作业 1：卫星数据爬虫与图像处理

龚舒凯 2022202790

1 基于 Python 的卫星数据获取

卫星图像数据来自哨兵 2 号卫星 (Sentinel-2)。哨兵 2 号卫星是一颗多光谱卫星，可以提供 13 个波段的、10m-60m 高分辨率的多光谱图像数据。在数据获取上，本作业使用Microsoft Planetary Computer提供的卫星数据的 API 接口进行数据爬虫。抓取下来的一张卫星数据包含以下主要组成部分：

- **id**：每张卫星图像的唯一标识符，例如'S2A_MSIL2A_20231201T023041_R046_T51RUP_20231201T073845'，表明卫星图像数据来自 S2A 卫星，拍摄时间为 2023 年 12 月 1 日。
- **properties**：包含了卫星数据的属性信息，如卫星数据的时间 (datetime)、平台 (Sentinel-2A 卫星)、云量 (cloud cover)、水体比例 (water percentage)、绿植比例 (vegetation percentage) 等。
- **geometry**：一个多边形类型 (Polygon) 的数据，包含了卫星图像的边界坐标信息。
- **assets**：卫星图像的多个波段数据，如 AOT(大气光学厚度)、B01-B12(13 个波段的多光谱数据)、SCL(地表覆盖类型)、WVP(水汽含量)、visual(真彩色图像)、rendered_preview(渲染预览图像) 等。
- **bbox**：API 调用者指定的卫星图的边界框信息。

1.1 卫星数据爬虫策略

在卫星图像获取这一部分，本作业采取的数据爬虫策略是：

1. 创建一个 Python 的 `requests.Session()` 和 `Retry` 对象实现一个持久化的、有重试机制的 HTTP 会话。
2. 通过 API 接口向 Planetary Computer 递交卫星类型 (sentinel-2-l2a)、时间范围、边界框信息 (经度范围 $[x_1, x_2]$，纬度范围 $[y_1, y_2]$) 和云量这些查询参数，获取到一个所有符合条件的卫星数据列表。
3. 通过遍历列表中的每一个卫星数据，先获取卫星图像的 id，再通过卫星数据的 assets 属性中的 rendered_preview 获取其下载链接。
4. 检查 rendered_preview 图像中是否有大片的空白（这是因为部分卫星数据是不规则的，统一为矩形图片后，图像中有很多无效的区域）。如果有，则跳过这张卫星图像，否则通过下载链接将符合条件的卫星图像数据下载到本地磁盘中，并以 id 为文件名保存。

卫星数据爬取的完整代码见 `crawler_stac.py` 文件。在本作业中，默认的参数为

- 卫星类型：sentinel-2-l2a
- 时间范围：2020-01-01 至 2023-12-31
- 边界框：经度范围 [121.21364, 122.03086]，纬度范围 [30.72784, 31.4577] (上海市)
- 云量：0-1%
- 空白区域阈值：0.3

- 爬取方式：多线程
- 保存内容：卫星的 rendered_preview 图像

以下展示了爬取下来前 3 张卫星图像的 rendered_preview 图像。

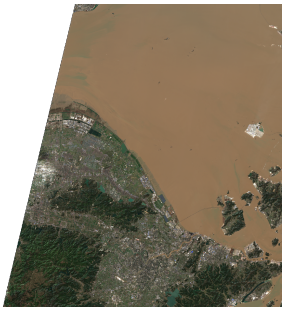

图 1: 20200210T022831

图 2: 20231015T023651

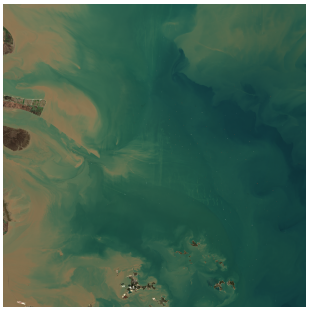

图 3: 20200220T022731

1.2 多线程、多进程、串行爬取的效率比较

在下载过程中，我们比较了串行下载、多进程下载和多线程下载的性能。（你可以在 python 终端中通过 `python crawler_stac.py --method threading` 或 `python crawler_stac.py --method multiprocessing` 或 `python crawler_stac.py --method serial` 来切换不同的下载方式。）

	尝试 1	尝试 2	尝试 3	尝试 4	尝试 5	平均	标准差
多线程	15.11982512	20.15151906	23.32113314	28.18130112	23.76642203	22.10804009	4.841731298
多进程	24.61101675	28.59665012	25.260607	24.81853008	32.77296829	27.21195445	3.505116654
串行	35.62296295	37.8040657	47.25472498	33.49508023	44.71747684	39.77886214	5.935816593

爬虫任务（网络请求与下载）是典型的 I/O 密集型任务。从实验结果来看，多线程爬虫在耗时上略优于多进程爬虫、显著优于串行爬虫。三者运行时间的标准差相近，运行结果稳定。**因此，多线程爬虫是最优的选择。**

上述实验结果出现的可能原因是：

- 在网络 I/O 任务中，**多线程**能够在网络等待期间切换任务，减少等待时间，且线程间的上下文切换开销比进程要小，因此速度更快。
- **多进程**方式虽然能充分利用多核 CPU 中的每个进程，但每个进程需要分配独立的内存空间，且进程间的通信和资源共享较线程更复杂。在系统资源（CPU、内存）有限的情况下，多进程的运行效率可能反而低于多线程。
- 串行方式完全是顺序执行的，每次只能处理一个请求，并且每个请求的网络延迟都会累积，从而导致总时间显著增加。相比于多线程和多进程，串行的表现最慢是可以预期的。

2 基于 OpenMP 的图像处理：卫星图像 NDVI 计算

接下来，本作业将基于 C 和 OpenMP 实现一个简单的卫星图像处理程序，计算卫星图像的 NDVI(Normalized Difference Vegetation Index)。NDVI 是一种用来评估植被覆盖程度的指标，其计算公式为：

$$\mathrm{NDVI} = \frac{\mathrm{NIR} - \mathrm{RED}}{\mathrm{NIR} + \mathrm{RED}} \tag{1}$$

其中，NIR 是近红外波段的反射率（对应 sentinel-2-l2a 的 B08 波段），RED 是红光波段的反射率（对应 sentinel-2-l2a 的 B04 波段）。NDVI 的取值范围为 $[-1,1]$，较高的数值表示植被覆盖程度较高，较低的数值表示植被覆盖程度较低（如水体、建筑物等）。哨兵 2 号卫星的波段 tif 图像是 3600*3600 的矩形图像，每个像素点的值是一个 16 位的整数，要获得 NDVI 的图像，我们要对每个像素点进行一次浮点数的计算：

$$\mathrm{NDVI}(i,j) = \frac{\mathrm{NIR}(i,j) - \mathrm{RED}(i,j)}{\mathrm{NIR}(i,j) + \mathrm{RED}(i,j)} \tag{2}$$

一共需要进行 $3600 * 3600$ 次计算，因此 **NDVI 计算是一个典型的计算密集型任务。**为了提高计算效率，我们可以使用 OpenMP 并行化计算过程。在本作业中，我们使用了 OpenMP 的 `parallel for` 指令来并行化计算 NDVI。

2.1 NDVI 计算的 OpenMP 实现

在 C 中直接读取 tif 图像需要使用 `libtiff` 库。而本作业希望以最基本的方式实现 NDVI 计算，故采取如下处理策略

1. 运行 `tif_to_bin.py`，将所有 `tif` 图像转换为二进制 `bin` 文件，以便 C 程序读取。
2. 编译计算 NDVI 的程序(并行版):`gcc -fopenmp -o ndvi_parallel ndvi_parallel.c` 编译 `ndvi_parallel.c`，生成可执行文件 `ndvi_parallel`。
3. `./ndvi_parallel` 运行 `ndvi_parallel`，生成 `NDVI.bin` 文件。
4. 运行 `bin_to_tif.py`，将 `NDVI.bin` 转换为 png 图像保存。

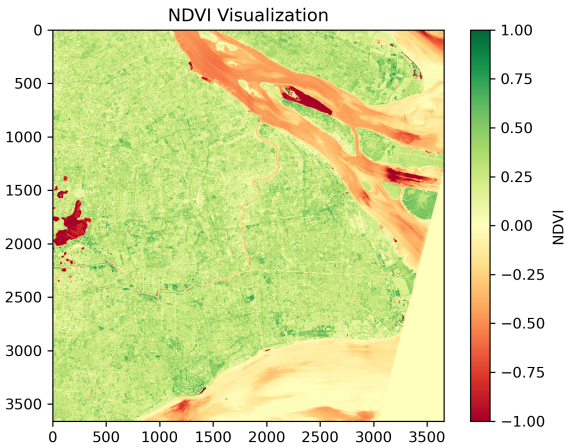

图 4: 经 ndvi_parallel.c 生成的 NDVI 图像

2.2 OpenMP 并行与串行的效率比较

在 NDVI 计算过程中,我们比较了串行计算(编译 `ndvi_parallel.c`)和 OpenMP 并行计算(编译 `ndvi_parallel.c`)的性能。在这一部分，我们将处理 116 个卫星图像文件夹（包括 B04.bin 和 B08.bin）。对于每个文件夹，我们分别计算 NDVI，比较两种计算方式的计算运行时间。

值得注意的是，由于不同文件夹处理工作量不均衡（部分没有成功下载 B04.tif 和 B08.tif 的文件夹工作量较少），本作业在 OpenMP 并行计算中使用了动态调度 (schedule(dynamic)) 让线程更加高效地分配任务，减少线程间的等待时间，尤其当；并且加了 `omp parallel critical` 来保证 logging 信息的输出是串行化的，防止多个线程同时访问 `stdout`。将可能产生瓶颈的文件操作减少到最小程度，专注于处理计算密集型部分（NDVI 计算）。

	尝试 1	尝试 2	尝试 3	尝试 4	尝试 5	平均	标准差
串行	13.36488	10.180874	13.823605	12.889765	12.841121	12.620049	1.42078451
OpenMP 并行	1.464132	2.251753	2.266411	1.494481	1.54808	1.8049714	0.415664913

可以看到，OpenMP 并行计算的平均运行时间明显短于串行计算，其可以充分利用多核 CPU 的优势，同时处理多个计算任务，从而提高计算效率。而串行计算则是逐个处理，效率较低。因此，**OpenMP 并行计算是最优的选择。**

附录

卫星数据爬虫代码

卫星数据爬虫相关代码存放在 `crawler/crawler_stac.py`，在命令行中输入相关参数即可以制定方式（多进程、多线程、串行）爬取指定区域、指定时间、指定云量、指定空白区域阈值的卫星图像数据：

```
python crawler_stac.py --method threading --bbox 121.21364 122.03086 30.72784 31.4577
--start_date 2020-01-01 --end_data 2023-12-31 --cloud_cover 1 --threshold 0.3

python crawler_stac.py --method multiprocessing --bbox 121.21364 122.03086 30.72784 31.4577
--start_date 2020-01-01 --end_data 2023-12-31 --cloud_cover 1 --threshold 0.3

python crawler_stac.py --method serial --bbox 121.21364 122.03086 30.72784 31.4577
--start_date 2020-01-01 --end_data 2023-12-31 --cloud_cover 1 --threshold 0.3
```

NDVI 并行计算代码

NDVI 并行计算代码存放在 `ndvi/`中，包括

- `local_folder`：存放全部的卫星波段，每张卫星图像的所有波段都储存在一个子文件夹中，结构如下所示：其中每一个下载下来的波段都储存为`.tif` 格式

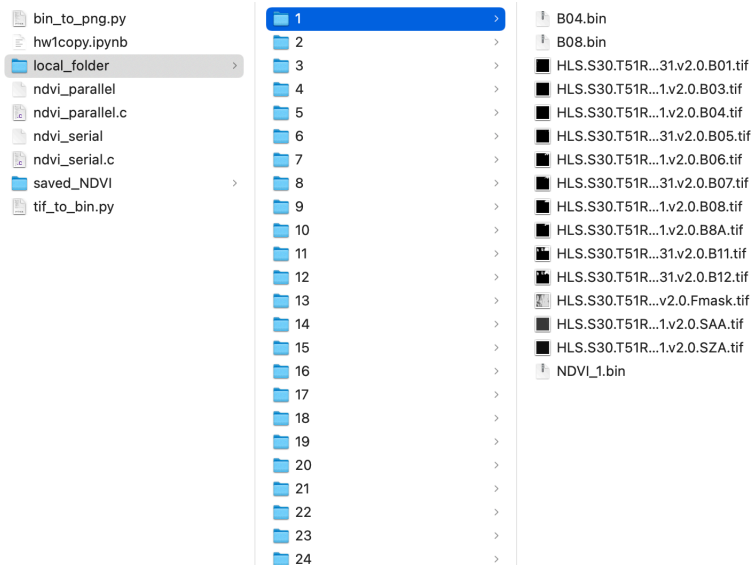

图 5: local_cluster 文件夹结构

- `saved_NDVI`：存放转化后的 NDVI 图像。
- `bin_to_png.py`：将并行计算后得到的 `NDVI.bin` 转化为 png 图像，保存至 `saved_NDVI` 文件夹中（并行实现）。
- `ndvi_parallel.c`：OpenMP 并行计算 NDVI 的 C 代码。
- `ndvi_serial.c`：串行计算 NDVI 的 C 代码。
- `tif_to_bin.py`：数据预处理，将`.tif` 格式的波段批量转化为 C 语言方便读写的二进制`.bin` 文件（并行实现）。

介于卫星图像数据过大，本文件夹中并未包括全部下载下来的卫星波段，只提供了少量作为运行样例。

本地与服务器配置

- 卫星数据爬虫在本地进行，本地配置为搭载 8 核 Apple M2 芯片的 MacBook Air。
- NDVI 计算在服务器上进行，服务器配置为搭载 64 核 Intel Xeon Gold 5218 CPU 的 Inspru NF5468M5 Linux 服务器。