搜索确定 α_k。

牛顿法:
在牛顿法中，搜索方向为 $p_k = -H^{-1}\nabla f(x_k)$，步长采用线搜索或单位步长。

输出记录：

- 每次迭代k的函数值
- 每次迭代的时间

用python代码实现

```python
import numpy as np
import time

#定义目标函数
```

```python
def rb_f(x):
    return sum((1 - x[:-1])**2 + 100 * (x[1:] - x[:-1]**2)**2)

#梯度函数
def rb_grad(x):
    grad = np.zeros_like(x)
    grad[:-1] = -2 * (1 - x[:-1]) - 400 * x[:-1] * (x[1:] - x[:-1]**2)
    grad[1:] += 200 * (x[1:] - x[:-1]**2)
    return grad

#目标函数二阶导，构建对称矩阵
def rb_hessian(x):
    n = len(x)
    H = np.zeros((n, n))
    for i in range(n - 1):
        H[i, i] += 2 - 400 * (x[i + 1] - 3 * x[i]**2)
        H[i, i + 1] += -400 * x[i]
        H[i + 1, i] += -400 * x[i]
    H[-1, -1] += 200
    return H

#最速下降法
#沿着函数梯度的负方向更新变量x
def gradient_descent(x0, max_iter=10):
    x = x0.copy()
    results = []
    for _ in range(max_iter):
        start_time = time.time()
        grad = rb_grad(x)
        alpha = 1e-3  #固定步长0.001
        x -= alpha * grad
        f_val = rb_f(x)
        results.append((f_val, time.time() - start_time))
    return results

#牛顿法
#通过Hessian矩阵提供曲率信息更新方向，二次收敛
def newton_method(x0, max_iter=10):
    x = x0.copy()
    results = []
    for _ in range(max_iter):
        start_time = time.time()
        grad = rb_grad(x)
        H = rb_hessian(x)
        try:
            p = np.linalg.solve(H, -grad)
        except np.linalg.LinAlgError:
            p = -grad
        x += p
        f_val = rb_f(x)
        results.append((f_val, time.time() - start_time))
    return results

#初始化
x0 = np.zeros(6)
gd_results = gradient_descent(x0)
```

```python
newton_results = newton_method(x0)

print(f"{'Iteration':<10}{'GD f(x)':<20}{'GD Time (s)':<20}{'Newton f(x)':<20}{'Newton Time (s)':<20}")
for i, (gd, nt) in enumerate(zip(gd_results, newton_results), 1):
    print(f"{i:<10}{gd[0]:<20.10f}{gd[1]:<20.10f}{nt[0]:<20.10f}{nt[1]:<20.10f}")

```

表格结果

迭代次数k	最速下降法	牛顿法
	$f(x^k)$	$f(x^k)$
1	4.981614	1.0×10^{2}
2	4.968427	4.93×10^{-30}
3	4.958531	0.000000
4	4.950723	0.000000
5	4.944244	0.000000
6	4.938612	0.000000
7	4.933517	0.000000
8	4.928762	0.000000
9	4.924221	0.000000
10	4.919811	0.000000
10次迭代两个方法所花时间	0.0005	0.0024

分析

1. **收敛速度**:
   - **牛顿法**在第2次迭代后已接近最优解（接近于零），显示了其利用二阶导数信息的快速收敛特性。
   - **最速下降法**收敛较慢，在10次迭代后函数值仍在下降，显示了其梯度方向选择的局限性。
2. **耗时比较**:
   - 每次迭代牛顿法的时间略高于最速下降法，但差距不大，计算效率较高。
   - 牛顿法在初期可能因 Hessian 矩阵计算占用更多时间，但随着迭代次数增加，其收敛快的优势更明显。

结论

- 牛顿法在函数收敛速度上显著优于最速下降法，但实现较复杂。
- 最速下降法实现简单，但收敛较慢，适合问题简单或对精度要求不高的场景。