

### תרגיל בית 3- מקבוצר

1. להלן התיאור של הפונקציות `naive all reduce`, `ring all reduce` ושל `async network`.

#### :Naive all reduce

תחילה, אנו מעתיקים את מערך ה-`send` לתוך מערך ה-`recv` (המערך הצובר). לאחר מכן, אנו עוברים בלולאה על כל שאר התהליכים (מ-0 עד `size`). אם האינדקס שווה ל-`rank` שלנו, אנו מדלגים. אחרת, בכל איטרציה, התהליך מבצע החלפה הדדית (`Sendrecv`) עם התהליך שהמזהה שלו הוא `src`: הוא שולח לו את מערך ה-`send` המקורי שלו, ומקבל ממנו את מערך ה-`send` שלו לתוך באפר זמני (`tmp`). לאחר הקבלה, אנו מפעילים את הפעולה `op` בין המידע שהצטבר ב-`recv` לבין המידע שהתקבל ב-`tmp`, ושומרים את התוצאה ב-`recv`. בסוף הלולאה, `recv` מכיל את ה-`Reductions` של כל התהליכים.

#### :Ring all reduce

המימוש שלנו מתבצע בשתי פאזות: פיזור של מערך ה-`send` בין כל התהליכים (`scatter-reduce`) ואז הפצה של כל המידע לכל התהליכים וחיבור על ידי הפעולה (`all gather`). אנו מחלקים את המערך ל-`p` צ'אנקים (כמספר התהליכים) כאשר התהליך האחרון מקבל את מה שנשאר מהחלוקה.

שלב ה-`scatter` מתבצע באופן הבא: ה-`current` מאותחל ל-`rank`. הלולאה רצה `p-1` פעמים. בכל איטרציה: שולחים את הצ'אנק שעליו מצביע `current` לשכן מימין (`right`). מעדכנים את `current` להיות הצ'אנק הקודם מעגלית:  $p \% (p + \text{current} - 1)$ . מקבלים מהשכן משמאל (`left`) באפר זמני (`tmp`). מבצעים `op` בין המידע שהתקבל לבין הצ'אנק המתאים במערך המקומי (הצ'אנק החדש שעליו מצביע `current`). בסוף שלב זה, התהליך `rank` מחזיק את התוצאה המלאה והסופית עבור הצ'אנק באינדקס  $p \% (\text{rank} + 1)$ .

בשלב ה-`gather`: מטרתנו להפיץ את הצ'אנקים המוכנים. אנו מאתחלים את המצביע `current` לאינדקס  $p \% (\text{rank} + 1)$  (זהו הצ'אנק שעבורו יש לנו כרגע את התוצאה הסופית). לאחר מכן אנו מבצעים לולאה של `p-1` איטרציות. בכל איטרציה: שולחים את הצ'אנק הנוכחי (`current`) לשכן מימין. מעדכנים את `current` אחורה מעגלית:  $p + (\text{current} - 1) \% p$ . מקבלים מהשכן משמאל את הצ'אנק החסר ושומרים אותו בבאפר זמני. מעתיקים את המידע שהתקבל (`tmp`) ישירות לתוך המערך במקום המתאים (ללא `op`).

בסיום הלולאה, מערך ה-recv בכל התהליכים מכיל את התוצאה המלאה של ה-Reduce.

### Async network

do\_worker: הפונקציה do\_worker אחראית על ביצוע שלב חישוב הגרדינטים וסנכרון מול ה-Masters. בתחילת הריצה, הפונקציה מחלקת את עומס העבודה ומחשבת כמה Batches על התהליך הנוכחי לבצע, כאשר התהליך האחרון ("השארית") לוקח על עצמו את כל ה-Batches הנותרים כדי להבטיח כיסוי מלא של הדאטה. בתוך הלולאה הראשית (עבור כל Epoch וכל Batch), ה-Worker מבצע Forward ו-Backward propagation ומחשב את הגרדינטים המקומיים. לאחר מכן, הוא שולח את הגרדינטים הללו באופן אסינכרוני (Non-blocking) באמצעות Isend ל-Masters הרלוונטיים, כאשר כל Master אחראי על תת-קבוצה של שכבות (לפי החלוקה  $layer\_index \% num\_masters$ ). מיד לאחר השליחה, ה-Worker מבצע האזנה אסינכרונית (lrecv) לקבלת המשקולות וההטיות המעודכנים מאותם Masters. הוא ממתין (Wait) עד שכל המידע החדש מגיע ונכתב ישירות לתוך הזיכרון של המודל המקומי (self.weights), ורק אז ממשיך ל-Batch הבא.

do\_master: מנהלת את הפרמטרים עבור השכבות שבאחריותה. תחילה, היא מאתחלת באפרים עבור הגרדינטים של השכבות הספציפיות שהוקצו לה. בלולאה הראשית, ה-Master לא יודע איזה Worker יסיים ראשון, ולכן הוא מאזין לקבלת גרדינטים מ-MPI\_ANY\_SOURCE עבור השכבה הראשונה שבאחריותו. ברגע שהודעה כזו מתקבלת, ה-Master מחלץ את ה-Rank של ה-Worker ששלח אותה ("נעילה" על ה-Worker), ואז מבצע lrecv יזום כדי לקבל את שאר הגרדינטים לאותו Batch ספציפית מאותו Worker. לאחר שכל הגרדינטים התקבלו, ה-Master מבצע צעד של Gradient Descent ומעדכן את המשקולות המקומיות שלו. לבסוף, הוא שולח את הפרמטרים המעודכנים בחזרה אך ורק ל-Worker שממנו קיבל את הגרדינטים (באמצעות Isend), וממתין לסיום השליחה לפני שהוא חוזר להאזין לבקשה הבאה.

```

(tf23-gpu) dashay@lambda:~/CDPwinter26/HW3$ srun -K -c 2 -n 4 --mpi=pmi2 --pty python3 main.py sync
Epoch 1, accuracy 59.2 %.
Epoch 2, accuracy 86.12 %.
Epoch 3, accuracy 90.36 %.
Epoch 4, accuracy 91.21 %.
Epoch 5, accuracy 91.75 %.
Time reg: 6.705892324447632
Test Accuracy: 91.17%
Epoch 1, accuracy 26.63 %.
Epoch 2, accuracy 46.07 %.
Epoch 3, accuracy 62.61 %.
Epoch 4, accuracy 83.48 %.
Epoch 5, accuracy 88.16 %.
Time sync: 13.11998200416565
Test Accuracy: 87.88%
(tf23-gpu) dashay@lambda:~/CDPwinter26/HW3$ srun -K -c 2 -n 8 --mpi=pmi2 --pty python3 main.py sync
Epoch 1, accuracy 62.22 %.
Epoch 2, accuracy 87.0 %.
Epoch 3, accuracy 89.28 %.
Epoch 4, accuracy 91.23 %.
Epoch 5, accuracy 92.35 %.
Time reg: 8.308148384094238
Test Accuracy: 91.99%
Epoch 1, accuracy 16.05 %.
Epoch 2, accuracy 49.44 %.
Epoch 3, accuracy 58.23 %.
Epoch 4, accuracy 62.53 %.
Epoch 5, accuracy 69.56 %.
Time sync: 12.497069597244263
Test Accuracy: 68.84%
(tf23-gpu) dashay@lambda:~/CDPwinter26/HW3$ srun -K -c 2 -n 16 --mpi=pmi2 --pty python3 main.py sync
Epoch 1, accuracy 52.82 %.
Epoch 2, accuracy 87.9 %.
Epoch 3, accuracy 90.39 %.
Epoch 4, accuracy 91.16 %.
Epoch 5, accuracy 92.38 %.
Time reg: 14.799089670181274
Test Accuracy: 92.25%
Epoch 1, accuracy 9.83 %.
Epoch 2, accuracy 24.58 %.
Epoch 3, accuracy 44.6 %.
Epoch 4, accuracy 62.08 %.
Epoch 5, accuracy 69.57 %.
Time sync: 35.758594036102295
Test Accuracy: 68.67%

```

3. על בסיס התוצאות, ניתן לראות כי המימוש המקבילי (Sync) איטי יותר מהמימוש הסדרתי (Original) בכל המקרים. כמו כן, המעבר ל-16 ליבות גרם להרעה משמעותית בביצועים. הסיבה המרכזית לכך שהמימוש הסינכרוני איטי יותר היא שזמן התקשורת (העברת הגרדינטים בין התהליכים ב-AllReduce) עולה על הזמן שנחסך על ידי חלוקת החישוב. המודל שבו משתמשים הוא קטן יחסית, והחישובים (Forward/Backward) מהירים מאוד. לכן, המחר של שליחת המידע ברשת וסנכרון התהליכים גבוה יותר מהתועלת שבמקביליות.

4. השיטה שלפיה נקבל ספידאפ היא חוק אמדהל. גודל הבעיה במקרה שלנו הוא גודל ה-dataset. החלק בתוכנית שאינו ניתן למקבול הוא החלק שבו אנחנו מחשבים מחדש את w,b. לא משנה בכמה נגדיל את ה-dataset עדיין חלק זה יהיה לא ניתן למקבול. נראה שהחלק הזה אינו זניח ככל שה-dataset גדל. בנוסף לכך, החלק שכן ממוקבל בתוכנית הוא חישוב הגראדיאנטים על כל batch. כל worker מחשב:

$$no. \text{ of batches} * \frac{mini \text{ batch size}}{N}$$

ולכן סך הכל כל ה-workers יחד מחשבים:

$$no. \text{ of batches} * \frac{mini \text{ batch size}}{N} * N = no. \text{ of batches} * mini \text{ batch size}$$

נשים לב שחלק הבעיה שניתן למקבול תלוי ב-no. Of batches וב-mini batch size שבהנחה והם נשארים קבועים ככל שמגדילים את ה-dataset החלק הממוקבל נשאר קבוע. ולכן קיבלנו כי ה-A בחוק אמדהל הוא קבוע. כדי לקבל speedup גם לפי השיטה של גוסטפסון נרצה להתנות את גודל ה-batch שכל עובד יעבוד עליו לפי גודל ה-data. בצורה כזו החלק הממוקבל כן יהיה תלוי בגודל הבעיה ונקבל ספידאפ.

5.

```
(tf23-gpu) dashay@lambda:~/CDPwinter26/HW3$ srun -K -c 4 -n 4 --mpi=pmi2 --pty python3 main.py async 2
Epoch 1, accuracy 56.98 %.
Epoch 2, accuracy 85.28 %.
Epoch 3, accuracy 90.29 %.
Epoch 4, accuracy 91.05 %.
Epoch 5, accuracy 92.46 %.
Time reg: 6.262248277664185
Test Accuracy: 92.21%
Epoch 1, accuracy 10.64 %.
Epoch 2, accuracy 9.83 %.
Epoch 3, accuracy 9.83 %.
Epoch 4, accuracy 9.83 %.
Epoch 5, accuracy 9.19 %.
Time async: 5.331676244735718
Test Accuracy: 92.03%
(tf23-gpu) dashay@lambda:~/CDPwinter26/HW3$ srun -K -c 4 -n 8 --mpi=pmi2 --pty python3 main.py async 2
Epoch 1, accuracy 46.43 %.
Epoch 2, accuracy 86.56 %.
Epoch 3, accuracy 89.64 %.
Epoch 4, accuracy 91.6 %.
Epoch 5, accuracy 92.33 %.
Time reg: 7.183706521987915
Test Accuracy: 91.71%
Epoch 1, accuracy 9.83 %.
Epoch 2, accuracy 9.83 %.
Epoch 3, accuracy 9.83 %.
Epoch 4, accuracy 9.83 %.
Epoch 5, accuracy 9.83 %.
Time async: 5.596585988998413
Test Accuracy: 9.8%
```

```

(tf23-gpu) dashay@lambda:~/CDPwinter26/HW3$ srun -K -c 4 -n 8 --mpi=pmi2 --pty python3 main.py async 4
Epoch 1, accuracy 46.63 %.
Epoch 2, accuracy 87.21 %.
Epoch 3, accuracy 90.54 %.
Epoch 4, accuracy 91.77 %.
Epoch 5, accuracy 92.55 %.
Time reg: 7.187877655029297
Test Accuracy: 92.34%
Epoch 1, accuracy 9.83 %.
Epoch 2, accuracy 9.83 %.
Epoch 3, accuracy 9.83 %.
Epoch 4, accuracy 9.83 %.
Epoch 5, accuracy 9.83 %.
Time async: 5.320249795913696
Test Accuracy: 88.85%

```

```

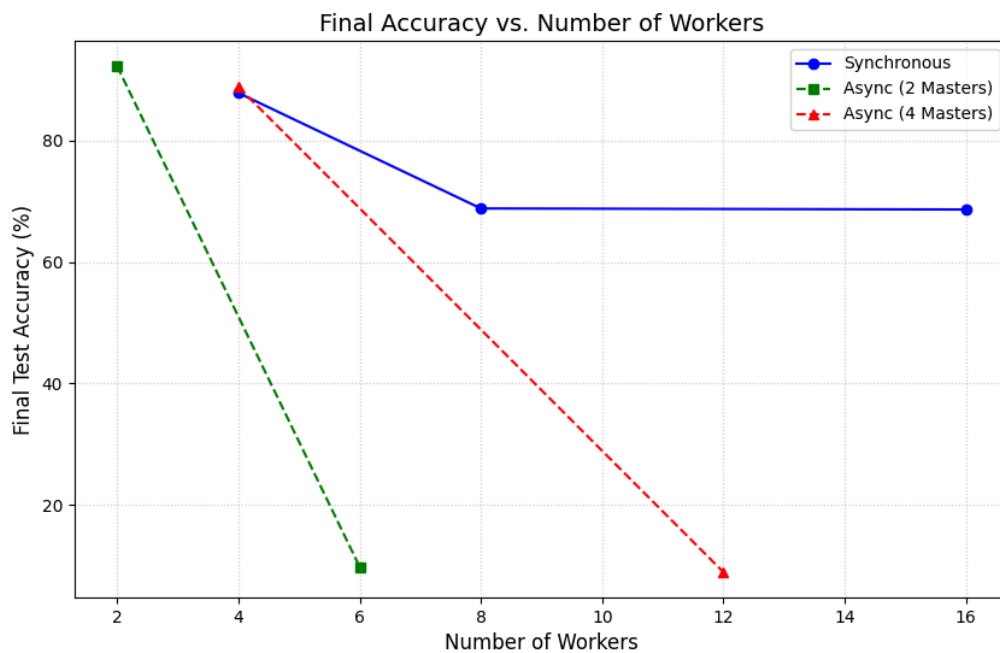
(tf23-gpu) dashay@lambda:~/CDPwinter26/HW3$ srun -K -c 2 -n 16 --mpi=pmi2 --pty python3 main.py async 4
Epoch 1, accuracy 49.99 %.
Epoch 2, accuracy 87.06 %.
Epoch 3, accuracy 90.03 %.
Epoch 4, accuracy 90.62 %.
Epoch 5, accuracy 92.69 %.
Time reg: 14.141512870788574
Test Accuracy: 91.71%
Epoch 1, accuracy 10.9 %.
Epoch 2, accuracy 10.9 %.
Epoch 3, accuracy 10.9 %.
Epoch 4, accuracy 10.9 %.
Epoch 5, accuracy 10.9 %.
Time async: 4.8748955726623535
Test Accuracy: 8.92%

```

6. בהשוואה בין הריצות השונות, ניכר כי המימוש האסינכרוני מציג שיפור עקבי ומשמעותי בזמני הריצה לעומת המימוש הסדרתי (המקורי), ואף עולה בביצועיו על המימוש הסינכרוני שנבחן קודם לכן. השיפור בולט במיוחד בריצה עם 16 ליבות, שם זמן הריצה צנח מ-14.14 שניות (סדרתי) ל-4.87 שניות בלבד. הסיבה המרכזית להאצה זו היא ביטול ה-Overhead של הסנכרון: בארכיטקטורה אסינכרונית, ה-Workers אינם ממתינים זה לזה ב-Barrier בסוף כל Batch, אלא ממשיכים לעבד נתונים באופן רציף (Non-blocking). בנוסף לכך מבחינת דיוק המודל, התוצאות חושפות את החיסרון המהותי של אימון אסינכרוני המכונה "Stale Gradients" (גרדינטים לא מעודכנים). כאשר מספר ה-Workers היה נמוך (2 או 4 Workers), המודל הצליח להתכנס לדייק גבוה (כ-92% ו-88.85% בהתאמה), הדומה לדייק הסדרתי. עם זאת, כאשר מספר ה-Workers גדל ל-6 (בתצורת 2 Masters) או ל-12 (בתצורת 16 Cores), הדיוק קרס לרמה של ניחוש אקראי (~9.8%). תופעה זו מתרחשת מכיוון שקצב העדכון של המשקולות ב-Master הופך למהיר מאוד ביחס לזמן החישוב של Worker בודד. עד ש-Worker מסיים לחשב את הגרדינט ושוחר אותו, המשקולות ב-Master כבר הספיקו להשתנות מספר רב של פעמים על סמך עדכונים מ-Workers אחרים. כתוצאה מכך, הגרדינט שנשלח מחושב ביחס לנקודה ישנה במרחב הפרמטרים, וכיוון העדכון שהוא מציע אינו רלוונטי עוד ואף מזיק להתקדמות, מה שמוביל להתבדרות המודל ולכישלון באימון.

7. אנו מפצלים את שרת הפרמטרים על פני מספר מכונות כדי למנוע צוואר בקבוק של תקשורת וחשוב, שכן שרת יחיד המקבל עדכוני גרדינטים תכופים ואסינכרוניים ממספר רב של Workers יגיע במהירות לרוויה ברוחב הפס וביכולת העיבוד שלו. חלוקת האחריות על שכבות המודל בין מספר Masters (תצורת Model Parallelism חלקית) מאפשרת לפזר את עומס התעבורה ברשת ואת חישובי עדכון המשקולות במקביל, ובכך מאפשרת למערכת לגדול ולתמוך במספר רב של Workers מבלי שביצועי האימון ייפגעו כתוצאה מהמתנה לתגובת השרת המרכזי.

8.



9. ההתבדרות באימון האסינכרוני כאשר מספר ה-Workers גדל נובעת מ- staleness gradients. כל Worker קורא את הפרמטרים מה-Master, מחשב גרדינטים ושולח אותם חזרה לעדכון. כאשר ישנם Workers רבים, קצב העדכון ב-Master הופך למהיר מאוד, כך שבזמן ש-Worker מסוים מחשב את הגרדינט, ה-Master כבר הספיק לקבל ולבצע עדכונים מ-Workers אחרים. כתוצאה מכך, הגרדינט שה-Worker מחזיר מחושב ביחס לנקודה ישנה במרחב הפרמטרים, שאינה תואמת עוד למצב הנוכחי של המודל. כלל שיש יותר Workers, הפער הזה גדל, והגרדינטים הישנים פועלים כרעש חזק המסיט את כיוון הירידה ב-Loss, מה שמוביל לחוסר יציבות, פגיעה בלמידה ולבסוף להתבדרות מוחלטת של המודל (כפי שנצפה בתוצאות הדיוק האקראי ב-16 ליבות).

10. נשווה את תוצאות הגישה הסינכורית והאסינכורנית. מבחינת זמנים ניתן לראות כי הגישה האסינכרונית טובה יותר לעומת הגישה הסינכרונית לזמן אימון המודל כפי שהרחבנו יותר בסעיף 6. מבחינת תוצאות הגישה הסינכרונית מביאה דיוק טוב ללא שונות גובהה בין הריצות כתלות מספר העובדים. זאת מכיוון שכשאר מסנכרנים את כל העובדים כולם מקבלים בכל איטרטציה את ה-w וה-b הנכונים. לעומת זאת, בגישה האסינכרונית יש לעיתים יש דיוק טוב ולפעמים לא כתלות במספר העובדים. ככל שיש יותר עובדים כך הגישה נשברת ואחוזי הדיוק קורסים הרחבנו על כך בשאלה 6 וגם בשאלה 9. זה גורם לגישה להיות פחות עקבית בדיוק שכן היא מספקת תלות במספר העובדים בהם היא משתמשת.

11.

```
(tf23-gpu) dashay@lambda:~/CDPwinter26/HW3$ srun -K -c 4 -n 2 --mpi=pmi2 --pty python3 allreduce_test.py
Testing array size: 4096
Naive all-reduce time: 0.0006334781646728516
Ring all-reduce time: 0.0001609325408935547
Comparing results...
Naive all-reduce correct: True
Ring all-reduce correct: True

Testing array size: 32768
Naive all-reduce time: 0.001264333724975586
Ring all-reduce time: 0.0006146430969238281
Comparing results...
Naive all-reduce correct: True
Ring all-reduce correct: True

Testing array size: 262144
Naive all-reduce time: 0.004481792449951172
Ring all-reduce time: 0.002457141876220703
Comparing results...
Naive all-reduce correct: True
Ring all-reduce correct: True

Testing array size: 2097152
Naive all-reduce time: 0.030797958374023438
Ring all-reduce time: 0.022222280502319336
Comparing results...
Naive all-reduce correct: True
Ring all-reduce correct: True
```

```
(tf23-gpu) dashay@lambda:~/CDPwinter26/HW3$ srun -K -c 4 -n 4 --mpi=pmi2 --pty python3 allreduce_test.py
Testing array size: 4096
Naive all-reduce time: 0.0007290840148925781
Ring all-reduce time: 0.0004775524139404297
Comparing results...
Naive all-reduce correct: True
Ring all-reduce correct: True

Testing array size: 32768
Naive all-reduce time: 0.0015478134155273438
Ring all-reduce time: 0.0010876655578613281
Comparing results...
Naive all-reduce correct: True
Ring all-reduce correct: True

Testing array size: 262144
Naive all-reduce time: 0.009649991989135742
Ring all-reduce time: 0.008933305740356445
Comparing results...
Naive all-reduce correct: True
Ring all-reduce correct: True

Testing array size: 2097152
Naive all-reduce time: 0.07873678207397461
Ring all-reduce time: 0.07883024215698242
Comparing results...
Naive all-reduce correct: True
Ring all-reduce correct: True
```



```

(tf23-gpu) dashay@lambda:~/CDPwinter26/HW3$ srun -K -c 4 -n 8 --mpi=pmi2 --pty python3 allreduce_test.py
Testing array size: 4096
Naive all-reduce time: 0.0016393661499023438
Ring all-reduce time: 0.0010004043579101562
Comparing results...
Naive all-reduce correct: True
Ring all-reduce correct: True

Testing array size: 32768
Naive all-reduce time: 0.0035986900329589844
Ring all-reduce time: 0.0018281936645507812
Comparing results...
Naive all-reduce correct: True
Ring all-reduce correct: True

Testing array size: 262144
Naive all-reduce time: 0.024361848831176758
Ring all-reduce time: 0.016876697540283203
Comparing results...
Naive all-reduce correct: True
Ring all-reduce correct: True

Testing array size: 2097152
Naive all-reduce time: 0.17928004264831543
Ring all-reduce time: 0.18152165412902832
Comparing results...
Naive all-reduce correct: True
Ring all-reduce correct: True

```

מהתוצאות עולה כי באופן כללי, מימוש ה-Ring All-Reduce יעיל ומהיר יותר ממימוש ה-Naive, כאשר היתרון בולט במיוחד במספר ליבות נמוך (2) ועבור גדלי מערך קטנים עד בינוניים.

ב-2 ליבות: ה-Ring All-Reduce מהיר יותר באופן עקבי בכל גדלי המערכים. לדוגמה, עבור המערך הגדול ביותר (כ-2 מיליון איברים), זמן הריצה היה כ-0.022 שניות ב-Ring לעומת כ-0.030 שניות ב-Naive. עבור מערכים קטנים, ה-Ring היה מהיר פי 4 בקירוב.

ב-4 ו-8 ליבות: המגמה נשמרת עבור מערכים קטנים ובינוניים, שם ה-Ring ממשיך להציג ביצועים עדיפים. עם זאת, עבור גודל המערך המקסימלי (2097152), אנו רואים התכנסות בביצועים ולעיתים אף יתרון קל ל-Naive (למשל ב-8 ליבות: 0.179s ל-Naive מול 0.181s ל-Ring).

מכאן נובע כי ה-Ring All-Reduce מיטיב לחלק את רוחב הפס על ידי פיצול המידע לצ'אנקים, מה שמקטין את העומס על כל חיבור בודד ומונע צווארי בקבוק האופייניים לגישה הנאיבית (שבה כל תהליך שולח את כל המידע לכולם). עם זאת, ככל שמספר התהליכים עולה, התקורה של ניהול מספר רב של הודעות קטנות (Latency) באלגוריתם הטבעת עשויה להשפיע, ולכן במערכים גדולים מאוד הפער מצטמצם בתנאי הרצה אלו.

12. במימוש הנאיבי של אלגוריתם All-Reduce, כל תהליך שולח את המערך המלא שלו (בגודל  $N$ ) לכל  $P - 1$  התהליכים האחרים ומקבל מהם את המערכים שלהם. מבחינת נפח תעבורה, כל תהליך בודד שולח ומקבל נתונים בכמות של  $(P - 1)N$ . סך כל המידע העובר ברשת בכל המערכת הוא הסכום עבור כל התהליכים, כלומר  $NP(P - 1)$ , שהם סדר גודל של  $O(P^2N)$ . תלות ריבועית זו במספר התהליכים גורמת לצוואר בקבוק משמעותי ברוחב הפס ככל שמספר המעבדים גדל, כפי שנצפה בתוצאות הניסוי.

13. אלגוריתם ה-Ring All-Reduce נועד לפתור את בעיית רוחב הפס של הגישה הנאיבית על ידי חלוקת המידע לצ'אנקים. האלגוריתם מורכב משני שלבים (All-Gather ו-Scatter-Reduce), שכל אחד מהם אורך  $P - 1$  צעדים. בכל צעד, כל תהליך שולח ומקבל רק צ'אנק אחד בגודל  $N/P$ . לכן, כמות המידע ששולח כל תהליך בודד היא  $2 * N/P(P - 1)$ , שזה בקירוב  $2N$ . סך כל המידע שעובר ברשת הוא  $2N(P - 1)$  שהוא סדר גודל של  $O(N * P)$ .

14. א. Sequential Consistency גורר Coherent-Causal Consistency: הטענה נכונה. הוכחה: עקביות סדרתית (SC) דורשת שכל הפעולות של כל המעבדים ייראו כאילו בוצעו בסדר סדרתי אחד (Total Order) המכבד את סדר התוכנית של כל מעבד. סדר גלובלי זה מכתוב בהכרח סדר חוקי לכל משתנה בנפרד (Coherence) ומכבד את יחסי הסיבה והתוצאה (Causal Consistency), שכן כל קריאה רואה את הכתיבה האחרונה בסדר הגלובלי הזה.

ב. Coherent-Causal Consistency גורר Sequential Consistency: הטענה שגויה. דוגמה נגדית: נחשב תרחיש עם שני משתנים  $x, y$  (מאותחלים ל-0) ושני תהליכים כותבים ושניים קוראים. תהליך 1 כותב  $W(x)1$ . תהליך 2 כותב  $W(y)1$ . תהליך 3 קורא  $R(x)1$  ואז  $R(y)0$  (רואה את  $x$  לפני  $y$ ). תהליך 4 קורא  $R(y)1$  ואז  $R(x)0$  (רואה את  $y$  לפני  $x$ ). תרחיש זה הוא Coherent (עבור כל משתנה בנפרד יש סדר חוקי: קודם אפס ואז אחד) והוא Causal (אין קשר סיבתי בין הכתיבות ל- $x$  ו- $y$ , אז מותר לתהליכים שונים לראות בסדר שונה), אך הוא אינו Sequential Consistency, שכן לא קיים סדר גלובלי יחיד של כל הפעולות שיכול להסביר את התצפיות הסותרות של תהליך 3 ותהליך 4.