به نام خدا

پدرام رستمی – ۸۱۰۱۰۰۳۵۳

گزارش تمرین اول درس یادگیری ماشین توزیع شده

سوال ۱.

الف) در این بخش کد مربوط به ضرب دو ماتریس min به کمک ۳ حلقه پیاده سازی شده است. در این کد تابع read_matrix برای خواندن ماتریس‌ها به روش row major و از تابع write_matrix برای نوشتن ماتریس نهایی به روش column major استفاده می‌شود. در شکل ۱ پیاده سازی این توابع قابل مشاهده است.

```python
def read_matrix(path):
    with open(path, 'r') as f:
        raw_matrix = [int(line.strip()) for line in f]
    matrix_size = int(math.sqrt(len(raw_matrix)))
    matrix = [[0] * matrix_size for _ in range(matrix_size)]
    for idx, val in enumerate(raw_matrix):
        matrix[int(idx // matrix_size)][int(idx % matrix_size)] = val
    return matrix, matrix_size

def write_matrix(matrix, matrix_size, path):
    output = ""
    for i in range(matrix_size):
        for j in range(matrix_size):
            output += str(matrix[j][i]) + '\n'
    with open(path, 'w') as file:
        file.write(output)
```

شکل ۱- پیاده سازی توابع خواندن ماتریس‌ها و نوشتن ماتریس نهایی

سپس برای پیاده سازی الگوریتم ضرب ماتریس‌ها، از حلقه‌ی اول برای iterate بر روی سطرهای ماتریس اول، از حلقه‌ی دوم برای iterate بر روی ستون‌های ماتریس دوم و از حلقه‌ی سوم برای iterate بر روی عناصر سطر و ستون برای ضرب کردن آن‌ها استفاده می‌شود. پیاده سازی این الگوریتم در شکل ۲ قابل مشاهده است.

```python
start_time = time.time()
parent_path = '/home/shared_files/CA1/'
A_matrix, A_matrix_size = read_matrix(parent_path + 'A_matrix_min.txt')
B_matrix, B_matrix_size = read_matrix(parent_path + 'B_matrix_min.txt')
assert A_matrix_size == B_matrix_size
C_matrix = [[0] * A_matrix_size for _ in range(A_matrix_size)]
for i in range(A_matrix_size):
    for j in range(A_matrix_size):
        for k in range(A_matrix_size):
            C_matrix[i][j] += A_matrix[i][k] * B_matrix[k][j]
write_matrix(C_matrix, A_matrix_size, "C_matrix_simple.txt")
end_time = time.time()
print(f"time: {end_time - start_time} (s)")
```

شکل ۲ - پیاده سازی الگوریتم رب دو ماتریس به کمک ۳ حلقه

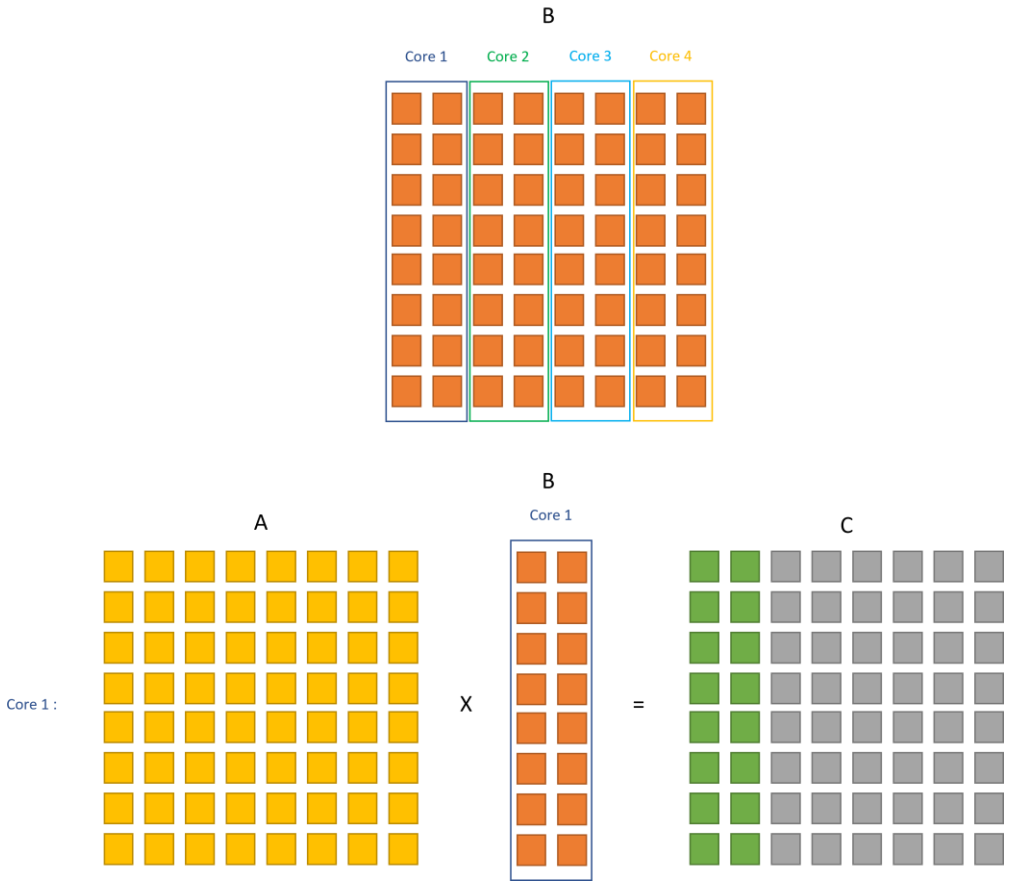
ب) در این بخش همچنان از توابع read_matrix و write_matrix بخش قبل برای خواندن و نوشتن ماتریس‌ها استفاده می‌شود. تنها تفاوت در این است که در تابع read_matrix نوع خروجی از لیست به آرایه‌ی numPy تغییر می‌کند. پیاده سازی الگوریتم این بخش بسیار راحت است و تنها کافی است از تابع matmul برای ضرب دو ماتریس استفاده شود. در این بخش حاصل ضرب ماتریس‌های min در فایل C_matrix_min_numpy.txt و حاصل ضرب ماتریس‌های عادی در فایل C_matrix_numpy.txt ذخیره می‌شوند. در شکل ۳ کد مربوط به این بخش آورده شده است.

```
parent_path = '/home/shared_files/CA1/'
start_time = time.time()
A_matrix, A_matrix_size = read_matrix(parent_path + 'A_matrix_min.txt')
B_matrix, B_matrix_size = read_matrix(parent_path + 'B_matrix_min.txt')
assert A_matrix_size == B_matrix_size
C_matrix = np.matmul(A_matrix, B_matrix)
end_time = time.time()
print(f"Min matrices multiplication time: {end_time - start_time} (s)")
write_matrix(C_matrix, A_matrix_size, "C_matrix_min_numpy.txt")

start_time = time.time()
A_matrix, A_matrix_size = read_matrix(parent_path + 'A_matrix.txt')
B_matrix, B_matrix_size = read_matrix(parent_path + 'B_matrix.txt')
assert A_matrix_size == B_matrix_size
C_matrix = np.matmul(A_matrix, B_matrix)
end_time = time.time()
print(f"Matrices multiplication time: {end_time - start_time} (s)")
write_matrix(C_matrix, A_matrix_size, "C_matrix_numpy.txt")
```

شکل ۳ - پیاده سازی ضرب ماتریس‌ها به کمک تابع matmul

ج) در این بخش پیاده سازی ضرب دو ماتریس به کمک کتابخانه‌ی mpi4py انجام می‌شود. برای پیاده سازی این الگوریتم، ماتریس اول (ماتریس A) به صورت کامل خوانده می‌شود و ماتریس دوم (ماتریس B) به از طریق ستون‌هایش به اندازه‌ی تعداد کل هسته‌ها تقسیم می‌شود و هر هسته بخشی که متناظر با رنکش است را می‌خواند. از ضرب ماتریس A در ستون‌های خوانده شده از ماتریس B ستون‌های متناظر از ماتریس C بدست می‌آیند. در شکل ۴ این الگوریتم برای ۴ هسته نشان داده شده است و در قسمت پایین آن بخشی که هسته‌ی اول انجام می‌دهد قابل مشاهده است.



شکل ۴- الگوریتم پیاده سازی شده ضرب دو ماتریس در حالی که از ۴ هسته استفاده شده باشد.

برای پیاده سازی کد این بخش، همچنان از تابع read_matrix برای خواندن ماتریس A استفاده می‌شود. همچنین برای نوشتن ماتریس نهایی همچنان از تابع write_matrix استفاده می‌شود با این تفاوت که در این بخش چون می‌خواهیم داده‌های هر هسته را به فایل اضافه کنیم، فایل در حالت a+ باز می‌شود. تابع read_matrix_partition برای خواندن بخشی از ماتریس پیاده سازی شده است که کد آن در شکل ۵ قابل مشاهده است.

```
def read_matrix_partition(path, matrix_size, col_min, col_max):
    raw_matrix = []
    with open(path, 'r') as f:
        for idx, line in enumerate(f):
            if col_min <= idx % matrix_size <= col_max:
                raw_matrix.append(int(line.strip()))
    matrix = np.array([np.array([0] * (col_max - col_min + 1)) for _ in range(matrix_size)])
    new_matrix_size = col_max - col_min + 1
    for idx, val in enumerate(raw_matrix):
        matrix[int(idx // new_matrix_size)][int(idx % new_matrix_size)] = val
    del raw_matrix
    return matrix
```

شکل ۵ - پیاده سازی تابع read_matrix_partition که تنها بخشی از ماتریس را می‌خواند.

در شکل ۶ الگوریتم اصلی این بخش قابل مشاهده است. در این بخش هر هسته ابتدا ماتریس A را می‌خواند و سپس بخش مربوط به خود از ماتریس B را می‌خواند و از ضرب این دو ماتریس در هم بخش متناظر از ماتریس C را می‌سازد. سپس به کمک دستور gather متغیرهای C همه‌ی هسته‌ها در متغیر all_C به ترتیب رنکشان قرار می‌گیرند و هسته‌ای که رنکش صفر است، ماتریس نهایی را نوشته و زمان اجرای الگوریتم را هم حساب می‌کند.

```
if rank == 0:
    start_time = time.time()
parent_path = '/home/shared_files/CA1/'
A, matrix_size = read_matrix(parent_path + 'A_matrix.txt')
B = read_matrix_partition(parent_path + 'B_matrix.txt', matrix_size, rank * int(matrix_size // size), ((rank + 1) * int(matrix_size // size)) - 1)
C = np.matmul(A, B)
del A, B
all_C = comm.gather(C, root=0)
if rank == 0:
    print(f"size: {len(all_C)}")
    for i in range(len(all_C)):
        write_matrix(all_C[i], int(matrix_size // size), matrix_size, "parallel.txt")
    print(f"time: {time.time() - start_time} (s)")
```

شکل ۶ - پیاده سازی الگوریتم ضرب ماتریس بر روی چند هسته

فایل sbatch این بخش در شکل ۷ قابل مشاهده است. در فایل حافظه‌ی رزرو شده برابر ۴۰۰ مگابایت در نظر گرفته شده است که مشکلی برای اجرا حتی با یک هسته هم به وجود نیاید. (در حالتی که تنها از یک هسته استفاده شود حدودا ۳۰۰ مگابایت حافظه نیاز است.) همچنین تعداد نودها و هسته‌ها برابر ۱ در نظر گرفته شده‌اند.

```
#!/bin/bash
#SBATCH --job-name=matrix_multiplication_parallel_1n1c
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --partition=partition
#SBATCH --mem=400
#SBATCH --output=matrix_multiplication_parallel_1n1c.out
echo "Jobs are started..."
srun --mpi=pmix_v4 python3 matrix_multiplication_parallel_1n1c.py
```

شکل ۷ - فایل sbatch در حالتی که از ۱ نود و ۱ هسته استفاده شود

برای حالتی که قصد داشته باشیم از ۱ نود و ۲ هسته استفاده کنیم، مقدار ntasks-per-node– برابر ۲ می‌شود.

د) در این بخش تمام اتفاقات بخش قبل تکرار می‌شود با این تفاوت که در فایل sbatch مقدار nodes– برابر ۲ و مقدار ntasks-per-node– برابر ۲ فرض می‌شوند.

ه) در این بخش زمان الگوریتم‌های ضرب ماتریس در حالت‌های مختلف مقایسه شده است. همانطور که مشخص است، ماتریس‌های اصلی با افزایش هسته‌ها و تسک‌ها زمان اجرایش کاهش می‌یابد و از ۲۴۹ ثانیه به ۸۰ ثانیه کاهش می‌یابد. در مقابل زمان اجرای ضرب ماتریس‌های min از ۰.۴۸ ثانیه به ۲.۴۱ ثانیه افزایش می‌یابد زیرا سایز این ماتریس‌ها کوچک است و موازی سازی آن‌ها تنها باعث افزایش زمان ضربشان می‌شود.

	ماتریس‌های اصلی	ماتریس‌های min
Simple	-------	7.30 (s)
Numpy	212.42 (s)	0.26 (s)
Parallel 1n1c	249.34 (s)	0.48 (s)
Parallel 1n2c	122.41 (s)	0.42 (s)
Parallel 2n2c	80.14 (s)	2.41 (s)

جدول ۱ – مقایسه زمانی الگوریتم‌های ضرب ماتریس در حالت‌های مختلف

در این سوال، نتیجه‌ی ضرب ماتریس‌های min در فایل‌هایی که پسوند min_ دارند ذخیره شده است.

سوال ۲.

الف) در این بخش کد مربوط به تخمین عدد PI به کمک ماژول Decimal و به روش مونت کارلو پیاده سازی شده است. کد این بخش در شکل ۸ قابل مشاهده است. در این حالت عدد PI برابر با مقدار زیر بدست آمد:

PI: 3.14133629999999985926706358441151678562164306640625

```python
import random
import time
from decimal import Decimal, getcontext


i = 40_000_000
counter = 0
getcontext().prec = 40
start_time = time.time()
for _ in range(i):
    a, b = random.uniform(-1, 1), random.uniform(-1, 1)
    if (a ** 2 ) + (b ** 2) <= 1:
        counter += 1
print(f"PI: {Decimal(4 * counter / i)}")
print(f"time: {time.time() - start_time} (s)")
```

شکل ۸ - پیاده سازی الگوریتم مونت کارلو برای تخمین عدد PI

ب) برای پیاده سازی کد این بخش، حلقه‌ی اصلی به تعداد تسک‌ها شکسته و بین همه پخش می‌شود. سپس در نهایت مقدار counter همه‌ی تسک‌ها با دستور reduce جمع شده و در هسته‌ای که رنکش صفر است ذخیره

می‌شود. در شکل ۹ پیاده سازی این الگوریتم قابل مشاهده است. در این بخش به علت این که حداکثر تسک‌های هر نود ۲ است، نمی‌توان از ۴ هسته‌ی یک نود استفاده کرد.

```python
from mpi4py import MPI
import random
import time
from decimal import Decimal, getcontext

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
i = 40000000
counter = 0
getcontext().prec = 40
if rank == 0:
    start_time = time.time()
for _ in range(int(i // size)):
    a, b = random.uniform(-1, 1), random.uniform(-1, 1)
    if (a ** 2 ) + (b ** 2) <= 1:
        counter += 1

all_counters = comm.reduce(counter, root=0)
if rank == 0:
    print(f"PI: {Decimal(4 * all_counters / i)}")
    print(f"time: {time.time() - start_time} (s)")
```

شکل ۹- پیاده سازی الگوریتم تخمین عدد PI به صورت موازی

ج) در این بخش از همان کد پایتون بخش قبل استفاده می‌شود. در فایل sbatch تعداد نودها برابر ۲ و تعداد تسک‌ها در هر نود هم برابر ۲ فرض می‌شود. در شکل ۱۰ فایل sbatch این بخش قابل مشاهده است. مقدار عدد PI هم در زیر قابل مشاهده است:

PI: 3.14175650000000006301092980720568448305130004882812 5

```
#!/bin/bash
#SBATCH --job-name=pi_parallel_2n2c
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=2
#SBATCH --partition=partition
#SBATCH --output=pi_parallel_2n2c.out
echo "Jobs are started..."
srun --mpi=pmix_v4 python3 pi_parallel_1n4c.py
```

شکل ۱۰ - فایل sbatch برای حالتی که از ۲ نود و هر نود ۲ هسته استفاده می‌شود.

د) در این بخش همچنان از کد پایتون بخش ب استفاده می‌شود. در حالتی که از یک نود و یک هسته استفاده می‌شود، مقادیر nodes و ntasks-per-node برابر ۱ فرض می‌شوند و در حالتی که از یک نود و ۲ هسته استفاده می‌شود، مقدار nodes ۱ و ntasks-per-node ۲ فرض می‌شود.

ه) در جدول ۲ مقایسه‌ی زمانی تخمین PI در حالت‌های مختلف نشان داده شده است. همانطور که مشخص است با افزایش تسک‌ها و هسته‌ها زمان اجرای الگوریتم کاهش می‌یابد.

	تخمین PI
Serial	109.20 (s)
Parallel 1n1c	106.70 (s)
Parallel 1n2c	54.43 (s)
Parallel 2n2c	28.62 (s)

جدول ۲ – مقایسه زمانی الگوریتم تخمین PI در حالت‌های مختلف

برای این سوال، برای بخش‌های 1n1c، 1n2c و 2n2c از همان کد بخش 1n4c استفاده شده است. برای همین تنها کد این بخش قرار گرفته است.