חישוב מקבילי

תוכנית רגילה בשפת ג'אבה רצה באופן סדרתי, מהתחלה לסוף. יש כמה סיבות שבגללן אנחנו רוצים להריץ תוכניות באופן מקבילי:

1. **תגובתיות**. אנחנו מריצים חישוב שלוקח הרבה זמן, ורוצים במקביל לעשות דברים אחרים במחשב. תגובתיות חשובה במיוחד בתוכנה שמשרתת כמה משתמשים (למשל שרת רשת - שלמדנו בשיעור הקודם). אם משתמש אחד מבצע פעולה שלוקחת הרבה זמן, אנחנו עדיין רוצים שהשרת יהיה פנוי לענות לבקשות של משתמשים אחרים.
2. **מגבלת זמן**. אם חישוב מסויים לוקח יותר מדי זמן, אנחנו רוצים שתהיה לנו אפשרות לעצור אותו מבחוץ. זה שימושי במיוחד למימוש אלגוריתמי "כל זמן" (any-time algorithms) - אלגוריתמי אופטימיזציה שרצים עד שעוצרים אותם, ומחזירים את התוצאה הטובה ביותר שהצליחו למצוא עד כה.
3. **מהירות**. היום, כמעט בכל מחשב יש שתי ליבות או יותר. חישוב מקבילי מאפשר לנו לנצל את הליבות הללו - לבצע חישובים שונים על כל ליבה כך שהתוכנית תשיג את מטרתה מהר יותר.

בעיה לדוגמה

נדגים את היתרונות של חישוב מקבילי בעזרת בעיה לדוגמה - **בעיית החלוקה**:

- נתונה רשימת מספרים ממשיים. יש לחלק אותה לשתי קבוצות כך שסכום המספרים בשתי הקבוצות שווה. אם זה בלתי אפשרי, יש למצוא את החלוקה שבה ההפרש בין הסכומים קטן ביותר.

יישום אפשרי של בעיה זו הוא חלוקה הוגנת של חפצים. שני אחים קיבלו בירושה אוסף של חפצים, לכל חפץ יש שווי אחר, והם רוצים לחלק את החפצים ביניהם כך שהשווי הכולל שכל אחד מקבל הוא זהה, או קרוב ביותר.

הבעיה הזו ידועה כבעיה קשה (ראו בויקיפדיה Partition problem). לא קיים אלגוריתם שפותר אותה בזמן פולינומיאלי. קיימים אלגוריתמי-קירוב שפותרים אותה ביעילות יחסית עבור קלטים מסויימים, אבל לא עבור כל הקלטים. בשיעור זה נפתור את הבעיה בעזרת האלגוריתם הפשוט הבא:

- עבור על כל החלוקות האפשריות של הקלט לשתי תת-קבוצות. לכל חלוקה כזאת:
  - חשב את הפרש הסכומים.
- החזר את החלוקה שבה המרחק קטן ביותר.

מספר החלוקות האפשריות הוא 2 בחזקת מספר האיברים, כך שהאלגוריתם הזה עלול לקחת הרבה זמן, לכן זו דוגמה טובה לחשיבות של תיכנות מקבילי.

האלגוריתם הבסיסי ממומש במחלקה Partition, במתודה best. באותה מחלקה, במתודה main ישנה בדיקה של האלגוריתם על מערכים באורך הולך וגדל. כצפוי, זמן הריצה גדל פי 2 בכל פעם שהמערך גדל ב-1. כשגודל המערך מגיע ל-30, זמן הריצה כבר ארוך למדי.

1. תגובתיות

נניח שיש לנו שרת-רשת (דומה לזה שראינו בשיעור הקודם), המאפשר ללקוחות להזין קלטים לבעיית החלוקה, ומחזיר להם תשובה.

השרת הבסיסי נמצא בקובץ PartitionServer0. כשלקוח שולח בעיה קשה שהפתרון שלה דורש הרבה זמן, השרת לא יכול להתייחס ללקוחות אחרים, והם לא מקבלים תשובה גם על בעיות קלות.

כדי לפתור את הבעיה נשתמש ב**חוטים**. קודם-כל נסביר, מה זה בכלל חוט? בשפת ג'אבה, חוט הוא קטע-קוד שרץ במקביל לתוכנית הראשית. טכנית, חוט הוא עצם מהמחלקה Thread. לחוט יש בנאי

המקבל ארגומנט מסוג Runnable. למדנו על Runnable בשיעור על ממשקים - זה בסה"כ ממשק עם מתודה אחת בלי ארגומנטים. הנה תוכנית שיוצרת שני חוטים וממשיכה לרוץ:

```
new Thread( () -> for(int i=0; i<100; ++i)
System.out.print("1") ).start();

new Thread( () -> for(int i=0; i<100; ++i)
System.out.print("2") ).start();

for (int i=0; i<100; ++i) System.out.print("3")
```

התוכנית הזאת אמורה לכתוב 1, 2, 3 בערבוביה; סדר הריצה של החוטים לא ידוע מראש.

עכשיו, נוסיף לשרת-הרשת שלנו חוטים. בכל פעם שהשרת מקבל בקשה, הוא מתחיל חוט חדש כדי לפתור אותה (ראו במחלקה PartitionServer1).

יש עוד שינוי אחד חשוב שחייבים לעשות כדי שנוכל ליהנות מהיתרונות של חוטים. ג'אבה מתנגדת לעצירת חוטים בכוח; חוטים צריכים לאפשר למערכת לעצור אותם. לכן, אנחנו צריכים לוודא שהאלגוריתם שלנו לפתרון בעיית החלוקה, מאפשר למערכת לעצור אותו זמנית כדי לאפשר לחוטים אחרים לרוץ. זה נעשה ע"י הכנסת הפקודה Thread.yield בתוך הלולאה שעוברת על כל החלוקות (ראו Partition.best).

הריצו את PartitionServer1. פיתחו שני חלונות, באחד הריצו קלט ארוך ובשני קלט קצר, וודאו שהקלט הקצר מתקבל.

כיום, לא מקובל להשתמש ישירות בחוטים. במקום זה, משתמשים ב**שירותי ביצוע** - ExecutorService. שירות-ביצוע מקבל מטלות (כגון Runnable) ומבצע אותן תוך שימוש בחוטים. מדיניות השימוש בחוטים משתנה לפי סוג המבצע: אפשר לבחור מבצע שמשתמש בחוט אחד, או במספר קבוע של חוטים ("ממחזר" חוטים לפי הצורך), או במספר משתנה של חוטים. ראו דוגמה במחלקה PartitionServer2. באיזה שירות-ביצוע כדאי להשתמש?

- אם חשובה לנו ה**תגובתיות** - אז נשתמש ב Executors.newCachedThreadPool(). זהו שירות-ביצוע שמתחיל חוטים בלי הגבלת מספר, לפי הצורך. כך נבטיח שתמיד יהיו במערכת מספיק חוטים לטפל בכל הבקשות.
- אם חשובה לנו יותר ה**יעילות** - אז נשתמש ב Executors.newFixedThreadPool(n), כאשר n הוא מספר הליבות במחשב שלנו. כך נבטיח שיהיה חוט אחד לכל ליבה ולא יהיה צורך בהחלפת חוטים על אותה ליבה.

2. הגבלת זמן

כעת נראה שימוש נוסף של תיכנות מקבילי - הגבלת זמן ביצוע של חישוב ממושך. אנחנו רוצים לחשב חלוקה טובה ככל האפשר, אבל אנחנו לא רוצים לחכות לנצח. אנחנו רוצים את החלוקה הטובה ביותר שאפשר לחשב, נניח, בשניה אחת.

הפתרון נמצא בקובץ PartitionWithInterruption0. הרעיון הוא להריץ חוט, לחכות שניה, ואז **להפריע** לחוט. פעולת ההפרעה נקראת interrupt.

כזכור, ג'אבה מתנגדת לעצירת חוטים בכפיה; בשפת ג'אבה אפשר רק לבקש מחוט בנימוס שיפריע לעצמו. לפיכך, השלב הראשון בפתרון הוא לשנות את הקוד של האלגוריתם כך שיאפשר הפרעה. זה נעשה באופן הבא. במתודה best, בתוך הלולאה שעוברת על כל החלוקות האפשריות, נכתוב:

```
if (Thread.interrupted())
    break;
```

כלומר, האלגוריתם בודק האם מישהו מנסה להפריע לחוט שהוא רץ בו, ואם כן, האלגוריתם יוצא מהלולאה. בסוף הלולאה הוא מחזיר את החלוקה הטובה ביותר שנמצאה עד כה. כלומר, מימשנו אלגוריתם anytime.

ההפרעה עצמה מתבצעת בתוכנית הראשית: יוצרים חוט, מפעילים אותו, מחכים כמה שרוצים, ואז מפריעים לחוט.

```
t = new Thread(task);
t.start();
Thread.sleep(1000);
t.interrupt();
```

הריצו את הקטע עם זמנים שונים, ותראו איך איכות הפתרון משתפרת ככל שנותנים לחוט יותר זמן לרוץ.

תרגיל (ארוך): שנו לשנות את שרת-הרשת כך שיאפשר ללקוח לבחור מתי להפסיק את החישוב. בלקוח יהיו שני כפתורים: "Submit" (כמו עכשיו) ו"Stop". כשהלקוח לוחץ על Stop באמצע חישוב, השרת יפסיק את החישוב ויחזיר את החלוקה הטובה ביותר שנצפתה עד כה.

3. חישוב מהיר

היתרון השלישי של חישוב מקבילי הוא ייעול ביצוע חישובים כבדים ע"י שימוש בכמה ליבות. נדבר על זה בשיעור הבא.

מקורות

- Cay Horstmann, "Core Java for the Impatient", chapter 10
- Udemy, Java Multithreading Course, 3 hours, https://www.udemy.com/java-multithreading/

סיכם: אראל סגל-הלוי.