

236370 תכנות מקבילי וUMBRELLA לעיבוד
נתונים ולמידת מכונה

דוח תרגיל בית 3

מリンה ינובסקי 324515659

קורן מעברי 207987314

שאלה 1**:Naïve_allreduce**

נקזה מקום למידע שיגיע מכל התהיליכים.

מכל תהיליך, נשלח את send לכל שאר התהיליכים.

התהיליך יוכל את כל המידע שכל שאר התהיליכים שלוו אליו (send) שנשלח משאר התהיליכים. לפwi שהטהיליך ימשיך בחישוב reduction, עליו לחכות לקבלת המידע.

לאחר שכל המידע הגיע לתהיליך, הוא יבצע reduction עם הפעולה skibl בקלט.

לאחר מכן נעתיק את התוצאה לתוכה buffer מהוקצתה לה, לתוכה recv. בסיום הפעולה לכל תהיליך יהיה buffer ובו תוצאה החסינית של כל המעלים מכל התהיליכים (allreduce) בדומה לתוצאה (allreduce).

:Ring_allreduce

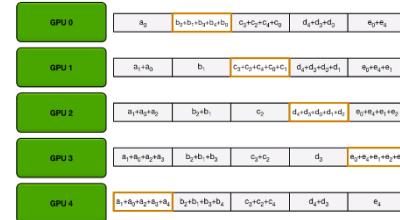
כל תהיליך כעת לא ישלח את כל העד של המידע לשיליחה, אלא חלקים ממנו. נחלק את העד לצ'אנקים כמספר התהיליכים, כי נרצה שעומס העבודה יתחלק בצורה כמה שיוונית בין כל התהיליכים.

נקזה גם כן העד למידע שהטהיליך יקבל בכל איטרציה.

נשים לב שהטהיליך יוכל מידע מטהיליך מסוים וישלח מידע לתהיליך מסוים. נקרה להם התהיליכים א' ו-ב' בהתאם. תהיליך א' הוא הקודם בסדר לתהיליך הנוכחי, וטהיליך ב' הוא הבא בסידור התהיליכים (באופן שריאנו בכיתה).

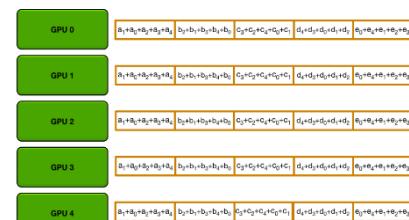
בהינתן ח' התהיליכים, נצטרך 1-ח איטרציות לביצוע הפעולות בצורה מעגלית (כפי שריאנו בכיתה). נשלח בכל איטרציה באופן מעגלי לתהיליך צ'אנק, ונתקבל צ'אנק שהגיע מטהיליך ב'. לאחר שנחנכה לקלטנו, נבצע reduction לצ'אנק המתאים לו עם הפעולה בקלט.

בסוף האיטרציות, נראה את העד של קבלת המידע כך:



כעת, נצטרך עוד 1-ח איטרציות להפצת צ'אנק המידע שמכיל את התוצאה הסופית בין כל התהיליכים. הפצת זו תבוצע גם כן בצורה מעגלית, כאשר כל תהיליך ישלח צ'אנק מקודכן לתהיליך א', יჩקה לקבלת צ'אנק מקודכן מטהיליך ב', ולאחר מכן ישמור את המידע שקיבל במקומות המתאים לו.

בסוף כל האיטרציות, נעתיק לתוכה buffer הrecv את התוצאה של החישוב. זה נראה כך בין כל התהיליכים:



```

# divide total number of batches between all the workers.
num_batches_per_worker = self.number_of_batches // self.num_workers
self.number_of_batches = num_batches_per_worker
# needed to set up num_of_batches in order to create minibatches properly!
# because num_of_batches minibatches are created when calling create_batches() method later in each epoch.

for epoch in range(self.epochs):
    # creating batches for epoch
    data = training_data[0]
    labels = training_data[1]
    mini_batches = self.create_batches(data, labels, self.mini_batch_size)
    for x, y in mini_batches:
        # do work - don't change this
        self.forward_prop(x)
        nabla_b, nabla_w = self.back_prop(y)

        # send nabla_b, nabla_w to masters
        # TODO: add your code

    for layer in range(self.num_layers):
        # we want to send the data to the master in charge of that layer.
        # we want to specify a tag indicating the layer number.
        # we will map the layers to tags as follows:
        # for biases: tag[layer] = 2 * layer, for weights: tag[layer] = 2 * layer + 1

        tag_base = 2 * layer
        master = layer % self.num_masters
        self.comm.Isend(nabla_w[layer], master, tag_base + 1)
        self.comm.Isend(nabla_b[layer], master, tag_base)

    # receive new self.weight and self.biases values from masters
    # TODO: add your code

    for layer in range(self.num_layers):
        tag_base = 2 * layer
        master = layer % self.num_masters

        req_w = self.comm.Irecv(self.weights[layer], master, tag_base + 1)
        req_w.Wait()

        req_b = self.comm.Irecv(self.biases[layer], master, tag_base)
        req_b.Wait()

```

```

# setting up the layers this master does
nabla_w = []
nabla_b = []
for i in range(self.rank, self.num_layers, self.num_masters):
    nabla_w.append(np.zeros_like(self.weights[i]))
    nabla_b.append(np.zeros_like(self.biases[i]))

self.number_of_batches = (self.number_of_batches // self.num_workers) * self.num_workers
# because we changed num_of_batches in workers- we need them to match each other

for epoch in range(self.epochs):
    for batch in range(self.number_of_batches):

        # wait for any worker to finish batch and
        # get the nabla_w, nabla_b for the master's layers
        # TODO: add your code

        # get the worker id to receive the message from
        status = MPI.Status()
        self.comm.Probe(source=MPI.ANY_SOURCE, tag=MPI.ANY_TAG, status=status)
        worker = status.Get_source()

        for i, layer in enumerate(range(self.rank, self.num_layers, self.num_masters)):
            tag_base = 2 * layer
            req_w = self.comm.Irecv(nabla_w[i], worker, tag_base + 1)
            req_w.Wait()

            req_b = self.comm.Irecv(nabla_b[i], worker, tag_base)
            req_b.Wait()

        # calculate new weights and biases (of layers in charge)
        for i, dw, db in zip(range(self.rank, self.num_layers, self.num_masters), nabla_w, nabla_b):
            self.weights[i] = self.weights[i] - self.eta * dw
            self.biases[i] = self.biases[i] - self.eta * db

        # send new values (of layers in charge)
        # TODO: add your code

        for layer in range(self.rank, self.num_layers, self.num_masters):
            tag_base = 2 * layer
            self.comm.Isend(self.weights[layer], worker, tag_base + 1)
            self.comm.Isend(self.biases[layer], worker, tag_base)

        self.print_progress(validation_data, epoch)

    # gather relevant weight and biases to process 0
    # TODO: add your code
    # if the master is not 0- send the calculations
    if self.rank != 0:
        for layer in range(self.rank, self.num_layers, self.num_masters):
            tag_base = 2 * layer
            self.comm.Isend(self.weights[layer], 0, tag_base + 1)
            self.comm.Isend(self.biases[layer], 0, tag_base)

    # if the master is 0- receive **from all other masters** the calculations and update
    if self.rank == 0:
        for other_master in range(1, self.num_masters):
            for layer in range(other_master, self.num_layers, self.num_masters):
                tag_base = 2 * layer
                recv_w_req = self.comm.Irecv(self.weights[layer], other_master, tag_base + 1)
                recv_w_req.Wait()
                recv_b_req = self.comm.Irecv(self.biases[layer], other_master, tag_base)
                recv_b_req.Wait()

```

הסבר למימוש חלק 2:

הסבר חלק כחול: נסמן ב- { חלקים שהוספנו לך.

הסבר חלק צהוב: לפי ההנחיות, כל העובדים ייחד צריכים לבצע num_of_batches באציגים בכל epoch. לכן, נחלק ביניהם את כמות הבאים.

לאחר חלק זה, נבצע epochים: ניצור בכל אחד את mini_batches עליהם אותו העובד יבצע חישובים, ונחשב גרדיאניםUm forward and back propagation

הסבר חלק ירוק: באיטרציות על כל השכבות, נרצה לשלוח כל שכבה לאMASTER שאחראי עליה. מהמשמעות הנתון בdo_master, כל MASTER שלו דרגה master_id, אחראי על כל שכבה ו שמיימת mod(num_masters)=master_id(i).

לכן נשלוח את הגרדיאנים nabla_b, nabla_w המתאים לכל שכבה. נסיף תג להודעה כדי ש渴בלת האMASTER מכל מופות אותה חוזרת לשכבה עבורה התקבלה (ולפרמטרים המותאים), לאחר מכן וכל MASTER אחראי על יתר שכבה אחת. לכן, כל Tag מומפה באופן חד-חד ערכי למספר שכבה, ובעזרת הזוגות שלו ניתן גם לזרה לאיזה פרמטר (w/b) התקבלה ההודעה אצל המMASTER.

הסבר חלק ירוק: בעת עליינו לקבל חוזרת לשכבה מהMASTER האחראי עליה המשקלים והbiases המעודכנים. לכן, ניעזר בכך מיפוי התגים כדי לקבל כל הودעה לbuffer המתאים לה לפ' שכבה, ונכחה לקבללה עם .wait()

הסבר חלק צהוב: באופן דומה נסמן ב- { חלקים שנוספו לך.

נקצה מקום לגרדיינטים שיתקבלו מכל שכבה עליה אחראי המMASTER, וכן אומדן מספר הבאים און תואם לעובדים.

עת, בכל אפקט, בכל batch:

הסבר חלק אדום: קיבל בסטטוס שנשלח לprobe את העובד שרצה לשוחה הודהה לMASTER זהה. אנו מבצעים פעולה זו כדי לאפשר קבלת הודעות מכל עובד שכבר שלח הודהה, מוביל לכפota סדר מסוים בין העובדים מהם קיבל הודהה. ככל שכבה, נכחנה לקבללה ההודהה לכל buffer מותאים עם wait ובעזרה מיפוי התגים כפי שהסביר.

לאחר חלק זה, יבצעו עדכון למשקלות ו biases עבופר השכבות עליהם אחראי אותו MASTER, בהתאם לערכיהם שקיבל מהעובד.

הסבר חלק ירוק: נשלח חוזרת לעובד את המשקלות ו biases המעודכנים לשכבות שהוא שלח לחישוב.

הסבר חלק כחול: בסיסם כל העבודה, קיבל מצב בו כל MASTER שומר את התוצאות של החישוב רק עבור השכבות עליהם אחראי. נרצה לאגד מידע זה תחת אותו "MASTER אב" שהוא התחילה עם rank 0, וכך נשלח אליו כל MASTER אחר את כל data שחייב עבור כל שכבה עליה אחראי, ובMASTER עם דירוג 0 נכחה עם wait לקבלת מידע זה.

הערה לחלק 2:

מצין שאנו משתמשים בחלק ב' עבור קריאה וכיתה בקריאות Isend, IrecvMPI שהן קריאות אוטונומיות על מנת לאפשר את התקשרות האסינכרונית הרצiosa.

שאלה 2

הרצה sync_allreduce :ring_allreduce.py מושתמשה במימוש sync_network.py

4 cores	8 cores	16 cores
<pre>Epoch 1, accuracy 55.02 %. Epoch 2, accuracy 86.14 %. Epoch 3, accuracy 89.14 %. Epoch 4, accuracy 89.97 %. Epoch 5, accuracy 92.09 %. Time reg: 5.114367723464966 Test Accuracy: 91.59% Epoch 1, accuracy 19.04 %. Epoch 2, accuracy 57.74 %. Epoch 3, accuracy 83.65 %. Epoch 4, accuracy 88.56 %. Epoch 5, accuracy 90.6 %. MPICH: Builtin communicator Time sync: 77.99466967582703 Test Accuracy: 90.09%</pre>	<pre>Epoch 1, accuracy 55.37 %. Epoch 2, accuracy 85.43 %. Epoch 3, accuracy 90.45 %. Epoch 4, accuracy 91.39 %. Epoch 5, accuracy 92.43 %. Time reg: 5.854595899581909 Test Accuracy: 92.07% Epoch 1, accuracy 9.91 %. Epoch 2, accuracy 29.95 %. Epoch 3, accuracy 69.69 %. Epoch 4, accuracy 86.38 %. Epoch 5, accuracy 90.38 %. MPICH: Builtin communicator Time sync: 87.59503555297852 Test Accuracy: 90.15%</pre>	<pre>Epoch 1, accuracy 50.95 %. Epoch 2, accuracy 86.41 %. Epoch 3, accuracy 89.87 %. Epoch 4, accuracy 91.38 %. Epoch 5, accuracy 92.34 %. Time reg: 5.247700452804565 Test Accuracy: 91.69% Epoch 1, accuracy 9.91 %. Epoch 2, accuracy 12.0 %. Epoch 3, accuracy 47.02 %. Epoch 4, accuracy 83.72 %. Epoch 5, accuracy 88.83 %. MPICH: Builtin communicator Time sync: 90.67174935340881 Test Accuracy: 88.61%</pre>

מצורפות הפקודות איתן בוצעה כל הרצה. החלק העליון מציג את הריצה הסינכרונית בעוד שהתחתון מציג את הריצה הסינכרונית.

שאלה 3

אם מרים את הרשת הסינכרונית עם מספר משתנה של עובדים, כאשר לכל עובד הקצנו 2 cores.

ניתן לראות כי באימון הרשת הרגילה, הזמן אחוז הדיק נשארים יחסית יציבים.

עבור הרשת הסינכרונית, ניתן לראות שכל שחקנים יותר עובדים - הדיק שלו יורד באימון על פני epochs, והזמן הדרוש לאימון עולה.

הסביר לירידת הדיק: גודל minibatch ברשת הוא 16, ולכן כל עובד בכל אחת מהגרסאות יעבוד על גודל num_of_workers/16. לעומת האימון המשמש ב4 עובדים עובד עם minibatch בגודל 4, והוא הינו גדול מ倍ן אחר minibatches בשאר הגרסאות.

מכיוון שככל שעבוד גרדיאנטים על חלק המידע (ה minibatches) שהוקצה לו, ועוד מחושב ממוצע על פני כל העובדים, מקבל שככל שהградיאנטים מחושבים על קבוצה קטנה יותר של דוגמאות- כך יש פחות הכללה בחישוב (ובחישוב הממוצע שמתבצע לאחר מכן), וכך ההתכנסות איטית יותר, וכן מקבל עבור אותו מספר epochs דיק נמוך יותר. ניתן לראות זאת בבירור עבור דיק שמודפס לכל epoch.

הסביר לעיליה בזמן האימון: הזמן שנדרש לאימון עולה מכיוון שככל שיש יותר עובדים, יש בינהם יותר תקשורת (מערכת עומסה כי יש נקודות פיק בהן צריכה להתבצע הרבה תקשורת) וכן אנו נדרשים לבצע יותר סינכרונים (ובפרט - יותר הפיצזת מידע בין כל העובדים עבורו reduce), דבר שעה להרשות זמן החישוב (כלומר: למראת המקובל בחישוב, אנחנו מקבלים תקורה ממשמעית בתקשורת שצריכה להתבצע באופן סדרתי).

השוואה עם האימון המקורי: באימון המקורי, הגרדיינטים מחושבים על ידי תהיליך אחד, ולכן אין תקשורת ואין סינכרון, דבר שלא מאט את העבודה האימון. כמו כן, הגרדיינטים מחושבים על קבוצת הדוגמאות (minibatch) הינו גדול מ倍ן כל הרצאות, הכוללת 16 דוגמאות, וכך ההתכנסות של האימון ברשת הזו מהיר יותר מהגרסאות האחרות ונקל עבורו דיק גבוה יותר מאשר אותו מספר epochs.

שאלה 4

האלגוריתם הסינכרוני מקבל speedup לפי העיקרון של האמדל: זאת מכיוון שהוא גודל הבעה קבוע, והחלק הנitinן למקובל קבוע. וכך, החלקים הסדרתיים בתוכנית (התקשורת והסינכרון) מגבלים את ההאצה ואף מאטים את התוכנית ככל שכמות העובדים גדלה.

כדי לקבל speedup לפי גוסטבסון, علينا לבצע scaling: ריצה להגדיל את הרשת עם הגדלה כמות העובדים, ואת כמות הדוגמאות עליהן מבוצע האימון, על מנת שהחלק הסדרתי לחולוטן בתוכנית (שוב, הסינכרון והתקשורת) ישאף ל-0 ככל שהבעה גדלה.

שאלה 5

הרצה :async_network.py

הרצה מקורית	2 masters, 4 cores	2 masters, 8 cores	4 masters, 8 cores	4 masters, 16 cores
יחסית זהה לכל כמות עובדים ומאסטרים	2 workers, 2 masters	6 workers, 2 masters	4 workers, 4 masters	12 workers, 4 masters
לפי כל פקודות ההרצה	srun -K -c 2 -n 4 --mpi=pmi2 --pty python3 main.py async 2	srun -K -c 2 -n 8 --mpi=pmi2 --pty python3 main.py async 2	srun -K -c 2 -n 8 --mpi=pmi2 --pty python3 main.py async 4	srun -K -c 2 -n 16 --mpi=pmi2 --pty python3 main.py async 4
Epoch 1, accuracy 56.3 %. Epoch 2, accuracy 86.35 %. Epoch 3, accuracy 89.44 %. Epoch 4, accuracy 91.16 %. Epoch 5, accuracy 92.18 %. Time reg: 5.4892418384552 Test Accuracy: 92.18%	Epoch 1, accuracy 9.61 %. Epoch 2, accuracy 9.61 %. Epoch 3, accuracy 9.61 %. Epoch 4, accuracy 9.61 %. Epoch 5, accuracy 9.61 %. MPICH: Builtin communicator 44 Time async: 3.4467759132385254 Test Accuracy: 92.03%	Epoch 1, accuracy 10.09 %. Epoch 2, accuracy 10.09 %. Epoch 3, accuracy 10.09 %. Epoch 4, accuracy 10.09 %. Epoch 5, accuracy 10.09 %. MPICH: Builtin communicator 44 Time async: 2.567652940750122 Test Accuracy: 11.35%	Epoch 1, accuracy 9.15 %. Epoch 2, accuracy 9.15 %. Epoch 3, accuracy 9.15 %. Epoch 4, accuracy 9.15 %. Epoch 5, accuracy 9.15 %. MPICH: Builtin communicator 44 Time async: 2.474914312362671 Test Accuracy: 88.24%	Epoch 1, accuracy 9.67 %. Epoch 2, accuracy 9.67 %. Epoch 3, accuracy 9.67 %. Epoch 4, accuracy 9.67 %. Epoch 5, accuracy 9.67 %. MPICH: Builtin communicator 44 Time async: 2.474914312362671 Test Accuracy: 9.8%

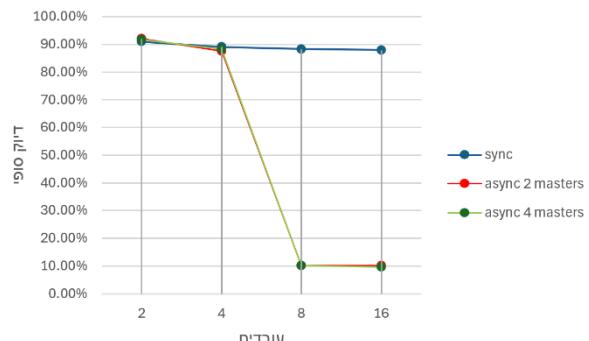
שאלה 6

תחליה, ניתן לראות שהרצה על ליבות רבות יותר מובילה להאצה בчисלוב. זאת מכיוון שעבודת כל העובדים לא תליה אחת בשניה, ככלומר כל עובד מבצע את עדכונו הגרדיינט שלו, מקבל מהMASTER גרדיאנטים חדשים, וממשיך בעבודה שלו. יש משמעותית פחות סyncronous מאשר בשיטה הקודמת (במקום לסגור בין כל העובדים, יש לסגור בין עובד והמאסטרים שלו), ויש מתקבל הרבה- גם בחישובי הגרדיינטים (batch) מתחלק בין העובדים ומחושב במקביל מבלי להסתנכרן) וגם בעדכון הפרמטרים (כל מאסטר מעדכן פרמטרים לשכבות עליין הוא אחראי, וזה קורה במקביל עבורו).

עם זאת, ניתן לראות ירידת בדיקת האימון עם אותו מספר epochs ככל שיש יותר עובדים. זאת מכיוון שהפרמטרים שאיתם העובד עושים שימוש בחישוב הגרדיינט ואז מחזיר את התוצאה לעובדים- לא בהכרח עדכנים- לא בהכרח עדכנים. בזמן שעבוד זה מחשב את הגרדיינטים, העובדים אחרים מס'ים את העבודה שלהם וועשים עדכנים לקובצת הפרמטרים. כפי שראינו בהרצאה, GAP (push远程参数更新) גודל ככל שיש יותר עובדים, מה שוביל לכך שרלוונטיות הגרדיינטים שנוחפים בשלב ההשנה פחותת טוביה, והדיק בAIMON יורד. בעיה זו נקראת gradient stainless.

שאלה 7

אם נשימוש בparameter server יחיד, תהייה בעיית סקלibility: ככל שנרצה מערכת גדולה ומהירה יותר, כך יווצר עומס על שרת הפרמטרים כי כל העובדים יפנו אליו לעדכנים, והוא יփוך להיות צואר בקבוק ויאט את המערכת. בעיה זו נקראת sharding. לכן, נרצה לחלק את העבודה בין כמה שרתים פרמטרים כך שכל חלק של הפרמטרים יחושב על ידי שרת שונה, מה שיפחית את העבודה מכל שרת.

שאלה 8**שאלה 9**

המיומש האסינכרוני יורד בדיקו בכל שכמות העובדים עולה בגל בעית hessian_stainless, עליה פירטו בשאלה 6 ☺

שאלה 10

נשווה בין הגישה הסינכרונית לגישה האסינכרונית.
 היותרן בגישה הסינכרונית הוא שהוא מספקת דיק גודל יותר, מכיוון שתמיד כל העובדים עובדים על קבוצת פרמטרים עדכניים.
 החיסרון בשיטה זו היא שכל שיש יותר עובדים, אנחנו צריכים לבצע יותר תקשורת בנקודות ספציפיות, מה שגדיל את החלק הסדרתי לביצוע, ומאט את המערכת (כפי שראינו, האטה אפיו ביחס לביצוע סדרתי רגיל)
 כמו כן, ראיינו בהרצאה את הבעיה הבאות:

בעיה בסקלביות של השיטה הסדרטיבית: אם נגדיל את הרשת, כל עובד יצטרך לעבוד על batch גדול יותר, מה שייגע בדיק הסופי. בנוסף, הגדלת הרשת גורמת לכך שיש לבצע יותר תקשורת בנקודות הסינכרון, מה שוביל למערכת עומסה בתפעול ווסף כל איטרציה. בעיה בחישוב מעלה ענן: יתכן משאבי ענן שיולטים להיות איטיים יותר כי הם משותפים עם אפליקציה שצריכה הרבה מהם, ולכן אין שימוש במשאב זה כעובד ייחודה תואמת ביחס לחלקים האחרים של האימון, מה שיאט עוד יותר את האימון הכללי.

היותרן בשיטה האסינכרונית הוא שהמקבול גבוה, וכן אנחנו נקבל האצה בזמן הנדרש לאימון ככל שיש מספר גודל יותר של עובדים (ומאסטרים המתאים להם ממוקם, אחרת יהיה sharding). עם זאת, החיסרון הוא שבגלל בעיית השarding, הדיק ירד בצורה משמעותית (כלומר - זו בעית סקלביות).

שאלה 11

2 cores	4 cores	8 cores
Testing array size: 4096 Naive all-reduce time: 0.003344297409057617 Ring all-reduce time: 0.0011382102966308594 Comparing results... Naive all-reduce correct: True Ring all-reduce correct: True Testing array size: 32768 Naive all-reduce time: 0.011167526245117188 Ring all-reduce time: 0.004137516021728516 Comparing results... Naive all-reduce correct: True Ring all-reduce correct: True Testing array size: 262144 Naive all-reduce time: 0.07677626609802246 Ring all-reduce time: 0.03612065315246582 Comparing results... Naive all-reduce correct: True Ring all-reduce correct: True Testing array size: 2097152 Naive all-reduce time: 1.146385669708252 Ring all-reduce time: 0.30754852294921875 Comparing results... Naive all-reduce correct: True Ring all-reduce correct: True	Testing array size: 4096 Naive all-reduce time: 0.0073566436767578125 Ring all-reduce time: 0.0033338069915771484 Comparing results... Naive all-reduce correct: True Ring all-reduce correct: True Testing array size: 32768 Naive all-reduce time: 0.05022382736206055 Ring all-reduce time: 0.007995843887329102 Comparing results... Naive all-reduce correct: True Ring all-reduce correct: True Testing array size: 262144 Naive all-reduce time: 0.44113993644714355 Ring all-reduce time: 0.07328367233276367 Comparing results... Naive all-reduce correct: True Ring all-reduce correct: True Testing array size: 2097152 Naive all-reduce time: 5.504987716674805 Ring all-reduce time: 0.5721452236175537 Comparing results... Naive all-reduce correct: True Ring all-reduce correct: True	Testing array size: 4096 Naive all-reduce time: 0.0144060926513672 Ring all-reduce time: 0.003125429153442383 Comparing results... Naive all-reduce correct: True Ring all-reduce correct: True Testing array size: 32768 Naive all-reduce time: 0.060242652893066406 Ring all-reduce time: 0.06520438194274902 Comparing results... Naive all-reduce correct: True Ring all-reduce correct: True Testing array size: 262144 Naive all-reduce time: 1.1353404521942139 Ring all-reduce time: 0.16941571235656738 Comparing results... Naive all-reduce correct: True Ring all-reduce correct: True Testing array size: 2097152 Naive all-reduce time: 9.18032169342041 Ring all-reduce time: 5.322966575622559 Comparing results... Naive all-reduce correct: True Ring all-reduce correct: True

ניתן לראות שמיון ring all reduce שווה למיון naive. זאת מכיוון שבמימוש הנאייבי אנחנו שולחים את כל המערך לכל המהליים, ככלומר יש הרבה תקשורת, ובמימוש של ring אנחנו שולחים חלקים מהההליים, וכך סך התקשורת נמוך יותר, דבר שמאפשר למערכת לעבוד מחר יותר ויעיל יותר.

שאלה 12

ניתוח סיבוכיות למימוש הנאייבי: בהינתן N מהליים DATA1 לשילוח בגודל M מכל אחד, כל מהלייר שולח N הודעות בגודל M . אך סיבוכיות המידע הנשלח מההלייר אחד היא $O(MN)$, ורק הסיבוכיות של כל המידע הנשלח מכל מהליים היא $O(MN^2) = O(MN^2)$.

שאלה 13

ניתוח סיבוכיות ל`reduce_all`: בהינתן N תהליכי DATAI לשילוח בגודל M מכל אחד, כל תהליכי שולח פעמיים (פעם אחת לחישובים ופעם נוספת להפצה) ב- $1-N$ איטרציות $\frac{M}{N}$ פיסות מידע, וכן סיבוכיות המידע הנשלח לתהליכי היא $O(M) = O(\frac{M}{N} \times N)$. וכך הסיבוכיות של המידע הנשלח מכל התהליכים היא $O(MN) = O(M) \times O(N)$.

שאלה 14

א. Seq. consistency => coherent causal consistency

הוכחה: הוכחנו בתרגול בשקף 26 ש $.seq.\text{consistency} \Rightarrow \text{causal consistency}$ -
 $.seq.\text{consistency} \Rightarrow \text{coherence}$

כמו כן, באותו השקף ראיינו $.seq.\text{consistency} \Rightarrow \text{coherence}$ המודל בשאלת היא חוקית גם תחת P שחוקית תחת seq היא חוקית תחת $causal$ ותחת $coherence$, וכך הכל לפי הגדרת המודל בשאלת היא חוקית גם תחתו.

ב. נפריר על ידי דוגמה נגדית:

Process 1	Process 2
Read y,1	Read x,2
Write x,2	Write y,1

התוכנית חוקית תחת $coherence$:

נראה שלכל משתנה מסו��ף, קיימים סידור חוקי לקריאות ולכתיבות.

סדר חוקי לא: $y,1 \rightarrow x,2 \rightarrow x,2 \rightarrow y,1$

סדר חוקי לע: $x,2 \rightarrow y,1 \rightarrow x,2 \rightarrow y,1$

התוכנית חוקית תחת $causal$:

אין כתיבות לאותו המשתנה בין שני התהליכים, ולכן הכתיבות הן לא בלתי תלויות, וכל סידור שליהן חוקי כי אין בכרח שכל התהליכים יראו אותן באותה סדרה.

לכן התוכנית חוקית גם כן תחת הגדרת המודל החדש בשאלת, תחת $coherent causal consistency$.

עם זאת, התוכנית לא חוקית תחת seq מכיוון שלא קיימים סידור לקריאות ולכתיבות שكونיסטנטי עם שני התהליכים (ראיינו עבור דוגמה זו בתרגול).