

STAT243 Lecture 6 Parallelization (Reduced)

1 Embarrassingly Parallel Problem

1.1 尴尬并行 (Embarrassingly Parallel, EP) 问题

- 定义

- 各任务可独立运行、互不依赖，也无需进程间通信。
- 可将总任务拆分为多个独立子任务并行执行，最后合并结果。

- 特征

- 同一段代码作用于不同数据子集。
- 各进程可能需要独立的随机数流（将在 Simulation 单元讨论）。

1.2 理想加速效果：线性加速 (Linear Speedup)

- 若一个任务原本需时间 (T)，并行化后使用 (p) 个 CPU：

- 理论执行时间 $\approx (T/p)$ (忽略少量通信与管理开销)。
- 加速比 (speedup) $\approx (p)$ ，即所谓 线性加速 (linear speedup)。
- 当加速比与 CPU 数量成比例时，说明并行化效率极高。

2 计算机体系结构 (Computer Architecture)

- 多处理器趋势

- 由于物理极限，单个处理器的速度提升变得困难，芯片行业转向在同一计算机上集成多个处理单元
- 多处理器 (multi-processor) 和多核心 (multi-core) 架构因此成为主流

- 多核心计算机 (Multi-core computers)

- 每个 CPU 通常包含多个核心 (core)
- 在个人电脑中，多个处理器或多核心共享同一主存 (shared memory)

- 超级计算机与集群 (Supercomputers and clusters)

- 由大量节点 (nodes) 组成，每个节点拥有独立的 CPU 与内存
- 节点间通过高速网络连接，采用 分布式内存 (distributed memory) 结构
- 内存访问原则：访问本地内存远快于跨节点通信

3 常用术语 (Terminology)

术语	含义
cores	单台机器或单节点中的多个处理单元，例如 AMD EPYC 7763 每个 CPU 含 64 核心
nodes	集群中的独立计算机，每个节点有自己的内存
processes	程序的执行实例，可并发运行，理想情况下进程数不超过核心数
workers	实际执行并行任务的进程，与 process 可互换使用
tasks	单个计算单元，每个 process 在某个 core 上执行一个或多个 task
threads	单个进程中的多执行路径，被视为轻量级进程
forking	创建子进程，拥有独立的内存与进程 ID；未修改的对象可能共享父进程内存
scheduler	管理集群任务的调度程序，例如 Slurm
load-balanced	所有核心在计算过程中都被充分利用

术语	含义
sockets	R 中通过 socket 通信实现并行 (如多个 Rscript 进程间的交互)

4 共享内存与分布式内存 (Shared vs Distributed Memory)

类型	结构	特点
共享内存 (Shared Memory)	多核心共享同一内存空间	无需显式消息传递，但新建进程时可能复制对象
分布式内存 (Distributed Memory)	各节点独立内存，通过网络连接	依赖消息传递 (Message Passing) 机制如 MPI

5 GPU 加速 (GPUs)

- GPU (图形处理单元) 最初用于图像渲染，具有大量简单计算单元，可进行 **大规模并行计算 (massively parallel computation)**
- GPGPU (General-Purpose GPU Computing)** 将 GPU 用于通用计算
- 使用方式：**
 - 多数研究者不直接编写 GPU 程序，而使用能自动利用 GPU 的软件，例如 TensorFlow, PyTorch, JAX
 - GPU 执行的函数是在 GPU **kernel** 中执行的，CPU 负责总体流程，将计算密集部分交由 GPU
 - GPU 与 CPU 主存独立，应避免频繁数据传输
 - 启动 kernel 有额外开销，应避免过多微小任务
- GPU 与 TPU 等协处理单元常被称为 **co-processors**，用于辅助 CPU 计算

6 并行化策略 (Parallelization Strategies)

6.1 影响并行化效率的主要因素

在评估一种并行计算方案是否有效时，需要考虑以下因素：

- 内存使用量**：不同进程所占用的总内存量是否超出机器容量
- 通信需求量**：进程之间需要交换多少数据
- 通信延迟 (latency)**：在进程之间传递数据或启动子进程时的延时
- 同步等待**：某些进程是否需要等待其他进程完成后才能继续执行

6.2 并行计算的基本原则与建议

6.2.1 单节点 vs 多节点

- 若能在单个节点上使用共享内存完成计算，通常比在多个节点上运行更快
- 即使多节点拥有更多核心，通信开销仍可能使速度下降
- 如果任务需要处理大量数据或占用高内存，单机拥有足够内存的情况下会比使用 Spark/Hadoop 等框架更快
- 但如果单节点内存不足，则必须采用分布式内存架构

⚠ Remark: 总结 ▾

单节点内存足够就用单节点, 否则用多节点

6.2.2 选择并行化的层次或维度

- 若存在嵌套循环, 一般只在一个层次上进行并行化
- 可以考虑是否已经使用了线程化线性代数 (threaded linear algebra), 以避免多层并行导致过度开销
- 通常情况下, 应并行化**外层循环**
- 目标是确保计算任务之间**负载均衡 (load-balanced)**, 同时**减少通信**

6.2.3 平衡通信开销与核心利用率

- 若**任务数量较少且执行时间不均**, 部分核心会空闲, 导致负载不均衡
- 若**任务数量过多且每个任务耗时极短**, 频繁启动与停止任务的开销会降低效率
- 因此需要在任务粒度和通信成本之间找到平衡

6.2.4 静态调度 (Prescheduling) vs 动态调度 (Dynamic Scheduling)

• 静态调度 (Prescheduling)

任务在开始时就预先分配给各个进程

- 适用于任务数量多且耗时相似的情况
- 优点: 减少通信开销
- 示例: 6 个任务与 3 个 worker
 - worker1 → 任务 1 和 4
 - worker2 → 任务 2 和 5
 - worker3 → 任务 3 和 6

• 动态调度 (Dynamic Scheduling)

任务按执行进度动态分配

- 适用于任务数量少或耗时差异较大的情况
- 优点: 提高负载均衡性
- 示例:
 - 初始时: worker1 任务 1, worker2 任务 2, worker3 任务 3
 - 当某个 worker 率先完成后, 再分配下一个任务 (如任务 4)

7 Task Package 的基本概念

• Task 概览

- Task 是 Python 中用于并行计算的框架, 可在单台或多台机器上实现任务并行化
- 其重要特性在于支持**分布式数据集 (distributed datasets)**:
- 数据可被拆分为多个块 (chunks/shards), 并行执行计算

• Task 核心理念

将“**并行化的定义**”与“**并行化的执行方式与资源**”解耦 (**abstraction**), 目标包括:

- 分离 what 与 how/where**
 - 在代码中定义要并行化的部分 (what)
 - 而具体如何与在哪里执行 (how/where) 由系统配置决定
- 可移植性**

- 同一段代码可在不同计算资源上运行，无需修改代码本身

8 并行后端 (Parallel Backends)

Logic

代码运行所依赖的计算资源称为 **后端 (backend)**

通过 **scheduler (调度器)** 控制并行化的执行方式，例如是否跨机器运行、每台机器使用的核心数等

示例：使用多个 Python 进程并行化计算

```
Python
1 import dask
2 dask.config.set(scheduler='processes', num_workers=4)
```

8.1 调度器类型对比表

类型	描述	支持多节点 multi-node	是否复制对象
synchronous	串行执行 (非并行)	否	否
threads	当前 Python 会话内的多线程	否	否
processes	多个后台 Python 进程	否	是
distributed	跨多节点的 Python 会话	是	是

⚠ Remark: 关于 GIL 与线程调度

- Python 有 **全局解释器锁 (GIL)**，阻止纯 Python 代码的并行执行
- 因此 **threads** 调度器只适用于 **底层由 C/C++/Cython 实现** 的运算，如 **numpy** 数组计算和 **pandas DataFrame** 操作
- 纯 Python 循环或逻辑不会真正并行化

8.2 使用 distributed 调度器的优势

- 可在单机使用 (如笔记本电脑)
- 相较于 **multiprocessing** 的优点包括：
 - 提供 **可视化监控仪表盘 (diagnostic dashboard)**
 - 更高效的内存复制与对象管理
- 进行 **并行 map 操作** 时必须使用 distributed 调度器

9 变量与 worker 的访问 (Variables and Workers)

Dask 会自动识别并复制代码中所需的 **包与全局变量** 到 worker 进程中，无需手动导入或传递

9.1 隐式使用 delayed 装饰器

```
Python
```

```
1 import dask
2 dask.config.set(scheduler='processes', num_workers=4, chunkszie=1)
3
4 import numpy as np
5 n = 10
6
7 @dask.delayed
8 def myfun(idx):
9     return np.random.normal(size=n)
10
11 tasks = []
12 p = 8
13 for i in range(p):
14     tasks.append(myfun(i)) # 延迟创建任务 (lazy task)
15
16 results = dask.compute(tasks) # 并行执行所有任务
```

说明：

- `@dask.delayed` 装饰器将函数转为惰性 (lazy) 任务
- 这些任务会被添加到 Dask 的调度图中 (**使用了 Dynamic Scheduling**)，由调度器自动分配到不同进程执行
- 变量 `n` 和包 `numpy` 自动可用，无需显式传入

9.2 显式使用 `delayed` 装饰器

无需事先为函数添加 `@delayed` 装饰器，也可以直接调用：



Python

```
1 tasks.append(dask.delayed(myfun)(i))
```

这种写法的好处是 `myfun()` 可以根据具体场景决定要不要惰性执行：

- 若无需使用 Dask，只调用 `myfun(i)` 即可正常运行
- 若使用 Dask，可通过 `dask.delayed()` 将函数变为可并行的惰性任务

10 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

10.1 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量



Python

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

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

10.2.1 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

10.2.2 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

- 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 ▾

小规模矩阵计算可能因线程启动与调度开销反而变慢

10.3 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

严谨的 CPU 与 GPU 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

10.4 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

10.4.1 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}")

```

11 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

11.1 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

11.2 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, chunkszie=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

11.3 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

⚠ Remark: 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

12 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

12.1 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

12.2 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"] +
21 np.random.normal(size=n)
22
23 n_folds = 10
24 seq = np.arange(n_folds)
25 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

13 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

13.1 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

13.1.1 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

13.2 静态分配 (Static Allocation)

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

```
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

13.3 强制动态分配 (Force Dynamic Allocation)

- 通过设置 `chunkszie=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)

13.4 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	

14 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

14.1 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))
```

14.2 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

14.2.1 示例：使用 SSHCluster 启动多节点并行

```
Python
from dask.distributed import Client, SSHCluster
# 第一个主机为 scheduler
cluster = SSHCluster([
    "gandalf.berkeley.edu",
    "radagast.berkeley.edu",
    "radagast.berkeley.edu",
    "arwen.berkeley.edu",
    "arwen.berkeley.edu"
])
c = Client(cluster)
```

14.2.2 在 SLURM 集群中的用法

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

14.2.3 示例任务

```
Python
def fun(i, n=10**6):
    return np.mean(np.random.normal(size=n))

n_tasks = 120
tasks = c.map(fun, range(n_tasks))
results = c.gather(tasks)
```

14.2.4 检查使用的节点

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

15 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

15.1 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

15.2 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)
```

⚠ Remark ▾

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

15.3 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)

15.4 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()
```

⚠ Remark ▾

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

16 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

16.1 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**

16.2 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)
```

16.3 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)
```

16.4 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