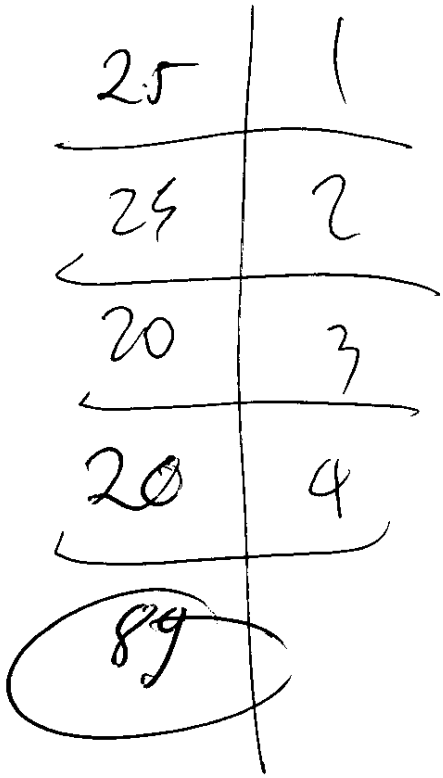
סמסטר ב'
יום שלישי 15 יוני 2010

תכנות מחשבים רבי מעבדים
מבחן סוף סמסטר – מועד א'

מרצה: ניר שביט
מתרגל: גיא קורלנד

<u>הוראות כלליות</u>

1. משך הבחינה **שלוש שעות.**
2. **יש לרשום ת"ז ומס' מחברת בראש כל עמוד.**
3. חומר עזר מותר בשימוש: כל חומר כתוב או מודפס שהבאת אתך.
4. יש לענות על כל השאלות.
5. ניתן להשתמש בכל קוד שהוזכר בשיעור או מופיע בספר.
6. כתוב את כל תשובותיך בטופס המבחן – המחברת לא תיבדק!
7. יש לקרוא בעיון את השאלות.
8. עזור לבודק המבחן לבדוק את עבודתך — **<u>כתוב בצורה נקייה ומסודרת (עדיף בעפרון!!!!)</u>**
9. הבחינה כוללת 9 עמודים כולל עמוד זה.



ב ה צ ל ח ה ! !

שאלה 1 (25):

נתונה תוכנית לחוט יחיד הרצה על מעבד יחיד המסוגל לבצע 3 GFlops. תוכנית זו הומרה כך ש-90% מהריצה הינה מקבילית ו-10% סידרתית.

(6).1 בהנתן מעבד בעל 16 ליבות שכל ליבה שלו מסוגלת לבצע 2 GFlops. מה ההאצה הפוטנציאלית?

$$Speedup = \frac{1}{[(1-0.9)+\frac{0.9}{16}]\cdot\frac{3}{2}} = \frac{16}{(1.6+0.9)\cdot\frac{3}{2}} = \frac{32}{7.5}$$

$\frac{32}{7.5} \simeq 4.266$

(7).2 בהמשך הוצע מעבד אסימטרי בעל 13 ליבות: 12 ליבות סימטריות המסוגלות לרוץ במהירות 1.6 GFlops וליבה ראשית המסוגלת לרוץ במהירות 3 GFlops. מי משני המעבדים, הסימטרי או האסימטרי, תעדיף?

$$Speedup = \frac{1}{0.1\cdot\frac{3}{3}+\frac{0.9}{13}\cdot\frac{3}{22.2/13}} = \frac{1}{0.1+\frac{2.7}{22.2}} = \frac{22.2}{2.22+2.7} = \frac{22.2}{4.92} \simeq 4.512$$

(12).3 הצע נוסחה כללית להאצה עבור מעבד אסימטרי (מספר ליבות סימטריות וליבה ראשית אחת). הנח כי הריצה המקורית הינה על מעבד יחיד המסוגל לבצע S1 פעולות בעוד במעבד האסימטרי המעבד הראשי יכול לבצע S2 פעולות וכל אחד מהמעבדים הסימטרים יכול לבצע S3 פעולות.

p – החלק המקבילי

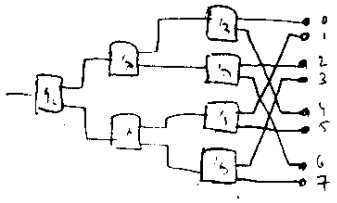
$$Speedup = \frac{1}{(1-p)\cdot\frac{S1}{S2}+\frac{p}{n}\cdot\frac{S1}{\frac{S2+(n-1)S3}{n}}} \Rightarrow$$

$$Speedup = \frac{1}{(1-p)\cdot\frac{S1}{S2}+\frac{p\cdot S1}{S2+(n-1)S3}}$$

שאלה 2 (25):

ראינו בכיתה את - Tree[k], k >= 4 diffracting tree network.
כמו כן ראינו את השימוש שלו למימוש מונה אטומי ע"י הצבת מונים בעלי הערך.

1.(10) האם המימוש של המונה linearizable בריצה בה לכל היותר k חוטים ניגשים לעץ? הוכח את טענתך או הראה דוגמא נגדית.



- תהי A נגיעה ראשון יעבור לפני ה- balancer בגובה 2 בדוגמה ב'.
- תהי B נגיעה שני, אבל לגובה 1 ויעבור לפני ה balancer בדוגמה ב' (בגובה 2)
- תהי C נגיעה שלישי, אבל אחרי A ו-B ויצא מהמונה לפני B.

אבל הנכונה 2 ולכן לא ייתכן שבהכרח יקבלו 0,1 ולכן הוא לינאריזבילי (לדוגמה C).

2.(15) נתון מונה המורכב מ-diffracting tree אשר הפלט שלו מוביל לעלים של combining tree ברוחב k, כך שכל חוט ניגש קודם ל-diffracting Tree ואז ל-combining tree ואז מחזיר את התוצאה המתקבלת מה-combining tree. האם מימוש זה linearizable בריצה בה לכל היותר k חוטים ניגשים לעץ? הוכח את טענתך או הראה דוגמא נגדית.

שאלה 3 (20):

מה הוא מספר הקונסנזוס של אובייקט ה-BDEQueue אשר ראינו מימוש שלו בכיתה (פרק 16 עמ' 383-384, ראה נספח)? תזכורת: רק חוט יחיד P יכול לקרוא ל- pushBottom ו-popBottom בעוד שאר החוטים (שונים מ-P) יכולים לבצע את popTop. במקרה ו- popTop נקרא במקביל לפעולות popTop אחרות או לפעולת popBottom, לפחות קריאה אחת תצליח להחזיר ערך אם התור לא ריק, וחלק מהקריאות האחרות יכולות להכשל עם ערך החזרה null. הוכח את טענתך.

מספר הקונסנזוס הוא: 2.

תחילה נראה פרוטוקול ל-2 חוטים, ואז נראה שלא יתכן לשלושה. הפרוטוקול:

- מאתחלים BDEQueue שיכיל איבר אחד כלשהו שמסמן ערך שונה מ-null.
- כל אחד משני החוטים שנכנסים לפרוטוקול תחילה מפרסמים את הערך שלהם במערך הרגיסטרים המשותפים, ואז מבצעים popTop. אם קיבלו ערך, מחזירים את הערך שלהם. אם קיבלו null, מחזירים את הערך של השני. הנכונות נובעת מכך שרק אחד יכול לקבל ערך, כי החוט הקודם שזכה בלקיחה החזיר את ערכו, וזה היה אחרי שפרסם אותו - לכן המפסיד בהכרח יראה את ערך המנצח במערך.

10/10

(שימוש בהוכחה רגילה) כדי להראות שאין פרוטוקול ל-3 חוטים. נניח ב"ש חוטים A,B,C הרצים הליך לפתרון קונצנזוס, שמשתמש ב BDEQ ורגיסטרים משותפים. כמו שנלמד, כמו בהוכחות דומות, יש מצב קריטי בו המהלך הבא של A יביא ל 0-valent ושל B ל-1-valent. אלו יהיו של אובייקט אחד יחיד, שכן הפעולות על שני אובייקטים נפרדים הן commute. נראה את כל הצעדים האפשריים:

נשים לב שלא יתכן 2 ביצועי push ו-pop ל-Bottom, לכן בה"כ נניח כי A [illegible].

[illegible]

10/10

- A: pushBottom, B: popTop: אם A עשה pushBottom קודם, הוא היה [illegible] לפני pop [illegible] בוולנט [illegible] A עשה קודם pushBottom, ואחר כך B מבצע pop [illegible] B ויבצע popTop, וזו לעומת מצב בו B ירוץ ראשון popTop - שני המצבים האחרונים לא ניתנים להבחנה אם נריץ אחריהם את C בריצה סולו, ולכן [illegible] 1-valent [illegible] סתירה.
- A: popBottom, B: popTop: כמו קודם, קודם A [illegible] B [illegible] A [illegible] - המצבים הסופיים לא ניתנים להבחנה ע"י C בריצה סולו - ושוב סתירה.

[illegible]

- A: popTop, B: popTop: וגם כאן, לא משנה מי מהם יבצע popTop קודם, בסוף שני המצבים לא יהיו ניתנים להבחנה ע"י C בריצה סולו ולכן גם כאן סתירה...

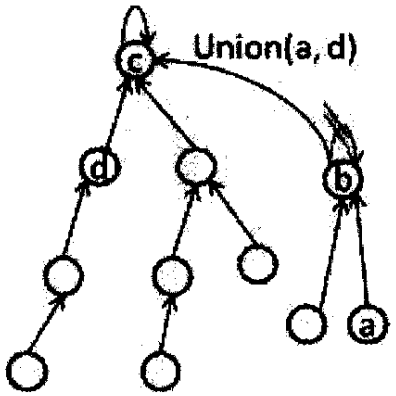
מכיוון שלא יתכן פרוטוקול קונצנזוס ל-3 חוטים ב-BDEQueue, לכן מספר הקונצנזוס שלו הוא בדיוק 2.

<u>שאלה 4 (30)</u>:

<u>הערה</u>: לצורך השאלה הנח מרחב חסום (מספר ידוע n) של ערכים בעלי מפתחות שונים.

האוביקט union-find-same הינו אוסף של קבוצות הממש שתי מטודות:
union(a,b) - מאחדת את כל הקבוצות המכילות את הערכים a ו-b.
find-same(a,b) - מחזירה true עם קיימת קבוצה המכילה את הערכים a ו-b ואחרת false.
בתחילת הריצה כל קבוצה מכילה איבר בודד ממרחב של n הערכים.

כפי שניתן לראות בתמונה להלן, קבוצה מתוארת ע"י עץ עם מסלולים מהעלים לשורש. המציין של שורש הוא מצביע לעצמו.
union(a,b) מוצאת השורשים של העצים המכילים את a ו-b ומשנה את המצביע משורש אחד לשני.
find-same(a,b) מסיירת בעץ מצמתים a ו-b בהתאמה לשורשם ובודקת האם יש להם שורש משותף.




```
public class NetworkSet{
   final public Node root = new Node();

   public NetworkSet(Object value){
      root.parent = root;
      root.value = value;
   }

   public static class Node{
      public Node parent;
      public Object value;
   }
}
```

(10).1 הצע מימוש linearizable מבוסס מנעולים למטודות union ו - find-same המסתמך על מנעול יחיד לכל הקבוצה.

```
public class UnionFindSame{
  Map<Object, Node> nodes = ...;

  public void union(Object a, Object b){
    Node aRoot = nodes.get(a);
    Node bRoot = nodes.get(b);
    try {   // lock Roots
      while (true){ aRoot.lock.lock(); if (aRoot == aRoot.parent) break;
                    else { aRoot.lock.unlock(); aRoot = aRoot.parent; }}
      while (true){ bRoot.lock.lock(); if (bRoot == bRoot.parent) break;
                    else { bRoot.lock.unlock(); bRoot = bRoot.parent; }}
      // change pointers
      aRoot.parent = bRoot;  // good even if aRoot == bRoot
    } finally {  // anyway - unlock
      aRoot.lock.unlock();
      bRoot.lock.unlock();
    }

  public boolean findSame(Object a, Object b){
    // check if a and b have the same root
    if (aRoot == bRoot) return true; else return false;
  }
}
```

(20).2 הצע מימוש wait-free linearizable למטודות union ו - find-same שבו union מבצע מספר קבוע של פעולות CAS.

```
public class UnionFindSame{
  Map<Object, Node> nodes = ...;

  public void union(Object a, Object b){
      AtomicReference<Node> aRoot = new AtomicReference<Node>(nodes.get(a));
                            bRoot =                          (nodes.get(b));

      for (int i = 0; i < n; i++){
          while (true){
              Node a = aRoot.get();
              if (a == a.parent.get()) break; else aRoot.set(a.parent); }
          [ " bRoot ]

          if (aRoot.get() == bRoot.get()) break;  // already unioned
          if (aRoot.get().parent.compareAndSet(aRoot.get(), bRoot.get())) break;
      }
  }

  public boolean findSame(Object a, Object b){
      AtomicReference<Node> aRoot = new AtomicReference<Node>(nodes.get(a));
                            bRoot =                          (nodes.get(b));

      while (true){
          Node a = aRoot.get()
          if (a == a.parent.get()) break;
          else aRoot.set(a.parent);
      }
          " (bRoot)

      return (aRoot.get() == bRoot.get());
  }
}
```

retry

validation

נספח

```
1.  public class BDEQueue {
2.      Runnable[] tasks ;
3.      volatile int bottom;
4.      AtomicStampedReference<Integer> top;
5.
6.      public BDEQueue(int capacity) {
7.          tasks = new Runnable[capacity];
8.          top = new AtomicStampedReference<Integer>(0, 0);
9.          bottom = 0;
10.     }
11.
12.     public void pushBottom(Runnable r){
13.         tasks [bottom] = r;
14.         bottom++;
15.     }
16.
17.     // called by thieves to determine whether to try to steal
18.     boolean isEmpty() {
19.         return ( top.getReference() < bottom);
20.     }
21. }
```

```
1.  public Runnable popTop() {
2.     int [] stamp = new int[1];
3.     int oldTop = top.get(stamp), newTop = oldTop + 1;
4.     int oldStamp = stamp[0], newStamp = oldStamp + 1;
5.     if (bottom <= oldTop)
6.        return null ;
7.     Runnable r = tasks[oldTop];
8.     if (top.compareAndSet(oldTop, newTop, oldStamp, newStamp))
9.        return r ;
10.    return null ;
11. }
12.
13. Runnable popBottom() {
14.    if (bottom == 0)
15.       return null ;
16.    bottom--;
17.    Runnable r = tasks[bottom];
18.    int [] stamp = new int[1];
19.    int oldTop = top.get(stamp), newTop = 0;
20.    int oldStamp = stamp[0], newStamp = oldStamp + 1;
21.    if (bottom > oldTop)
22.       return r ;
23.    if (bottom == oldTop) {
24.       bottom = 0;
25.       if (top.compareAndSet(oldTop, newTop, oldStamp, newStamp))
26.          return r ;
27.    }
28.    top. set (newTop,newStamp);
29.    bottom = 0;
30.    return null ;
31. }
```