

### תרגיל בית 3 - מקבוזר

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

#### :Naive all reduce

תחילה, אנו מעתיקים את מערך `h-send` לטור מערך `h-recv` (המערך הצובר). לאחר מכן, אנו עוברים בלולאה על כל שאר התהליכים (`src` מ-0 עד `size`). אם האינדקס שווה `l-rank` שלנו, אנו מגדגים. אחרת, בכל איטרציה, התהלייר מבצע החלפה הדידית (`Sendrecv`) עם התהלייר שהמזהה שלו הוא `src`: הוא שולח לו את מערך `h-send` המקורי שלו, ומקבל ממנו את מערך `h-send` שלו לטור 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 `Reductions` של כל התהלייכים.

#### :Ring all reduce

הימוש שלנו מtbצע בשתי פазות: פיזור של מערך `h-send` בין כל התהלייכים (`scatter-reduce`) ואז הפצה של כל המידע לכל התהלייכים וחיבור על ידי הפעולה (`all gather`). אנו מחלקים את המערך ל- $k$  צ'אנקים (כמספר התהלייכים) כאשר התהלייר האחרון מקבל את מה שנשאר מחלוקת.

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

בשלב `gather`: מטרתנו להפיץ את הצ'אנקים המוכנים. אנו מאתחלים את המוצבע `current` לאינדקס  $(k \% 1) + rank$  (זהו הצ'אנק שעבורו יש לנו כרגע את התוצאה הסופית). לאחר מכן אנו מבצעים לולאה של  $1-k$  איטרציות. בכל איטרציה: שולחים את הצ'אנק הנוכחי (`current`) לשכן ימיין. מעדכנים את `current` אחורה מעגלית:  $(+ k \% 1) - current$ . מקבלים מהשכן משמאל את הצ'אנק החסר ושומרים אותו באפר זמני. מעתיקים את המידע שהתקבל (`tmp`) לשירות לטור המערך במקום המתאים (ללא `op`).

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

### Async network

do\_worker: הפונקציה do\_worker אחראית על ביצוע שלב חישוב הגרדיניטים וסנכרון מול ה-Masters. בתחלת הריצה, הפונקציה מחלקת את עומס העבודה ומחשבת כמה Batches על התהיליך הנוכחי לבעץ, כאשר התהיליך האחרון ("השarity") לוקח על עצמו את כל ה-Batches הנותרים כדי להבטיח כייסוי מלא של הדאטה. בתוך הלולאה הראשית (עבור כל Epoch וכן כל Batch), ה-Worker מבצע Backward propagation ו-Forward propagation והוא שולח את הגרדיניטים הללו באמצעות Isend אסינכריוני (Non-blocking) במאזעות self.weights ל-Masters הרלוונטיים, כאשר כל Master אחראי על תת-קבוצה של שכבות (לפי החלוקה num\_masters % layer\_index). מיד לאחר השילחה, ה-Worker מבצע האזנה אסינכריונית (Irecv) לקבלת המשקولات וההתוויות 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. תחילתה, היא מתחילה באפרים עבור הגרדיניטים של השכבות הספציפיות שהוקזו לה. בולולאה הראשית, ה-Worker לא יודע איזה Master יסימן ראשון, ולכן הוא מאמין לקבלת גרדיניטים מ-MPI\_ANY\_SOURCEüber השכבה הראשונה שבאחריותו. ברגע שהודעה כזו מתיקבלת, ה-Worker מחלץ את ה-Rank של ה-Worker של אותה ("נעילה" על ה-Worker), וזה מבצע Irecv יזום כדי לקבל את שאר הגרדיניטים לאותו Batch ספציפית מאותו Worker. לאחר שכל הגרדיניטים התקבלו, ה-Worker מבצע צעד של Gradient Descent ומעדכן את המשקولات המקומיות שלו. לבסוף, הוא שולח את הפרמטרים המעודכנים בחזרה אך ורק ל-Worker שמננו קיבל את הגרדיניטים (במאזעות Isend), וממתין לסיום השילחה לפני שהוא חוזר להاذין לבקשת 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 המקורי (Original) ייתר מהמימוש הסדרתי (Original) בכל המקרים. כמו כן, המעבר ל-16 ליבות גרם להרעה משמעותית בביטויים. הסיבה המרכזית לכך שהמימוש הסינכרוני איטי יותר היא שזמן התקשרות (ההעברת הגרדיננטים בין התהליכיים ב-*AllReduce*) עולה על הזמן שנחסר על ידי חלוקת החישוב. המודל שבו משתמשים הוא קטן יחסית, והчисובים (Forward/Backward) מהירים מאוד. לכן, המחיר של שליחת המידע בראש וסנכוון התהליכיים גבוה יותר מהתועלת שבמקביליות.

4. השיטה שלפיה קיבל סfidaf היא חוק אמדהיל. גודל הבעה במקורה שלנו הוא גדול ה-data-set. החלק בתוכנית שאינו ניתן למקובל הוא החלק שבו אנחנו מחשבים מחדש את b,w. לא משנה בכמה נגדיל את ה-data-set עדין חלק זה יהיה לא ניתן למקובל. נראה שהחלק הזה אינו זניח כל SHA-data-set גדול. בנוסף לכך, החלק שכן ממקובל בתוכנית הוא חישוב הגראדיינטס על כל batch.

כל worker מחשב:

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

ולכן סך הכל כל ה-workers יחד מחשבים:

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

נשים לב שהחלק הבעה שנייתן למקובל תלוי ב-Of batches. So ו-ב-mini batch size. שבנה מהם נשארים קבועים ככל שմגדילים את ה-data-set החלק הממקובל נשאר קבוע. ולכן קיבלנו כי ה-A בחוק אמדהיל הוא קבוע. כדי לקבל speedup גם לפיה השיטה של גוסטפסון נרצה להתנות את גודל ה-batch שכל עובד יעבד עליו לפחות ה-data. בצורה כזו החלק הממקובל כן יהיה תלוי בגודל הבעה ונתקבל סfidaf.

## 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%]

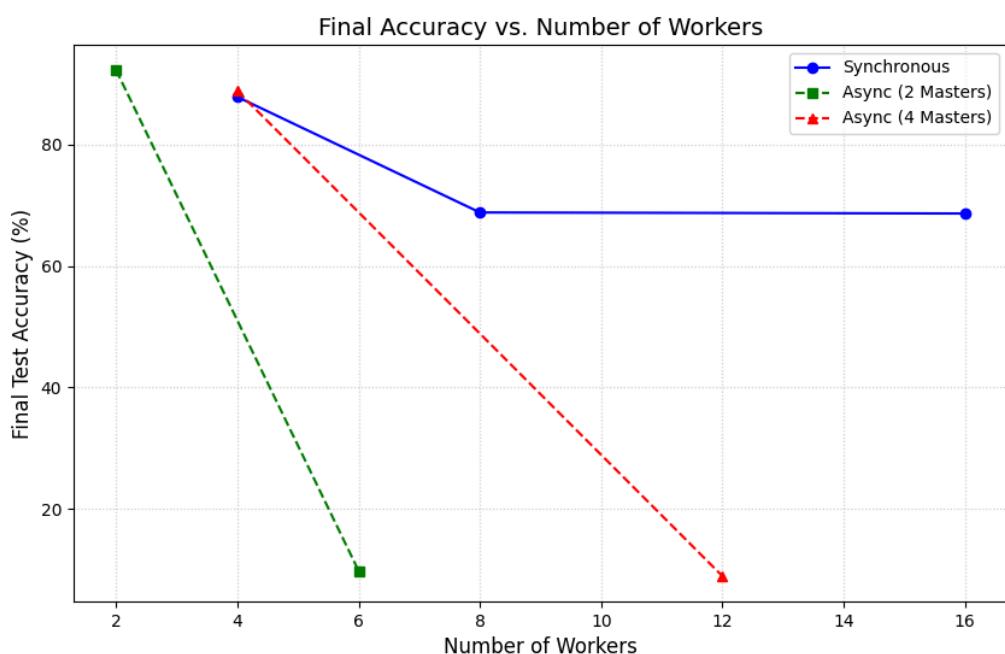
Srun: force terminated job 348896
[(tf23-gpu) dashay@lambda:~/CDPwinter26/HW3$ srun -K -c 2 -n 16 --mpi=pmi2 --pty python3 main.py async 4
Epoch 1, accuracy 49.99 %.
Epoch 2, accuracy 87.06 %.
Epoch 3, accuracy 90.03 %.
Epoch 4, accuracy 90.62 %.
Epoch 5, accuracy 92.69 %.
Time reg: 14.141512870788574
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 הrizות השונות, ניכר כי המימוש האסינכרוני מציג שיפור עקבי ומשמעותי בזמן הrizה לעומת המימוש הסדרתי (המקור), ואף עולה בביצועו על המימוש הסינכרוני שנבחן קודם לכן. השיפור בולט במיוחד בrizה עם 16 ליבות, שם זמן הrizה צנחה מ-14.14 שניות (סדרתי) ל-4.87 שניות בלבד. הסיבה המרכזית להאצה זו היא ביטול ה-Overhead של הסyncronization: בארכיטקטורה אסינכרונית, ה-Workers אינם ממתינים זה לזה ב-Barrier בסוף כל Batch, אלא ממשיכים לעבוד נתונים באופן רציף (Non-blocking). בנוסף לכך מבחינת דיווק המודל, התוצאות חושפות את החיסרון המהותי של אימון אסינכרוני המכונה "Stale Gradients" (גרדינטים לא מעודכנים). כאשר מספר ה-Workers יהיה נמוך (2 או 4, גרדינטים לא מעודכנים). לעומת זאת, כאשר מספר ה-Workers יהיה גבוה (כ-92%-ו-88.85% בהתאם), הדומה לדיווק המודל הצליל להתכנס לדיווק גבוה (כ-92%-ו-88.85% בהתאם), הדומה לדיווק הסדרתי. עם זאת, כאשר מספר ה-Workers גדול יותר (Masters 2-6 (בתצורת 2 Masters) או 12-16 (בתצורת 16 Cores), הדיווק קרס לרמה של ניחוש אקראי (~9.8%). תופעה זו מתרכשת מכיוון שקצב העדכון של המשקولات ב-Master הופך ל מהיר מאוד ביחס בזמן החישוב של Worker בודד. עד ש-Worker מסוים לחשב את הגרדינט ושולח אותו, המשקولات ב-Master כבר הספיקו להשתנות מספר רב של פעמים על סמך עדכנים מ-Workers אחרים. כתוצאה מכך, הגרדינט שנשלח מחושב ביחס לנקודת ישנה במרחב הפרמטרים, וכיוון העדכון שהוא מציע אינו רלוונטי עוד ואף מזיק להתקדמות, מה שmobiel להתבדרות המודל ולכישלון באימון.

7. אנו מפצלים את שרת הפרמטרים על פני מספר מכונות כדי למנוע צוואר בקבוק של תקשורת וחישוב, שכן שרת יחיד המקבל עדכוני גרדינטיים תכופים ואסינכרוניים מספר רב של Workers יגיע במהירות לרוחב הפס וביקולת העבודה שלו. חלוקת האחראיות על שכבות המודל בין מספר Masters (צורת Model Parallelism חלקית) מאפשרת לפזר את עומס התעבורה בראשת ותיחזוק עדכון המשקولات במקביל, ובכך מאפשרת למערכת לגודל ולתמוך במספר רב של Workers מבלי שביצוע האימון יפגיעו כתוצאה מהמתנה לתגובה השרת המרכזי.

8.



9. ההתבדרות באימון האסינכרוני כאשר מספר ה-Workers גדול נובעת מ-*staleness gradients*. כל Worker קורא את הפרמטרים מה-Master, מחשב גרדינטים ושולח אותם חזרה לעדכון. כאשר ישנו Workers רבים, קצב העדכון ב-Master הופך למהיר מאוד, כך שזמן ש-Worker מסוים מחשב את הגרדינט, ה-Master כבר הספיק לקבל ולבצע עדכנים מ-Workers אחרים. כתוצאה לכך, הגרדינט שה-Worker מחזיר מחושב ביחס לנקודה שונה במרחב הפרמטרים, שאינה תואמת עוד למצב הנוכחי של המודל. ככל שיש יותר Workers, הפער זהה גדול, והגרדינטים הישנים פועלם כרעש חזק המסייע את כיוון הירידה ב-Loss, מה שmobiel לחוסר יציבות, פגיעה בلمידה ולבסוף להתבדרות מוחלטת של המודל (כפי שנצפה בתוצאות הדיקן האקדמי ב-16 ליבות).

10. נשווה את תוצאות הגישה הסינכורית והאסינכורנית. מבחינת זמינים ניתן לראות כי הגישה האסינכורנית טוביה יותר לעומת הגישה הסינכורנית לזמן אימון המודל כפי שהרchnנו יותר בסעיף 6. מבחינת תוצאות הגישה הסינכורנית מביאה דיקט טוב ללא שונות גובהה בין הריצות כתלות במספר העובדים. זאת מכיוון שכשאר מסנכרנים את כל העובדים יכולים לקבלים בכל איטרטציה את ה-a וה-b הנכונים. לעומת זאת, בגישה האסינכורנית יש לעיתים יש דיקט טוב ולפעמים לא כתלות במספר העובדים. ככל שיש יותר עובדים כך הגישה נשברת ו אחוז הדיקט קורסים הרחובנו על כך בשאלת 6 וגם בשאלת 9. זהגורם לגישה להיות פחות יעכיבית בבדיקה 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: 0.00893305740356445
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, שימוש ה-All-Reduce Ring יעיל ומהיר יותר מימוש ה-Naive, כאשר היתרון בולט במיוחד במקרים רבים (2) ועבור גודלי מערכות קטנים עד בינוניים.

ב-2 ליבות: ה-All-Reduce Ring מהיר יותר באופן עקבי בכל גודלי המערכות. 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 מערכות קטנות, ה-Ring היה מהיר פי 4 בקירוב.

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

מכאן נובע כי ה-All-Reduce Ring מיטיב לחלק את רוחב הפס על ידי פיצול המידע לצ'אנקים, מה שמקטין את העומס על כל חיבור בודד ומונע צווארי בקבוק האופייניים לגישה הנאייתית (שבה כל תחילה שולח את כל המידע לכלם). עם זאת, ככל שמספר התחיליכים עולה, התקורה של ניהול מספר רב של הודעות קטנות (Latency) באlgorigithm הטעetta עשויה להשפיע, ולכן במקרים גדולים מאוד הפער מצטמצם בתנאי הריצה אלו.

12. בmimeosh הנאיבי של אלגוריתם All-Reduce, כל תהליך שולח את המערך המלא שלו (בגודל  $N$ ) לכל  $1 - P$  התהיליכים האחרים ומקבל מהם את המרכיבים שלהם. מבחינת נפח תעבורת, כל תהליך בודד שולח ומקבל נתונים בכמות של  $N(1 - P)$ . סך כל המידע העובר בראשת בכל המערכת הוא הסכום עבור כל התהיליכים, כלומר  $PN(1 - P)$ , שהם סדר גודל של  $(N^2P)O$ . תלות ריבועית זו במספר התהיליכים גורמת לצואර בקבוק משמעותי ממשמעותי ברוחב הפס ככל שמספר המעבדים גדול, כפי שנצפה בתוצאות הניסוי.

13. אלגוריתם-h 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 אורק  $1 - P$  צעדים. בכל צעד, כל תהליך שולח ומקבל רק צ'אנק אחד בגודל  $P/N$ . לכן, כמות המידע ששולח כל תהליך בודד היא  $2 * (1 - P/N)P$ , שהיא בקירוב  $N/2$ . סך כל המידע שעובר בראשת הוא  $(1 - P)2N$  שהוא סדר גודל של  $(P * N)O$ .

14. א. הטענה Coherent-Causal Consistency גורר Sequential Consistency. הטענה נכונה: עקביות סדרתית (SC) דורשת שכל הפעולות של כל המעבדים יראו כאילו בוצעו בסדר סדרתי אחד (Total Order) המכבד את סדר התוכנית של כל מעבד. סדר גלובלי זה מכתיב בהכרח סדר חוקי לכל משתנה בנפרד (Coherence) ומכבד את יחס הסיבת וההתוצאה (Causal Consistency), שכן כל קרייה רואה את הכתיבה האחרונה בסדר הגלובלי זהה.

ב. הטענה Coherent-Causal Consistency גורר Sequential Consistency. הטענה נכונה: דוגמה נגדית: נחשב תרחיש עם שני משתנים  $x, y$  (מאוחלים ל-0) ושני תהליכים כתובים ושניים קוראים. תהליך 1 כותב  $1(x)W$ . תהליך 2 כותב  $1(y)W$ . תהליך 3 קורא  $1(x)R$  ואז  $0(y)R$  (רואה את  $x$  לפני  $y$ ). תהליך 4 קורא  $1(y)R$  ואז  $0(x)R$  (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 הכתיבה ל- $x$  ו- $y$ ), אך מותר לתהיליכים שונים לראותם בסדר שונה), אך הוא אינו Sequential Consistency, שכן לא קיים סדר גלובלי יחיד של כל הפעולות שיכל להסביר את התוצאות הסותרות של תהליך 3 ותהליך 4.