Name : David Meriin

ID: 304472038

Name: Yossef Yakobi

ID: 301752267

# Report

* בחרנו בMaxBucketSize = 16 , לאחר שעשינו מספר ניסויים עם buckets בגדלים 1 ,2 ,4 ,8,16,32,64. וגילינו כי 16 מביא לתוצאות הטובות ביותר .
* בחרנו בשימוש Backoff lock with random queue , שכן ראינו בתרגיל הקודם שזה המימוש הטוב יותר למנעול ברוב המקרים.
* אנו מבצעים resize כאשר גודל כל bucket מגיע לגודלו המקס' מכיוון שבטבלאות גדולות הסיכוי ליפול לאותו bucket בזמן שהוא מלא ובזמן שיש resize הוא נמוך. כך מאפשרים מספר resizes קטן יחסית , ומגדילים את הטבלה רק כשצריך באמת.

נציג עבור כל פונקציה בכל מימוש האם היא מקיימת wait-free/lock-free/Deadlock-free/Startvation-free.

**Locking -**

* add -קיים כאן blocking, ולכן Deadlock - free, נועל מנעול אחד. מי שמגיע ראשון לוקח , אין הבטחה להוגנות בין מנעולים.
* remove - קיים כאן blocking, ולכן Deadlock - free, נועל מנעול אחד. מי שמגיע ראשון לוקח , אין הבטחה להוגנות בין מנעולים.
* contains - קיים כאן blocking, ולכן Deadlock - free, נועל מנעול אחד. מי שמגיע ראשון לוקח , אין הבטחה להוגנות בין מנעולים.
* resize - קיים כאן blocking, ולכן Deadlock - free, נועל את כל המנעולים . אך מכיוון שנועל את המנעולים לפי הסדר, פשוט יחכה במידה ומנעול נעול וימשיך לנעול כאשר מנעול זה ישתחרר.

**Optimistic -**

* add -קיים כאן blocking, ולכן Deadlock - free, נועל מנעול אחד. מי שמגיע ראשון לוקח , אין הבטחה להוגנות בין מנעולים.
* remove - קיים כאן blocking, ולכן Deadlock - free, נועל מנעול אחד. מי שמגיע ראשון לוקח , אין הבטחה להוגנות בין מנעולים.
* contains - wait-free. מכיוון שresize מעדכן את table, בפעולה האחרונה שלו . וcontains ניגש לtable בפקודה הראשונה שלו כל מצב חפיפה בין 2 הפעולות הוא לינאריזיבלי. הפונ' לא נועלת כלל ולכן אין כאן blocking. במידה וcontains חופף לadd או remove , גם תמיד ניתן יהיה למצוא נקודת לינאריזיציה (אם רץ במקביל למחיקה או הוספה כל תשובה של קיים או לא תתקבל). אין כאן צורך בנעילה מחדש במקרה של כשלון , כי אין נקודה שהיא לא לינאריזיבלית.
* resize - קיים כאן blocking, ולכן Deadlock - free, נועל את כל המנעולים . אך מכיוון שנועל את המנעולים לפי הסדר, פשוט יחכה במידה ומנעול נעול וימשיך לנעול כאשר מנעול זה ישתחרר.

**Lock Free -**

* add -
* remove
* contains
* resize

**Linear Probe -**

* add - קיים כאן blocking, ולכן Deadlock - free, נועל שני מנעולים. מי שמגיע ראשון לוקח , אין הבטחה להוגנות בין מנעולים. מכיוון שבנעילת המנעול השני , הפונ' תוותר במידה והמנעול השני נעול והindex של המנעול השני קטן מהמנעול הראשון . כל זאת בכדי לשמר את דרך הפעולה כי קודם נועלים את המנעול בעל הindex הקטן יותר , כמו שמתבצע בפעולת resize.
* remove- קיים כאן blocking, ולכן Deadlock - free, נועל שני מנעולים. מי שמגיע ראשון לוקח , אין הבטחה להוגנות בין מנעולים. מכיוון שבנעילת המנעול השני , הפונ' תוותר במידה והמנעול השני נעול והindex של המנעול השני קטן מהמנעול הראשון . כל זאת בכדי לשמר את דרך הפעולה כי קודם נועלים את המנעול בעל הindex הקטן יותר , כמו שמתבצע בפעולת resize.
* contains - wait-free. מכיוון שresize מעדכן את table, בפעולה האחרונה שלו . וcontains ניגש לtable בפקודה הראשונה שלו כל מצב חפיפה בין 2 הפעולות הוא לינאריזיבלי. הפונ' לא נועלת כלל ולכן אין כאן blocking. במידה וcontains חופף לadd או remove , גם תמיד ניתן יהיה למצוא נקודת לינאריזיציה (אם רץ במקביל למחיקה או הוספה כל תשובה של קיים או לא תתקבל)
* resize- קיים כאן blocking, ולכן Deadlock - free, נועל את כל המנעולים . אך מכיוון שנועל את המנעולים לפי הסדר, פשוט יחכה במידה ומנעול נעול וימשיך לנעול כאשר מנעול זה ישתחרר. מכיוון שadd וremove מכריחים נעילה לפי הסדר ( במידה והמנעול השני נעול), לא יהיה כאן deadlock.

1. קיבלנו עבור ParallelNoLoad ב8 חוטים - 1399 pkts/ms. בתרגיל התכנותי הראשון קיבלנו עבור 10 חוטים קצב של כ 1000 pkts/ms . הסיבה העיקרית לקצב כה נמוך בתרגיל הראשון הייתה שנזרקות הרבה שגיאות מכיוון שהתורים ריקים וכך השגיאות מאטות את התכנית . כעת, כל חוט יכול לגשת לכל התורים ( בתרגיל הראשון לכל חוט היה תור משלו). פיזור רנדומלי של בחירת התור יכולה לגרום להתנגשויות כך שחוט מסויים יחכה במנעול בזמן שאחר מוציא איבר ממנו. המתנה זו יכולה דווקא לתרום במקרה זה כי יכולות להגרם פחות שגיאות של תורים ריקים כאשר כל אחד מוציא רק מהתור שלו.
2. קיבלנו את התוצאות :

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | Serial | Locking | Optimistic | LockFree | LinearProbe |
| Mostly Reads | | 213 | 190 | 190 | 189 | 197 |
| Heavy Writes | | 213 | 186 | 187 | 180 | 182 |
| Mostly Reads ratio | |  | 0.892019 | 0.892019 | 0.887324 | 0.924883 |
| Heavy Writes ratio | |  | 0.873239 | 0.887324 | 0.84507 | 0.85446 |

אנו רואים כי עבור חוט 1 , optimistic ו Locking מראים ביצועים דומים שכן ההבדל היחידי הוא ש optimistic לא נועל. עבור חוט 1 לא תהיה המתנה למנעול שכבר נעול , ולכן ההבדל לא מורגש. ישנה אולי האטה קלה בנעילה עצמה של המנעול ושחרורו ולכן optimistic יציג לעיתים ביצועים טובים יותר ( כמו שרואים בHeavyWrites , שלoptimistic יש קצב טוב יותר). בשני המקרים ישנה ירידה בתעבורה כאשר אנו בHeavyWrites. זאת מכיוון שיותר כתיבות גורמות ליותר resizes , מה שגורם לנעילת יותר מנעולים ושחרורם, שזו בעצם התקורה היתרה לחוט אחד בהשוואה למימוש serial.

LockFree בשני המקרים בעל התעבורה הנמוכה ביותר , כי הוא בעל הלוגיקה המורכבת ביותר, שהיא מיותרת עבור חוט אחד כמובן. כלומר מספר הפעולות הנוספות לעומת המימוש Serial הוא הגדול ביותר כאן.

LinearProbe בעל הביצועים הטובים ביותר עבור קריאות מרובות. במימוש זה ניגשים למערך ורצים עליו במקום לרוץ על רשימה מקושרת כמו בשאר המימושים. cache locality תורם כאן וניתן להניח כי יש במימוש זה פחות cache misses , במיוחד בחוט אחד כאשר אין חוטים אחרים שמשנים את הטבלה. בHeavyWrites , אנו רואים כי הוא לא המימוש הטוב ביותר . זאת מכיוון שפעולות resize כבדות במימוש זה שכן הם מעתיקות את כל הטבלה מחדש ומכפילות את גודלה. הקצאת הזכרון כאן היה יותר גדולה ממימושים אחרים, שכן במימוש זה אין פוינטרים לbuckets אלא כל הbuckets מוכלים במערך. בבניית המערך מגודל 2 יהיו יותר resizes ממימושים אחרים. לדוגמה במימוש locking במערך בגודל 2 יהיו 2 תאים כאשר כל תא מצביע לBucket בגודל 16. אבל עבור LinearProbe יהיו סך הכל 2 תאים בהתחלה כך שברור שהוא יעשה resize הרבה יותר מהר מאשר המימוש Locking.

3. קיבלנו את התוצאות :

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Mostly Reads | |  |  |  |  |  |
|  | Num Threads | Serial | Locking | Optimistic | LockFree | LinearProbe |
| p=0.5 | 1 | 211 | 171 | 181 | 193 | 202 |
| p=0.75 | 1 | 218 | 191 | 188 | 200 | 194 |
| p=0.99 | 1 | 219 | 185 | 201 | 206 | 194 |
| p=0.5 | 4 |  | 556 | 581 | 590 | 578 |
| p=0.75 | 4 |  | 554 | 602 | 602 | 585 |
| p=0.99 | 4 |  | 565 | 578 | 578 | 598 |
| p=0.5 | 10 |  | 950 | 1300 | 1184 | 1299 |
| p=0.75 | 10 |  | 1230 | 1391 | 1379 | 1364 |
| p=0.99 | 10 |  | 1182 | 1228 | 1238 | 1305 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| **Ratio** |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| p=0.5 | 1 |  | 0.810427 | 0.85782 | 0.914692 | 0.957346 |
| p=0.75 | 1 |  | 0.876147 | 0.862385 | 0.917431 | 0.889908 |
| p=0.99 | 1 |  | 0.844749 | 0.917808 | 0.940639 | 0.885845 |
| p=0.5 | 4 |  | 2.635071 | 2.753555 | 2.796209 | 2.739336 |
| p=0.75 | 4 |  | 2.541284 | 2.761468 | 2.761468 | 2.683486 |
| p=0.99 | 4 |  | 2.579909 | 2.639269 | 2.639269 | 2.730594 |
| p=0.5 | 10 |  | 4.50237 | 6.161137 | 5.611374 | 6.156398 |
| p=0.75 | 10 |  | 5.642202 | 6.380734 | 6.325688 | 6.256881 |
| p=0.99 | 10 |  | 5.39726 | 5.607306 | 5.652968 | 5.958904 |

נציג עבור כל קונפיגורציה טבלה משל עצמה.

**Mostly Reads:**

**p = 0.5**

**p=0.75**

**p=0.99**

אנו רואים כי המימוש Locking תמיד מראה את הביצועים הנמוכים ביותר , ובאופן משמעותי כאשר הhit rate הוא 0.5. זאת מכיוון שבמימוש זה בפונקצית contains ישנה נעילת read. נעילת read נועלת את כל הwriters. ובזמן שמתקיים חיפוש של איבר לא ניתן לכתוב אל הbucket עליו מתבצע החיפוש ( וגם לא ניתן לעשות resize) . כאשר הhit rate הוא 0.5, נקבל את הביצועים הגרועים ביותר מכיוון שהנעילה מתבצעת לאורך יותר זמן , שכן צריך לעבור על כל הbucket עד שיודעים באופן חד משמעי כי האיבר אינו נמצא שם. נעילה לאורך זמן גורמת לכך שפחות פעולות יכולות להתבצע על bucket זה.

לעומת המימוש Locking , נסתכל על optimistic. מימוש זה שונה מlocking רק בכך שהוא אינו נועל כאשר הוא מחפש איבר. בפעולת החיפוש איננו נועלים , וכן בריצות אלו רוב הפעולות הן פעולות קריאה. כלומר הרבה מן הפעולות מתבצעות במקביליות אמיתית ללא נעילה. וכן רואים כי יחד עם linear probe הוא מראה את הביצועים הטובים ביותר עבור p=0.5, p =0.75.

Linear probe - מימוש זה בדומה לoptimistic אינו נועל בקריאה ולכן ניתן לראות ביצועים טובים. הסיבה שזהו גם המימוש בעל הביצועים הטובים ביותר כאשר p=0.99 היא מכיוון שהוא בנוי בצורת מערך ללא פוינטרים לbuckets, ישנו cache locality. משמע אם איבר קודם חופש בbucket כלשהו ונמצא (בסבירות גבוהה), פעם הבאה שיחפשו איבר אחר בbucket זה לא תהיה גישה נוספת לזכרון מכיוון שהאיבר החדש כבר נטען עם האיבר הישן לcache. יתרון זה לא קיים אצל מימושים אחרים שכן הם עובדים עם מנגנון chaining שלא מבטיח קרבה בין איברים. מכיוון שבp=0.99 כמעט תמיד האיבר ימצא במבנה הנתונים , ה cache locality תורם רבות גם לקריאות הבאות.

Lock Free - אנו רואים כי הביצועים של מימוש זה פחות טובים מoptimistic,LockFree כאשר p=0.5. זה נובע מכך שבזמן חיפוש אם הbucket המיועד לא קיים , יש ליצור את הbucket (ואולי ליצור עוד buckets כתוצאה מכך). פעולה זו גוזלת זמן ויכולה לקרות במיוחד כאשר איבר לא קיים , כי ישנה סבירות כי הbucket אליו הוא משויך לא קיים גם כן. אנו רואים שעבור p=0.75,p=0.99 ביצועי המימוש דומים לoptimistic , שכן בhit rate גבוה אין צורך ליצור הרבה buckets וגם כמו לoptimistic אין כאן את היתרון של ה cache localityשל linear probe ולכן ביצועי המימוש לא משתווים לאלו של linear probe כאשר p=0.99..

אנו רואים כי עבור 10 חוטים דווקא כאשר p=0.75 ישנו את הspeedup הגבוה ביותר ביחס לserial. ניתן להסביר את תופעה זו ע"י כך שעצם העובדה שישנו hitrate גבוה יותר מ0.75

( p =0.99), יכולה לגרום להאטה קטנה של כל המערכת. זאת מכיוון שהפעולות חיפוש מסתיימות אכן יותר מהר (אחוזי פגיעה גבוהים יותר), זה גורם לכך שישנם יותר הוצאות מהתורים, כלומר יותר "מאבקים" על מנעולים ובזבוז זמן בלחכות למנעול שיתפנה. בנוסף, ישנו סיכוי גבוה יותר לקבל emptyException במשיכה מתור, שכן הworkers עובדים מהר ויש הרבה מהם , והdispatcher אולי לא מספיק להזין אותם, לכן התורים מתרוקנים מהר יותר וקופצות שגיאות ( שנתפסות ע"י catch שידוע שפעולה זו היא פעולה כבדה).