Operating Systems – 234123

Homework Exercise 3 – Dry

Name: Nadav Orzech

ID: 311549455

Email: nadav.or@campus.technion.ac.il

Name: Roni Englender

ID: 312168354

Email: roni.en@campus.technion.ac.il

חלק ראשון

שאלה 1

א. התכונות שהקטע קריטי מפר במימוש:
Progress (התקדמות) – במידה וחוט נמצא בתוך הקטע הקריטי, לאחר נעילת המנעול, ויתרחש מצב של נפילת חוט טרם שיחרר את המנעול, המנעול ישחרר נעול ללא אפשרות להפתח ושאר החוטים שרוצים לבצע את הקטע הקריטי לעולם לא יצליחו להכנס. משמע, קיבלנו deadlock.
Fairness (הוגנות) – אין התייחסות לאיזה חוט מהחוטים הממתינים למנעול ייכנס הבא בתור, כלומר אין סדר ברור וידוע לכניסת החוטים למנעול וגם אין חסם למספר הפעמים שחוטים אחרים יכולים להיכנס למנעול לפני החוט הנוכחי. לכן במידה ויש מספר רב של חוטים שמחכים להיכנס, ייתכן וייווצר מצב של הרעבה לחוטים מסוימים.

ב. בעיית הPerformance הקיימת במימוש הנ"ל היא שהמנעול מבצע לולאה במדיניות busy-wait, דבר הלוקח זמן חישוב של המעבד, ומכיוון שנתון בשאלה כי הקוד רץ על ליבה אחת אנחנו נתקע כל פעם עד שהחוט הממתין יסיים את הקוונטום שלו. מכיוון שנתון כי קטע הקוד ארוך וכבד חישובית, ידרשו מספר רב של החלפות הקשר, ולכן החוט המבצע ימתין זמן רב למעבד. לעומת זאת, אם קטע הקוד יהיה קצר ומהיר, ידרשו פחות החלפות הקשר, ולכן הבעיה תהיה פחות משמעותית.

שאלה 2

התכונה שמופרת במימוש הנתון, היא **מניעה הדדית** ניקח לדוגמה תרחיש שבו נכנס חוט ראשון, נועל את המנעול וממשיך לקטע הקריטי, ולאחר מכן חוטים רבים מנסים להיכנס גם כן לקטע הקריטי, וייתכן שנגיע למצב של פתיחת הנעילה בעקבות הגעת המונה ל-MAX_ITER, טרם סיום ריצת הקטע הקריטי של החוט הראשון. כלומר ייתכן מצב בו יהיה יותר מחוט אחד בקטע הקריטי.

שאלה 3

הקוד הנתון ידפיס ערכים בין 10 ל-55

הסבר: הקוד ידפיס את ערך ה- sum רק לאחר שהוא יבצע join לחוט העשירי. במידה ואין סדר בכניסת החוטים לפי ה- scheduler ייתכן מצב שבו החוט העשירי יכנס לקטע הקריטי ראשון, יעדכן את ערך ה-sum ומיד לאחר הקטע הקריטי ייצא לפעולה join. במצב זה רק הוא עדכן את ערך ה-sum לפני ביצוע ההדפסה ולכן יודפס המספר 10. בכל מצב אחר, הסכום שיודפס בסיום הוא סכום מספרי החוטים שהספיקו להיכנס עד שחוט מספר 10 סיים את ריצתו. לכן, במצב בו החוטים נכנסים בסדר הנכון לפי מספרי החוט, נקבל סכום מקסימלי שהוא 55. בכל מצב אחר נקבל סכום שהוא טווח בין 10 ל-55 כתלות בכמות ובאילו חוטים סיימו את ריצתם.

שאלה 4

הקוד הנתון ידפיס ערכים בין 2 ל-200

הסבר: את הערך המקסימלי נוכל לקבל במצב בו 2 החוטים לא מפריעים אחד לשני לעדכן את ערך ה-result וכיוון שכל אחד מבצע את הפעולה 100 פעמים נקבל את הערך 200.

נסביר את התרחיש בו נוכל לקבל 2 –

- חוט 1 מחשב באיטרציה הראשונה result+1 כאשר תוצאת החישוב היא 1
- לפני שחוט 1 מספיק לבצע את ההשמה ל-result מתבצע החלפת הקשר לחוט 2

- חוט 2 מבצע 99 איטרציות של סכימה והשמה כאשר בסיומן result = 99
- מתבצעת החלפת הקשר שוב וחוט 1 מסיים את ההשמה הקודמת ומעדכנת את result ל-1
- מתבצעת שוב החלפת הקשר וכעת חוט 2 מבצע את האיטרציה האחרונה בלולאה ומחשב result+1 כאשר תוצאת החישוב כעת היא 2
- לפני שחוט 2 מספיק לבצע את ההשמה ל-result מתבצע החלפת הקשר לחוט 1
- חוט 1 מבצע את כל האיטרציות הנותרות ומעדכן את ערך ה-result
- מתבצעת שוב החלפת הקשר וכעת חוט 2 מבצע השמה של הערך 2 שחישב לפני כן ומסיים גם את ריצתו
- 2 החוטים סיימו את ריצתם ומודפס הערך 2

<u>שאלה 5</u>

בקוד הנתון אין צורך להגן בעזרת משתנה סנכרון על sum מכיוון שבשאלה זו מדובר בתהליכים שונים ולא חוטים כמו בסעיפים הקודמים. לתהליכים שונים יש מחסניות שונות וכתובות שונות בזיכרון בשונה מחוטים שחולקים את אותו מרחב זיכרון. ולכן על אף ש-sum הוא משתנה גלובלי אין צורך להגן על כתיבה או קריאה ממנו.

חלק שני

סעיף א':

```
typedef struct mutex {
    singlephore h;
} mutex;

//init lock value to 0 for an open lock
void mutex_init(mutex* m) {
    singlephore_init(&m->h);
}

//first thread will skip the while loop and decrease the value
//the next thread will wait until the value increases again (in unlock)
void mutex_lock(mutex* m) {
    H(&(m->h),0,-1);
}

//the thread will skip the while loop and increas back the value
void mutex_unlock(mutex* m) {
    H(&(m->h),-1,1);
}
```

סעיף ב' - בונוס:

```
typdef struct condvar{
    mutex m;
    singlephore h;
    int num_of_extra_signals;
    int num_of_waiting;

public:
    void cond_init(condvar* c){
        singlephore_init(&c->h);
        c->num_of_extra_signals=0;
        c->num_of_waiting=0;
    }
    void cond_signal(condvar* c){
        H(&c->h, INT_MIN, 1);

        mutex_lock(&c->m);
        if(c->num_of_waiting==0)
            c->num_of_extra_signals++;
        mutex_unlock(&c->m);
    }
    void cond_wait(condvar* c, mutex* m){
        mutex_unlock(&m);

        mutex_lock(&c->m);
        c->num_of_waiting++;
        mutex_unlock(&c->m);

        if(c->num_of_extra_signals>0)
            H(&c->h, INT_MIN, -(c->num_of_extra_signals));
        H(&c->h, 0, -1);

        mutex_lock(&c->m);
        c->num_of_waiting--;
        c->num_of_extra_signals=0;
        mutex_unlock(&c->m);

        mutex_lock(&m);
    }
};
```

סעיף ג':

הפתרון שגוי כיוון שקריאות "מיותרות" ל- cond_signal יכולות לפגוע בתקינות המשתנה תנאי, כלומר קריאות נצברות של signal יגרמו לכך שה-wait לא יגרום לנעילה. נראה תרחיש בו הפתרון שגוי:

נקרא ל-cond_signal פעמיים למשל ללא חוטים ממתינים בcond_wait, במצב כזה הגדלנו את ה-value ב-2 לכן התנאי ב-cond_wait לא יתקיים ויוכלו 2 חוטים להיכנס לקטע הקריטי, דבר הפוגע ב- mutual exclusion.

חלק שלישי

שאלה 1

א. לפי הערכתנו המנגנון היה אמור לפעול בצורה הבאה:

- בכל דור במשחק היצרן טוען n משימות ל-PCQueue כשבכל טעינת משימה הוא מגדיל את הערך working ב-1 ולאחר מכן עובר להמתין שכל המשימות יבוצעו על די הצרכנים.
- כל עוד קיימות משימות הצרכנים מבצעים אותן ובסיום כל משימה הצרכן מוריד את ערך ה-working ב-1.
- היצרן יודע שהמשימות הסתיימו כאשר ה- working חוזר להיות 0.
- בסיום כל דור הלוחות curr ו- next מוחלפים ועוברים לדור הבא.

במנגנון זה לעומת ה-Semaphore ההמתנה לא מתבצעת בשלב הקטנת המונה, אלא על ידי פונקציה אחרת (wait) בצורה יזומה. בצורה כזו נוכל להגדיל את מונה החוטים העובדים ללא הגבלה וללא נעילה של קטע הקוד הקריטי. לעומת זאת ב-Semaphore בדיקת החסם מתבצעת כבר בשלב הקטנת המונה, לכן אם הגענו לחסם המוגדר החוט ייכנס ישירות לתור ההמתנה ולא ימשיך לקטע הקריטי. בנוסף, ההמתנה במנגנון הנתון מתבצעת על ידי המתנה פעילה של busy-wait ולא על ידי תור המתנה כמו ב-Semaphore.

ב. בעיה של Race Condition – כיוון שעדכון ערך ה-working לא מתבצע באופן אטומי או תחת נעילה, עלולה להיווצר תחרות בין החוטים בפרט בין היצרן לצרכן. ייתכן מצב בו יצרן מכניס משימה ובמקביל צרכן מוציא משימה, לכן שינוי ערך המונה working לא קבוע, הערך הסופי של המונה עלול להתנהג בצורה לא צפויה.
בעיה של Mutual Exclusion – בהמשך לבעיה הנ"ל שתיארנו, ייתכן מצב שערך ה- working יעודכן ל-0 טרם בוצעו כל המשימות, במצב כזה היצרן יעבור לבצע החלפת לוחות, כאשר עדיין יש חוט צרכן המבצע את המשימה שלו בחוט הנוכחי.
תרחישי Deadlock – בהמשך לבעיית ה-RC שתיארנו לעיל, ראינו כי המונה working יכול לקבל ערך השונה מ-0. נדגים 2 תרחישים אפשריים עבור working=1 ועבור working=-1

1. Working = 1: במנגנון הנתון ייתכן מצב בו בעקבות החלפת הקשר ביצוע ה-decrease לאחת המשימות לא יתעדכן כראוי. במצב כזה בסיום ביצוע המשימות יתקיים מצב בו working=1, ובקריאה ל-wait נתקע בלולאה אין סופית שהיא למעשה ה-deadlock.
2. Working = -1: באותו אופן כמו בתרחיש הקודם, בעקבות החלפת הקשר יתקיים מצב בו ה-increase לא יתבצע כראוי, בקריאה ל- wait יתקיים כי working=-1 ונתקע שוב בלולאה אין סופית – מצב deadlock.

ג. 1.

```
class Barricade{
private:
        int working;
        pthread_mutex_t mutex;

public:
        Barricade() : working(0) {
            pthread_mutex_init(&mutex,NULL);
        }
        void increase(){
        //updating working value is executed inside lock
            pthread_mutex_lock(&mutex);
            working++;
            pthread_mutex_unlock(&mutex);
        }
        void decrease(){
        //updating working value is executed inside lock
            pthread_mutex_lock(&mutex);
            working--;
            pthread_mutex_unlock(&mutex);
        }
        void wait(){
            while(working){}
        }
};
```

2. הפתרון שהצענו של הוספת מנעול mutex גרמה לבעיית performance, כעת כל פעולה של increase/decrease דורשת נעילה של המנעול ובמצב בו N – מספר החוטים האפקטיבי גדול מאוד נגיע לפרקי זמן המתנה ארוכים בהמתנה לשחרור המנעול.

3.

```
class Barricade{
private:
        int working;
        pthread_mutex_t mutex;

public:
        Barricade() : working(0) {
            pthread_mutex_init(&mutex,NULL);
        }
        void increase(){
        //updating working value is executed atomicly
            atomicAdd(&working,1);
        }
        void decrease(){
        //updating working value is executed atomicly
            atomicAdd(&working,-1);
        }
        void wait(){
            while(CAS(&working, 0, 0)){}
        }
};
```

4. הפתרון ב-(3) עדיף על הפתרון ב-(1) כיוון שבשני הפתרונות יש עדכון של ערך הworking שמתבצע באופן אטומי (בין אם במנעול או ע"י פעולה אטומית), אך הפתרון ב-(3) עדיף מבחינת performance, כיוון שפעולה אטומית מתבצעת יותר מהר מנעילה של מנעול ושחרורו. בנוסף, בפתרון (3) אין המתנה של חוטים אחרים לשחרור המנעול, המתנה שגוזלת זמן מעבד.

ד. בונוס:

```
producer
1. Init PCQueue p, c
2. Init fields curr, next
3. for t=0 -> t=n_generations
        for i=0 -> i=N
            p.push(job);

        for i=0 -> i=N          //this replaced the busy-wait loop
            c.pop();            //pops a finished job

        //if we reached this point then all threads finished their jobs
        swap(curr, next);



Consumer(one of N)
while(1)
    job j = p.pop();            //blocked here if queue is empty
    execute j;
    c.push(job_finished);       //anounces that one job is finished
```

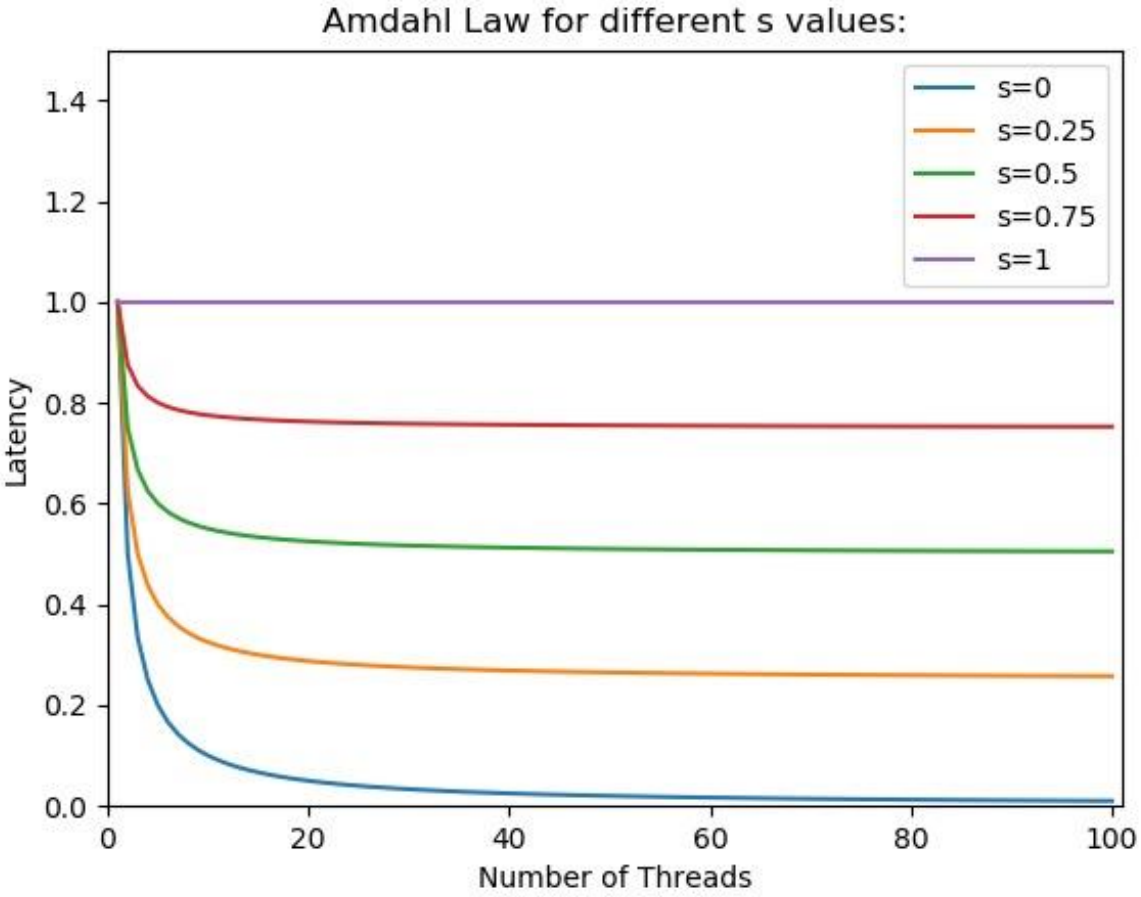
מנגנון הסנכרון הנ"ל מנצל 2 PCQueues לטובת הסנכרון ההדדי בין יצרן לצרכן כפי שנדרש. התור הראשון – p משמש להכנסת משימות על ידי היצרן, ומצד שני הוצאתן וביצוען על ידי הצרכן. התור השני – c משמש כאינדיקציה ליצרן מתי כל המשימות של הדור הנוכחי הסתיימו ומאפשר לו להחליף את הלוחות ולהתקדם לדור הבא. בסיום ביצוע משימה כל צרכן מכניס ערך המסמן כי סיים את המשימה לתור c, ורק כאשר היצרן הצליח להוציא כמות ערכים שכאלה שזהה לכמות המשימות שהכניס (על ידי לולאת ה- for השנייה שמבצעת אותו מספר איטרציות) הוא יוכל להמשיך להתקדם בביצוע האלגוריתם.

להלן פתרון מתוקן:

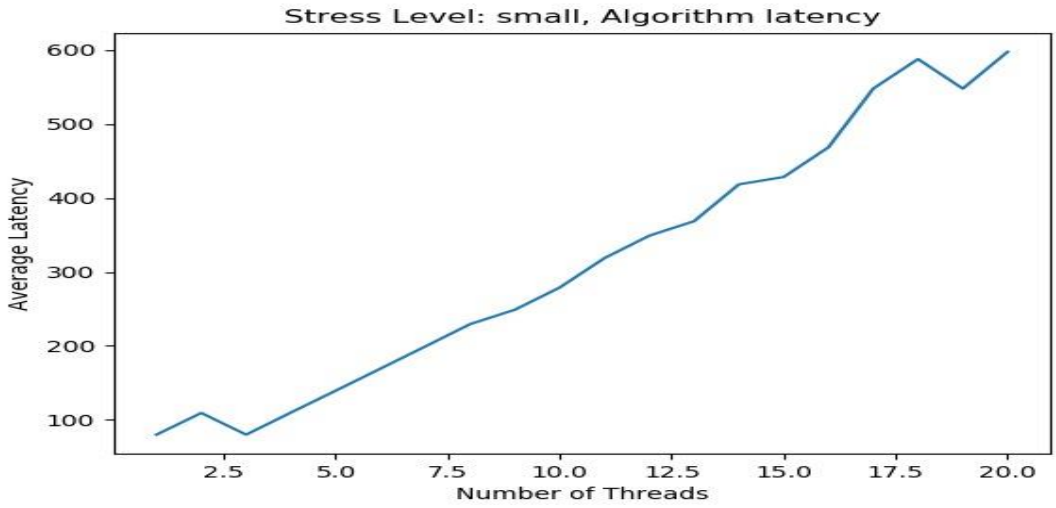
```
producer
1. Init PCQueue p
2. Init condvar cond
3. Init mutex m
4. Init fields curr, next, job_finished
5. for t=0 -> t=n_generations
        for i=0 -> i=N
            p.push(job);

        mutex_lock(&m);
        while(job_finished<N)   //while not all jobs executed
            cond_wait(&cond,&m);
        job_finished=0;         //reset for next gen
        mutex_unlock(&m);

        //if we reached this point then all threads finished their jobs
        swap(curr, next);



Consumer(one of N)
while(1)
    job j = p.pop();            //blocked here if queue is empty
    execute j;

    mutex_lock(&m);
    job_finished++;             //anounces that one job is finished
    cond_signal(&cond);         //and signals the producer to check if gen is finished
    mutex_unlock(&m);
```

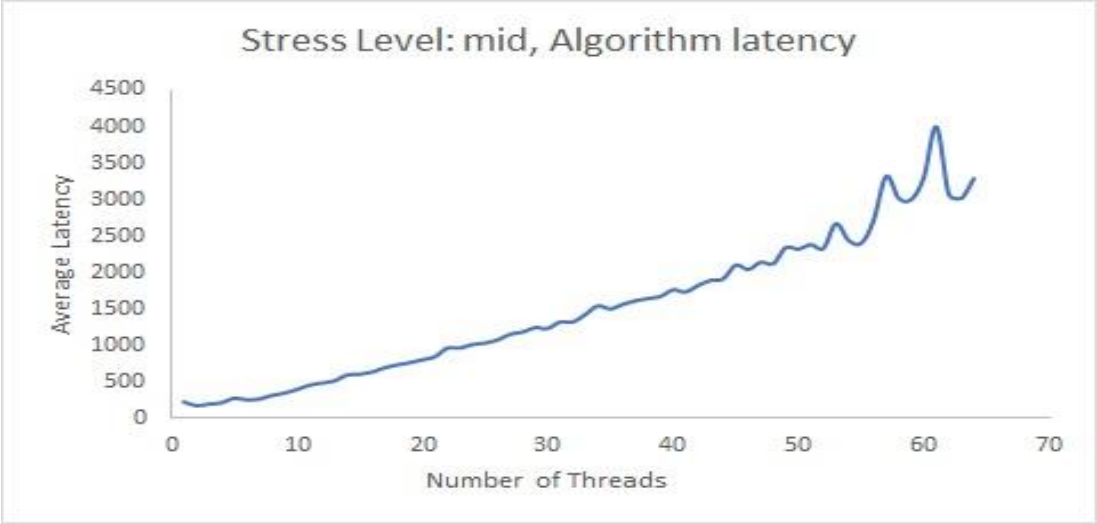
שאלה 2

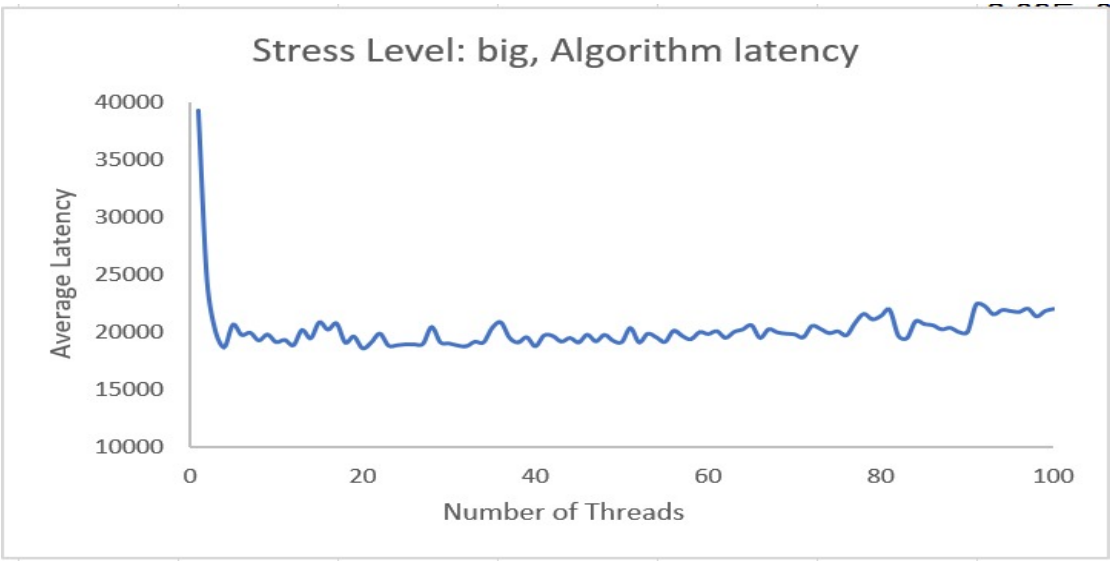
סעיפים (א) + (ב):



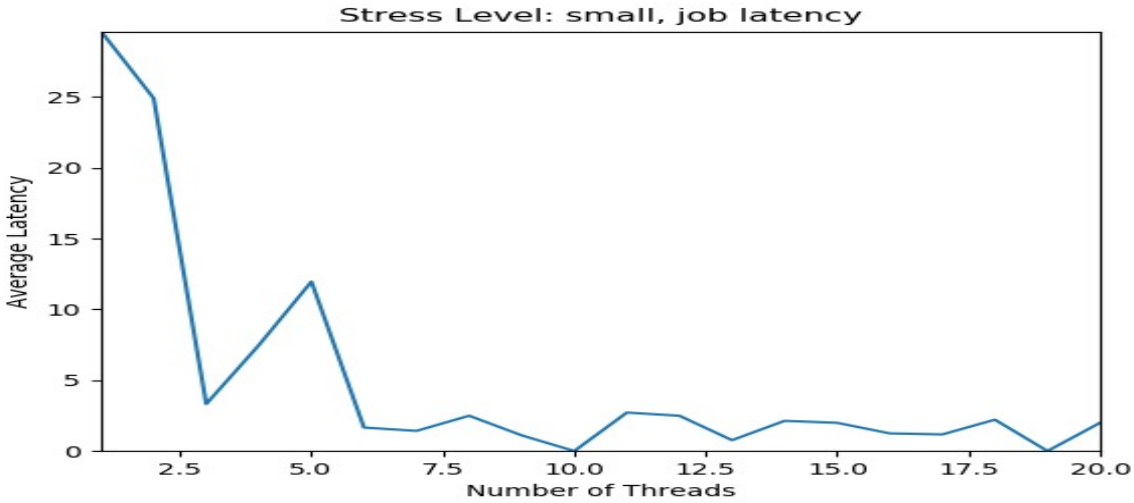
ג. (i)

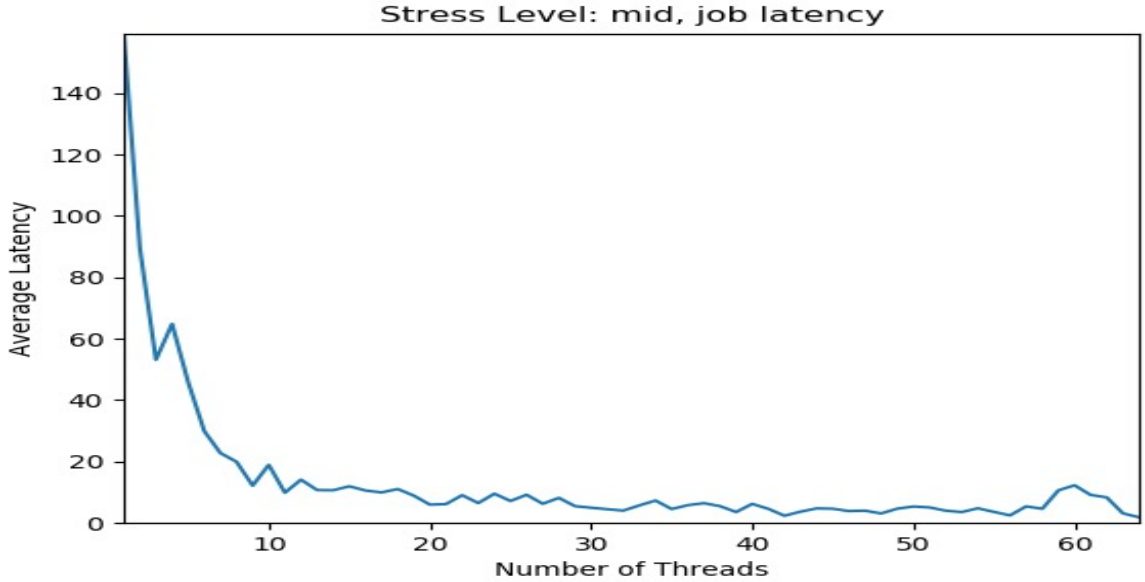

Stress Level: small, Algorithm latency
Average Latency
600
500
400
300
200
100
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Number of Threads

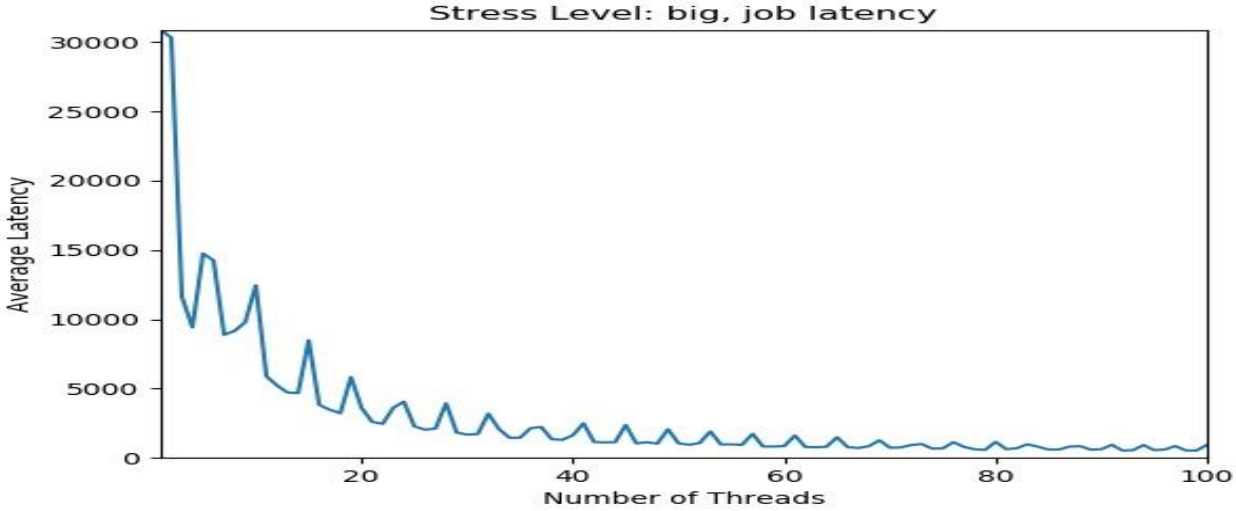

Stress Level: mid, Algorithm latency
Average Latency
4500
4000
3500
3000
2500
2000
1500
1000
500
0
0
10
20
30
40
50
60
70
Number of Threads


Stress Level: big, Algorithm latency
Average Latency
40000
35000
30000
25000
20000
15000
10000
0
20
40
60
80
100
Number of Threads

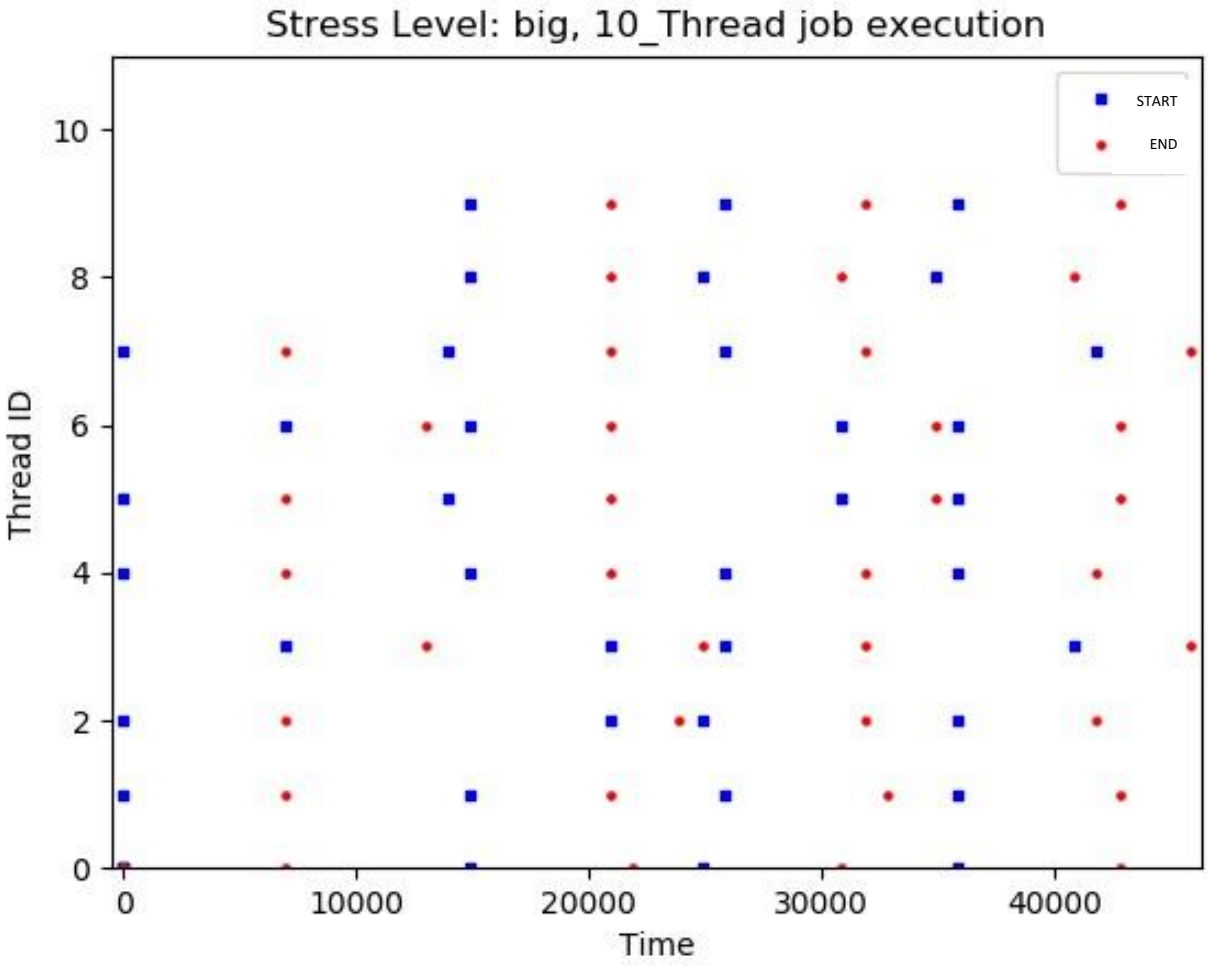
(ii)


Stress Level: small, job latency
Average Latency
Number of Threads
0
5
10
15
20
25
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0


Stress Level: mid, job latency
Average Latency
Number of Threads
0
20
40
60
80
100
120
140
10
20
30
40
50
60


Stress Level: big, job latency
Average Latency
Number of Threads
0
5000
10000
15000
20000
25000
30000
20
40
60
80
100

(iii)


Stress Level: big, 10_Thread job execution
START
END
Thread ID
Time
0
2
4
6
8
10
0
10000
20000
30000
40000

ד. <u>ניתוח הגרפים:</u>

הלוח הקטן (small):
<u>Latency(A)</u> – ניתן לראות כי עבור לוח זה כמות החוטים המיטבית לחישוב אלגוריתם A בזמן אופטימלי היא בין 1 ל-3. נשים לב שכאשר מגדילים את מספר החוטים מעבר ל-3 זמן החישוב עולה במגמה חדה ולינארית ביחס לכמות החוטים. ניתן להסביר את המגמה הזאת כי עבור לוח קטן עלות ההחלפת הקשר בין החוטים השונים גדולה יותר משמעותית מעלות חישוב העבודה בהיקף קטן. לכן יתבצעו הרבה החלפות הקשר כאשר כל חוט מבצע חישוב קטן מאוד. יתכן גם כי עם כמות חוטים גדולה מתבצעות יותר המתנות על מנעולים מה שגורם לעלייה בLatency.
<u>Latency(j)</u> – כאן, בשונה מ-Latency(A) ככל שכמות החוטים גדלה, ה-Latency(j) יורד. ניתן להסביר זאת כי שככל שכמות החוטים עולה כך כמות השורות שכל חוט מטפל בה יורדת, ולכן זמן החישוב של כל חוט יורד כפי שניתן לראות בגרף.
לפי הניתוח הנ"ל ניתן להסיק כי **מספר החוטים האופטימלי עבור הלוח הקטן הוא 1 עד 3.**

הלוח הבינוני (mid):
<u>Latency(A)</u> – ניתן לראות כי גם כאן, בדומה ללוח הקטן, כאשר מגדילים את מספר החוטים מעבר ל-6 זמן החישוב של האלגוריתם עולה באופן לינארי ביחס לכמות החוטים. הסיבה לעלייה היא אותה סיבה שגורמת לעליה בלוח הקטן שכבר תיארנו, אלא שבלוח זה מכיוון שהוא גדול יותר נדרשים יותר חוטים בשביל שעלות ההחלפות הקשר תעלה על עלות החישוב. לכן העלייה מתחילה רק מעבר לחוט ה-6.
<u>Latency(j)</u> – באותו אופן כמו בלוח הקטן, גם כאן ה-Latency(j) יורד ככל שכמות החוטים עולה, וההסבר זהה.
לפי הניתוח הנ"ל ניתן להסיק כי **מספר החוטים האופטימלי עבור הלוח הקטן הוא 1 עד 6.**

הלוח הגדול (big):
<u>Latency(A)</u> – בלוח הגדול, יש ירידה חדה בזמני הlatency מחוט 1 עד החוט ה5. כאשר יש כמות חוטים קטנה, כל חוט מקבל מספר גדול של שורות לכן זמן החישוב שלו ארוך יותר וזמן החישוב הכולל יותר ארוך. מכמות חוטים של 5 ומעלה, זמן הlatency יחסית יציב ואינו משתנה כאשר כמות החוטים ממשיכה לעלות. למרות שככל שכמות החוטים עולה וכל חוט מקבל tile יותר קטן לחישוב וזמן החישוב יורד, הזמן הכולל של אלגוריתם A נשאר יציב כתוצאה מהזמן שלוקח לבצע החלפות הקשר והמתנה על מנעולים, ולכן הLatency הכולל נשאר יציב למרות כמויות החוטים השונות.
<u>Latency(j)</u> – באותו אופן כמו בלוח הקטן, גם כאן ה-Latency(j) יורד ככל שכמות החוטים עולה, וההסבר זהה. נשים לב כי כאן קיימות קפיצות קטנות בגרף לאורך המגמה. ניתן להסביר זאת כי חלוקת השורות משתנה כתלות במספר החוטים כאשר החוט האחרון מקבל את שארית החלוקה. לכן נסיק כי עבור מספרי חוטים שמשאירים שארית חלוקה בכמות השורות גדולה יותר נקבל ממוצע של ה-Latency איטי יותר במעט.
לפי הניתוח הנ"ל ניתן להסיק כי **מספר החוטים האופטימלי עבור הלוח הקטן הוא 5 ומעלה.**

נשים לב שיש שוני מהותי בחישוב הLatency(A) בין הלוח הקטן והבינוני לעומת הלוח הגדול. כפי שהסברנו, בלוחות הקטנים כאשר עומס החישוב קטן יותר אין יתרון גדול למקביליות החישוב, ולכן ככל שעולה מספר החוטים, הlatency עולה. לעומת זאת בלוח הגדול העומס גדול יותר, ולכן הרווח מהמקביליות יהיה גדול יותר מעלות החלפות ההקשר, וככל שמספר החוטים גדל כך הlatency משתפר.

גרפי הLatency(j) מתנהגים דומה לגרף אמדל שהתקבל בסעיפים א' ו-ב'. על אף ש-latency(A) בלוחות הקטן והבינוני עולה בגלל החלפות ההקשר, latency(j) מתייחס לכל חוט בנפרד ולא לוקח בחשבון את זמן המעבד שנדרש להחלפות. כתוצאה מכך ככל שמספר החוטים גדל כך גודל החישוב של כל חוט יורד ולכן נקבל אחוזים גבוהים יותר של מקביליות ונוכל לזהות את התכונות של גרף אמדל.

בנוסף, latency(A) של הלוח הגדול גם כן מתנהג באופן דומה לגרף אמדל, וזה בגלל כמות העבודה הגדולה אשר מאפשרת עבודה מקבילית אופטימלית בין החוטים לעומת הגרפים של הלוחות הקטנים.

במידה והיינו רוצים להוסיף מעבדים נוספים, סביר להניח כי בלוח הגדול, ה Latency היה משתפר מכיוון שאנחנו למעשה נוריד את עלות החלפות ההקשר (פחות חוטים ירוצו על כל מעבד). באותו אופן בלוחות הקטנים יותר סביר להניח שמגמת העלייה שתיארנו תמשיך להתקיים, אך ככל הנראה תתחיל לעלות בנקודת זמן מאוחרת יותר, מכיוון שעומס החישוב אינו גדול מאוד מלכתחילה הוספת מקביליות תתרום עד לשלב מסוים ותפחית את כמות החלפות ההקשר, עד שכל חוט יבצע חישוב קטן מאוד במקביל, והוספת עוד מעבדים לא תשנה את המצב.

את גרף הscatter plot הרצנו על המחשב האישי. מהגרף ניתן לראות כי בנק' זמן נתונה רצים במקביל 8 חוטים, בהסקה ראשונית ניתן לומר כי התוכנית רצה על 8 ליבות. לאחר שבדקנו את דגם המעבד של המחשב, גילינו כי למחשב יש 4 ליבות וכל ליבה יכולה להריץ 2 חוטים במקביל, דבר שתואם את תוצאות הגרף.
בנוסף ניתן להסיק כי timeslice הוא לפחות 8,000 מילי-שניות מכיוון שזהו הזמן המקסימלי מתחילת הרצת עבודה (נקודה כחולה) ועד סיום העבודה (נקודה אדומה) של 8 חוטים במקביל (שזהו המספר המקסימלי של החוטים האפשריים שאבחנו). אם ה-timeslice היה יותר קטן הייתה מתבצעת החלפת הקשר והיינו מצפים לראות נקודה כחולה שניה ברצף בטרם הופעת נקודה אדומה, כי חוט אחר היה מתחיל לבצע חישובים.