תכנות מרובה ליבות – תרגול חזרה (#14)

מועד א', 2007:

שאלה 1:

רבע מהתוכנית מקבילית. כאשר מוסיפים עוד ועוד מעבדים הרווח פר מעבד ירד. לכן סביר שבין 1-3 – תוספת של מעבד אחד תתן את הספידאפ פר מעבד הכי טוב. חישוב:

$$s = \frac{1}{1 - p + \frac{p}{n}}$$

- עבור שני מעבדים (קניית מעבד אחד) - $s_2 = \frac{1}{(1-0.25)+\frac{0.25}{2}} = 1.14$
- הוספת עוד שניים או שלושה מעבדים מורידה את הספידאפ - ... s_3

מועד ב', 2007:

שאלה 1:

XBit – שלושה מצבים, 0,1,e. כתיבה כשהוא עם e קובעת את מצבו לעד.

- פתרון קונצנזוס בינארי לאינסוף תהליכים: מנסים כתיבה ואז קוראים – כולם יחזירו אותו ערך כי רק הראשון שינה את ערכו.
- הפתרון: עזרה – כל אחד מפרסם את הביט שלו. מתחיל לכתוב ולקרוא מערך של XBits

שאלה 4:

1. המימוש הפנימי של getAndIncrement כולל לולאה עם CAS עד שמצליחים (לא למדנו, זה מ-2007). לכן זה lock free ולא wait free.

מועד ב', 2008:

שאלה 5:

ניתוח המימושים השונים של pool המוצגים. הפרמטרים עליהם נסתכל:

- Cache
- Bus
- Memory consumption
- Contention
- Latency – כמה משלמים במסלול (בלי קשר ל-contention; כמו שב-combining tree הולכים log N עד למעלה ולמטה).
- Sibgle thread
- Cache architecture

מימוש ראשון: שימוש ב-stack מבוסס lock free list.

מימוש שני: משתמשים במערך עליו מגרילים מקום ראנדומלי ומנסים לקחת את האובייקט ב-CAS.

שני המימושים האלה הם lock-free תחת הנחה שיש איברים להוציא או שיש מקום שאפשר להחזיר אליו. ההוצאה לא בטוח lock free כי צריך להמתין להחזרת אובייקט. אם מניחים פיזור ראנדומלי במימוש השני אז הוצאה היא לוק פרי.

Cache:

אם מניחים שהמערך מספיק קטן ונכנס ב-cache, המימוש השני יהיה יותר טוב כי כמות ה-miss יהיה נמוך. ניתן להתייחס עם או בלי padding. אין תשובה אחת... צריך לפרט הנחות ותחתיהן תשובות. במימוש הראשון כל פעם שמשנים את הראש יש miss.

Bus:

יש נקודה אחת שכאשר יש עליה עדכון יש storm על ה-bus – אינווליד‌ציה לכולם. במימוש השני כל אחד מחפש מקום אחר אז יש פחות עומס על ה-bus בכל שינוי. כאן שוב כדאי להתייחס להאם המערך תופס cache line יחיד או לא.

Memory:

המימוש השני תופס זיכרון קבוע (טוב? רע?).

Contention:

כשכמות החוטים הרבה יותר גדולה מהזיכרון, במימוש השני יהיה contention גבוה מאוד כמעט כמו במימוש הראשון. אם כמות החוטים קטנה מכמות האובייקטים המימוש השני טוב יותר – יהיה קל יותר לקחת ללא contention.

Latency:

אם כמות החוטים ממש קטנה מכמות האובייקטים, זה דומה לראשון. מצד שני אם כמות האובייקטים קטנה מכמות החוטים ה-latency יהיה יותר גבוה – יותר זמן עד שחוט ימצא אובייקט לקחת. המצב הפוך כשמדובר בפעולת הכנסה למערך ולא הוצאה ממנו (נרצה יותר חוטים מאובייקטים כדי שיהיה הרבה מקום במערך והכנסה תהיה מיידית – latency נמוך).

Single thread:

המימוש הראשון עדיף כי הכל מתבצע מיידית ומהר. בשני, אם יש מלא אובייקטים זה יהיה דומה לראשון. המצב הפוך למימוש השני כשמדובר בפעולת ההכנסה ולא ההוצאה.

NUMA:

אין כל כך הבדל בין המימושים. מי שהקצה את הזיכרון – הוא יושב קרוב אליו ואז גם כך כל החוטים יצטרכו לגשת אליו. הסיכוי לעבודה לוקאלית ולא remote בערך כמו למימוש הראשון. מסתבך.

מועד ב', 2009:

שאלה 1:

נתון counting-sort: טווח הערכים ידוע מראש, אלגוריתם עובד ב-O(n).

העבודה T_1: סך העבודה הוא $T_1 = 2T_1\left(\frac{n}{2}\right) + \Theta(1) = \Theta(n)$. ה-$\Theta(1)$ זה בגלל שמבצעים לולאה על מספר קבוע – count (גודל הטווח).

המסלול הקריטי T_∞: $T_\infty = T_\infty\left(\frac{n}{2}\right) + \Theta(n) = \Theta(n)$