חישוב מקבילי מהיר

בשבוע שעבר דיברנו על שני יתרונות של חישוב מקבילי. השבוע נדבר על יתרון השלישי: ביצוע מהיר של חישובים כבדים ע"י שימוש בכמה ליבות. מספר הליבות במחשב שולחני ממוצע הוא 2-4, אבל אפשר היום לשכור מחשבים בענן (למשל בשירות AWS) עם 64 ליבות ויותר, כך שההאצה יכולה להיות משמעותית מאד. נתחיל מדוגמה פשוטה. נסו קודם לפתור אותה בעצמכם בעזרת הידע שלכם על חוטים.

תרגיל כיתה. נתון מערך array ובו 100 מיליון מספרים ממשיים. רוצים לחשב את סכום המספרים בשלישית: sum{i=0...a.length}{array[i]^3}.
החישוב הסדרתי הפשוט לוקח (על המחשב שלי) 8 שניות; מיצאו דרך לבצע את החישוב ב-4 שניות.
בידקו את האלגוריתם שלכם ע"י דוגמה פשוטה, למשל, מלאו את המערך שלכם ב-2 ע"י Arrays.*fil*(array, 2) *l* ובידקו שהתוצאה הסופית היא 800 מיליון.

בקובץ lesson7/ParallelSum נמצאים שני פתרונות:

פתרון **יצרן-צרכן** - ע"י חוטים ותורים

הפתרון הראשון משתמש בשני חוטים. כל חוט מחשב את הסכום בחצי מהמערך, ושומר אותו במשתנה נפרד:

```
double[] sums = new double[2];
Thread t1 = new Thread(() -> {
     int sum = 0;
     for (int i=0; i<array.length/2; ++i)
          sum += Math.pow(array[i], 3);
     sums[0] = sum;
});
t1.start();
Thread t2 = new Thread(() -> {
     int sum = 0;
     for (int i=array.length/2; i<array.length; ++i)
          sum += Math.pow(array[i], 3);
     sums[1] = sum;
});
t2.start();
```

כדי לחשב את הסכום הכולל, התוכנית הראשית צריכה לחכות ששני החוטים יסתיימו. ניתן לעשות זאת ע"י הפקודה join:

```
t1.join();
t2.join();
double sum = sums[0]+sums[1];
```

על המחשב שלי, הפתרון השני לוקח קצת יותר מ-4 שניות. למה יותר מ-4? - כי יש תקורה מסויימת לשימוש בשני חוטים. כאשר עוברים לחישוב מקבילי, כדאי תמיד לבדוק האם ועד כמה אנחנו חוסכים.

הערה: למעלה השתמשנו במערך בגודל 2 עבור שני הסכומים. אילו היינו משתמשים בשני משתנים, למשל sum0, sum1, היינו מקבלים שגיאת קומפילציה. הסיבה היא, שמשתנים מקומיים של מתודה נמצאים על המחסנית הפנימית של המעבד, והמחסנית הזאת נמחקת כשהמתודה מסתיימת, אולם החוטים

ממשיכים לרוץ גם אחרי שהמתודה מסתיימת. לעומת זאת, אובייקטים הנוצרים ע"י new, כגון מערכים, נמצאים על הערימה, והערימה קיימת גם אחרי שהמתודה מסתיימת.

דרך יותר בטוחה להשיג אותה מטרה היא להשתמש ב-BlockingQueue. זהו סוג של תור - אפשר להכניס לתוכו דברים בצד אחד ולהוציא אותם בצד השני באותו סדר. אולם ל-BlockingQueue יש מתודה מיוחדת בשם take: אם התור ריק, המתודה תחכה עד שיגיע איבר חדש לתור (יש גם מתודה בשם poll שמחכה זמן קצוב מראש שיגיע איבר חדש, ואז מפסיקה לחכות ומחזירה ערך null). כדי להשתמש במבנה זה, נחליף את השורה הראשונה (שמגדירה את sums) ב:

```
BlockingQueue<Integer> sums = new ArrayBlockingQueue<>(2);
```

ובמקום שכל חוט ישים את הסכום שלו באחד מאיברי המערך, הוא ישתמש ב-offer:

```
sums.offer(sum);
```

את שלוש השורות האחרונות (שמחכות שהחוטים יסתיימו) נחליף ב:

```
double sum = sums.take() + sums.take();
```

שימו לב - כבר אין צורך בפעולת join. השורה האחרונה מחכה עד ששני סכומי-הביניים יגיעו לתור, ואז לוקחת אותם ומחברת אותם. למידע נוסף על BlockingQueue ראו בתיעוד של אורקל.

הפתרון הזה הוא דוגמה לתבנית שנקראת **יצרן-צרכן** (Producer-Consumer): אפליקציה שבה חלק מהחוטים הם יצרנים של ערכים, וחוטים אחרים הם צרכנים של אותם ערכים. המשתנה BlockingQueue משמש לתיאום בין יצרנים לצרכנים.

חידה: מיצאו דוגמאות נוספות של אפליקציות יצרן-צרכן.

פתרון **מיפוי-צמצום** - ע"י זרמים מקביליים

פתרון שני ושונה מהותית מהראשון הוא להשתמש ב**זרמים** (Streams). **זרם** בג'אבה הוא סדרה של אובייקטים. הוא דומה ל**אוסף**, אבל יש לו מתודות מיוחדות שמאפשרות לבצע פעולות על כל הסדרה באופן מקבילי. המתודות מתחלקות לשני סוגים עיקריים הנקראים **מיפוי** (map) **וצמצום** (reduce):

- המתודה map מבצעת פעולה מסויימת על כל איבר בזרם. למשל, השורה:
  ```
  map(x -> Math.pow(x, 3))
  ```
  יוצרת זרם חדש שבו כל איבר x מועלה בחזקת 3.
- המתודה reduce מצמצמת את כל איברי המערך לאיבר אחד. למשל, השורה:
  ```
  reduce((x,y)->x+y)
  ```
  מצמצמת כל שני איברים סמוכים x,y לאיבר אחד שהוא הסכום שלהם x+y, ומבצעת פעולה זו שוב ושוב עד שנשאר רק איבר יחיד (שהוא הסכום של כל האיברים בזרם).

הכוח של שתי הפעולות הללו מתגלה כשמבצעים אותן זו אחר זו. לדוגמה, את הבעיה שבתרגיל הכיתה ניתן לפתור בשורה אחת כך:

```
double sum = Arrays.stream(array)
      .parallel()
      .map(x -> Math.pow(x, 3))
      .reduce((x,y)->x+y)
      .getAsDouble();
```

משמעות השורה היא: "קח את המערך, הפוך אותו לזרם, הפוך אותו למקבילי, העלה כל איבר במערך בחזקת 3, סכם את כל האיברים, והפוך את התוצאה למספר ממשי". על המחשב שלי (עם 4 ליבות), החישוב הזה לוקח רק 2.5 שניות - המחשב יודע לבד להשתמש בחוטים בהתאם למספר הליבות!

הפתרון הזה הוא דוגמה לתבנית שנקראת Map-Reduce. יש הרבה אלגוריתמים שניתן להציג כשילוב של פעולות מיפוי (=פעולות על כל איבר בנפרד) ופעולות צימצום (הפיכת כל האיברים לאיבר אחד). אלגוריתמים כאלה

הערות:

- מתודה חשובה נוספת השייכת לסוג של פעולות-מיפוי היא filter - סינון. היא מפעילה תנאי מסויים על כל איבר בנפרד, ויוצרת זרם חדש שבו מופיעים רק האיברים המקיימים את התנאי.
- ניתן להשתמש ב-reduce רק עם פעולות **אסוציאטיביות** - פעולות שבהן סדר הביצוע לא משפיע על התוצאה: a*(b*c) = (a*b)*c. האסוציאטיביות מאפשרת לבצע את הפעולות באופן מקביל או סדרתי בכל סדר שהוא, והתוצאה תישאר זהה. למשל, חיבור הוא אסוציאטיבי, ולכן אפשר לחשב סכום של ארבעה איברים בחוט אחד: (((a+b)+c)+d) או בשני חוטים: ((a+b)+(c+d)) והתוצאה תהיה זהה (בטח כשלמדתם מתמטיקה לא הבנתם למה צריך ללמוד על אסוציאטיביות ומה זה קשור למדעי המחשב, נכון? אז הנה, אסוציאטיביות בהחלט קשורה למדעי המחשב. היא מאפשרת לנו למקבל אלגוריתמים ולחסוך המון זמן).
- בגלל החשיבות הרבה של חישוב מקבילי בימינו, יותר ויותר מתכנתים מתרגלים לעבוד מלכתחילה עם זרמים ולא עם לולאות. מאז המצאת הזרמים, הלולאות כבר די "מחוץ לאופנה". הבעיה בלולאה היא, שהיא מכתיבה למחשב בדיוק באיזה סדר לבצע את הפעולות - הוא חייב לעבור על המערך החל מאיבר 0 ועד לאיבר ה-100 מיליון באופן סדרתי. לעומת זאת, זרם רק אומר למחשב **מה** לעשות ולא **איך** לעשות את זה, וכך משאיר לו חופש למקבל את החישוב.

דוגמאות נוספות

- ניזכר בבעיית החלוקה מהשיעור הקודם. אנחנו צריכים למצוא את החלוקה שבה הפרש הסכומים הוא הקטן ביותר. ניתן למקבל את הפתרון בשתי הדרכים שראינו למעלה. דרך א - **יצרן-צרכן** (ראו PartitionThreas). נשתמש בשני חוטים יצרנים: כל חוט יחשב את החלוקה הטובה ביותר בחצי מהתחום. החוט הראשי יחכה לתוצאות של שני החוטים ויחזיר את החלוקה הטובה ביותר מבין השתיים. דרך ב - **מיפוי-צמצום** (ראו PartitionStreams). במקרה זה הזרם ההתחלתי הוא אוסף האינדקסים האפשריים: `IntStream.range(0, numOfPartitions)`, ואנחנו מפעילים עליו רק פעולת צמצום השומרת את האינדקס הטוב ביותר: `.reduce( (i,j) -> diff(values,i)<diff(values,j)? i: j )`.
- **תרגיל בית קל**: איך למקבל חישוב של n! עבור n מאד גדול? (שימו לב, צריך להשתמש ב-BigInteger כדי להחזיק את התוצאה הנכונה).
- **תרגיל בית קשה**: כיתבו תוכנית המציירת פרקטל, לדוגמה, הפרקטל של מנדלברוט (קיראו באינטרנט איך לחשב אותו). הגדילו את מספר החוטים. האם הציור מתבצע במהירות רבה יותר כשיש כמה חוטים? נסו לבצע זאת באפליקציית שרת-לקוח: הלקוח מבקש מהשרת לחשב ערכים בתחום מסויים, והשרת מחזיר לו את הערכים. מה קורה כשהשרת משתמש בחוט אחד? בכמה חוטים?

מקורות

- Cay Horstmann, “Core Java for the Impatient”, chapters 10,8.
- Dane Cameron, "A Software Engineer Learns Java and OOP", chapter 30
- Peter Taylor, "Forking Factorials", https://codegolf.stackexchange.com/a/1641/12019

סיכם: אראל סגל-הלוי.