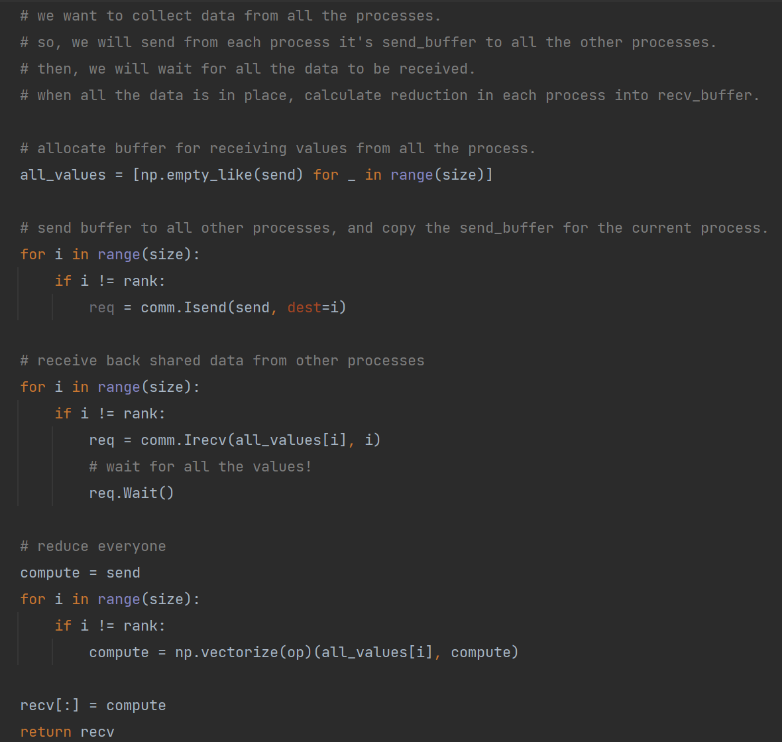
**236370 תכנות מקבילי ומבוזר לעיבוד נתונים ולמידת מכונה**

**דוח תרגיל בית 3**

מרינה ינובסקי 324515659

קורן מעברי 207987314

**שאלה 1**

**Naïve\_allreduce:**

נקצה מקום למידע שיגיע מכל התהליכים.

מכל תהליך, נשלח את send לכל שאר התהליכים.

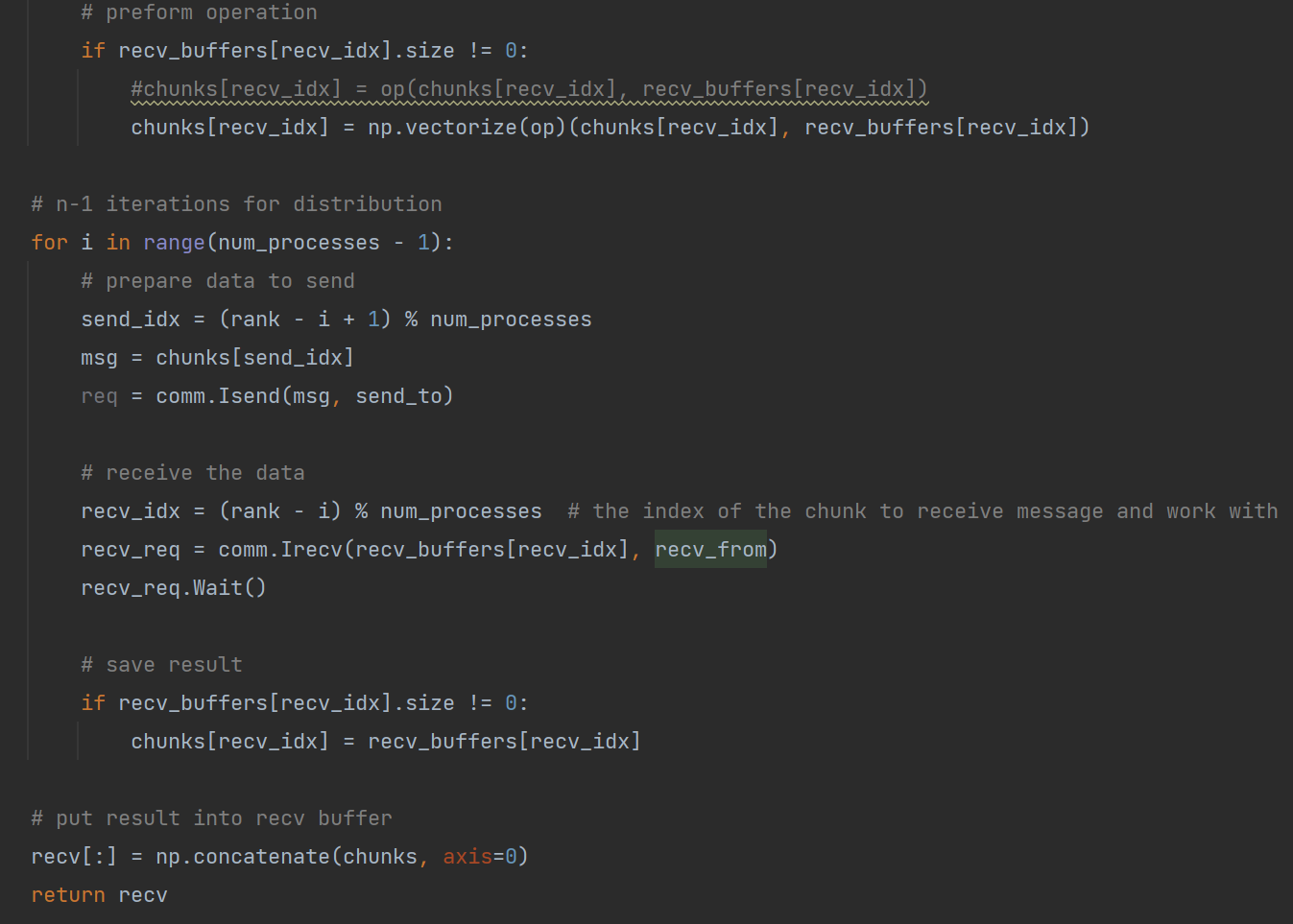
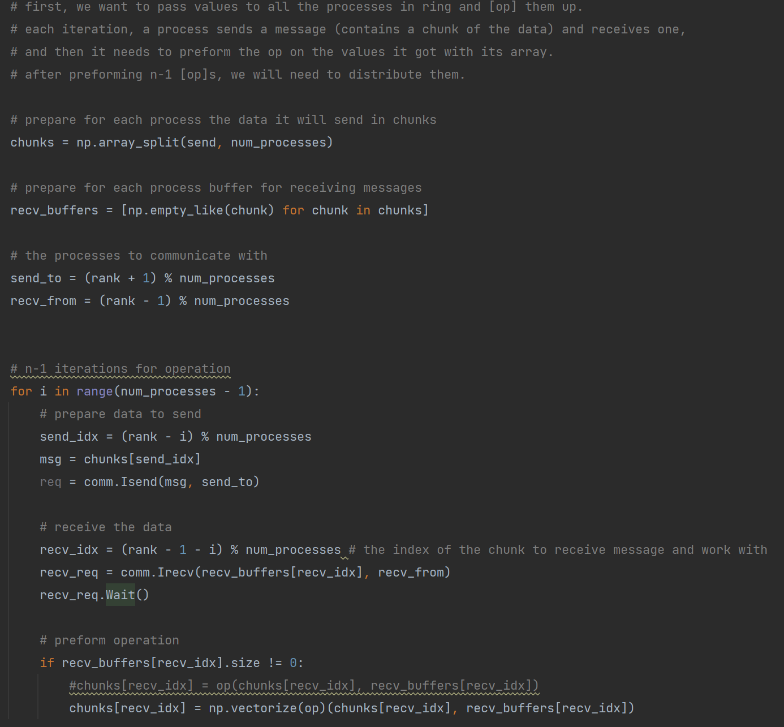
התהליך יקבל את כל המידע שכל שאר התהליכים שלחו אליו (הsend שנשלח משאר התהליכים). לפני שהתהליך ימשיך בחישוב הreduction, עליו לחכות לקבלת המידע.

לאחר שכל המידע הגיע לתהליך, הוא יבצע reduction עם הפעולה שקיבל בקלט.

לאחר מכן נעתיק את התוצאה לתוך הbuffer שהוקצה לה, לתוך recv.

בסיום הפעולה לכל תהליך יהיה buffer ובו תוצאת הreduction של כל המערכים מכל התהליכים (בדומה לתוצאת allreduce).

**Ring\_allreduce:**

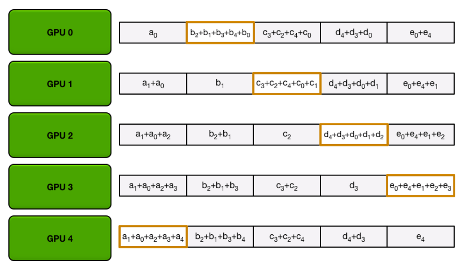


כל תהליך כעת לא ישלח את כל הbuffer של המידע לשליחה, אלא חלקים ממנו. נחלק את הbuffer לצ'אנקים כמספר התהליכים, כי נרצה שעומס העבודה יתחלק בצורה כמה שיותר שוויונית בין כל התהליכים.

נקצה גם כן buffer למידע שהתהליך יקבל בכל איטרציה.

נשים לב שתהליך יקבל מידע מתהליך מסויים וישלח מידע לתהליך מסויים. נקרא להם תהליכים א' ו-ב' בהתאמה. תהליך א' הוא הקודם בסדר לתהליך הנוכחי, ותהליך ב' הוא הבא בסידור התהליכים (באופן שראינו בכיתה).

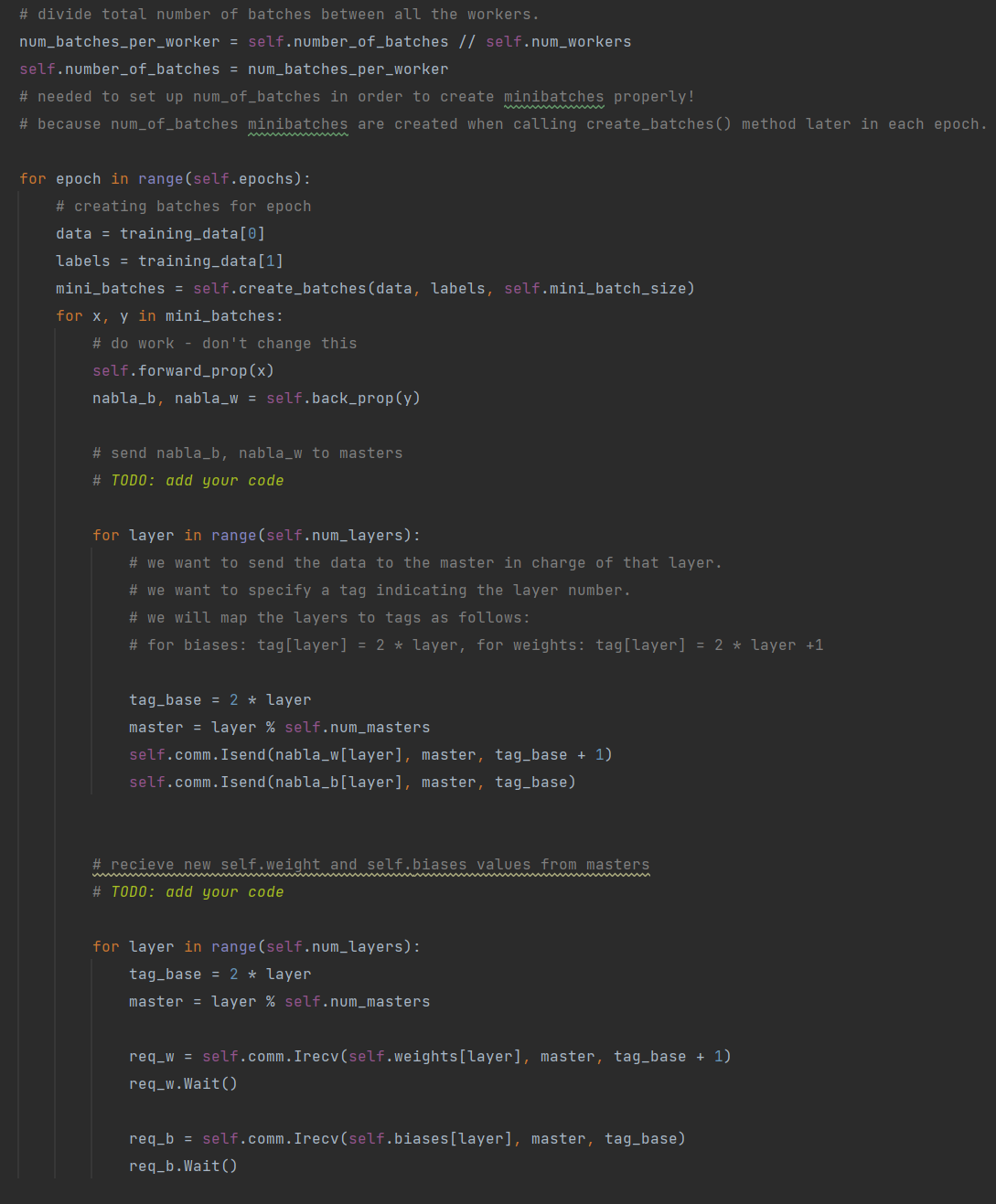
בהינתן n תהליכים, נצטרך n-1 איטרציות לביצוע הפעולות בצורה מעגלית (כפי שראינו בכיתה). נשלח בכל איטרציה באופן מעגלי לתהליך א' צ'אנק, ונקבל צ'אנק שהגיע מתהליך ב'. לאחר שנחכה לקבלתו, נבצע reduction לצ'אנק המתאים לו עם הפעולה בקלט.

בסוף האיטרציות, נראה את הbuffer של קבלת המידע כך:

כעת, נצטרך עוד n-1 איטרציות להפצת צ'אנק המידע שמכיל את התוצאה הסופית בין כל התהליכים. הפצה זו תתבצע גם כן בצורה מעגלית, כאשר כל תהליך ישלח צ'אנק מעודכן לתהליך א', יחכה לקבלת צ'אנק מעודכן מתהליך ב', ולאחר מכן ישמור את המידע שקיבל במקום המתאים לו.

בסיום כל האיטרציות, נעתיק לתוך buffer ה recv את התוצאה של החישוב. Buffer זה יראה כך בין כל התהליכים:

**הסבר למימוש חלק 2:**



**Do\_worker:** נסמן ב- } חלקים שהוספנו לקוד.

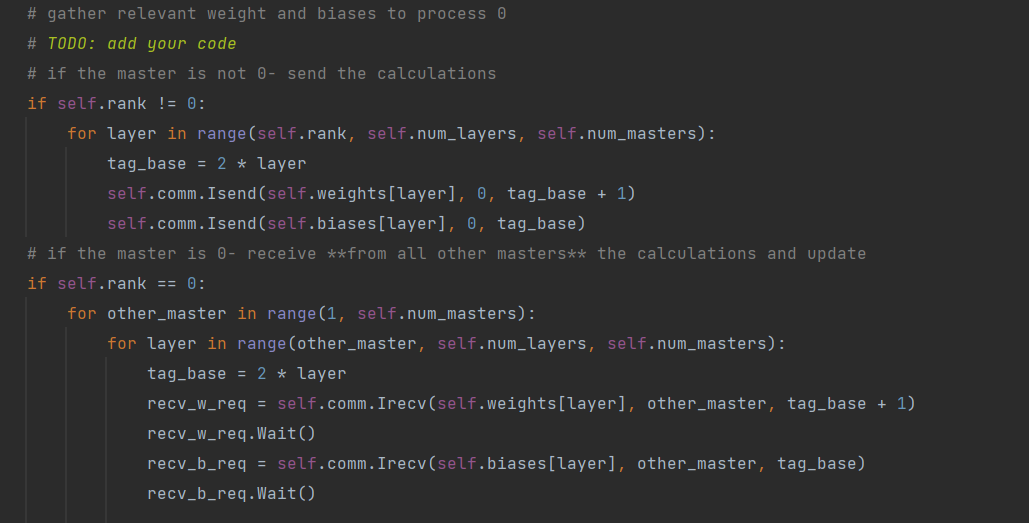
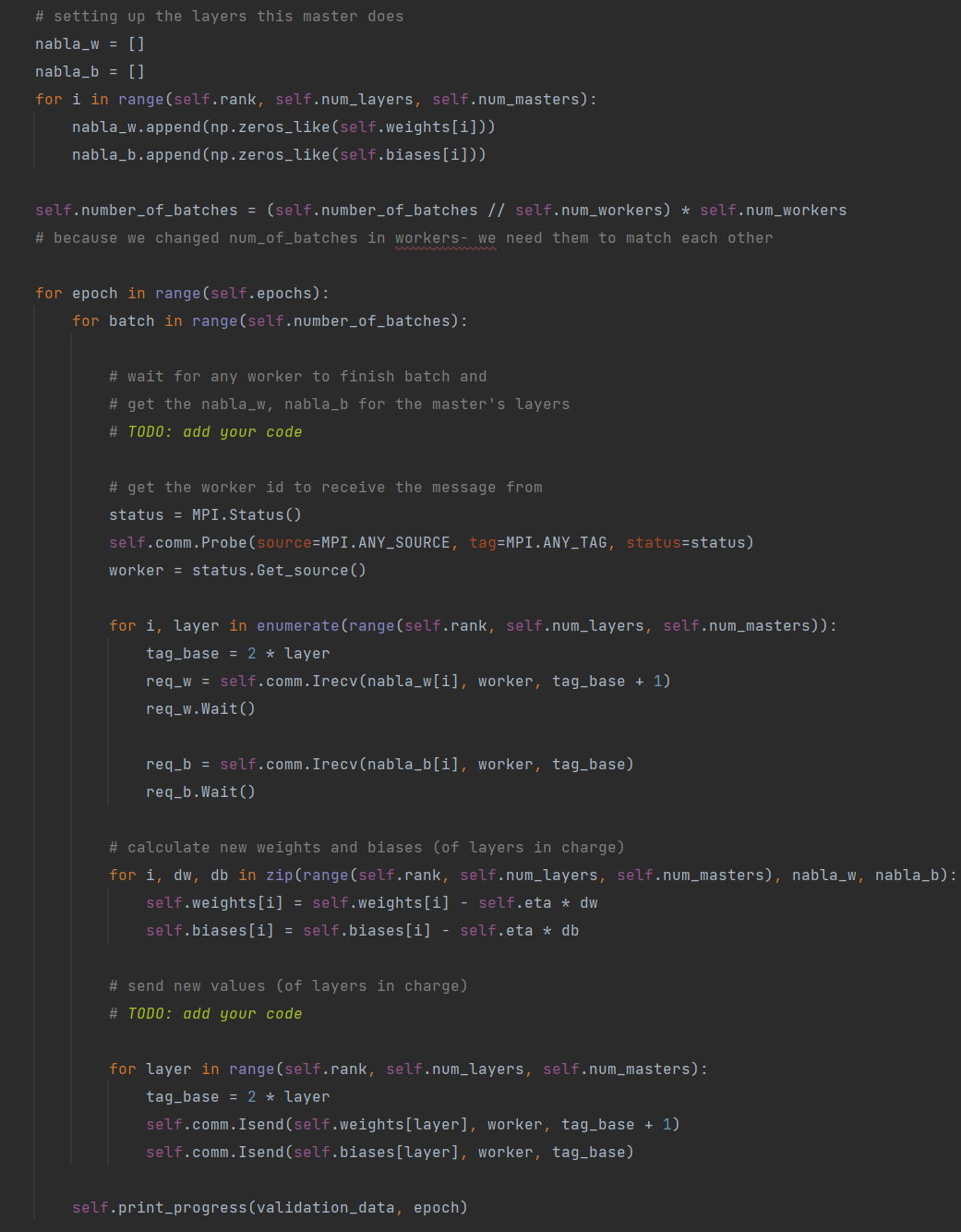
הסבר חלק כחול: לפי ההנחיות, כל העובדים יחד צריכים לבצע num\_of\_batches באצ'ים בכל epoch. לכן, נחלק בינהם את כמות הבאצ'ים.

לאחר חלק זה, נבצע epochים: ניצור בכל אחד את הmini\_batchים עליהם אותו העובד יבצע חישובים, ונחשב גרדיאנטים עם forward and back propagation.

הסבר חלק צהוב: באיטרציות על כל השכבות, נרצה לשלוח כל שכבה למאסטר שאחראי עליה. מהמימוש הנתון בdo\_master, כל מאסטר שלו דרגה master\_id, אחראי על כל שכבה i שמקיימת (i)mod(num\_masters)=master\_id.

לכן נשלח את הגרדיאנטים nabla\_b, nabla\_w המתאימים לכל שכבה. נוסיף תג להודעה כדי שבקבלתה אצל המאסטר נוכל למפות אותה חזרה לשכבה עבורה התקבלה (ולפרמטרים המתאימים), מאחר וייתכן וכל מאסטר אחראי על יותר משכבה אחת. לכן, כל תג מומפה באופן חד-חד ערכי למספר שכבה, ובעזרת הזוגיות שלו ניתן גם לזהות לאיזה פרמטר (nabla\_b/w) התקבלה ההודעה אצל המאסטר.

הסבר חלק ירוק: כעת עלינו לקבל חזרה לכל שכבה מהמאסטר האחראי עליה את המשקלים והbiases המעודכנים. לכן, ניעזר באותו מיפוי התגים כדי לקבל כל הודעה לbuffer המתאים לה לפי שכבה, ונחכה לקבלתה עם wait.



**Do\_master:** באופן דומה נסמן ב-} חלקים שנוספו לקוד.

נקצה מקום לגרדיאנטים שיתקבלו מכל שכבה עליה אחראי המאסטר, ונוודא שמספר הבאצ'ים אכן תואם לעובדים.

כעת, בכל אפוק, בכל batch:

הסבר חלק אדום: נקבל בסטטוס שנשלח לprobe את העובד שרוצה לשלוח הודעה למאסטר הזה. אנו מבצעים פעולה זו כדי לאפשר קבלת הודעות מכל עובד שכבר שלח הודעות, מבלי לכפות סדר מסויים בין העובדים מהם נקבל הודעות. לכל שכבה, נחכה לקבלת ההודעה לכל buffer מתאים עם wait ובעזרת מיפוי התגיות כפי שהוסבר.

לאחר חלק זה, יבוצעו עדכונים למשקולות ולbiases עבוdaר השכבות עליהן אחראי אותו מאסטר, בהתאם לערכים שקיבל מהעובד.

הסבר חלק ירוק: נשלח חזרה לעובד את המשקולות והbiases המעודכנים לשכבות שהוא שלח לחישוב.

הסבר חלק כחול: בסיום כל העבודה, נקבל מצב בו כל מאסטר שומר את התוצאות של החישוב רק עבור השכבות עליהן אחראי. נרצה לאגד מידע זה תחת אותו "מאסטר אב" שהוא התהליך עם rank 0, ולכן נשלח אליו מכל מאסטר אחר את כל הdata שחישב עבור כל שכבה עליה אחראי, ובמאסטר עם דירוג 0 נחכה עם wait לקבלת מידע זה.

הערה לחלק 2:

נציין שאנחנו משתמשים בחלק ב' עבור קריאה וכתיבה בקריאות Isend, Irecv שהן קריאות MPI אסינכרוניות על מנת לאפשר את התקשורת האסינכרונית הרצויה.

**שאלה 2**

הרצת sync\_network.py שמשתמשת במימוש ring\_allreduce:

|  |  |  |
| --- | --- | --- |
| 16 cores | 8 cores | 4 cores |
| srun -K -c 2 -n 16 --mpi=pmi2 --pty python3 main.py sync | srun -K -c 2 -n 8 --mpi=pmi2 --pty python3 main.py sync | srun -K -c 2 -n 4 --mpi=pmi2 --pty python3 main.py sync |

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

**שאלה 3**

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

ניתן לראות כי באימון הרשת הרגילה, הזמן ואחוזי הדיוק נשארים יחסית יציבים.

עבור הרשת הסנכרונית, ניתן לראות שככל שמקצים יותר עובדים- הדיוק שלה יורד במקצת (גם על פני epochs), והזמן הדרוש לאימון עולה.

הסבר לירידת הדיוק: גודל minibatch ברשת הוא 16, ולכן כל עובד בכל אחת מהגרסאות יעבוד על minibatch בגודל 16/num\_of\_workers. כלומר- האימון שמשתמש ב4 עובדים עובד עם minibatch בגודל 4, והוא הכי גדול מבין שאר הminibatches בשאר הגרסאות.

מכיוון שכל עובד מחשב גרדיאנטים על חלק המידע (ה minibatches) שהוקצה לו, ואז מחושב ממוצע על פני כל העובדים, נקבל שככל שהגרדיאנטים מחושבים על קבוצה קטנה יותר של דוגמאות- כך יש פחות הכללה בחישוב (ובחישוב הממוצע שמתבצע לאחר מכן), ולכן ההתכנסות איטית יותר, ונקבל עבור אותו מספר epochs דיוק נמוך יותר. ניתן לראות זאת בבירור עבור דיוק שמודפס לכל epoch.

הסבר לעלייה בזמן האימון: הזמן שנדרש לאימון עולה מכיוון שככל שיש יותר עובדים, יש בינהם יותר תקשורת (מערכת עמוסה כי יש נקודות פיק בהן צריכה להתבצע הרבה תקשורת) וכן אנו נדרשים לבצע יותר סנכרונים (ובפרט- יותר הפצות מידע בין כל העובדים עבור הreduce), דבר שעולה לנו בזמן החישוב (כלומר: למרות המקבול בחישוב, אנחנו מקבלים תקורה משמעותית בתקשורת שצריכה להתבצע באופן סדרתי).

השוואה עם האימון המקורי: באימון המקורי, הגרדיאנטים מחושבים על ידי תהליך אחד, ולכן אין תקשורת ואין סנכרון, דבר שלא מאט את עבודת האימון. כמו כן, הגרדיאנטים מחושבים על קבוצת הדוגמאות (minibatch) הכי גדולה מבין כל ההרצות, הכוללת 16 דוגמאות, ולכן ההתכנסות של האימון ברשת הזו מהיר יותר מהגרסאות האחרות ונקבל עבורו דיוק גבוה תחת אותו מספר epochs.

**שאלה 4**

האלגוריתם הסינכרוני מקבל speedup לפי העיקרון של אהמדל: זאת מכיוון שגודל הבעיה קבוע, והחלק הניתן למקבול קבוע. ולכן, החלקים הסדרתיים בתוכנית (התקשורת והסנכרונים) מגבילים את ההאצה ואף מאטים את התוכנית ככל שכמות העובדים גדלה.

כדי לקבל speedup לפי גוסטבסון, עלינו לבצע scaling: נרצה להגדיל את הרשת עם הגדלת כמות העובדים, ואת כמות הדוגמאות בminibatch עליהן מבוצע האימון, על מנת שהחלק הסדרתי לחלוטין בתוכנית (שוב, הסנכרון והתקשורת) ישאף ל0 ככל שהבעיה גדלה.

**שאלה 5**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **4 masters, 16 cores** | **4 masters, 8 cores** | **2 masters, 8 cores** | **2 masters, 4 cores** | **הרצה מקורית** |
| 12 workers, 4 masters | 4 workers, 4 masters | 6 workers, 2 masters | 2 workers, 2 masters | יחסית זהה לכל כמות עובדים ומאסטרים |
| srun -K -c 2 -n 16 --mpi=pmi2 --pty python3 main.py async 4 | srun -K -c 2 -n 8 --mpi=pmi2 --pty python3 main.py async 4 | srun -K -c 2 -n 8 --mpi=pmi2 --pty python3 main.py async 2 | srun -K -c 2 -n 4 --mpi=pmi2 --pty python3 main.py async 2 | לפי כל פקודות ההרצה |
|  |  |  |  |  |

הרצת async\_network.py:

**שאלה 6**

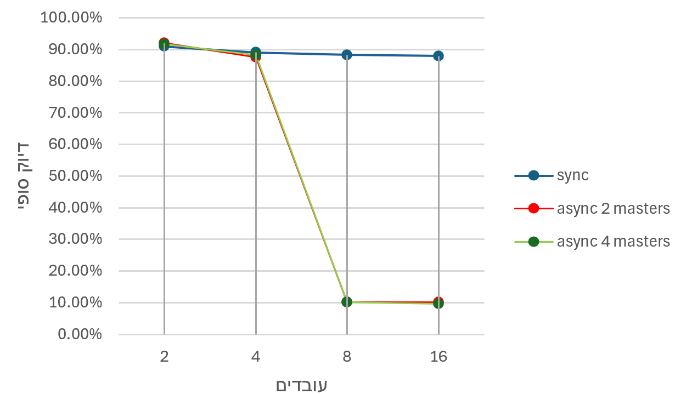
תחילה, ניתן לראות שהרצה על ליבות מרובות יותר מובילה להאצה בחישוב. זאת מכיוון שעבודת כל העובדים לא תלויה אחת בשניה, כלומר כל עובד מבצע את עדכוני הגרדיאנט שלו, מקבל מהמאסטר גרדיאנטים חדשים, וממשיך בעבודה שלו. יש משמעותית פחות סנכרונים מאשר בשיטה הקודמת (במקום לסנכרן בין כל העובדים, יש לסנכרן בין עובד והמאסטרים שלו), ויש מקבול רב- גם בחישובי הגרדיאנטים (הbatch מתחלק בין העובדים ומחושב במקביל מבלי להסתנכרן) וגם בעדכון הפרמטרים (כל מאסטר מעדכן פרמטרים לשכבות עליהן הוא אחראי, וזה קורה במקביל עבורן).

עם זאת, ניתן לראות ירידה בדיוק האימון עם אותו מספר epochs ככל שיש יותר עובדים. זאת מכיוון שהפרמטרים שאיתם העובד הi משתמש לחישוב הגרדיאנט ואז מחזיר את התוצאה לעדכון- לא בהכרח עדכניים. בזמן שעובד זה מחשב את הגרדיאנטים, עובדים אחרים מסיימים את העבודה שלהם ועושים עדכונים לקבוצת הפרמטרים. כפי שראינו בהרצאה, הGAP (=כמה הפרמטרים במצב push רחוקים ממה שהיו במצב pull) גדל ככל שיש יותר עובדים, מה שמוביל לכך שרלוונטיות הגרדיאנטים שנדחפים בשלב הpush פחות טובה, והדיוק באימון יורד. בעיה זו נקראת gradient stainless.

**שאלה 7**

אם נשתמש בparameter server יחיד, תהיה בעיית סקלביליות: ככל שנרצה מערכת גדולה ומהירה יותר, כך יווצר עומס על שרת הפרמטרים כי כל העובדים יפנו אליו לעדכונים, והוא יהפוך להיות צוואר בקבוק ויאט את המערכת. בעיה זו נקראת sharding. לכן, נרצה לחלק את העבודה בין כמה שרתי פרמטרים כך שכל חלק של הפרמטרים יחושב על ידי שרת שונה, מה שיפחית את העבודה מכל שרת.

**שאלה 8**



.

**שאלה 9**

המימוש האסינכרוני יורד בדיוקו ככל שכמות העובדים עולה בגלל בעיית הstainless, עליה פירטנו בשאלה 6😊

**שאלה 10**

נשווה בין הגישה הסנכרונית לגישה האסינכרונית.

הייתרון בגישה הסינכרונית הוא שהיא מספקת דיוק גדול יותר, מכיוון שתמיד כל העובדים עובדים על קבוצת פרמטרים עדכניים.

החיסרון בשיטה זו היא שככל שיש יותר עובדים, אנחנו נדרשים לבצע יותר סנכרונים בנקודות ספציפיות, מה שמגדיל את החלק הסדרתי לביצוע, ומאט את המערכת (כפי שראינו, האטה אפילו ביחס לביצוע סדרתי רגיל)

כמו כן, ראינו בהרצאה את הבעיות הבאות:

בעיה בסקלביליות של השיטה הסדרתית: אם נגדיל את הרשת, כל עובד יצטרך לעבוד על batch גדול יותר, מה שיפגע בדיוק הסופי. בנוסף, הגדלת הרשת גורמת לכך שיש לבצע יותר תקשורת בנקודות הסנכרון, מה שיוביל למערכת עמוסה בתחילת וסוף כל איטרציה. בעיה בחישוב מעל ענן: ייתכנו משאבי ענן שיכולים להיות איטיים יותר כי הם משותפים עם אפליקציה שצריכה הרבה מהם, ולכן חלק אימון שישתמש במשאב זה כעובד יחווה תאוטה ביחס לחלקים האחרים של האימון, מה שיאט עוד יותר את האימון הכולל.

הייתרון בשיטה האסינכרונית הוא שהמקבול גבוה, ולכן אנחנו נקבל האצה בזמן הנדרש לאימון ככל שיש מספר גדול יותר של עובדים (ומאסטרים המתאימים להם כמובן, אחרת יהיה sharding). עם זאת, החיסרון הוא שבגלל בעיית הstainless, הדיוק ירד בצורה משמעותית (כלומר- זו בעיית סקלביליות).

**שאלה 11**

|  |  |  |
| --- | --- | --- |
| **8 cores** | **4 cores** | **2 cores** |
|  |  |  |

ניתן לראות שring all reduce מהיר יותר מnaive all reduce. זאת מכיוון שבמימוש הנאיבי אנחנו שולחים את כל המערך לכל התהליכים, כלומר יש הרבה תקשורת, ובמימוש של ring אנחנו שולחים חלקים מהמערך לחלק מהתהליכים, ולכן סך התקשורת נמוך יותר, דבר שמאפשר למערכת לעבוד מהר יותר ויעיל יותר.

נתייחס ל2 הריצות האחרונות ב8 cores: כפי שניתו לראות בהן, דווקא המימוש הנאיבי מהיר יותר פי 2. זאת מכיוון שעם יותר עובדים, יש יותר סבבי תקשורת שצריכים להתבצע, ומכיוון שהמערכים מאוד גדולים בריצות אלו, זה יוצר עומס בתקשורת.

**שאלה 12**

ניתוח סיבוכיות למימוש הנאיבי: בהינתן N תהליכים וDATA לשליחה בגודל M מכל אחד, כל תהליך שולח N-1 הודעות בגודל M. לכן סיבוכיות המידע הנשלח מתהליך אחד היא , וסך הסיבוכיות של כל המידע הנשלח מכל התהליכים היא .

**שאלה 13**

ניתוח סיבוכיות לring\_allreduce: בהינתן N תהליכים וDATA לשליחה בגודל M מכל אחד, כל תהליך שולח פעמיים (פעם אחת לחישובים ופעם נוספת להפצה) ב N-1 איטרציות פיסות מידע, ולכן סיבוכיות המידע הנשלח לתהליך היא . וסך הסיבוכיות של המידע הנשלח מכל התהליכים היא .

**שאלה 14**

1. Seq. consistency => coherent causal consistency

הוכחה: הוכחנו בתרגול בשקף 26 ש - seq.consistency=>causal consistency.

כמו כן, באותו השקף ראינו seq. consistency => coherence.

לכן תוכנית P שחוקית תחת seq היא חוקית תחת coherence ותחת causal, וסך הכל לפי הגדרת המודל בשאלה היא חוקית גם תחתיו.

1. נפריך על ידי דוגמה נגדית:

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

התוכנית חוקית תחת coherence:

נראה שלכל משתנה משותף, קיים סידור חוקי לקריאות ולכתיבות.

סידור חוקי לx: Read y,1 -> Write x,2 -> Read x,2 -> Write y,1

סידור חוקי לy: Read x,2 -> Write y,1 -> Read y,1 -> Write x,2

התוכנית חוקית תחת causal:

אין כתיבות לאותו המשתנה בין שני התהליכים, ולכן הכתיבות הן לא בלתי תלויות, וכל סידור שלהן חוקי כי אין בכרח שכל התהליכים יראו אותן באותו הסדר.

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

עם זאת, התוכנית לא חוקית תחת seq. consistency מכיוון שלא קיים סידור לקריאות ולכתיבות שקונסיסטנטי עם שני התהליכים (ראינו עבור דוגמה זו בתרגול).