南昌大学超算俱乐部考核题（Python）的实验报告

FROM 回归天空
QQ：903928770
微信：night-helianthus

考核要求：

超算选拔考核题

在本考核中，您需要在虚拟机环境中完成以下任务，任务涉及安装操作系统、配置并行计算环境、实现并优化 Python 并行任务，并撰写实验报告。

任务要求

1. **安装虚拟机：**
    - 在虚拟机中安装 Ubuntu 22.04 LTS 操作系统。
    - 配置虚拟机的网络连接，确保可以正常联网。
2. **安装 Python 并配置 HPC 环境：**
    - 安装 Python 3.x。
    - 安装并配置并行计算相关库（如 `mpi4py` 和 `multiprocessing`），确保环境能够支持并行。
3. **实现并行计算任务：**
    - 使用 `mpi4py` 实现一个大规模矩阵的乘法。

- 使用 `multiprocessing` 实现多进

3. **实现并行计算任务：**
    - 使用 `mpi4py` 实现一个大规模矩阵的乘法。
    - 使用 `multiprocessing` 实现多进程计算完成同样的任务，对比其与 `mpi4py` 的性能差异。
    - 确保程序可以处理至少 10,000 x 10,000 的矩阵运算。
4. **优化与加速：**
    - 优化 Python 代码，使用不同的并行策略提高计算效率。
    - 测试不同进程数对任务执行时间的影响，记录不同并行度下的性能表现。
    - 结合 NumPy 或 SciPy 等高效库，进一步提升计算速度。
5. **性能测试与对比：**
    - 对比 `mpi4py` 和 `multiprocessing` 在不同数据规模和并行度下的性能表现。

- 分别测试在不同规模下的执行时间，收集每种方案的资源使用情况（如 CPU 使用率和内存占用）。

6. **数据记录与可视化：**
   - 收集不同并行策略和并行度的运行结果和性能数据。
   - 将数据记录到 CSV 文件中。
   - 使用 Python 绘制矢量图，展示不同并行策略在不同任务规模下的性能差异。
7. **报告撰写：**
   - 撰写一份详细报告，内容应包括：
     - 实验环境的搭建过程（虚拟机安装、环境配置等）。
     - 不同并行策略的性能对比及其对大规模矩阵运算的加速效果。
     - 优化方案及其带来的性能提升。
     - 数据可视化部分（附图表）。
     - 实验过程中遇到的问题及解决方案。
   - 报告必须采用 LaTeX 或 Markdown

格式撰写。

提交要求

- 将完整的实验报告和源代码上传至个人 GitHub 仓库。
- 提交报告的 PDF 文件及仓库链接。

实验完成部分可以使用 AI 工具，但是报告书写部分请亲自完成！

一.环境搭建

1.配置虚拟机：在虚拟机中安装 Ubuntu 22.04 LTS 操作系统。

使用VMware Workstation，首先在镜像站（如清华源）中下载ubuntu镜像文件，根据安装指南完成vm安装，创建ubuntu64位虚拟机，由于要进行多进程测试，将处理器部分的处理器数量设置为4，每个处理器内核数为4，网络设置为最方便使用的NAT模式，安装ubuntu系统，在Software&updates中换源（如阿里云），完成基本配置。

2.安装IDE: pycharm

第一种最容易理解的办法是到类似应用商店的 ubuntu software 中下载，只需搜索+点击下载即可。首先确保更新，然后搜索pycharm community edition，终端更新指令如下。

```
sudo apt update
```

第二种则是通过snap包管理器，终端进行安装

```
sudo apt update
sudo apt install snapd
sudo snap install pycharm-communiy --classic
```

完成后终端输入来启动并检查pycharm的安装

```
pycharm-community
```

如果想要更新pycharm,终端指令如下

```
sudo snap refresh pycharm-community
```

3.pycharm虚拟环境中必要的环境配置

Python3的安装
打开终端检查是否有python3

```
python3 --version
```

若没有，进行安装

```
sudo apt update
sudo apt install python3
```

pip包管理器安装
创建一个新项目后，打开终端，进行pip包管理器安装

```
sudo apt update
sudo apt install python3-pip
```

安装必要的库
打开项目终端，输入安装指令(这里只以numpy为例)

```
pip install numpy
```

安装MPI
终端安装mpich

```
sudo apt-get update
sudo apt-get install mpich
```

检查是否安装mpich

```
mpicc -v
```

再安装mpi4py库

```
pip install mpi4py
```

4.git的环境搭建

github与git的配置，登录github，配置Ubuntu的.ssh文件，将公钥上传github，方便后期git上传与拉取终端命令生成ssh对：

```
ssh-keygen -t rsa -b 4096 -C "邮箱地址"
```

配置完ssh后测试与github的连接情况

```
ssh -T git@github.com
```

生成项目并克隆仓库到本地

```
cd ~/PycharmProjects/pythonProject
git clone git@github.com:SXP-Simon/the-repo-for-NCUSCC.git
```

若要实现ubuntu无障碍连接到www.github.com,推荐使用金钱方面无痛但是需要配置一点证书的watt tooltik。至此，已经基本搭建测试所需要的环境了。

二.考核部分的multiprocessing与mpi4py的比较

在此声明，我的本次实验中选择的矩阵乘法计算方法为最原始的手撕矩阵方法，使用for循环的嵌套，时间复杂度为O(n^3),至于原因我在后文实验过程中遇到的问题部分会做出回答。由于时间复杂度过大，我选择采取numba的njit装饰器对python语法进行c类语言转译加速，将尽量进行控制变量比较。
分析部分不会将比较两者，只比较自身不同规模的进程数效果。多方案比较将在分析部分后呈现。

基于各个库实现的矩阵乘法并行计算逻辑大致如下：

> 1. matrix_multiple函数实现矩阵乘法方法；
> 2. split_matrix函数实现矩阵分块，方便并行计算进行；
> 3. parallel_matrix_multiple函数调用多进程方法，最后合并结果矩阵。

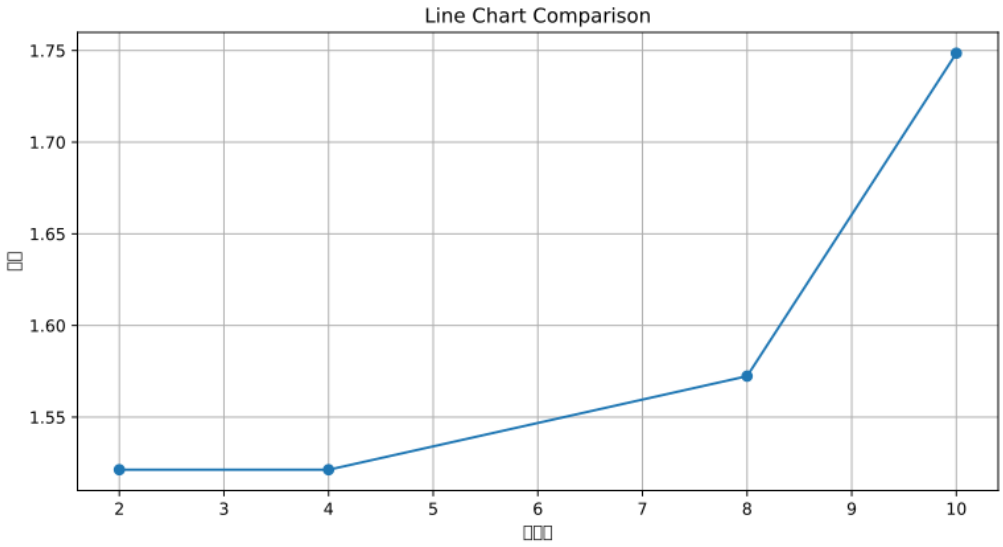
1.multiprocessing库的分析

multiprocessing中使用简洁的语法来调用多核编程方法，简化了实现多进程编程的复杂性，适合单机玩家享受。
这里我选择比较有效的进程池方法与异步方法来进行并行计算，采用with方法来自动管理资源。

```
import numpy as np
from multiprocessing import Pool
import time
from numba import njit

@njit
def matrix_multiply(A_block, B_block):
    # 初始化结果矩阵块
    result_block = np.zeros((A_block.shape[0], B_block.shape[1]))
    # 使用基本的矩阵乘法实现
    for i in range(A_block.shape[0]):
        for j in range(B_block.shape[1]):
            for k in range(A_block.shape[1]):
                result_block[i, j] += A_block[i, k] * B_block[k, j]
    return result_block
@njit()
def split_matrix(A, B, num_splits):
    # 计算每个子块的大小
    split_size_A = A.shape[0] // num_splits
    split_size_B = B.shape[1] // num_splits
    # 分割矩阵 A 和 B
    A_splits = [np.ascontiguousarray(A[i*split_size_A:(i+1)*split_size_A]) for i
in range(num_splits)]
```

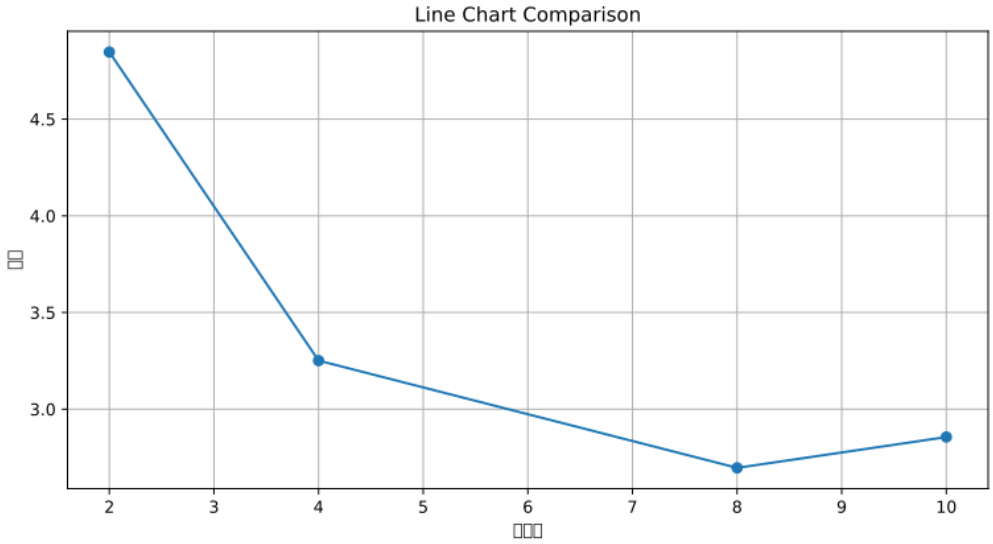
```
    B_splits = [np.ascontiguousarray(B[:, i*split_size_B:(i+1)*split_size_B])
for i in range(num_splits)]
    return A_splits, B_splits

def parallel_matrix_multiply(A, B, num_splits):
    A_splits, B_splits = split_matrix(A, B, num_splits)

    # 创建进程池
    with Pool(processes=num_splits) as pool:
        # 使用 starmap 进行并行计算
        results = pool.starmap(matrix_multiply, [(A_splits[i], B_splits[j]) for
i in range(num_splits) for j in range(num_splits)])

    # 初始化结果矩阵
    final_result = np.zeros((A.shape[0], B.shape[1]))

    # 将子块结果合并到最终结果矩阵中
    split_size_A = A.shape[0] // num_splits
    split_size_B = B.shape[1] // num_splits
    for idx, (i, j) in enumerate([(i, j) for i in range(num_splits) for j in
range(num_splits)]):
        final_result[i*split_size_A:(i+1)*split_size_A, j*split_size_B:
(j+1)*split_size_B] = results[idx]

    return final_result

if __name__ == "__main__":
    n = 10000
    #矩阵大小声明
    A = np.random.rand(n, n)
    B = np.random.rand(n, n)
    num_splits = 8
    #进程数声明

    starttime = time.time()
    result = parallel_matrix_multiply(A, B, num_splits)
    print("Time taken for parallel matrix multiply with numba:", time.time() -
starttime)

    print(result.shape)
    #验证矩阵形状
    np.testing.assert_allclose(result, np.dot(A, B))
    #验证计算结果
```

multiprocessing结果



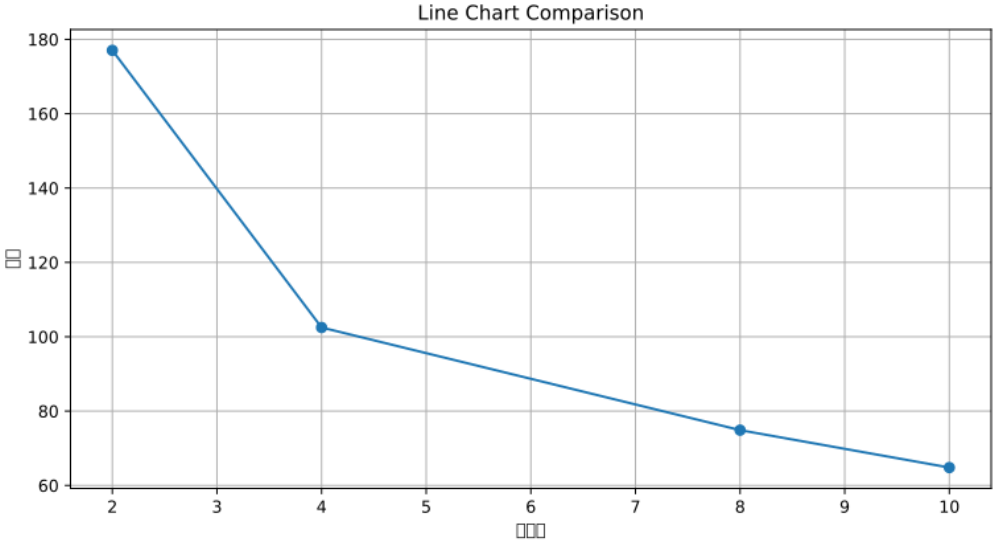
multiprocessing库实现1000*1000规模并行矩阵乘法

横轴为实验进程数，纵轴为实验时间（s）



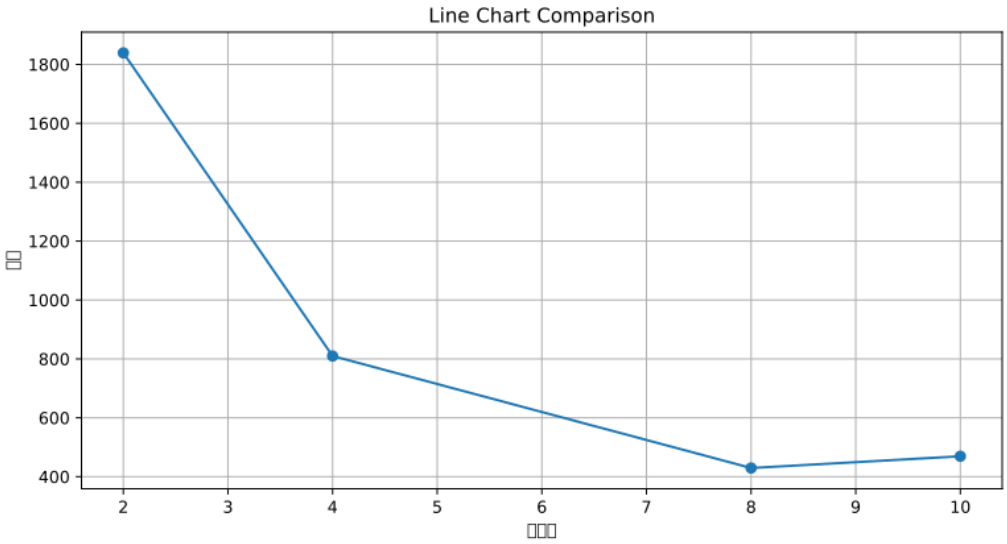
multiprocessing库实现2000*2000规模并行矩阵乘法

横轴为实验进程数，纵轴为实验时间（s）



multiprocessing库实现5000*5000规模并行矩阵乘法

横轴为实验进程数，纵轴为实验时间（s）



multiprocessing库实现10000*10000规模并行矩阵乘法

横轴为实验进程数，纵轴为实验时间（s）

经分析：
在数据规模较小时，并行引起的资源开销占主导地位，进程越多，时间越长，并行效率越低。
在数据规模较大时，并行运算带来的加速效果明显，总体上进程越多，时间缩短，效率提高。

拓展一个多进程库：joblib

joblib是一个基于multiprocessing库的并行实现，特别优化了在数组处理和磁盘缓存发方面，语法简洁，容易上手。在服务启动后，第一次运行可能会比后续运行多耗时一些，因为需要分配进程。但一旦初始化完成，后续的并行计算将更加高效。

安装joblib

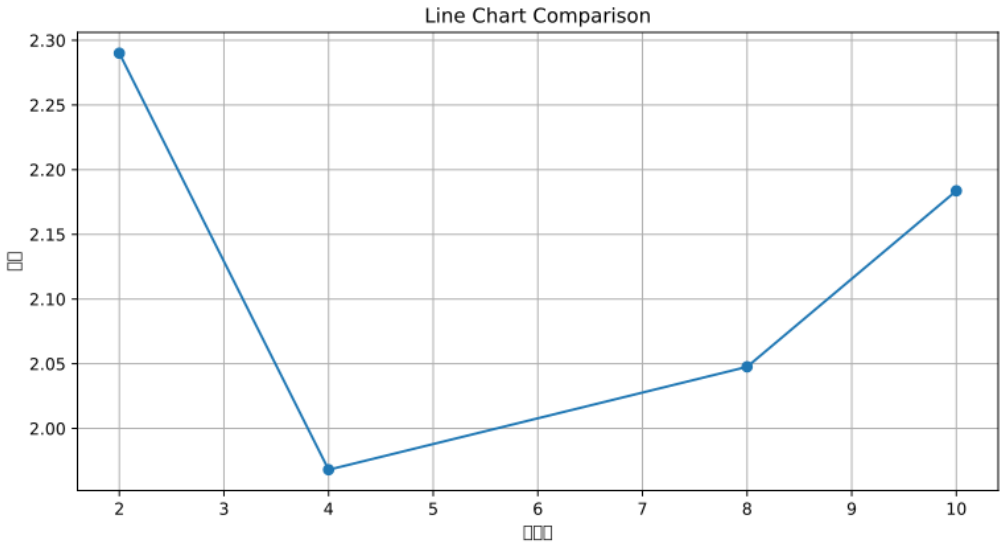
```
pip install joblib
```

joblib并行框架例子

```
import numpy as np
from joblib import Parallel, delayed
import time
from numba import njit

@njit()
def matrix_multiply(A_block, B_block):
    # 初始化结果矩阵块
    result_block = np.zeros((A_block.shape[0], B_block.shape[1]))
    # 使用基本的矩阵乘法实现
    for i in range(A_block.shape[0]):
        for j in range(B_block.shape[1]):
            for k in range(A_block.shape[1]):
                result_block[i, j] += A_block[i, k] * B_block[k, j]
    return result_block
```

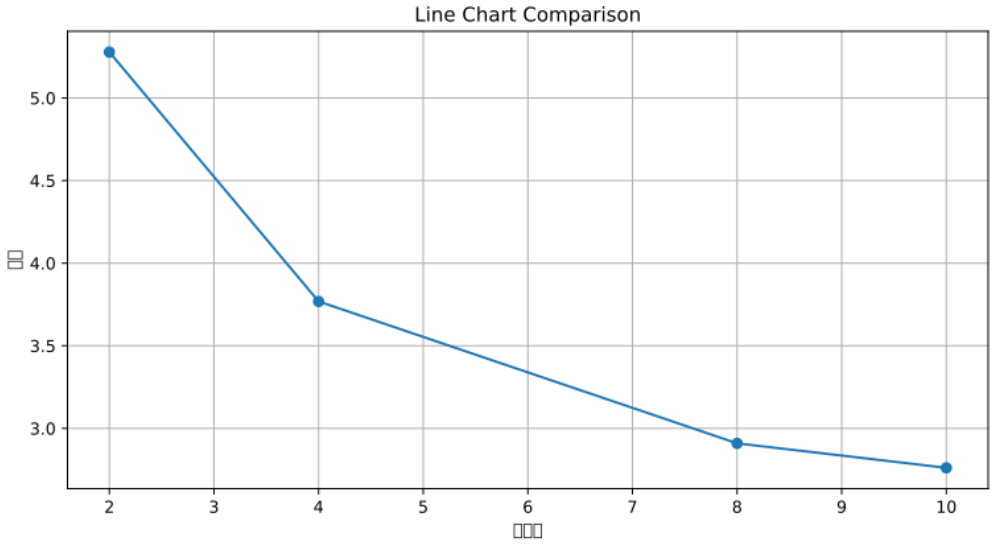
```
@njit()
def split_matrix(A, B, num_splits):
    # 计算每个子块的大小
    split_size_A = A.shape[0] // num_splits
    split_size_B = B.shape[1] // num_splits
    # 分割矩阵 A 和 B
    A_splits = [np.ascontiguousarray(A[i*split_size_A:(i+1)*split_size_A]) for i
in range(num_splits)]
    B_splits = [np.ascontiguousarray(B[:, i*split_size_B:(i+1)*split_size_B])
for i in range(num_splits)]
    return A_splits, B_splits

def parallel_matrix_multiply(A, B, num_splits):
    A_splits, B_splits = split_matrix(A, B, num_splits)

    # 使用 joblib 进行并行计算
    results = Parallel(n_jobs=num_splits)(
        delayed(matrix_multiply)(A_splits[i], B_splits[j])
        for i in range(num_splits)
        for j in range(num_splits)
    )

    # 初始化结果矩阵
    final_result = np.zeros((A.shape[0], B.shape[1]))

    # 将子块结果合并到最终结果矩阵中
    split_size_A = A.shape[0] // num_splits
    split_size_B = B.shape[1] // num_splits
    for idx, (i, j) in enumerate([(i, j) for i in range(num_splits) for j in
range(num_splits)]):
        final_result[i*split_size_A:(i+1)*split_size_A, j*split_size_B:
(j+1)*split_size_B] = results[idx]

    return final_result

if __name__ == "__main__":
    n = 2000
    A = np.random.rand(n, n)
    B = np.random.rand(n, n)
    num_splits = 8

    starttime = time.time()
    result = parallel_matrix_multiply(A, B, num_splits)
    print("Time taken for parallel matrix multiply with joblib:", time.time() -
starttime)

    starttime = time.time()
    np.dot(A, B)
    print("Time taken for numpy dot product:", time.time() - starttime)

    print(result.shape)
    np.testing.assert_allclose(result, np.dot(A, B))
```

joblib结果（这里不做详细介绍，分别为1000，2000规模的矩阵乘法）



joblib库实现1000*1000规模并行矩阵乘法

横轴为实验进程数，纵轴为实验时间（s）



joblib库实现2000*2000规模并行矩阵乘法

横轴为实验进程数，纵轴为实验时间（s）

joblib数据量小时无优势，数据量大时优势明显，而且在多进程资源优化方面优势明显，
后文将用图像直观呈现。

2.mpi4py库分析

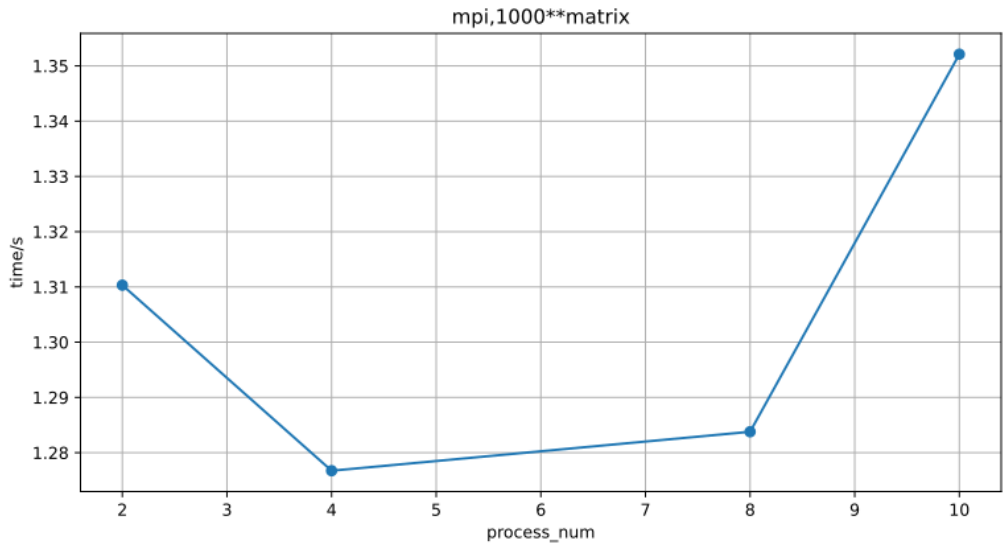
mpi4py是一个在python中实现MPI标准的库，提供面向对象接口使得python程序可以利用多处理器进行并行运算，但是比较适合联机玩家（多机跨节点，服务器层面），单机玩家使用时优势不明显。我在实验过程中使用了进程间集体通信和非阻塞通信等方法进行了测试。

```
import numpy as np
from mpi4py import MPI
import time
from numba import njit
```

```
@njit()
def matrix_multiply(A_block, B_block):
    # 初始化结果矩阵块
    result_block = np.zeros((A_block.shape[0], B_block.shape[1]))
    # 使用基本的矩阵乘法实现
    for i in range(A_block.shape[0]):
        for j in range(B_block.shape[1]):
            for k in range(A_block.shape[1]):
                result_block[i, j] += A_block[i, k] * B_block[k, j]
    return result_block
@njit()
def split_matrix(A, B, num_splits):
    # 计算每个子块的大小
    split_size_A = A.shape[0] // num_splits
    split_size_B = B.shape[1] // num_splits
    # 分割矩阵 A 和 B
    A_splits = [np.ascontiguousarray(A[i*split_size_A:(i+1)*split_size_A]) for i
in range(num_splits)]
    B_splits = [np.ascontiguousarray(B[:, i*split_size_B:(i+1)*split_size_B])
for i in range(num_splits)]
    return A_splits, B_splits

def parallel_matrix_multiply(A, B, num_splits):
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()

    # 计算每个进程的块索引
    block_per_process = (num_splits * num_splits) // size
    start_block = rank * block_per_process
    end_block = (rank + 1) * block_per_process

    # 分割矩阵
    A_splits, B_splits = split_matrix(A, B, num_splits)

    # 初始化结果矩阵
    final_result = np.zeros((A.shape[0], B.shape[1]))

    # 每个进程计算自己的块
    requests = []
    for block_idx in range(start_block, end_block):
        i = block_idx // num_splits
        j = block_idx % num_splits
        result_block = matrix_multiply(A_splits[i], B_splits[j])

        # 使用非阻塞发送将结果发送给根进程
        req = comm.Isend(result_block, dest=0, tag=block_idx)
        requests.append(req)

    # 根进程收集所有结果
    if rank == 0:
        split_size_A = A.shape[0] // num_splits
        split_size_B = B.shape[1] // num_splits
        for block_idx in range(num_splits * num_splits):
            i = block_idx // num_splits
            j = block_idx % num_splits
            result_block = np.zeros((split_size_A, split_size_B))
            req = comm.Irecv(result_block, source=MPI.ANY_SOURCE, tag=block_idx)
```

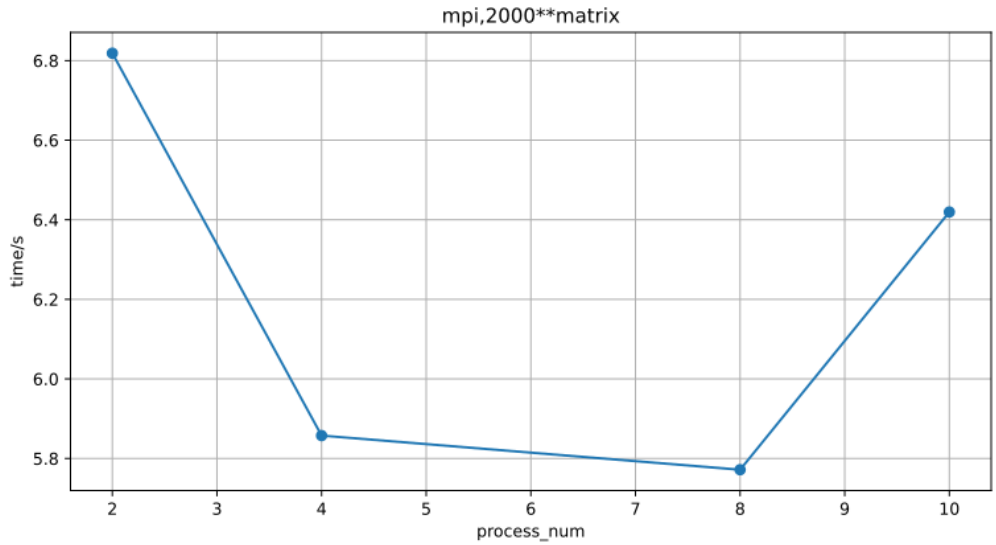
```
            req.Wait()
            final_result[i*split_size_A:(i+1)*split_size_A, j*split_size_B:
(j+1)*split_size_B] = result_block

    # 等待所有非阻塞发送完成
    MPI.Request.Waitall(requests)

    return final_result

if __name__ == "__main__":
    n = 10000
    A = np.random.rand(n, n)
    B = np.random.rand(n, n)
    num_splits = 10

    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()

    if rank == 0:
        starttime = time.time()
        result = parallel_matrix_multiply(A, B, num_splits)
        print("Time taken for parallel matrix multiply with MPI (non-
blocking):", time.time() - starttime)

        starttime = time.time()
        np.dot(A, B)
        print("Time taken for numpy dot product:", time.time() - starttime)

        print(result.shape)
        np.testing.assert_allclose(result, np.dot(A, B))
```

mpi4py结果（由于10000规模时mpi方法时间太长，不方便计量，所以没有成功收集，10进程下mpi方法时间超过了1800s,2进程下超过了5400s,效果较差。）



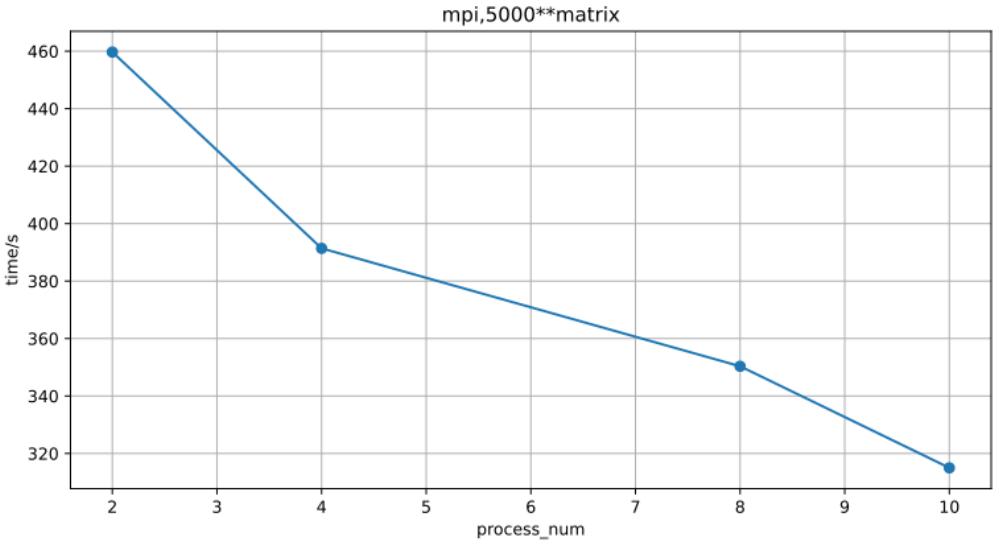
mpi4py库实现1000*1000规模并行矩阵乘法

横轴为实验进程数，纵轴为实验时间（s）



mpi4py库实现2000*2000规模并行矩阵乘法

横轴为实验进程数，纵轴为实验时间（s）



mpi4py库实现5000*5000规模并行矩阵乘法

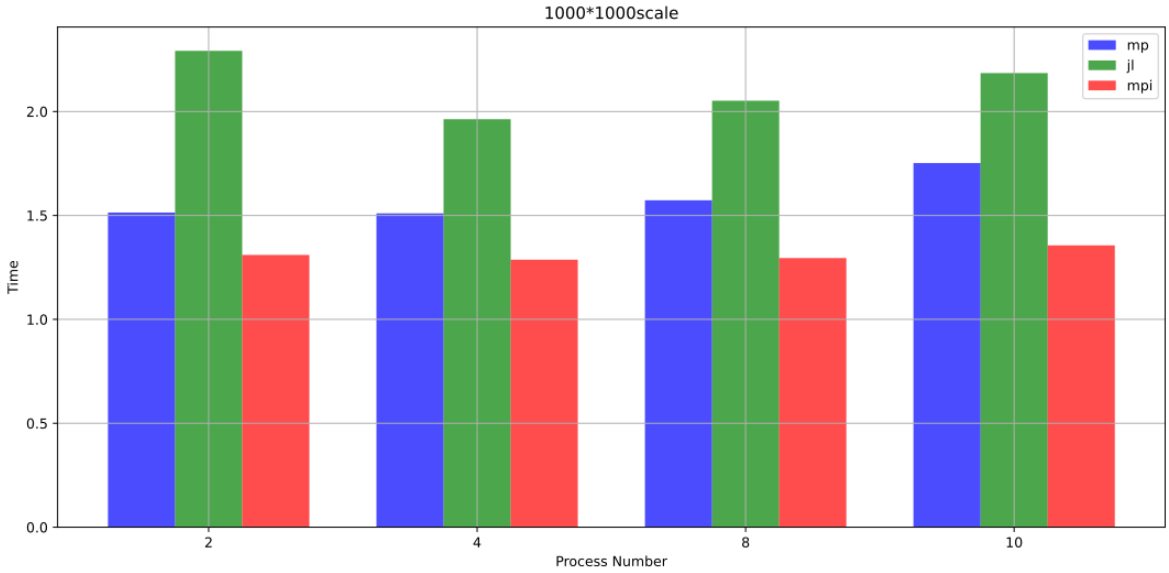
横轴为实验进程数，纵轴为实验时间（s）

经分析：
在数据规模较小时，并行引起的资源开销占主导地位，进程越多，时间越长，并行效率越低。
在数据规模较大时，并行运算带来的加速效果明显，总体上进程越多，时间缩短，效率提高。

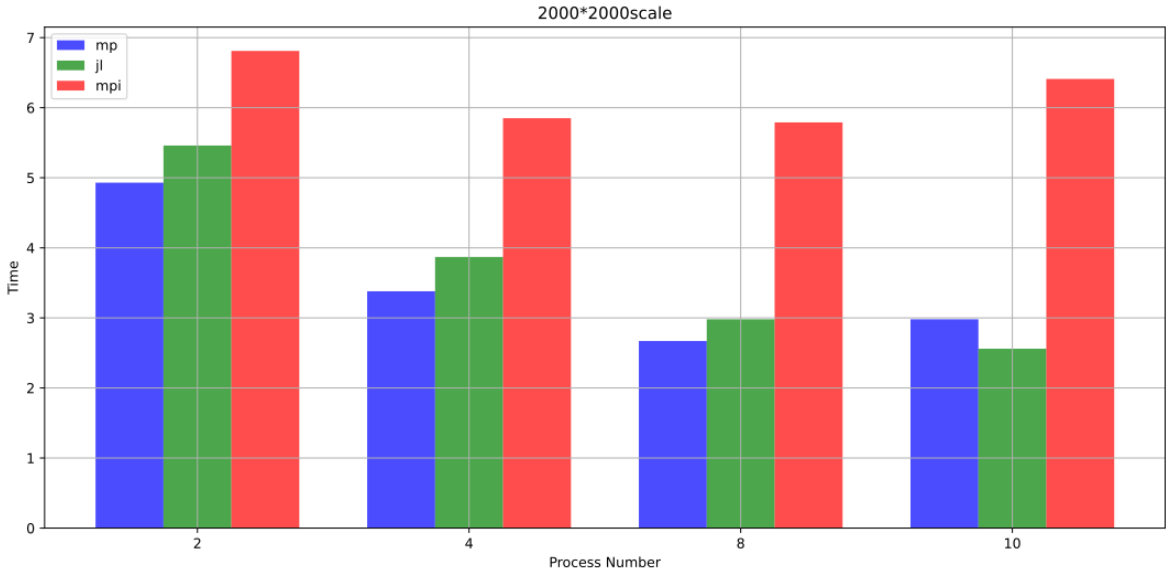
3.多方案的比较

（赛博斗蛐蛐环节）

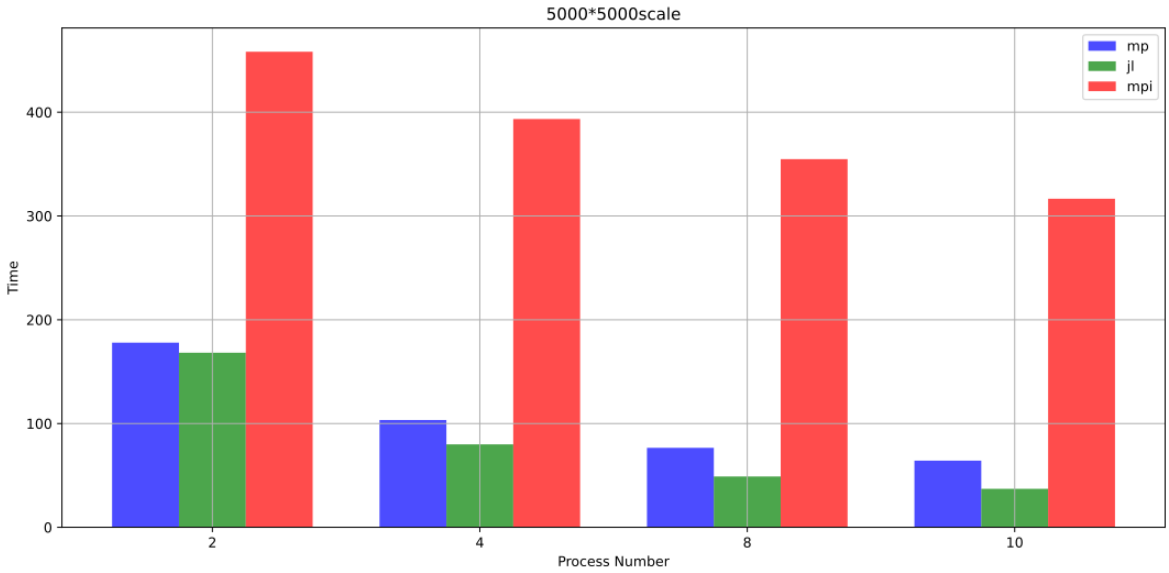
下方图表中，**蓝色**的mp代表**multiprocessing**库，**绿色**的jl代表**joblib**库，**红色**的mpi代表**mpi4py**库。



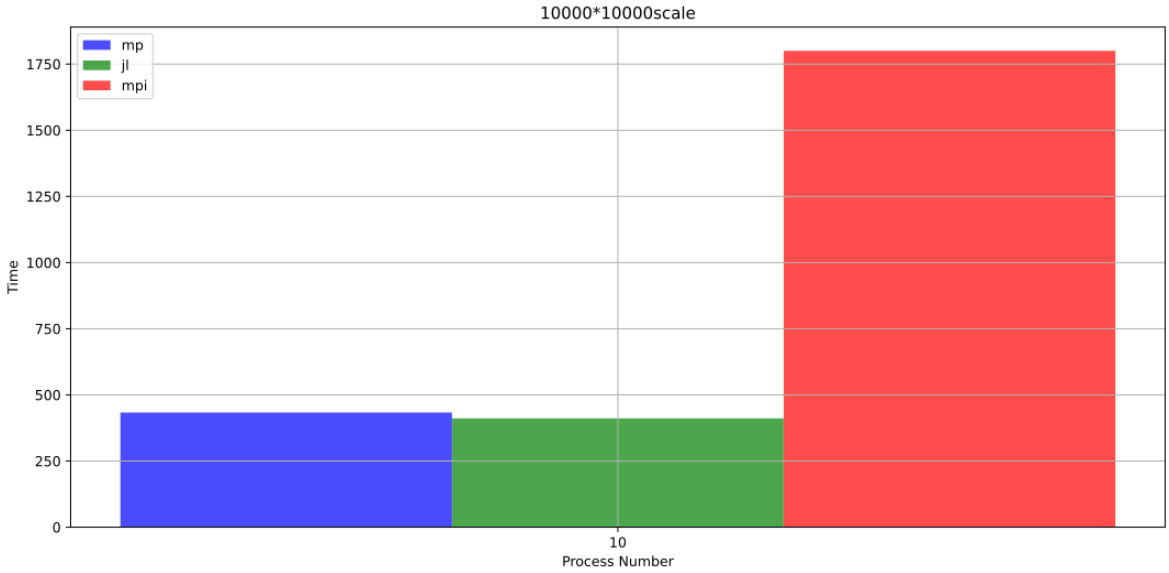
1000*1000矩阵乘法比较效果



2000*2000矩阵乘法比较效果



5000*5000矩阵乘法比较效果



10000*10000矩阵乘法比较效果

分析：
数据规模小时，mpi进程间非阻塞通讯提高了效率，良好利用了并行资源，速度较快；
数据规模大时，mpi方法由于单机局限，造成根进程瓶颈，主进程负担过大，并行效率下降明显。
multiprocessing库在数据规模大时效率明显高于mpi4py库，但是joblib还是比
单纯的multiprocessing高效。但是这并不表明mpi在大数据量时计算完全比不了
multiprocessing。

网上查询结果获取的结论如下：

“如果你的应用主要在单机上运行，且不需要跨节点的分布式计算，
multiprocessing 可能是一个更简单、更易上手的选择。
如果你的应用需要在多节点的集群上运行（如服务器层面），
或者需要更高效的进程间通信和更好的可扩展性，mpi4py 可能是更好的选择。”

三.实验过程中遇到的问题

1.调用已有实现矩阵乘法的库函数进行实验带来的问题

使用numpy.dot()方法计算矩阵乘法确实很快，不要任何处理，可以达到10000*10000规模的矩阵乘法5秒内结束，SciPy库里的dgemm方法也是差不多，但是如果我想参考它们并行的效率，那么所有多进程方法将跑不过单进程。由于害怕对实验结果分析有不好影响，最后，我选择最原始的办法，时间复杂度为O(n^3)，但是由于时间过长，采用numba的njit装饰器加速。

```
@njit
def matrix_multiply(A_block, B_block):
    # 初始化结果矩阵块
    result_block = np.zeros((A_block.shape[0], B_block.shape[1]))
    # 使用基本的矩阵乘法实现
    for i in range(A_block.shape[0]):
        for j in range(B_block.shape[1]):
            for k in range(A_block.shape[1]):
                result_block[i, j] += A_block[i, k] * B_block[k, j]
    return result_block
```

下面是使用numpy.dot()方法和scipy.linalg.blas中的dgemm方法实现并行矩阵乘法的示例。

```
#numpy内置方法完成矩阵乘法

import numpy as np
from multiprocessing import Pool
import time

def matrix_multiply(A, B):
    return A @ B

def split_matrix(A, B, num_splits):
    # 计算每个子块的大小
    split_size_A = A.shape[0] // num_splits
    split_size_B = B.shape[1] // num_splits
    # 分割矩阵 A 和 B
    A_splits = [np.ascontiguousarray(A[i*split_size_A:(i+1)*split_size_A]) for i
in range(num_splits)]
    B_splits = [np.ascontiguousarray(B[:, i*split_size_B:(i+1)*split_size_B])
for i in range(num_splits)]
    return A_splits, B_splits

def parallel_matrix_multiply(A, B, num_splits):
    # 使用 with 语句管理进程池
    with Pool(processes=num_splits) as pool:
        A_splits, B_splits = split_matrix(A, B, num_splits)
        results = []

        for i in range(num_splits):
            for j in range(num_splits):
                # 并行计算矩阵块的乘法
                result = pool.apply_async(matrix_multiply, (A_splits[i],
B_splits[j]))
                results.append((i, j, result))

        # 初始化结果矩阵
        final_result = np.zeros((A.shape[0], B.shape[1]))

        # 将子块结果合并到最终结果矩阵中
        split_size_A = A.shape[0] // num_splits
        split_size_B = B.shape[1] // num_splits
        for i, j, result in results:
            final_result[i*split_size_A:(i+1)*split_size_A, j*split_size_B:
(j+1)*split_size_B] = result.get()

        return final_result

if __name__ == "__main__":
    n = 10000
    A = np.random.rand(n, n)
    B = np.random.rand(n, n)
    num_splits = 8

    starttime = time.time()
    result = parallel_matrix_multiply(A, B, num_splits)
    print("Time taken:", time.time() - starttime)

    starttime = time.time()
    np.dot(A, B)
```

```
    print("Time taken:", time.time() - starttime)
    print(result.shape)
    np.testing.assert_allclose(result, np.dot(A, B))
```

```
#scipy中的矩阵乘法方法
import numpy as np
from scipy.linalg.blas import dgemm
import time

import numpy as np
from multiprocessing import Pool
import time
from scipy.linalg.blas import dgemm

def matrix_multiply(A, B):
    result = dgemm(alpha=1.0, a=A, b=B)
    return result

def split_matrix(A, B, num_splits):
    # 计算每个子块的大小
    split_size_A = A.shape[0] // num_splits
    split_size_B = B.shape[1] // num_splits
    # 分割矩阵 A 和 B
    A_splits = [np.ascontiguousarray(A[i*split_size_A:(i+1)*split_size_A]) for i
in range(num_splits)]
    B_splits = [np.ascontiguousarray(B[:, i*split_size_B:(i+1)*split_size_B])
for i in range(num_splits)]
    return A_splits, B_splits

def parallel_matrix_multiply(A, B, num_splits):
    # 使用 with 语句管理进程池
    with Pool(processes=num_splits) as pool:
        A_splits, B_splits = split_matrix(A, B, num_splits)
        results = []

        for i in range(num_splits):
            for j in range(num_splits):
                # 并行计算矩阵块的乘法
                result = pool.apply_async(matrix_multiply, (A_splits[i],
B_splits[j]))
                results.append((i, j, result))

        # 初始化结果矩阵
        final_result = np.zeros((A.shape[0], B.shape[1]))

        # 将子块结果合并到最终结果矩阵中
        split_size_A = A.shape[0] // num_splits
        split_size_B = B.shape[1] // num_splits
        for i, j, result in results:
            final_result[i*split_size_A:(i+1)*split_size_A, j*split_size_B:
(j+1)*split_size_B] = result.get()

        return final_result

if __name__ == "__main__":
    n = 10000
```

```
    A = np.random.rand(n, n)
    B = np.random.rand(n, n)
    num_splits = 8

    starttime = time.time()
    result = parallel_matrix_multiply(A, B, num_splits)
    print("Time taken:", time.time() - starttime)

    starttime = time.time()
    np.dot(A, B)
    print("Time taken:", time.time() - starttime)
    print(result.shape)
    np.testing.assert_allclose(result, np.dot(A, B))
```

由于并行效率分析结果不佳，两个方法都被废弃。但是当要进行大规模矩阵乘法时，这两个都是很好的方法。

2.OpenMPI安装困难

采用一堆方法安装最后没有成功，报错一直是'no module named mpi4py'，最后换用简单易用的MPICH解决。
具体实现见前面MPICH的安装。

3.ubuntu系统中安装的matplotlib进行数据可视化时调用plt.show()方法报错。

警告信息表明Matplotlib 当前使用的是 agg后端，这是一个非GUI后端，通常用于生成图形文件而不是在屏幕上显示图形。如果你想要显示图形，你需要确保Matplotlib 使用的是一个 GUI后端，比如 TkAgg 、 Qt5Agg等。可以通过在代码中显式设置后端来解决这个问题，如:import matplotlib matplotlib.use('TkAgg')。import matplotlib.pyplot as plt确保这个设置在导入 pyplot之前进行。

其实也没有必要这样，只要不调用plt.show()就行，我们只需要将svg矢量图保存即可。具体可视化示例可见仓库*save.py*文件。

```
plt.savefig('文件名。svg',format='svg')
```

四.尾声

1.SciPy库linalg模块的浅显涉猎

scipy.linalg 是 SciPy 库中的一个模块，它提供了大量的线性代数运算功能。这个模块基于 NumPy 数组，并且与 NumPy 的 numpy.linalg 模块有很多相似之处，但 scipy.linalg 提供了更多的功能，特别是针对稀疏矩阵和一些高级线性代数运算。

> scipy.linalg 的一些主要功能包括：
>
> ·矩阵分解：包括奇异值分解（SVD）、QR 分解、LU 分解、Cholesky 分解等。这些分解对于解决线性方程组、最小二乘问题、特征值问题等非常重要。
>
> ·求解线性方程组：提供了多种方法来求解线性方程组，包括使用 LU 分解、QR 分解和求解器专门针对稀疏矩阵。
>
> ·特征值和特征向量：可以计算矩阵的特征值和特征向量，这对于许多科学和工程问题非常重要，比如稳定性分析、振动分析等。

> ·矩阵函数：提供了一些矩阵函数，如矩阵的平方根、指数、对数等。
>
> ·优化和正则化：提供了一些用于优化和正则化的工具，如最小二乘法和 Tikhonov 正则化。
>
> 后期学习可以进行一些深入了解。

2.矩阵乘法一种算法优化--*Strasssen*算法

*Strassen*算法：
Strassen算法是一种分治算法，用于加速矩阵乘法。它将两个 2×2 ,2×2 矩阵的乘法分解为7次较小的矩阵乘法，从而减少所需的乘法次数。Strassen算法的基本思想是将原始问题分解为更小的子问题，然后递归地解决这些子问题。

Strassen算法的步骤如下：

```
1.将输入的矩阵A，B分割为四个n/2,n/2的子矩阵；
2.使用7次矩阵计算乘法计算10个中间矩阵；
3.合并这些中间矩阵以得到结果矩阵；
4.递归的思想。
```

Strassen算法的实现较为复杂，需要处理子矩阵的分割和合并。在实际应用中，Strassen算法在小矩阵上可能不如传统算法快，因为递归的开销可能超过节省的乘法次数。但对于大型矩阵，Strassen算法可以显著提高效率。

下面是一个Strassen算法的Python实现示例。这个实现考虑了矩阵大小为2的幂的情况，并使用递归方法来计算矩阵乘法。

```python
import numpy as np

def strassen(A, B):
    # 基本情况：当矩阵大小为1时，直接计算乘积
    if len(A) == 1:
        return A * B

    # 分割矩阵
    n = len(A)
    half = n // 2
    A11 = A[:half, :half]
    A12 = A[:half, half:]
    A21 = A[half:, :half]
    A22 = A[half:, half:]
    B11 = B[:half, :half]
    B12 = B[:half, half:]
    B21 = B[half:, :half]
    B22 = B[half:, half:]

    # 计算P1到P7
    P1 = strassen(A11 + A22, B11 + B22)
    P2 = strassen(A21 + A22, B11)
    P3 = strassen(A11, B12 - B22)
    P4 = strassen(A22, B21 - B11)
    P5 = strassen(A11 + A12, B22)
    P6 = strassen(A21 - A11, B11 + B12)
    P7 = strassen(A12 - A22, B21 + B22)

    # 合并结果
    C11 = P1 + P4 - P5 + P7
```

```
    C12 = P3 + P5
    C21 = P2 + P4
    C22 = P1 - P2 + P3 + P6

    # 构造结果矩阵
    C = np.vstack((np.hstack((C11, C12)), np.hstack((C21, C22))))

    return C

# 示例
A = np.array([[1, 2], [3, 4]])
B = np.array([[2, 0], [1, 3]])

result = strassen(A, B)
print(result)
```

3.Numba利用装饰器cuda.jit利用GPU加速

*numba*库实现了对CUDA的python绑定，允许开发者使用Python编写CUDA内核，并将其自动编译为可以在GPU上运行的代码。

下面是一个利用numba调用CUDA完成矩阵乘法的方法

```
from numba import cuda
import numpy as np

@cuda.jit
def matrixMulCUDA(A, B, C, n):
    row, col = cuda.grid(2)
    if row < n and col < n:
        sum = 0
        for i in range(n):
            sum += A[row, i] * B[i, col]
        C[row, col] = sum

# 设置矩阵大小
n = 1024
A = np.random.rand(n, n)
B = np.random.rand(n, n)
C = np.zeros((n, n))

# 设置CUDA网格和块的大小
threads_per_block = (16, 16)
blocks_per_grid_x = int(np.ceil(n / threads_per_block[0]))
blocks_per_grid_y = int(np.ceil(n / threads_per_block[1]))
blocks_per_grid = (blocks_per_grid_x, blocks_per_grid_y)

# 分配设备内存并复制数据
d_A = cuda.to_device(A)
d_B = cuda.to_device(B)
d_C = cuda.device_array((n, n))

# 调用CUDA内核
matrixMulCUDA[blocks_per_grid, threads_per_block](d_A, d_B, d_C, n)

# 将结果复制回主机
C = d_C.copy_to_host()
```

```
# 输出结果
print(C)
```

4.其他多进程方法

1. pathos模块
2. concurrent.futures模块

但是由于考核时间紧张，只是粗略看了一点，其实与multiprocessing库方法类似。

最后的保留节目

特别鸣谢

·**太阳王子*THINKER-ONLY***[*https://github.com/THINKER-ONLY*](https://github.com/THINKER-ONLY)
·**浩神*Howxu***[*https://github.com/HowXu*](https://github.com/HowXu)
·***orchid***[*https://github.com/orchiddell0*](https://github.com/orchiddell0)
·**彩彩*CAICAIIs***[*https://github.com/CAICAIIs*](https://github.com/CAICAIIs)
·***longtitle*仙贝**[*https://github.com/tinymonster123*](https://github.com/tinymonster123)
·**客服小祥**[*https://github.com/hangone*](https://github.com/hangone)