به نام خداوند جان و خرد

دانشگاه تهران

دانشکدگان فنی

دانشکده مهندسی برق و کامپیوتر

سامانه‌های یادگیری‌ماشین توزیع شده

تمرین شماره ۱

نام و نام خانوادگی: علی خرم فر

شماره دانشجویی: ۸۱۰۱۰۲۱۲۹

مهرماه ۱۴۰۳

فهرست مطالب

فهرست جداول

۱_ پاسخ سوال شماره ۱

در این سوال، هدف محاسبه مقدار تقریبی عدد رادیکال ۲ با استفاده از سری تیلور است.

۱-۱_ پاسخ بخش الف)

از آنجا که این کد بدون استفاده از sbatch و به صورت مستقیم در محیط اجرا شده است، خروجی‌ها مستقیماً در ترمینال چاپ شده‌اند و در فایل .out ذخیره نشده‌اند.

```
Sum of Series: 1.414213562373095
Execution time: 18.69730305671692 seconds
```

زمان اجرای کد برای محاسبه ۵۰۰۰ جمله از سری حدودا برابر با **۱۸.۷** ثانیه است.

۱-۲_ پاسخ بخش ب)

در این نسخه از کد، دو متغیر fact_k و fact_2k_plus_1 تعریف شده‌اند که فاکتوریل‌های !k و !(2k+1) را ذخیره می‌کنند. این متغیرها به جای محاسبه دوباره در هر حلقه، در هر مرحله با استفاده از مقادیر قبلی به‌روز می‌شوند.توان ۲ و سایر محاسبات همچنان در هر مرحله انجام می‌شوند، اما با توجه به اینکه محاسبات فاکتوریل کمتر از ۸ ثانیه شد از اجرای آن صرف نظر شد هرچند که این پیاده‌سازی نیز در پوشه Additional موجود است.

```
Sum of Series: 1.414213562373095
Execution time: 4.892261505126953 seconds
```

مقدار محاسبه شده برای جمع سری پس از ۵۰۰۰ جمله همانند بخش الف برابر با ۱.۴۱۴۲۱۳۵۶۲۳۷۳۰۹۵ و زمان اجرای کد بهینه‌شده برابر با **۴.۸۹** ثانیه است.

۱-۳_ پاسخ بخش ج)

در این بخش، الگوریتم مرحله الف به صورت توزیع شده اجرا شد. در این روش، بار محاسباتی به دو بخش تقسیم شد. هسته اول محاسبات جملات ۱ تا ۲۵۰۰ و هسته دوم محاسبات جملات ۲۵۰۱ تا ۵۰۰۰ را انجام داد و به نحوی کد نوشته شد که برای هر مقدار n این تقسیم بندی به خوبی انجام شود.

توضیحات استفاده از کتابخانه MPI و Slurm

دستورات MPI نقش اصلی در توزیع بار محاسباتی و جمع‌آوری نتایج ایفا می‌کنند. COMM_WORLD شامل هسته‌ها در کلاستر است. تمام هسته‌ها از طریق این ارتباط قادر به برقراری ارتباط با یکدیگر هستند. rank = comm.Get_rank()عدد rank هر هسته را تعیین می‌کند. rank یک عدد منحصر

به فرد برای هر هسته است که از ۰ شروع می‌شود و size = comm.Get_size() تعداد کل هسته‌ها را برمی‌گرداند.

```
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
```

comm.reduce نتایج محاسباتی همه هسته‌ها را در هسته اصلی جمع‌آوری می‌کند. عملیات MPI.SUM نشان می‌دهد که مقادیر sum_core از هر هسته با هم جمع می‌شوند و نتیجه نهایی به هسته ۰ فرستاده می‌شود.

comm.Bcast این دستورات برای پخش مقادیر از هسته اصلی به سایر هسته‌ها استفاده می‌شود که در بخش بعدی تمرین استفاده خواهیم کرد.

برای اجرای کدی که با کمک کتابخانه MPI نوشتیم دو راه وجود دارد. یکی استفاده از دستور mpiexec که در ویندوز برای تست کدهای خود استفاده کردیم یکی هم mpirun که در کلاسترها و محیط آن استفاده شد. ولی کدهای نهایی به کمک Slurm و دستور Sbatch اجرا شد. برای هر کد، اسکریپت مربوط به آن نیز پیوست شد. مثلا برای قسمت بعدی که یک نود و دو هسته نیاز داریم اسکریپت زیر نوشته شد:

```
/!#bin/bash
#SBATCH --job-name=sqrt_c_1
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=2
#SBATCH --partition=partition
#SBATCH --output=sqrt_c_1.out

echo "Job is started"

srun --mpi=pmix_v4 python3 sqrt_c_1.py

echo "Job is done"
```

اجرا بر روی یک نود و دو هسته

در کد، این تقسیم‌بندی با استفاده از محاسبه مقادیر start و end برای هر هسته انجام می‌شود که بر اساس rank هسته و تعداد جملات n مشخص می‌شود. در نهایت، نتایج محاسبات هر دو هسته با استفاده از تابع MPI.reduce جمع شده و به هسته اصلی (root 0) فرستاده و نتیجه نهایی توسط آن چاپ می‌شود.

```
Rank:  1 Start:  2500 End:  5000
Rank:  0 Start:  0 End:  2500
Sum of Series: 1.414213562373095
Execution time: 15.944663286209106 seconds
```

اجرای موازی با دو هسته باعث توزیع بار محاسباتی شد و زمان اجرای الگوریتم **۱۵.۹۴** ثانیه شد. این مقدار نسبت به اجرای تک هسته‌ای بهبود یافت.

اجرا بر روی دو نود و دو هسته

```
Rank:  3 Start:  3750 End:  5000
Rank:  1 Start:  1250 End:  2500
Rank:  0 Start:  0 End:  1250
Sum of Series: 1.4142135623730950
Execution time: 10.272013902664185 seconds
Rank:  2 Start:  2500 End:  3750
```

زمان اجرای کل برنامه برابر با **۱۰.۲۷** ثانیه شد، که نسبت به اجرای یک نود و دو هسته کاهش زمان محسوسی را نشان می‌دهد.

این کاهش زمان در این بخش از تمرین به دلیل تقسیم مساوی بار محاسباتی بین نودها و هسته‌ها بوده و پردازش موازی به بهبود عملکرد الگوریتم کمک کرده‌است.

۴-۱_ پاسخ بخش د)

الگوریتم پیاده‌سازی شده در این بخش با استفاده از بهینه‌سازی فاکتوریل‌ها و تقسیم بار محاسباتی بین چندین هسته ، کارایی بیشتری نسبت به حالت اولیه دارد و زمان اجرای کلی را به طور قابل ملاحظه‌ای کاهش می‌دهد.

اجرا بر روی یک نود و دو هسته

این بار نیز از روش قطعه بندی متوالی برای توزیع بار محاسباتی استفاده شد، اما از الگوریتم بهینه‌شده مرحله ب (استفاده از فاکتوریل‌های قبلی برای کاهش محاسبات) استفاده کردیم و مشابه مراحل قبلی، بار محاسباتی بین دو هسته یک نود تقسیم شد. هر هسته نیمی از جملات را محاسبه کرد.

```
Sum of Series: 1.4142135623730950
Execution time: 4.051809787750244 seconds
```

با استفاده از دو هسته و بهره‌گیری از الگوریتم بهینه‌شده، زمان اجرای برنامه به **۴.۰۵** ثانیه کاهش یافت.

اجرا بر روی دو نود و دو هسته

```
Sum of Series: 1.4142135623730950
Execution time: 2.5781447887420654 seconds
```

با اجرای برنامه بر روی دو نود و هر نود با دو هسته، زمان اجرای محاسبات به **۲.۵۸** ثانیه کاهش یافت.

۵-۱_ پاسخ بخش ه)

در اجرای قبلی (بخش ج)، محاسبات به صورت متوالی بین هسته‌ها تقسیم می‌شد، که باعث ایجاد عدم توازن در بار محاسباتی می‌شد که دلیل آن این است که محاسبه فاکتوریل اعداد بزرگتر بار محاسباتی

بیشتری خواهند داشت و باید تمام جملات قبلی هم محاسبه شوند در صورتی که تابع State ها را ذخیره نمی‌کند. برای رفع این مشکل، هر هسته به جای محاسبه یک بازه‌ی متوالی از جملات، بر اساس rankش در فواصل ثابت و به صورت متناوب جملات را پردازش می‌کند. مثلا هسته ۰ ابتدا جمله ۰، سپس جمله ۴، و به همین ترتیب ادامه می‌دهد.

```
Sum of Series: 1.414213562373095
Execution time: 4.7031683921813965 seconds
```

در مقایسه با اجرای قبلی که زمان اجرا ۱۰.۲۷ ثانیه بود، زمان اجرای جدید به **۴.۷۰** ثانیه کاهش یافت یعنی تقریبا نصف شد که بسیار نتیجه جالبی بود.

۶-۱_ نتیجه‌گیری

بخش	مقدار خروجی	زمان اجرا	توضیحات
الف	۱.۴۱۴۲۱۳۵۶۲۳۷۳۰۹۵	۱۸.۶۹۷	اجرای بدون بهینه‌سازی روی یک هسته، زمان بالا
ب	۱.۴۱۴۲۱۳۵۶۲۳۷۳۰۹۵	۴.۸۹۲	بهینه‌سازی فاکتوریل، کاهش زمان
ج- یک نود دو هسته	۱.۴۱۴۲۱۳۵۶۲۳۷۳۰۹۵	۱۵.۹۴۴	اجرای موازی با دو هسته، بهبود متوسط
ج -دو نود دو هسته	۱.۴۱۴۲۱۳۵۶۲۳۷۳۰۹۵	۱۰.۲۷۲	اجرای موازی با دو نود و دو هسته، توزیع بهتر
د-یک نود دو هسته	۱.۴۱۴۲۱۳۵۶۲۳۷۳۰۹۵	۴.۰۵۲	الگوریتم بهینه‌شده روی یک نود و دو هسته، سریع‌تر
د - دو نود دو هسته	۱.۴۱۴۲۱۳۵۶۲۳۷۳۰۹۵	۲.۵۷۸	بهینه‌سازی و اجرای روی دو نود و دو هسته، سریع‌ترین اجرا
ه	۱.۴۱۴۲۱۳۵۶۲۳۷۳۰۹۵	۴.۷۰۳	توزیع بهتر بار محاسباتی ، بهبود قابل‌توجه

جدول ۱ مقایسه خروجی‌های مختلف الگوریتم‌های محاسبه سری

بررسی نتایج اجرای الگوریتم در بخش‌های مختلف نشان می‌دهد که استفاده از بهینه‌سازی‌ها و توزیع مناسب بار محاسباتی تاثیر زیادی در کاهش زمان اجرا داشته است. در بخش‌های اول مثل الف، اجرای کد بدون بهینه‌سازی باعث زمان‌های طولانی شد، اما با استفاده از بهینه‌سازی محاسبات فاکتوریل در کنار توزیع بار محاسباتی شاهد کاهش چشمگیری در زمان بودیم.

۲_پاسخ سوال شماره ۲

رگرسیون لجستیک یکی از الگوریتم‌های پرکاربرد در یادگیری ماشین است که برای طبقه‌بندی استفاده می‌شود. مثلاً در اینجا هدف پیش‌بینی این است که آیا داده‌های ورودی متعلق به دسته ۱ هستند یا صفر. در ادامه به بررسی کد نوشته شده برای پیاده‌سازی این الگوریتم خواهیم پرداخت.

۲-۱_ پاسخ بخش الف)

کد زیر یک پیاده‌سازی ساده از رگرسیون لجستیک به صورت عادی و سریال است که بر روی کلاستر اجرا شد:

```
x = np.load('data.npy')
y = np.load('labels.npy')

np.random.seed(۴۲)

n = np.arange(x.shape[0])
np.random.shuffle(n)

x = x[n]
y = y[n]

s = int(0.2 * x.shape[0])

X_train = x[:-s]
X_test = x[-s:]
y_train = y[:-s]
y_test = y[-s:]

def sigmoid(z):
  return 1 / (1 + np.exp(-z))

start_time = time.time()

lr = 0.01
iters = 1000

w = np.zeros(X_train.shape[1])
b = 0

for _ in range(iters):
  z = np.dot(X_train, w) + b
  y_pred = sigmoid(z)
  dw = np.dot(X_train.T, (y_pred - y_train)) / X_train.shape[۰]
  db = np.sum(y_pred - y_train) / X_train.shape[۰]

  w -= lr * dw
  b -= lr * db

z_test = np.dot(X_test, w) + b
y_pred_test = sigmoid(z_test)
y_pred_cls = np.where(y_pred_test > 0.5, 1, 0)

accuracy = np.sum(y_pred_cls == y_test) / len(y_test)
```

در این کد، ابتدا داده‌های ورودی و برچسب‌ها از فایل‌های data.npy و labels.npy که از فایل تمرین دریافت شدند، بارگذاری می‌شوند. سپس برای اطمینان از اینکه داده‌ها به‌طور تصادفی برای آموزش و آزمایش انتخاب شوند، داده‌ها به صورت تصادفی ترتیبشان تغییر می‌دهیم. پس از این مرحله، داده‌ها به دو بخش

تقسیم می‌شوند: ۸۰ درصد برای آموزش مدل و ۲۰ درصد برای تست. تابع Sigmoid هم پیاده‌سازی شد تا خروجی‌ها به مقادیر بین ۰ و ۱ تبدیل شوند.

در مرحله بعد، در یک حلقه یادگیری انجام می‌شود. وزن‌ها و بایاس در ابتدا به صفر مقداردهی می‌شوند. سپس در هر تکرار پیش‌بینی‌های مدل انجام شده و گرادیان‌های مربوط به وزن‌ها و بایاس محاسبه و به‌روزرسانی می‌شوند. این به‌روزرسانی‌ها بر اساس نرخ یادگیری تعیین‌شده ۰.۰۱ انجام می‌دهیم. بعد از یادگیری، مدل روی داده‌های تست ارزیابی می‌کنیم. پیش‌بینی‌ها ابتدا به صورت مقادیر احتمالاتی بین ۰ و ۱ محاسبه شده و سپس به کلاس‌های ۰ یا ۱ تبدیل می‌شوند.

در نهایت، دقت مدل از طریق مقایسه پیش‌بینی‌ها با لیبل‌های واقعی داده‌های تست محاسبه می‌شود.

```
Execution time: 0.6508533954620361 seconds
Accuracy: 0.952
```

دقت **۹۵.۲** درصدی مدل نشان می‌دهد که الگوریتم عملکرد خوبی در طبقه‌بندی داده‌ها داشته و زمان اجرای آن نیز **۰.۶۵** ثانیه بوده است.

۲-۲_ پاسخ بخش ب)

الگوریتم رگرسیون لجستیک به صورت مشابه با مرحله الف پیاده‌سازی شده است، اما این بار داده‌ها با استفاده از کتابخانه mpi4py بین هسته‌های مختلف تقسیم شده و نتایج هر هسته در نهایت توسط نود اصلی جمع آوری و وزن‌ها آپدیت می‌شوند. در ادامه به بررسی کد مربوطه می‌پردازیم:

در ابتدا داده‌های ورودی بین هسته‌ها تقسیم شده‌اند و هر هسته بخشی از داده‌ها را برای آموزش مدل استفاده می‌کند. در اینجا مشخص نیست که فرض سوال چیست اگر هدف حریم خصوصی است که اینجا فرض میکنیم ابتدا این هر نود فقط دیتایی که قرار است بین آنها تقسیم شود را از اول داشته. یک راه دیگر این است که یک گره مسئول تقسیم داده ها با کمک Scatter شود که این پیاده‌سازی دوم نیز در پوشه Additional قرار داده شد.

در هر مرحله از یادگیری، گرادیان محلی مربوط به هر هسته محاسبه شده و سپس این مشتق ها توسط MPI.Reduce به هسته اصلی فرستاده می‌شوند تا جمع و میانگیری شوند. هسته اصلی سپس وزن‌ها و بایاس به‌روز شده را محاسبه و مجدداً با استفاده از MPI.Bcast به سایر هسته‌ها ارسال می‌کند.

در پایان، هر هسته با استفاده از داده‌های آزمایشی خود مدل را ارزیابی می‌کند و دقت local محاسبه می‌شود. دقت‌های local توسط MPI.Gather به هسته اصلی ارسال می‌شوند تا میانگین دقت، دقت هر هسته و زمان اجرای کل گزارش شود.

```
def sigmoid(z):
  return 1 / (1 + np.exp(-z))

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

data = np.load('data.npy')
labels = np.load('labels.npy')

data_split = np.array_split(data, size)
labels split = np.array split(labels, size)

X_local = data_split[rank]
y_local = labels_split[rank]

np.random.seed(۴۲)

n = np.arange(X_local.shape[0])
np.random.shuffle(n)

X_local = X_local[n]
y_local = y_local[n]

s = int(0.2 * X_local.shape[0])

X_train_local = X_local[:-s]
X_test_local = X_local[-s:]
y_train_local = y_local[:-s]
y_test_local = y_local[-s:]

lr = 0.01
iters = 1000
n_samples, n_features = X_train_local.shape
w = np.zeros(n_features)
b = np.zeros(1)  # For Bcast Problem

start_time = time.time()

for _ in range(iters):
  z = np.dot(X_train_local, w) + b[۰]
  y_pred = sigmoid(z)
  dw_local = np.dot(X_train_local.T, (y_pred - y_train_local)) / n_samples
  db_local = np.sum(y_pred - y_train_local) / n_samples

  dw_total = np.zeros(n_features)
  db_total = np.zeros(۱)

  comm.Reduce(dw_local, dw_total, op=MPI.SUM, root=0)
  comm.Reduce(db local, db total, op=MPI.SUM, root=0)

  if rank == 0:
    dw total /= size
    db_total /= size
    w -= lr * dw_total
    b -= lr * db_total

  comm.Bcast(w, root=0)
  comm.Bcast(b, root=0)

end_time = time.time()
exec_time = end_time - start_time

z_test = np.dot(X_test_local, w) + b[۰]
y_pred_test = sigmoid(z_test)
y_pred_cls = np.where(y_pred_test > 0.5, 1, 0)

local_accuracy = np.sum(y_pred_cls == y_test_local) / len(y_test_local)

all_accuracies = comm.gather(local_accuracy, root=0)
all_times = comm.gather(exec_time, root=0)
```

```
if rank == 0:
  total_end_time = time.time()
  total_exec_time = total_end_time - start_time
  print("Total execution time:", total_exec_time, "seconds")

  for i in range(size):
    print("Node", i, ": Local Accuracy =", all_accuracies[i], "Execution Time =",
all_times[i], "seconds")

  avg_accuracy = np.mean(all_accuracies)
  avg time = np.mean(all times)

  print("Average Accuracy:", avg_accuracy)
  print("Average Execution Time per Node:", avg_time, "seconds")
```

هرچند که می‌شد بجای Reduce ابتدا کل را با Gather دریافت و سپس میانگین گیری کرد ولی این جمع ضمنی به افزایش سرعت کمک می‌کرد

نتایج اجرا:

زمان اجرای کل: ۱.۲۷۶۵ ثانیه

دقت هسته ۰: ۹۳.۹٪ زمان اجرا: ۱.۲۷ثانیه

دقت هسته ۱: ۹۶.۲٪ زمان اجرا: ۱.۲۷ثانیه

دقت متوسط: ۹۵.۰۵٪

زمان اجرای متوسط هر هسته: ۱.۲۷۵۷ ثانیه

اگرچه موازی‌سازی معمولاً برای بهبود سرعت استفاده می‌شود، در این پیاده‌سازی زمان اجرای موازی بیشتر از حالت سریال مرحله الف شده است. این افزایش زمان به دلیل سربار ارتباطی بین هسته‌ها است. در اینجا، هر هسته باید به طور مداوم با هسته اصلی ارتباط برقرار کند تا گرادیان خود را ارسال کند و منتظر بماند تا وزن‌های جدید را دریافت کند. این فرآیندهای ارتباطی باعث افزایش زمان اجرا شده که در این حالت، موازی‌سازی به جای کاهش زمان، منجر به افزایش سربار و در نتیجه افزایش زمان اجرا شده است.

در نتیجه، برای داده‌های کوچک و با فرض هدف حریم خصوصی و محیطی که ارتباطات بین هسته‌ها بسیار زیاد است، ممکن است موازی‌سازی همیشه به نفع کارایی نباشد و همانطور که در اینجا دیدیم، سربار ارتباطات می‌تواند زمان اجرا را بیشتر کند.

۳-۲_ پاسخ بخش ج) اجرای رگرسیون لجستیک بر روی دو نود و دو هسته

نتایج اجرا:

زمان اجرای کل برنامه: ۶.۸۰ ثانیه

دقت هسته‌ها:

هسته ۰: ۹۶.۲٪ - زمان اجرا: ۶.۷۳ ثانیه

هسته ۱: ۹۵.۴٪ - زمان اجرا: ۶.۷۸ ثانیه

هسته ۲: ۹۷.۲٪ - زمان اجرا: ۰.۸۰ ثانیه

هسته ۳: ۹۶.۲٪ - زمان اجرا: ۱.۷۴ ثانیه

دقت متوسط: ۹۶.۲۵٪

زمان اجرای متوسط هر هسته: ۴.۰۱ ثانیه

در این اجرا، زمان اجرای کل افزایش یافته است. همانطور که مشاهده می‌شود، هسته‌های ۰ و ۱ زمان اجرای بسیار بالاتری نسبت به هسته‌های ۲ و ۳ دارند. به نظر می‌رسد که نود دوم به دلیل کمتر بودن سربار ارتباطی، زمان اجرای کمتری داشته باشد.

توجه شود که در اینجا نام هسته بر اساس یک حلقه است و نام تخصیص یافته توسط Slearn نیست.

۴-۲_ نتیجه‌گیری

بررسی نتایج اجرای الگوریتم رگرسیون لجستیک در مراحل مختلف نشان می‌دهد که استفاده از موازی‌سازی همیشه منجر به بهبود زمان اجرا نمی‌شود. در مرحله الف که الگوریتم به صورت سریال اجرا شد، زمان اجرای بسیار سریع و دقت مناسبی بدست آمد. دز قسمت ب، الگوریتم با استفاده از کتابخانه mpi4py به صورت موازی اجرا شد، اما زمان اجرای موازی به دلیل سربار ارتباطی بین هسته‌ها افزایش یافت. در نهایت، در مرحله ج که الگوریتم روی دو نود و دو هسته اجرا شد، ، زمان اجرای کل افزایش یافت اما دقت الگوریتم همچنان بالا باقی ماند.

به طور کلی، موازی‌سازی برای بهبود کارایی مفید است، اما در صورتی که داده‌ها کوچک باشند یا ارتباطات بین نودها زیاد باشد، موازی‌سازی می‌تواند باعث افزایش زمان اجرا شود، همانطور که در این آزمایش‌ها مشاهده شد ولی ملاک‌های دیگری مثل حفظ حریم خصوصی کاربران این مورد را توجیه‌پذیر می‌کند خصوصا برای داده‌های حساس مثل پزشکی. در جداول زیر خلاصه نتایج را مشاهده می‌کنیم:

مرحله	تعداد نود	تعداد هسته	میانگین دقت	زمان اجرای کل	زمان اجرای میانگین
(الف)	1	1	95.2	0.65	0.65
(ب)	1	2	95.05	1.2765	1.2757
(ج)	2	2(هر نود)	96.25	6.80	4.01

جدول ۲ نتایج رگرسیون لجستیک به صورت سریال و موازی

اجرای سریال در مرحله الف سریع‌ترین زمان را داشت (۰.۶۵ ثانیه). در قسمت ب (موازی‌سازی)، به دلیل سربار ارتباطی بین هسته‌ها، زمان اجرا کمی افزایش یافت (۱.۲۷۶۵ ثانیه). در نهایت، در مرحله ج که با دو نود اجرا شد، زمان اجرای کل به ۶.۸۰ ثانیه افزایش یافت.

۳_ پاسخ سوال شماره ۳

در این سوال، از دو کتابخانه مختلف برای شتابدهی محاسبات جبر خطی با استفاده از Numpy استفاده کردیم:

MKL (Math Kernel Library): کتابخانه شتاب‌دهنده جبر خطی توسعه‌یافته توسط اینتل.

OpenBLAS: یک پیاده‌سازی متن باز از BLAS برای انجام محاسبات ماتریسی و جبر خطی.

۳-۱_ ایجاد و نصب محیط‌ها:

برای نصب محیط‌های مختلف از conda استفاده سد. مراحل زیر برای هر محیط انجام داده شد که کد کامل ترمینال مربوطه نیز پیوست شده است.

نصب و ایجاد محیط با استفاده از MKL:

```
conda create --name mkl_env
conda activate mkl_env
conda install numpy mkl
```

نصب و ایجاد محیط با استفاده از OpenBLAS:

```
conda create --name openblas_env
conda activate openblas_env
conda install numpy blas=*=openblas -c conda-forge
```

باتوجه به اجرای print(np.__config__.show()) توسط فایل Bench_status از اینکه هر Numpy از چه ابزاری استفاده می‌کند را مطمئن شدیم


```
" Build Dependencies}:"
"  blas}:"
"   name": "mkl-sdl,"
"   found": true,
"   version": "2023.1,"
"   detection method": "pkgconfig,"
"   include directory": "C:/Users/Alikh/anaconda3/envs/mkl_env/Library/include,"
"   lib directory": "C:/Users/Alikh/anaconda3/envs/mkl_env/Library/lib,"
"   openblas configuration": "unknown,"
"   pc                                  file                                  directory":
"C:\\b\\abs_47rhpfptu4\\croot\\numpy_and_numpy_base_1728047654646\\_h_env\\Library\\lib\\pkgco
nfig"
,{
```



```
" Build Dependencies}:"
"  blas}:"
"   name": "blas,"
"   found": true,
"   version": "3.9.0,"
"   detection method": "pkgconfig,"
"   include directory": "C:/Users/Alikh/anaconda3/envs/openblas_env/Library/include,"
"   lib directory": "C:/Users/Alikh/anaconda3/envs/openblas_env/Library/lib,"
"   openblas configuration": "unknown,"
"   pc file directory": "D:\\bld\\numpy_1728664343104\\_h_env\\Library\\lib\\pkgconfig"
,{
```


۲-۳_ کدهای بنچمارک برای محاسبه ضرب و معکوس ماتریس

در ادامه کدهای استفاده‌شده برای بنچمارک محاسبات ضرب و معکوس ماتریس را اجرا کردیم:

نتایج اجرا

عملیات	اندازه ماتریس	زمان اجرای MKL	زمان اجرایOpenBLAS
ضرب ماتریس	۱۰۰۰۰ در ۱۰۰۰۰	7.75	6.74
ضرب ماتریس	۲۰۰۰۰ در ۲۰۰۰۰	67.04	56.15
معکوس ماتریس	۱۰۰۰۰ در ۱۰۰۰۰	10.16	10.29
معکوس ماتریس	۲۰۰۰۰ در ۲۰۰۰۰	88.42	84.16

جدول ۳ نتایج بنچمارک

نتایج بنچمارک نشان داد که برخلاف انتظار، OpenBLAS در اجرای عملیات ضرب ماتریس‌ها بهتر عمل کرد و زمان اجرای کمتری نسبت به MKL داشت. این تفاوت به خصوص در ابعاد بزرگتر ماتریس بیشتر محسوس است. از سوی دیگر، در محاسبات مربوط به معکوس ماتریس‌ها، هر دو کتابخانه عملکرد مشابهی داشتند و تفاوت زمانی بین آن‌ها اندک بود.

۴_پیاده‌سازی‌های اضافی برای رفع ابهام

برای برخی سوالات که دچار ابهام شدم که طبق خواسته تمرین است یا خیر پیاده‌سازی‌های مختلفی انجام شد که در پوشه Additional قرار داده شدند.

۴-۱_ پیاده سازی بدون استفاده از تابع فاکتوریل کتابخانه MATH

در این پیاده‌سازی اضافه‌sqrt_a.py ، کد فاکتوریل جداگانه اضافه شد:

```
def factorial(num):
    result = 1
    for i in range(1, num + 1):
        result *= i
    return result
```

که نتیجه زیر حاصل شد. بسیار کندتر از استفاده از کتابخانه math شد. همچنین از دستور Time جداگانه استفاده شد که باتوجه به شباهت با کتابخانه از اجرای آن برای تمام کدهای قبلی صرف نظر شد.

```
khorramfar@raspberrypi-dml0:~/ca1/Additional $ time python sqrt_a.py
Sum of Series: 1.414213562373095
Execution time: 101.89859127998352 seconds

real    1m42.025s
user    1m41.990s
sys     0m0.028s
```

۴-۲_ بهینه‌سازی بیشتر محاسبه سری

برای بهینه‌سازی بیشتر توان ۲ هم توسط یک متغیر محاسبه شد.

۴-۳_ پیاده سازی رگرسیون لجستیک با تقسیم داده‌ها به کمک Scatter

باتوجه به اینکه هدف سوال در حفظ حریم خصوصی کمی نقض شد سعی کردیم این پیاده‌سازی هم انجام دهیم. در این کد، ابتدا داده‌ها تنها توسط نود روت (رتبه ۰) بارگذاری می‌شود. سپس با استفاده از دستور comm.scatter، داده‌ها بین نودها توزیع می‌شوند. فایل های جدید در پوشه Additional پیوست شدند. نتایج به صورت زیر هستند که تقریبا مشابه قسمت قبلی است به دلیل اتمام ددلاین فرصت بررسی کامل فراهم نبود.

```
khorramfar@raspberrypi-dml0:~/ca1/Additional $ cat  LogReg_b.out
Job is started
Rank: 0 samples: 5000
Rank: 0 Training size: 4000 Test size: 1000
Total execution time: 1.4904148578643799 seconds
Node 0 : Local Accuracy = 0.939 Execution Time = 1.4896907806396484 seconds
Node 1 : Local Accuracy = 0.962 Execution Time = 1.4896414279937744 seconds
Average Accuracy: 0.9504999999999999
Average Execution Time per Node: 1.4896661043167114 seconds
Rank: 1 samples: 5000
Rank: 1 Training size: 4000 Test size: 1000
Job is done


khorramfar@raspberrypi-dml0:~/ca1/Additional $ cat  LogReg_c.out
```

```
Job is started
Rank: 1 samples: 2500
Rank: 1 Training size: 2000 Test size: 500
Rank: 0 samples: 2500
Rank: 0 Training size: 2000 Test size: 500
Total execution time: 1.0440466403961182 seconds
Node 0 : Local Accuracy = 0.962 Execution Time = 0.931605339050293 seconds
Node 1 : Local Accuracy = 0.954 Execution Time = 0.9313344955444336 seconds
Node 2 : Local Accuracy = 0.972 Execution Time = 0.9301486015319824 seconds
Node 3 : Local Accuracy = 0.962 Execution Time = 0.7371718883514404 seconds
Average Accuracy: 0.9624999999999999
Average Execution Time per Node: 0.8825650811195374 seconds
Rank: 2 samples: 2500
Rank: 2 Training size: 2000 Test size: 500
Rank: 3 samples: 2500
Rank: 3 Training size: 2000 Test size: 500
```

در طول این تمرین، برای بخش‌هایی مانند یادآوری فرمول‌های رگرسیون لجستیک و همچنین برای آشنایی بیشتر با محیط‌های Conda، از چت‌بات استفاده کرده‌ام و چت‌بات صرفاً نقش راهنما را داشته است.

برای رفع غلط‌های املایی و رعایت نیم‌فاصله در گزارش نیز از ابزار پاک‌نویس استفاده شد.