

STAT243 Lecture 6.5 Illustrating the Principles in Specific Case Studies

1 场景 1：单模型拟合 (One Model Fit)

具体情境：需要对数据拟合一个统计或机器学习模型，例如随机森林或回归模型

一般情境：并行化单个任务的执行

1.1 场景 1A：模型本身支持并行化

- 一些算法或库本身支持多核并行，只需通过参数启用
- 例如 scikit-learn 中的 `RandomForestClassifier` 可以通过 `n_jobs` 参数控制使用的线程数



Python

```
1 import sklearn.ensemble  
2 help(sklearn.ensemble.RandomForestClassifier)
```

⚠ Remark ↴

可以在查看模型文档时，寻找以下关键词：`threads`, `processes`, `cores`, `cpus`, `jobs` 等

1.2 场景 1B：并行化线性代数 (Parallelized Linear Algebra)

1.2.1 BLAS 与 LAPACK 简介

- BLAS (Basic Linear Algebra Subprograms)**
用于基础线性代数运算的底层库 (Fortran/C 实现)
- LAPACK (Linear Algebra PACKage)**
构建在 BLAS 之上，提供更复杂的矩阵分解算法

1.2.2 常见的高性能 BLAS 实现

名称	特点	许可
Intel MKL	免费教育许可，高度优化	免费 (教育用途)
OpenBLAS	开源，跨平台	免费
Apple Accelerate / vecLib	macOS 自带，优化良好	免费

1.2.3 BLAS 的多线程支持

- 这些 BLAS 实现除了单核性能高，还能在线性代数计算中自动多线程
- 前提是：
 - 程序正确链接到多线程 BLAS
 - 控制通用线程数的环境变量 `OMP_NUM_THREADS` 未被设置成 1

⚠ Remark ↴

- macOS 使用 `VECLIB_MAXIMUM_THREADS` 而不是 `OMP_NUM_THREADS`

- 若使用的是 Intel MKL，则需要设置 `MKL_NUM_THREADS`

1.2.4 Python 中的多线程线性代数

- NumPy 与 SciPy 在底层调用 BLAS/LAPACK
- 使用优化版本 (MKL 或 OpenBLAS) 能显著提高性能
- BLAS 线程化配置取决于 Python 与 NumPy 的安装方式

1.2.5 示例代码



Python

```

1 import numpy as np
2 import time
3
4 x = np.random.normal(size=(6000, 6000))
5
6 start_time = time.time()
7 x = np.dot(x.T, x)
8 U = np.linalg.cholesky(x)
9 elapsed_time = time.time() - start_time
10 print("Elapsed Time (8 threads):", elapsed_time)

```

若将 `OMP_NUM_THREADS` 设置为 1 再运行，可比较单线程与多线程性能差异，实验结果如下：

设置	<code>OMP_NUM_THREADS</code> 值	实测时间
单线程	1	17.6 s
多线程（显式 8）	8	4.7 s
默认（未设置）	None	8.8 s

⚠ Remark ▾

小规模矩阵计算可能因线程启动与调度开销反而变慢

1.3 场景 1C：GPU 加速线性代数 (GPUs and Linear Algebra)

- 大规模矩阵运算是 GPU 的理想应用场景
- PyTorch、TensorFlow、JAX 等框架可在 CPU 与 GPU 上自动切换执行
- 内部通过不同的 **kernel (计算核)** 实现相同操作的 CPU/GPU 版本

1.3.1 示例：PyTorch GPU 加速矩阵乘法



Python

```

1 import torch
2
3 start = torch.cuda.Event(enable_timing=True)
4 end = torch.cuda.Event(enable_timing=True)
5
6 gpu = torch.device("cuda:0")
7 n = 10000
8 x = torch.randn(n, n, device=gpu)
9 y = torch.randn(n, n, device=gpu)

```

```

10
11 # GPU 计时
12 start.record()
13 z = torch.matmul(x, y)
14 end.record()
15 torch.cuda.synchronize()
16 print(start.elapsed_time(end))    # 约 120 ms
17
18 # CPU 对比
19 cpu = torch.device("cpu")
20 x = torch.randn(n, n, device=cpu)
21 y = torch.randn(n, n, device=cpu)
22
23 start.record()
24 z = torch.matmul(x, y)
25 end.record()
26 torch.cuda.synchronize()
27 print(start.elapsed_time(end))    # 约 18 秒

```

GPU 计算约 0.1–0.2 秒，CPU 计算约 18 秒，速度提升约 **100 倍以上**

更严谨的对比应考虑：

- GPU 默认使用 **4 字节浮点数 (float32)**
- CPU 线程数、内存带宽及浮点精度设定

1.4 场景 1D：向量化计算的并行化 (Parallelized Vectorized Calculations)

- 框架如 PyTorch、TensorFlow、JAX 可自动在 CPU 或 GPU 上并行化向量化操作
- 在 Unit 5 中，我们使用 JAX 对大型一维数组执行了向量化计算
- 若 GPU 可用，JAX 会自动将计算调度到 GPU

1.4.1 示例：JAX 向量化 GPU 加速



Python

```

1 import time
2 import jax
3 import jax.numpy as jnp
4
5 def myfun_jnp(x):
6     y = jnp.exp(x) + 3 * jnp.sin(x)
7     return y
8
9 n = 500_000_000
10 key = jax.random.key(1)
11 x_jax = jax.random.normal(key, (n,)) # 默认 32-bit 浮点数
12 print(x_jax.platform())
13
14 # GPU 计算
15 t0 = time.time()
16 z_jax1 = myfun_jnp(x_jax).block_until_ready()
17 t_gpu = round(time.time() - t0, 3)
18
19 # CPU 计算
20 cpu_device = jax.devices('cpu')[0]
21 with jax.default_device(cpu_device):
22     key = jax.random.key(1)

```

```
23     x_jax = jax.random.normal(key, (n,))
24     print(x_jax.platform())
25     t0 = time.time()
26     z_jax2 = myfun_jnp(x_jax).block_until_ready()
27     t_cpu = round(time.time() - t0, 3)
28
29     print(f"GPU time: {t_gpu}\nCPU time: {t_cpu}")
```

运行结果：

```
1 GPU time: 0.015
2 CPU time: 4.709
```

- GPU 明显更快，即使 JAX 在 CPU 上自动使用多线程
- 若强制让 JAX 使用 CPU，可尝试：



Python

```
1 jax.config.update('jax_platform_name', 'cpu')
```

但在某些环境下可能无效

2 场景 2：三个预测模型并行运行 (Three Prediction Methods on One Dataset)

具体情境：需要对同一数据拟合三种不同的统计或机器学习模型

一般情境：并行化少量任务

2.1 可选方案

- **方案 1：单核对应单模型**

若有三核，可将每个模型分配给一个核心并行运行

- **方案 2：结合场景 1 的思路**

若可用核心更多，可同时并行多个模型的内部运算

在集群环境下，可分配**一个节点对应一个模型**，并在节点内部进行并行化

2.2 示例：使用 Dask 的 processes 调度器



Python

```
1 import dask
2 import time
3 import numpy as np
4
5 def gen_and_mean(func, n, par1, par2):
6     return np.mean(func(par1, par2, size=n))
7
8 dask.config.set(scheduler='processes', num_workers=3, chunksize=1)
9
10 n = 250_000_000
11 tasks = [
12     dask.delayed(gen_and_mean)(np.random.normal, n, 0, 1),
13     dask.delayed(gen_and_mean)(np.random.gamma, n, 1, 1),
14     dask.delayed(gen_and_mean)(np.random.uniform, n, 0, 1)
15 ]
16
```

```
17 t0 = time.time()
18 results = dask.compute(tasks)
19 print(time.time() - t0)
```

示例结果：约 16.03 秒

单线程顺序运行同样的任务约需 13.83 秒

思考

为什么没有达到理想的三倍加速？

可能原因包括：

- 数据分块与通信开销
- CPU 资源竞争或内存带宽瓶颈
- 各任务耗时略有差异，造成负载不均

2.3 延迟计算与同步机制 (Lazy Evaluation, Synchronicity, and Blocking)

- **延迟计算 (lazy evaluation)**
 - `dask.delayed()` 创建的对象并不会立即执行
 - 它只是计算任务的“图结构 (computational graph)”表示
- **同步计算 (synchronous execution)**
 - `dask.compute()` 会等待所有任务完成后才返回结果
 - 主进程在此期间被阻塞 (blocking call)
- **异步计算 (asynchronous execution)**
 - 若主进程在 worker 执行时即可继续运行，即称为非阻塞 (non-blocking)
- **关于 `chunksize` 参数**
 - 设置 `chunksize=1` 使每个 worker 立即启动一个任务
 - 若不设置，Dask 会将任务分组以减少任务启动的开销
 - 但在任务数量少的情况下，默认分组可能妨碍并行化效率

2.4 延迟计算与计算图的作用

- 延迟计算与**依赖图 (computational graph)**高度相关
- 当一个任务依赖另一个任务的结果时，Dask 会根据依赖关系自动确定执行顺序
- 整个任务流在运行前被优化为一个有向无环图 (DAG)，确保执行顺序正确且资源利用高效

3 场景 3：10 折交叉验证 (10-fold Cross-Validation) 与有限核心数

具体情境：在 10 折交叉验证中运行预测模型

一般情境：使用并行 map 进行任务分配

3.1 思路

- 每个折 (fold) 是一个独立的模型拟合任务
- 可通过**parallel map** 同时在多个核心上运行
- 使用**distributed scheduler**，即使在单机上也能实现并行

3.2 示例：Dask 并行交叉验证

```
Python
1 import numpy as np
2 import pandas as pd
3 from sklearn.ensemble import RandomForestRegressor
4 from sklearn.model_selection import KFold
5
6 def cv_fit(fold_idx):
7     train_idx = folds != fold_idx
8     test_idx = folds == fold_idx
9     X_train = X.iloc[train_idx]
10    X_test = X.iloc[test_idx]
11    Y_train = Y[train_idx]
12    model = RandomForestRegressor()
13    model.fit(X_train, Y_train)
14    predictions = model.predict(X_test)
15    return predictions
16
17 np.random.seed(1)
18 n, p = 1000, 50
19 X = pd.DataFrame(np.random.normal(size=(n, p)), columns=[f"X{i}" for i in range(1, p+1)])
20 Y = X["X1"] + np.sqrt(np.abs(X["X2"] * X["X3"])) + X["X2"] - X["X3"] +
np.random.normal(size=n)
21
22 n_folds = 10
23 seq = np.arange(n_folds)
24 folds = np.random.permutation(np.repeat(seq, 100))
```

```
Python
1 from dask.distributed import Client, LocalCluster
2 n_cores = 2
3 cluster = LocalCluster(n_workers=n_cores)
4 c = Client(cluster)
5
6 tasks = c.map(cv_fit, range(n_folds))
7 results = c.gather(tasks)
```

⚠ Remark

- 若核心数少于任务数 (如 10 折但仅 4 核), 则部分核心需轮流执行任务
- 后续场景将介绍可提升效率的动态任务分配策略

4 场景 4：不同预测方法的并行化 (Parallelizing Over Prediction Methods)

情境：不同预测模型或任务的执行时间差异较大

目标：通过合理任务分配策略提升整体效率

4.1 动态分配 (Dynamic Allocation)

- 动态分配逐个启动任务, 而非预先分组
- 适用于任务执行时间差异较大的情形

- 可减少“部分核心提前完成而空闲”的情况

4.1.1 示例：慢任务与快任务混合运行



Python

```

1 import numpy as np
2 import scipy.special
3 import time
4 from dask.distributed import Client, LocalCluster
5
6 n_cores = 4
7 cluster = LocalCluster(n_workers=n_cores)
8 c = Client(cluster)
9
10 # 4 个慢任务, 12 个快任务
11 n = np.repeat([5*10**7, 10**6, 10**6, 10**6], 4)
12
13 def fun(i):
14     print(f"Working on {i}.")
15     out = np.mean(scipy.special.gammaln(np.exp(np.random.normal(size=n[i]))))
16     print(f"Finishing {i}.")
17     return out
18
19 t0 = time.time()
20 tasks = c.map(fun, range(len(n)))
21 results = c.gather(tasks)
22 print(time.time() - t0)
23 cluster.close()

```

运行结果约 4.6 秒, 明显提升

- 动态分配依赖调度器实时管理任务, 可实现良好的负载均衡
- 若任务数量少, 仍可能出现多个慢任务被分配到同一核心的情况

4.2 静态分配 (Static Allocation)

- 默认情况下, `processes` 调度器按批次分配任务 (默认 `chunksize=6`)
- 若执行时间差异大, 可能造成不均衡

 Remark ▾

难以研究 Dask 是如何将任务分配到 chunk 中的 (通过重复运行发现似乎存在一些随机性)



Python

```

1 import dask, time
2 dask.config.set(scheduler='processes', num_workers=4)
3
4 tasks = [dask.delayed(fun)(i) for i in range(len(n))]
5 t0 = time.time()
6 results = dask.compute(tasks)
7 print(time.time() - t0)

```

示例结果约 5.45 秒

4.3 强制动态分配 (Force Dynamic Allocation)

- 通过设置 `chunksize=1` 可强制动态分配
- 适合任务耗时差异较大时使用

Python

```
1 dask.config.set(scheduler='processes', num_workers=4, chunksize=1)
2
3 tasks = [dask.delayed(fun)(i) for i in range(len(n))]
4 t0 = time.time()
5 results = dask.compute(tasks)
6 print(time.time() - t0)
```

示例结果约 5.55 秒 (与默认设置相近, 但动态分配可在部分情况下更快)

4.4 选择策略: 静态 vs 动态分配

情况	推荐策略	原因
任务耗时差异大	动态分配 (Dynamic)	避免某些核心空闲
任务数量多且耗时相近	静态分配 (Static)	减少调度开销 (~1ms/任务)
使用单机 + processes 调度器	默认静态, 可通过 <code>chunksize=1</code> 切换为动态	
使用 distributed 调度器	默认动态, 若需静态需自行分批提交任务	

补充: 在 R 的 `future` 包中, 静态分配是默认行为

5 场景 5: 多方法的 10 折交叉验证 (10-fold CV Across Multiple Methods)

具体情境: 你正在运行一个集成预测方法 (如 SuperLearner 或 Bayesian model averaging), 在 10 折交叉验证中包含多个统计或机器学习模型

一般情境: 需要并行化嵌套任务 (nested tasks) 或大量任务, 最好能跨多台机器运行

5.1 场景 5A: 嵌套并行化 (Nested Parallelization)

- 当任务具有双层循环结构 (如“折 × 方法”) 时, 可以通过“扁平化循环 (flattening the loops)”来简化并行化
- 例如原始代码使用双层循环:

Python

```
1 for fold in range(n):
2     for method in range(M):
3         ### code here
```

- 可以将其展开为单层循环:

Python

```
1 for idx in range(n*M):
2     fold = idx // M
3     method = idx % M
4     ### code here
```

- 或在使用 Dask 时，直接在嵌套循环中创建延迟任务列表：

```
 Python
1 for fold in range(n):
2     for method in range(M):
3         tasks.append(dask.delayed(myfun)(fold, method))
```

- 在 R 中，**future** 包提供了更直接的接口来处理嵌套循环的并行化

5.2 场景 5B：跨多节点的并行化 (Parallelizing Across Multiple Nodes)

- 若你能访问由多台机器组成的集群（如 Linux 集群），Dask 可在多个节点上启动 worker 并进行任务调度
- 适用于：
 - 嵌套任务数量庞大
 - 或单层任务数量极多的情况

5.2.1 示例：使用 SSHCluster 启动多节点并行

```
 Python
1 from dask.distributed import Client, SSHCluster
2
3 # 第一个主机为 scheduler
4 cluster = SSHCluster([
5     "gandalf.berkeley.edu",
6     "radagast.berkeley.edu",
7     "radagast.berkeley.edu",
8     "arwen.berkeley.edu",
9     "arwen.berkeley.edu"
10 ])
11 c = Client(cluster)
```

5.2.2 在 SLURM 集群中的用法

```
 Python
1 import subprocess
2 machines = subprocess.check_output("srun hostname", shell=True,
3                                     universal_newlines=True).strip().split('\n')
4 machines = [machines[0]] + machines
```

5.2.3 示例任务

```
 Python
1 def fun(i, n=10**6):
2     return np.mean(np.random.normal(size=n))
3
4 n_tasks = 120
5 tasks = c.map(fun, range(n_tasks))
6 results = c.gather(tasks)
```

5.2.4 检查使用的节点

```
 Python
```

```
1 import subprocess
2 c.gather(c.map(lambda x: subprocess.check_output("hostname", shell=True),
3                 range(4)))
4 cluster.close()
```

| 6 场景 6：大规模数据的分层分析 (Stratified Analysis on a Large Dataset)

具体情境：你在一个极大的数据集上执行分层分析 (stratified analysis)，希望避免不必要的数据复制

一般情境：在并行计算中，尽量减少数据副本以节省内存与时间

| 6.1 数据复制的问题

- 在单节点上使用多进程时，每个进程通常都会复制一份原始数据
- 原因：不同进程拥有独立的内存空间，无法直接共享主进程中的对象
- 数据复制会导致：
 - 巨大的内存占用
 - 不必要的计算延迟

| 6.2 示例：使用 processes 调度器 (每任务复制一次)

Python

```
1 import dask, os, time, numpy as np
2
3 def do_analysis(i, x):
4     print(f"Object id: {id(x)}, process id: {os.getpid()}")
5     return np.mean(x)
6
7 n_cores = 4
8 x = np.random.normal(size=5*10**7)
9
10 dask.config.set(scheduler='processes', num_workers=n_cores, chunksize=1)
11
12 tasks = [dask.delayed(do_analysis)(i, x) for i in range(8)]
13 t0 = time.time()
14 results = dask.compute(tasks)
15 print(time.time() - t0)
```

输出显示每个进程都生成了不同的对象 ID，意味着数据被多次复制

示例结果：约 8.66 秒

| 6.3 使用 threads 调度器避免复制

- 在线程模式下，所有 worker 共享同一内存对象 (无复制)
- 速度大幅提升

Python

```
1 dask.config.set(scheduler='threads', num_workers=n_cores)
2 tasks = [dask.delayed(do_analysis)(i, x) for i in range(8)]
```

```
3 t0 = time.time()
4 results = dask.compute(tasks)
5 print(time.time() - t0)
```

输出显示所有线程使用相同的对象 ID

示例结果：约 0.11 秒

⚠ Remark ▾

注意：在多线程模式下，不应修改共享数据，否则可能引发竞争条件 (race condition)

6.3.1 使用 distributed 调度器实现“每 worker 一份副本”



Python

```
1 from dask.distributed import Client, LocalCluster
2 cluster = LocalCluster(n_workers=n_cores)
3 c = Client(cluster)
4
5 x = dask.delayed(x) # 延迟传递数据，确保每个 worker 仅复制一次
6 tasks = [dask.delayed(do_analysis)(i, x) for i in range(8)]
7 t0 = time.time()
8 results = dask.compute(tasks)
9 print(time.time() - t0)
10 cluster.close()
```

输出约 1.4 秒，性能优于多进程模式

但 Dask 会给出警告，提示传输的数据量较大 (381 MiB)，建议使用 `scatter()` 等方式优化

7 场景 7：模拟研究与并行随机数生成 (Simulation Study with Parallel RNG)

具体情境：运行 n=1000 次模拟，每次包含两个模型拟合

一般情境：在并行环境中安全地生成随机数，避免不同进程使用相同的随机序列

7.1 问题背景

- 计算机生成的是**伪随机数 (pseudo-random numbers)**，由确定性算法产生
- 若多个进程使用相同的随机数种子 (seed)，可能产生重叠序列
- 为保证模拟的独立性，需为每个任务分配**唯一且不重叠的随机序列**

7.2 方法一：使用 PCG64 生成器的 `jumped()` 方法



Python

```
1 bitGen = np.random.PCG64(1)
2 rng = np.random.Generator(bitGen)
3 rng.random(size=3)
```

- 可通过 `jumped()` 前进生成器状态：



Python

```
1 bitGen = np.random.PCG64(1)
2 bitGen = bitGen.jumped(1)
3 rng = np.random.Generator(bitGen)
4 rng.normal(size=3)
```

- 两次连续跳跃等价于 `jumped(2)` :



Python

```
1 bitGen = np.random.PCG64(1)
2 bitGen = bitGen.jumped(1)
3 bitGen = bitGen.jumped(1)
4 rng = np.random.Generator(bitGen)
5 rng.normal(size=3)
```

| 7.3 方法二：使用 Mersenne Twister (MT19937)



Python

```
1 bitGen = np.random.MT19937(1)
2 bitGen = bitGen.jumped(1)
3 rng = np.random.Generator(bitGen)
4 rng.normal(size=3)
```

| 7.4 并行随机数生成策略

为每个任务分配不同的子序列：



Python

```
1 def myrandomfun(i):
2     bitGen = np.random.PCG64(1)
3     bitGen = bitGen.jumped(i)
4     rng = np.random.Generator(bitGen)
5     # 任务内部生成随机数
```

或使用 `SeedSequence.spawn()` 自动创建独立 RNG:



Python

```
1 n_tasks = 10
2 sg = np.random.SeedSequence(1)
3 rngs = [np.random.Generator(np.random.PCG64(s)) for s in sg.spawn(n_tasks)]
4
5 def myrandomfun(rng):
6     z = rng.normal(size=5)
```

- 这样每个任务使用独立的随机数子流
- 在 R 中，可使用 `rlecuyer` 包实现同样功能，基于 **L'Ecuyer 算法**，其随机数周期极长并安全分割为子序列