HW3 Dry Part

Shiran saada

301731998

shiransaada@campus.technion.ac.il

Yair Shachar

200431260

syairsha@campus.technion.ac.il

חלק ראשון:

שאלה 1:

סעיף א':

נענה על השאלה לפי סדר החשיבות של התכונות המופרות , תחילה נתייחס לתכונה הכרחית :

Progress: לפי ההסבר אם יש חוטים שמוכנים לבצע לבסוף אחד מהם יצליח לבצע , כלומר לא נוצר מצב של Deadlock למשל .

ניתן לראות במימוש הנתון כי יכול להיווצר מצב אשר חוט מבצע את הפעולה Unlock ומאחר והפעולה שבה היא לא פעולה אטומית , במהלך תהליך ההצבה למשתנה lockVal לפני שהוא מקבל את הערך 1 מתבצעת החלפת הקשר וחוט אחר מתחיל לבצע למשל את פעולת Lock , כעת אם החוט הראשון שהתחיל לבצע Unlock נפל מסיבה מסוימת , החוט השני וכל חוט אחר שינסה לבצע פעולת Lock יכנס ללולאה אינסופית מאחר והערך ב-lockVal ישאר 0 והמנעול עדיין יחשב נעול על ידי חוט אחר שנפל .

בנוסף קיימות תכונות רצויות שגם נפגעות :

Fairness: לפי ההסבר לעיל, ניתן לראות שיכול להיווצר מצב של Deadlock כפי שתיארנו למעלה ולכן גם אם יהיה חוט שרוצה לבצע את הקטע הקריטי הוא לא יצליח לעולם ויתקע בלולאה אינסופית.

Order: מאותה סיבה בדיוק לא קיים סדר ידוע וברור לזמני הכניסה של החוטים לקטע הקריטי.

סעיף ב':

לפי הסעיף הקודם ניתן לראות שקיימת בעיית Performance בביצוע פעולת Lock מאחר ואם המנעול נעול , החוטים ימשיכו לבצע BusyWait עד אשר המנעול ישתחרר במקום לצאת לחכות בתור המתנה ולכן בעצם גורמים לחוסר יעילות גדול של זמן מעבד . אם יש מספר גדול של חוטים הממתינים יבוצעו בינהם גם החלפות הקשר בעלות זמן מעבד רק בשביל שימשיכו לבצע את ה-BusyWait. לכן גם אם יש קטע קריטי קצר אבל יש מקרה בו הרבה חוטים ממתינים יבוצעו בינהם החלפות הקשר רק בשביל שיוכלו לבדוק האם המנעול שוחרר ולכן קיים בזבוז זמן מעבד רב ללא תלות באורך הקטע הקריטי.

שאלה 2:

שוב נענה על השאלה לפי סדר החשיבות של התכונות המופרות , תחילה נתייחס לתכונה הכרחית :

Mutual exclusion: המשמעות היא שבכל רגע נתון לא יכול להיות יותר מחוט אחד בתוך הקטע הקריטי, נשים לב כי בזמן שחוט ביצע נעילה והתחיל את הקטע הקריטי שלו חוט אחר הצליח לפרוץ את המנעול ולהיכנס לקטע הקריטי גם הוא , לצורך העניין אם מספר חוטים חיכו למנעול וכל אחד מהם הגדיל את המונה , מספיק חוט אחד שהגדיל את המונה ומספר MAX_ITER ופרץ אותו .

בנוסף קיימות תכונות רצויות שגם נפגעות :

Order: ניתן לראות לפי ההסבר במניעה הדדית כי אם יש הרבה חוטים ולפי הסדר הם מגדילים את המונה יכול להיות שחוט שזהו לא תורו הגדיל את המונה למספר הנדרש על מנת לפרוץ אותו והוא נכנס לקטע הקריטי לפני החוטים שלפניו בתור , אנחנו לא יכולים להבטיח במימוש זה שמדובר בחוט הראשון שממתין בתור ומעבר לכך לא יכולים להבטיח שום סדר כלשהו כי הסדר נקבע כתלות בריצת התוכנית .

שאלה 3:

הקוד הנתון עלול להדפיס ערכים שונים כתלות בהחלפות ההקשר שמתבצעות על החוטים שמבצעים את הפונקציה Workload .

<u>המקרה המקסימלי -</u> המקרה הרגיל הוא המקרה בו p_val מקבל את ערכי כל החוטים לפי הסדר במערך וכל חוט מבצע את הקטע הקריטי עם הערך המתאים לו , במקרה זה נקבל שבסופו של דבר כל חוט יבצע את הפונקציה עם ערך ה-id המתאים לו וSum יכיל :

Sum = 1+2+3+4+5+6+7+8+9+10 = 55

<u>המקרה המינימלי -</u> המקרה בו כל חוט מתחיל לבצע את הפונקציה עם הקטע הקריטי אבל לפני שהוא מספיק לשנות את הערך של p_val מתבצעת החלפת הקשר לחוט הבא , וככה ממשיכות החלפות ההקשר עד שמגיעים לחוט מספר 10 שמצליח לעדכן את p_val ולאחר עדכון Sum שיכיל את הערך 10 החוט יסיים את הפונקציה ולא יהיה לו למי לעשות join ולכן הוא ידפיס את Sum שמכיל את הערך 10 . נשים לב שבמידה והיה מתעדכן בהתחלה ערך אחר למשל 1 , והפונקציה הייתה מסתיימת החוט היה ממתין לשאר החוטים ככה שהמקרה המינימלי הבא הקטן ביותר הוא החלפות הקשר שוב עד 10 ואז Sum היה מכיל :

Sum = 1+10 = 11

לכן הערך המינימלי הוא התרחיש בו Sum מכיל 10.

נשים לב שיש אופציות רבות להחלפות ההקשר ולכן יכולים להיכתב סכומים שונים נוספים ב-Sum.

שאלה 4:

תחילה נציין מספר הבחנות על קטע הקוד :

*הפונקציה do_calc , בהנחה ש-result מאותחל ל-0 , מגדילה אותו ובסופה של הפונקציה המשתנה result יכיל 100 (כלומר מוסיפה לו 1 100 פעמים).

*התוכנית מאתחלת שני חוטים שמבצעים את הפונקציה do_calc , ולאחר מכן ממתינים לסיום החוטים ולבסוף מדפיסים את result.

הקוד הנתון עלול להדפיס ערכים שונים כתלות בהחלפות ההקשר שמתבצעות על החוטים שמבצעים את הפונקציה.

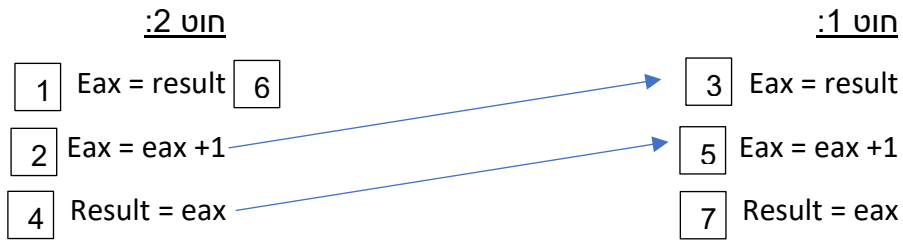
<u>המקרה המקסימלי -</u> המקרה הרגיל הוא המקרה בו result גודל ב-1 100 פעמים בחוט הראשון ו-100 פעמים בחוט השני . במקרה זה נקבל שבסופו של דבר כל חוט יבצע את הפונקציה ו-result יכיל :

result = 100+100 = 200

<u>המקרה המינימלי -</u> נשים לב שהפקודה result = result +1 היא לא פקודה אטומית ולכן יכולות להתבצע במהלכה החלפות הקשר בין החוטים , כפי שראינו בתרגול בכתה הפקודה :
result = result +1 מורכבת מ:

1) Eax = result
2) Eax = eax +1
3) Result = eax

נשים לב לביצוע אפשרי של החלפות הקשר בין החוטים :



במקרה הנ"ל הפקודות יתבצעו בסדר הזה והחיצים הם בעצם החלפות הקשר לדוגמא , ובסוף מצב זה תתבצע רק איטרציה אחת של החוט האחרון ו-result יכיל את הערך המינימלי 1 .

<u>שאלה 5:</u>

לפי מה שלמדנו בתרגול ובהרצאה , תהליכים שונים מחוטים בכך שהם לא חולקים את אותו מרחב הזכרון , תהליכים בעת החלפת הקשר מבצעים החלפה של כל מרחב הזכרון כולל המחסנית והרגיסטרים וכו' ואילו חוטים נמצאים תחת אותו מרחב הזכרון מאחר והם שייכים לאותו התהליך , ולכן בעת החלפת הקשר הם רק מחליפים את המחסנית.

לכן בקוד הנתון אין צורך להגן על Sum מאחר ולכל תהליך שנוצר יש העתק נפרד של המשתנה אשר לא משפיע על הערך של ההעתקים האחרים בתהליכים אחרים . ולכן אין צורך לעול את הערך מאחר ולא מדובר באותו מקום בזכרון .

חלק שני:

סעיף א':

נממש מנעול למניעה הדדית בעזרת Singelphore :

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

 void mutex_init(mutex* m) {
    singlephore_init(&m->h);
}
 void mutex_lock(mutex* m) {
    H(&m->h,0,-1);
}
void mutex_unlock(mutex* m) {
    if(m->h.value == 0){
        return;
    }
    H(&m->h,-1,1);
}
```

נפרט על המימוש :

אתחול – אתחול ה-mutex מתבצע על ידי אתחול הסינגלפור , כלומר מבצע השמת 0 לתוך ה-value.

כאשר מנסים לבצע נעילה יש שני מצבים :

הראשון הוא שהחוט רוצה לנעול , אבל המנעול פתוח כלומר (value = 0) ולכן לא ניכנס ללולאה בתנאי ה-while של הפונקציה H ונשנה את הערך למינוס 1.

השני הוא המצב בו החוט מנסה לנעול אבל המנעול לא פנוי ואז בביצוע הפונקציה H בתנאי הלולאת while אנחנו ניכנס ללולאה ונבצע yield .

כאשר מנסים לבצע שחרור של המנעול יש שני מצבים :

הראשון הוא שהחוט רוצה לפתוח את המנעול , אבל המנעול לא תפוס , אנחנו פשוט נסיים את הפונקציה לפי הבדיקה של תנאי ה- if.

השני הוא המצב בו החוט רוצה לפתוח את המנעול והוא תפוס , בביצוע הפונקציה H תנאי הלולאה לא יתקיים (1->1-) ולכן אנחנו נגדיל את הערך של value חזרה ל-0 (כלומר פותחים את המנעול).

סעיף ב':

```
typedef struct condvar {
  mutex cond_lock;
  singlephore h;
  int num_waiting;
} condvar;


// Initilize the condition variable
void cond_init(condvar* c) {
  mutex_init(&c->cond_lock);
  singlephore_init(&c->h);
  c->num_waiting = 0;
}

// Signal the condition variable
void cond_signal(condvar* c) {
  mutex_lock(&c->cond_lock);
  if(c->num_waiting > 0){
    H(&c->h,MIN_INT,1);
    c->num_waiting--;
  }
  mutex_unlock(&c->cond_lock);
}

// Block until the condition variable is signaled. The mutex m must be locked by the
// current thread. It is unlocked before the wait begins and re-locked after the wait
// ends. There are no sleep-wakeup race conditions: if thread 1 has m locked and
// executes cond_wait(c,m), no other thread is waiting on c, and thread 2 executes
// mutex_lock(m); cond_signal(c); mutex_unlock(m), then thread 1 will always recieve the
// signal (i.e., wake up).

void cond_wait(condvar* c, mutex* m) {
  mutex_unlock(m);
  c->num_waiting++;
  mutex_lock(&c->cond_lock);
  H(&c->h,1,-1);
  mutex_unlock(&c->cond_lock);
  mutex_lock(m);
}
```

הסבר :

הטיפוס מכיל 3 שדות – מנעול כדי לאפשר Mutual Exclusion בעת גישה למשתנה התנאי היכול להיות משותף לכמה חוטים ובעת גישה לsignal ול-wait .

Num_waiting : כמות החוטים הממתינים לתנאי .

סינגלפור – שיאפשר לחוט לבצע yield ויגדיל את value בעת שליחת signal (בעצם מאפשר לנו להוציא ולהחזיר מהתור המתנה)

הפונקציות :

Init – אתחול כל השדות , והשמת 0 למספר הממתינים .

Signal – ביצוע נעילה , לאחר מכן בדיקה האם יש ממתינים בתור (כדי למנוע את הבעיה הקיימת בסעיף ג' – עשינו אותו לפני סעיף ב') , אם יש ממתינים אז בעצם מגדילים את value ועל ידי כך מעירים מישהו מהמתנה ובנוסף מקטינים את מספר הממתינים , לבסוף משחררים את המנעול .

Wait – שחרור המנעול שבשימוש על ידי החוט , הגדלת מספר החוטים הממתינים וקריאה לפונקציה H שלבסוף תוביל ל yield (לפני הקריאה נועלים ואחר כך משחררים כדי למנוע גישה בו"ז לקטע הקריטי) , לאחר מכן נעילת המנעול .

אם נדרש כי יש כמה חוטים שממתינים אז מבצעים מספר פעמים signal ככמות הממתינים .

סעיף ג':

נשים לב שבעצם מה שאנחנו מנסים לבצע (שימוש במשתנה תנאי) בכלל לא מתקיים בצורה תקינה. נשים לב שהפונקציה cond_signal לא מבצעת yield לשום תהליך מאחר וכל תהליך שיכנס עם ערך value כלשהו לא יצליח לקיים את תנאי הלולאה , תמיד יתקיים bound < h->value מאחר ומאתחלים את bound עם הערך INT_MIN ולכן אף תהליך לא יוותר על המעבד . ניתן דוגמא לתרחיש שגוי שיכול לקרות :

מיד לאחר אתחול המערכת כאשר value = 0 , אם חוט יבצע signal הערך של value יעלה ל- 1 , ואז אם חוט ינסה לבצע wait הוא לא יצליח לבצע את הפעולה מאחר ו- value = 1 > 0 ולכן לא יוותר על המעבד וימשיך לרוץ בסתירה למה שהמערכת הייתה אמורה לבצע במידה והייתה עובדת בצורה תקינה .

הבעיה כאן היא שבמערכת תקינה בעת שליחת signal אם אין חוט שממתין אז לא קורה כלום , אבל כאן הלוגיקה מושפעת ולכן חוט שירצה לבצע wait יפגע מחוט שביצע signal כאשר אף אחד לא המתין .

בעיה נוספת היא שביצוע wait בפעם הראשונה לא יתבצע כנדרש מאחר ו-value = 0 ולא מתקיים התנאי value < bound = 0 ולכן הוא לא יצליח לוותר על המעבד .

חלק שלישי:

שאלה 1:

סעיף א':

רעיון כללי:
כוונותו העיקרית של פיראס היא שהחוט הראשי ייצר N משימות שיכנסו לתור הPCQ ובכל דור יחכה לסיום כל N העבודות ולאחר מכן יחליף בין המטריצה הנוכחית למטריצה הבאה ויחזור על התהליך כמספר הדורות המבוקש.

הסבר:

- לולאת הfor הפנימית:
  - תרוץ N פעמים כאשר N הוא מספר המשימות הדרושות לביצוע בדור הנוכחי.
    בכל איטרציה יבצע Push לעבודה הנוכחית אל תוך הPCQ וכן b.increase שיגדיל את ערך הbarricade ב-1.
    סה"כ N איטרציות.
- לאחר סיום הכנסת N עבודות לPCQ, החוט הראשי יבצע b.wait ובכך יחכה כל עוד ערך הbarricade שונה מ0.
- ביצוע עבודה ע"י תהליך:
  - כאשר תהליך יתחיל לעבוד, יבצע pop מהPCQ ויבצע את החישובים הנדרשים על הjob שקיבל. לאחר שיסיים את החישובים יבצע b.decrease ויקטין את ערך הbarricade ב-1
    סה"כ N עבודות וN תהליכים לכן באופן תאורטי ערך הbarricade יוקטן ב1 N פעמים.

הרעיון של פיראס היה שהתהליך הראשי יבצע N increase פעמים וסה"כ החוטים יבצעו N decrease פעמים ולכן התהליך הראשי ישוחרר מהb.wait בתום ביצוע העבודה הNית.

ההבדל העיקרי של הbarricade מהsemaphore הוא שערך הsemaphore נקבע מראש וערך הbarricade אינו חסום.

סעיף ב'

1. RaceCondition: יכול להגרם כתוצאה מ2 חוטים המנסים לבצע ()decrease או ()increase.

   השדה working של barrier הוא משאב משותף בין החוטים, נשים לב שאין מעטפת מנעול המגנה על קטע קריטי בו מקטינים או מגדילים את השדה הזה, לכן, מאחר ופעולות אלו אינן אטומיות תתכן החלפת בזמן ביצוען ולכן יתכן סדר תזמונים בין החוטים המשבש את לוגיקת הקוד ביחס למשאב המשותף.

2. Mutual Exclusion: במקרה פרטי של סעיף 1 (RC), יתכן מצב שבו החוט הראשי (PRODUCER) יבצע push לעבודה האחרונה שנותרה וכן חוט A יתחיל בביצועה, הPRODUCER יבצע b.increase וכן חוט B שסיים את עבודתו יבצע b.decrease כאשר working=1. במקרה בו החוט B המבצע decrease "ינצח", נקבל כי working=0.כעת, החוט PRODUCER יצא מיד מהלולאה שבb.wait (כי working == 0) וימשיך להחלפת המטריצות

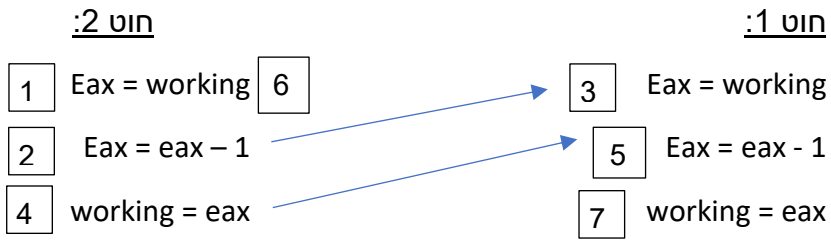
למרות שעדיין חוט A המבצע חישובים על עבודה של הדור הנוכחי (ז"א חישוב הלוח curr טרם הסתיים בזמן החלפת המטריצות)

3. Deadlock :

כפי שראינו בתרגול בכתה למשל הפקודה : working = working - 1 מורכבת מ:

1) Eax = working
2) Eax = eax - 1
3) working = eax

נשים לב לביצוע אפשרי של החלפות הקשר בין החוטים :



נסתכל למשל על תרחיש בו 0 < working = g < N , ומתרחש ביצוע של החלפת הקשר כמתואר לעיל, יתכן מצב בו שני חוטים מבצעים decrease אך בפועל ההקטנה מתבצעת פעם אחת בלבד, לאחר סיום כל החוטים נגיע למצב ש- working = 1 > 0 ולכן ניכנס למצב של המתנה אין סופית לסיום הדור הנוכחי וזהו deadlock .

בצורה דומה יתכן מצב בו אנו מנסה לעשות ()increase ע"י החוט הראשי (PRODUCER) וכן בדיוק באתו הזמן חוט A מסיים את עבודתו ומנסה לעשות ()decrease כך שהערך מתייצב על ערך הincrease. בסוף התהליך הנ"ל שוב נקבל כי working > 0 ולכן נכנס למצש של המתנה אין סופית לסיום הדור הנוכחי וזהו deadlock.

סעיף ג':

1) נוסיף שדה של מנעול mutex על מנת להגן על קטעי הקוד הקריטיים במחלקת Barricade

```
class Barrier {
 private:
      pthread_mutex_t mutex;
      int working;
 public:
    Barrier(){
        working =0;
    }
    increase(){
      pthread_mutex_lock(&mutex);
      working++;
      pthread_mutex_unlock(&mutex);
    }
    decrease(){
      pthread_mutex_lock(&mutex);
      working--;
      pthread_mutex_unlock(&mutex);
    }
    //using a lock has a heavy weight, so it is better not to lock , in the worst case
    //the process will be awakened
    wait(){
      while(working != 0){}
    }
}

```

נתקן את הפסאודו קוד , על ידי הוספת משתנה בוליאני בשם not_finished , המשתנה ישמש את הצרכן לדעת מתי הסתיימה העבודה (בסוף כל הדורות).

```
/* Correction of the psuedo code */

/***** Correction of producer *****/
Producer:
bool not_finished = 1;
Init Barricade b
Init PCQueue p
Init fields curr,next
for t = 0 -> t = n_generations
    for i = 0 -> i = N
        p.push(job);
        b.increase();
    b.wait();
    swap(curr,next);
not_finished = 0;


 /***** Correction of consumer *****/
Consumer(One of N)
while (not_finished){ //instead of while(1)
    job j=p.pop()//block here if queue is empty
    execute j
    b.decrease();
/*rest of the code is the same*/
```

2) הסיבה לכך שהמימוש לא יעיל מבחינת הביצועים היא שעלינו להמתין עד סיום העבודה של כל התהליכים על מנת להחליף את המטריצות ולהתקדם לדור הבא. את ההמתנה אנחנו מבצעים בצורת busy wait וכך מונעים חלוקת משימות חדשות לפני סיום הדור הנוכחי. ההמתנה הזו מבזבזת זמן מעבד יקר, כיוון שביצוע החישובים על המטריצות הוא ארוך ו"יקר" חישובית. לכן המתנה במהלך החישוב היא לא כדאית ובמקום נכון יותר להוציא את התהליך להמתנה ולהעיר אותו ע"י שימוש בsignal כמו שלמדנו בכתה .

3) כך יראה המימוש באמצעות פעולות אטומיות וbusy wait , נשתמש בפעולות אטומיות לטובת הגדלת והקטנת waiting וכך לא תהיה סכנה של החלפת הקשר , בפונקציה wait בעצם נשתמש בפעולה CAS ע"י כך שאם waiting מכיל 0 נכניס לו את הערך 0 והלולאה תעצור , אחרת לא נכניס לו את הערך החדש 0 והפונקציה תמשיך להחזיר ערך ששונה מ-0 וה-busy wait ימשיך לפי הלוגיקה המתאימה .

```
class Barricade {
 private:
      int working;
 public:
      Barricade(){
        working = 0;
    }
      increase(){
        atomicAdd(&working,1);
    }
      decrease(){
        atomicAdd(&working,-1);
    }
    //using a lock has a heavy weight, so it is better not to lock , in the worst case
    //the process will be awakened
      wait(){
       while(CAS(&working,0,0)){}
    }
};
```

4) מימוש 3 עדיף על מימוש 1 מאחר ועקב שימוש בפעולות אטומיות אנחנו מונעים בעצם המון החלפות הקשר מיותרות בהן חוטים פשוט יראו כי המנעול תפוס ויבצעו החלפת הקשר נוספת , בעצם הפעולות האטומיות יגמרו לתכנית לרוץ בצורה יעילה ביותר עם החלפות הקשר רלוונטיות שלא "מתבזבזות לחינם" , מנגד במימוש 1 עם המנעול יהיו המון החלפות הקשר שהתוצאה שלהן תהיה לגלות שהמנעול תפוס והן לא יכולות לבצע כלום – בסופו של דבר יבוזבז זמן מעבד יקר ורב .

סעיף ד:

נחזיק שני מבני PCQUEUE, נסמנם PCQ_A ו PCQ_B כאשר A מחזיק בתוכו עבודות המחכות לביצוע ו B מחזיק עבודות שסיימו להתבצע.

האלגוריתם יפעל כך:

PRODUCER: בצע N פעולות PCQ_A.push(job)
CONSUMER: בצע PCQ_A.pop=job והתחל חישובים, לאחר ביצוע העבודה, בצע PCQ_B.push(job)
PRODUCER: בצע N פעולות PCQ_B.pop()

הקוד יכול להראות בפסודו קוד בצורה הבאה:

PRODUCER	CONSUMER

For(; i < N ; i++){
PCQ_A.Push(job_n);
}
Int j=0
While(j < N){
PCQ_B.pop()
j++
}
Go to next generation
	Job = PCQ_A.pop();
	
	“ Do job “
	
	PCQ_B.push(job)

במימוש הנ"ל תמיד יהיה סנכון בהכנסות והוצאות של עבודות ובנוסף מהגדרת האלגוריתם תהליך האב יצא מהלולאה וימשיך לדור הבא רק כאשר יוציא N עבודות שהסתיימו ולכן לא יצטרך לדאוג שהחישובים הנוכחיים עדיין בביצוע.

- בכדי לבצע מימוש מינימאלי נבצע את האלגוריתם הבא:
נחזיק מנעול גלובלי ומשתנה גלובלי finished שיאותחל ל0 בכל תחילת דור ויספור את מספר העבודות שסיימו

PRODUCER	CONSUMER

For(; i < N ; i++){
PCQ.Push(job_n);
}
Pthread_mutex_lock(&global_lock)
While(finished != N){
Pthread_cond_wait(&condition, &global_lock)
}
Pthread_mutex_unlock(&global_lock)
Go to next generation
Finished = 0
	Job = PCQ_A.pop();
	
	“ Do job “
	
	Pthread_mutex_lock(&global_lock)
	Finished++
	If(finished == N){
	Pthread_cond_signal(&condition)
	}
	Pthread_mutex_unlock(&global_lock)

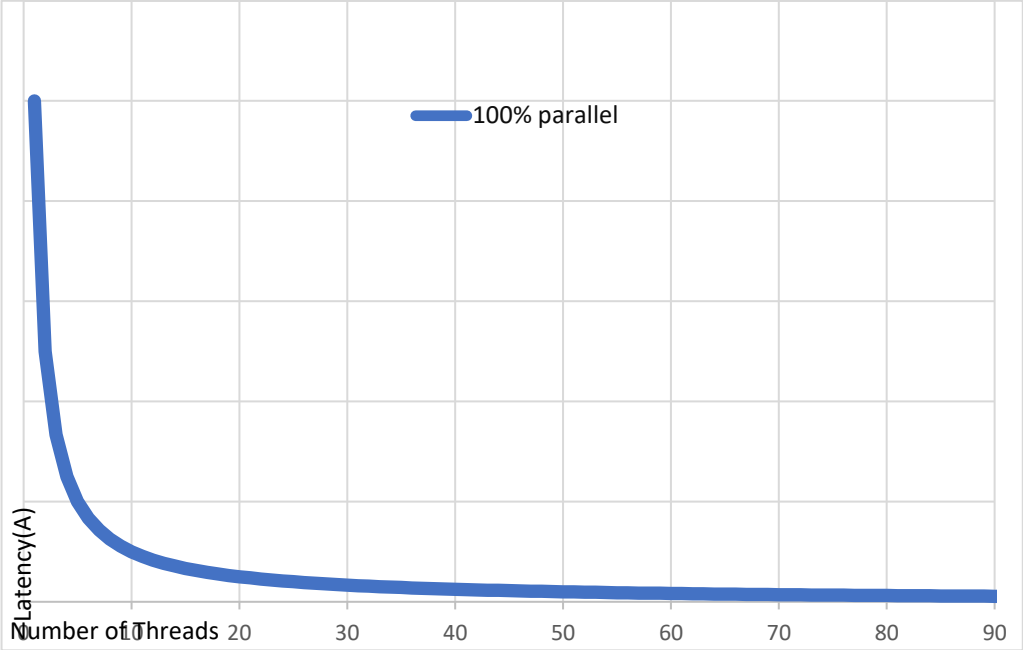
הקטע הקריטי הוא בשינוי של המשתנה finished, נשמור עליו בעזרת מנעול גלובלי, המימוש יעיל יותר מכיוון שהחוט הראשי לא נמצא בbusy wait אלא יוצא להמתנה ומחכה לסימון שיתקיים כאשר בדיוק כל העבודות הסתיימו ואפשר להמשיך לדור הבא.

שאלה 2

סעיף א:

מהחוק של אמדל נקבל כי: Latency(A) = Latency(A)*(s+(1-s)/n) כאשר s הוא החלק היחסי של האלגוריתם שהנו סדרתי. (0<=s<=1)

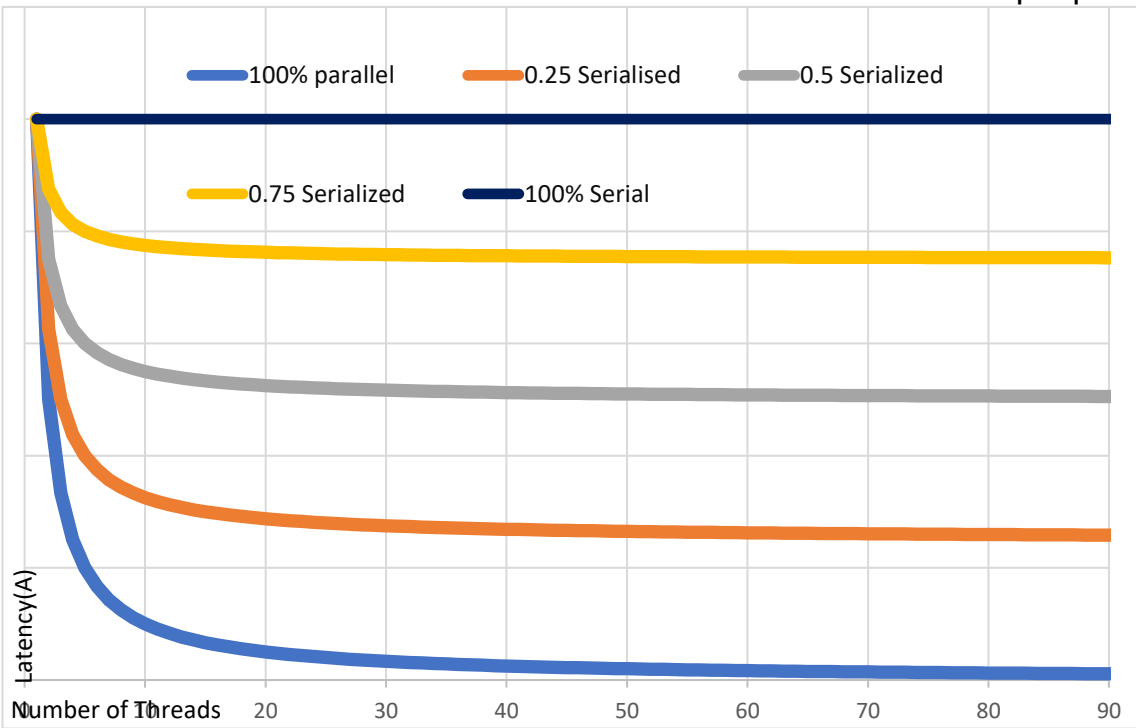
כאשר האלגוריתם הוא מקבילי לחלוטין, נקבל כי s=0 ולכן Latency(A)=1/n כאשר n הוא מספר החוטים הרצים במקביל.



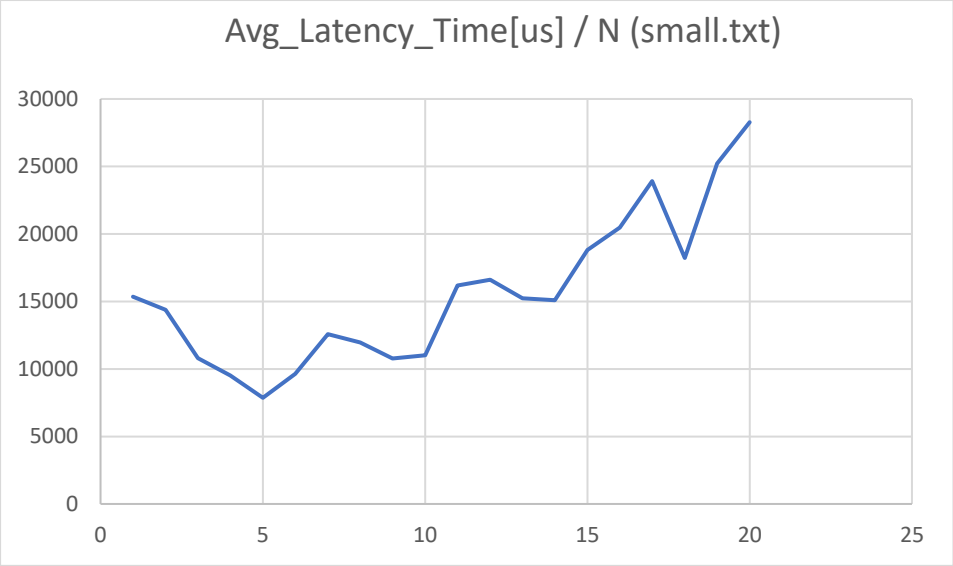
סעיף ב

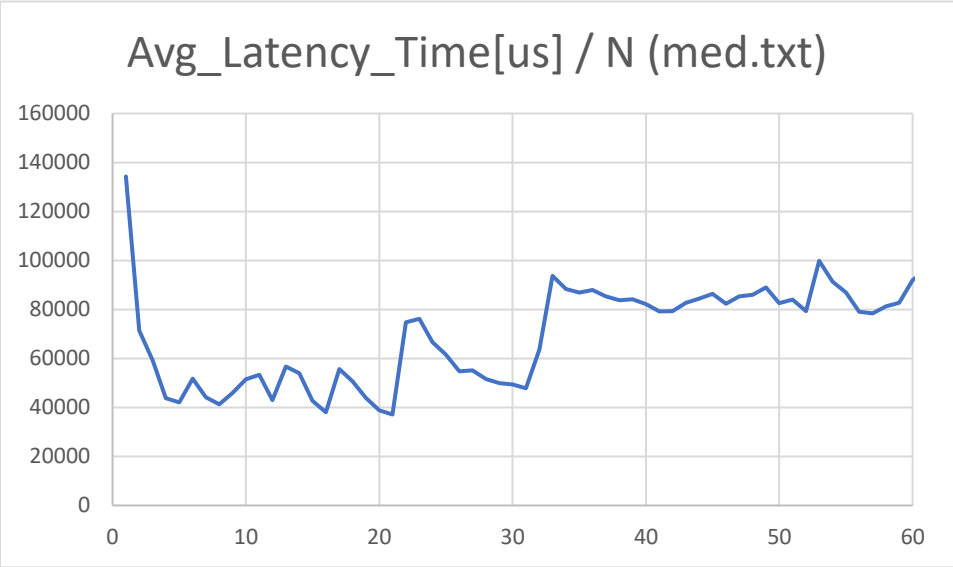
ניתן לראות כי תאורטית ככל שאחוז המקבול של האלגוריתם גדול יותר, כך האלגוריתם יפעל מהר יותר ככל שנוסיף חוטים שיפעלו במקביל.

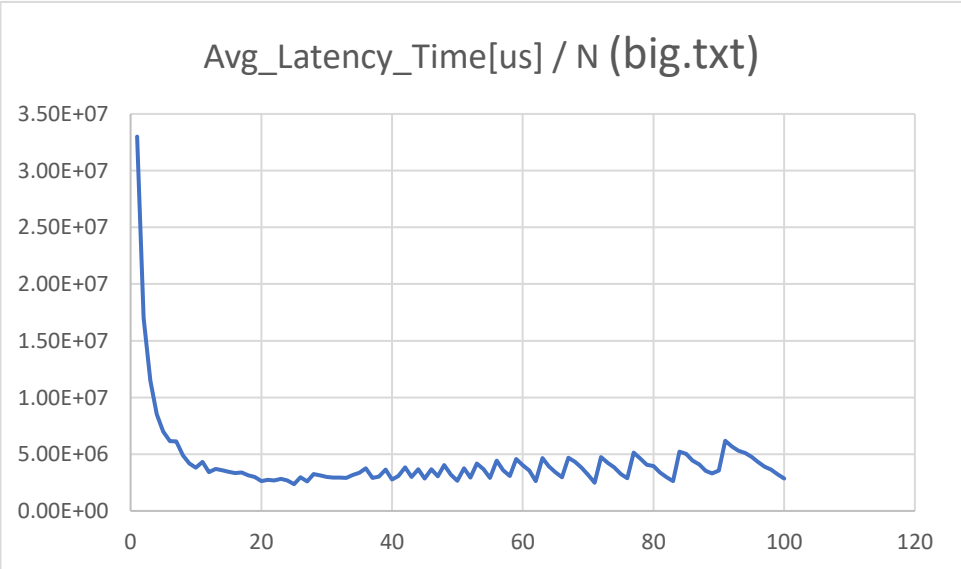
כאשר s=1 (ז"א שהאלגוריתם סדרתי לחלוטין) נקבל כי מספר החוטים שנוסיף אינו משפיע על סך זמן ביצוע האלגוריתם.

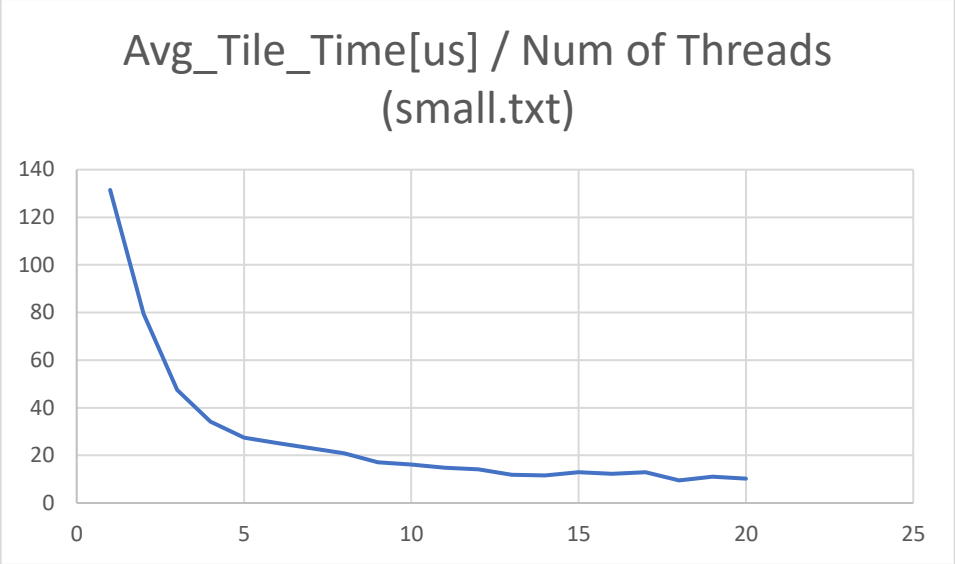


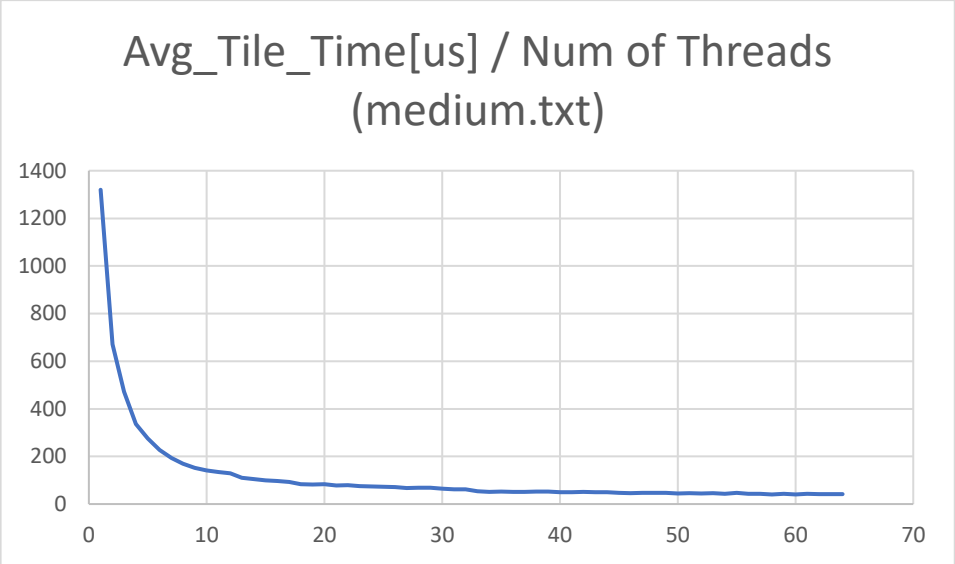
סעיף ג- ניתוח ביצועים של האלגוריתם שנכתב בחלק הרטוב:

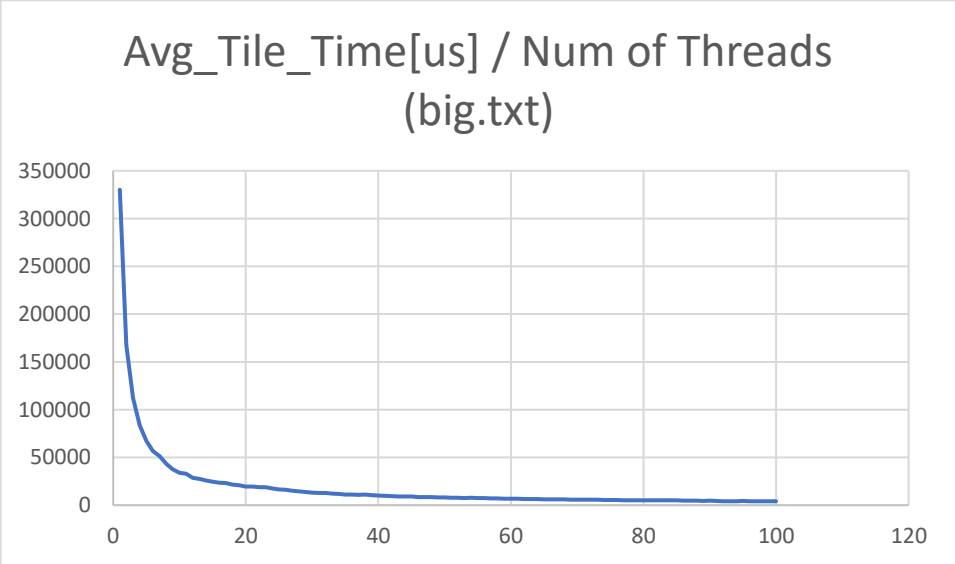


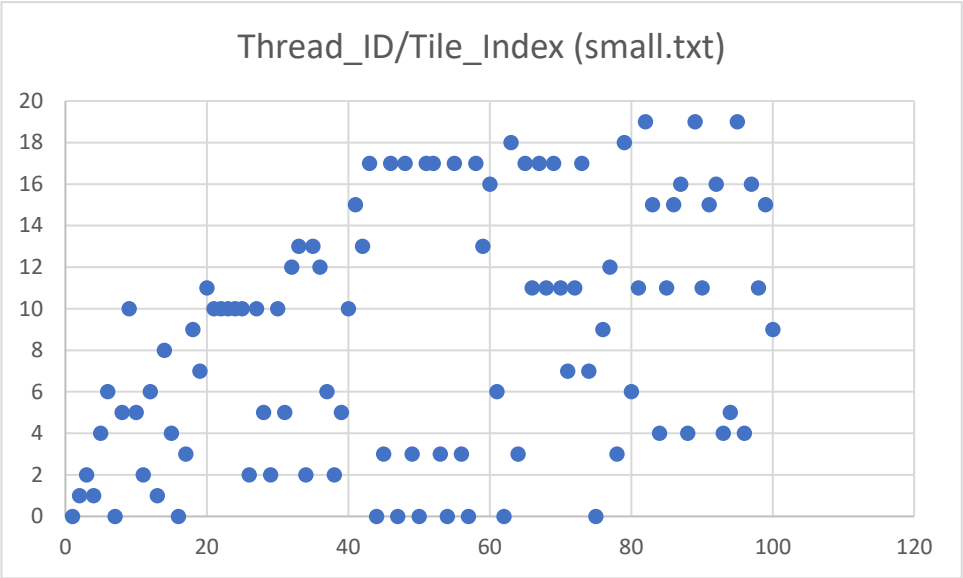


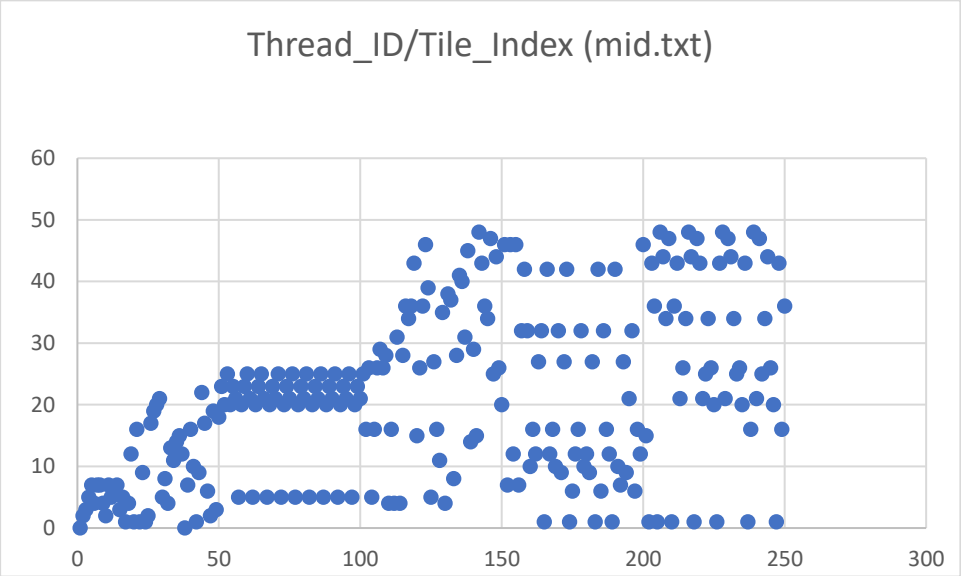


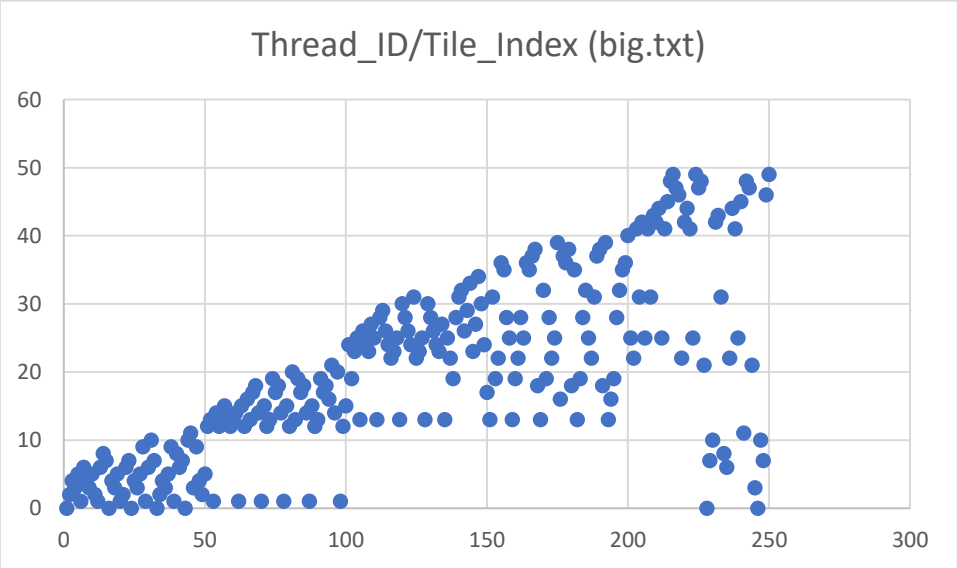

Avg_Tile_Time[us] / Num of Threads (small.txt)
140
120
100
80
60
40
20
0
0
5
10
15
20
25


Avg_Tile_Time[us] / Num of Threads (medium.txt)
1400
1200
1000
800
600
400
200
0
0
10
20
30
40
50
60
70


Avg_Tile_Time[us] / Num of Threads (big.txt)
350000
300000
250000
200000
150000
100000
50000
0
0
20
40
60
80
100
120


Thread_ID/Tile_Index (small.txt)
20
18
16
14
12
10
8
6
4
2
0
0
20
40
60
80
100
120


Thread_ID/Tile_Index (mid.txt)
60
50
40
30
20
10
0
0
50
100
150
200
250
300


Thread_ID/Tile_Index (big.txt)
60
50
40
30
20
10
0
0
50
100
150
200
250
300

- <u>גרף Average Latency (A) כתלות במספר החוטים N</u>

ניתן לראות כי ישנה מגמת ירידה משמעותית בזמן חישוב ממוצע של דור כתלות בכמות החוטים המבצעים את החישובים, עד למספר מסויים של חוטים שלאחריו לא חל שיפור ואף הוספה דווקא גורעת מזמן חישוב הדור הממוצע.
הדבר נובע מכך שבשלב מסויים אנו מתחילים לשלם יותר זמן על החלפות הקשר בין החוטים ועל הזמן שבו אנו פותחים\נועלים ומחכים למנעולים מאשר להרוויח מכמות החוטים הגדולה, בנוסף ניתן לראות שלאחר ירידה מסויימת אנו מגיעים ל"רוויה" של המעבדים ולמספר המקסימלי של החוטים היכולים לעבוד במקביל.

- <u>בגרף Average Latency (j) כתלות במספר החוטים N</u>

ניתן לראות כי הזמן הממוצע שולקח לחשב עבודה j הולך וקטן בצורה מתמדת מכיוון שככל שיש לנו מספר גדול יותר חוטים כך מספר העבודות גדל (עד למקסימום של גובה הטבלה) וכך כל עבודה דורשת חישוב של פחות שורות של המטריצה, לכן ככל שנגדיל את כמות החוטים נקבל עבודה ממוצעת יותר מהירה על כל עבודה.
במידה ונגדיל את מספר החוטים כך שיהיו יותר מגודל הטבלה, נצפה לראות התייצבות של הזמן הממוצע על מספר קבוע מכיוון שלא ניתקן להקטין את גודל העבודות יותר מזה.

- <u>כמות אידאלית של חוטים</u>
  - **בגרף הקטן:** ניתן לראות שכמות האידאלית של חוטים עבור זמן ממוצע של דור נמוך היא באזור 5-6, מכיוון שהגרף קטן עלות החלפות ההקשר והמנעולים דווקא פוגעת בנו כאשר אנו מגדילים מאוד את מספר החוטים.
  - **בגרף הבינוני:** גם כאן ניתן לראות שקיימת כמות אידאלית של חוטים הנמצאת בטווח שבין 20ל-25 חוטים, והוספת חוטים מכאן לא משפרת ואף עלולה להזיק לתוצאות החישוב.
  - **בגרף הגדול:** מכיוון שבגרף זה גודל החישוב של כל עבודה מאוד גדול, עלות החלפות ההקשר והמנעולים נהיית זניחה ונראה שככל שאנו מוספים יותר חוטים אנו מגיעים לתוצאות מהירות וטובות יותר, עד כדי המקסימום חוטים שהמעבדים שלנו יכולים למקבל.

- **<u>הגרפים Thread_ID/Tile_Index:</u>** ניתן לראות שבכל שלב בביצוע האלגוריתם ישנם מספר מצומצם של חוטים אשר מקבלים עבודות ומבצעים אותן, הדבר נובע מכך שיש לנו מספר מוגבל של מעבדים ולכן כמות החוטים שיכולים לעבוד במקביל הוא קטן. אנו משערים שדרך שבה נבחרים החוטים לעבוד נקבע ע"י הטיים סלייס שלהם, כאשר לרוב חוט יוכל לסיים יותר מעבודה אחת בטיים סלייס שלו ולכן יתועדף להבחר שוב ולסיים את הטיים סלייס שלו אם הוא ויתר על המעבד כאשר סיים.
לאחר שיסיים את זמן המעבד המוקצה לו, יתועדף אחריו חוט אשר כלל לא ביצע עבודות וכך חלוקת העבודה תתחלק בין החוטים. ניתן להסיק עם כן שכמות הליבות הקיימות במערכת ההפעלה דומה למספר החוטים אשר נראים כעובדים בסמוך אחד לשני ולכן נסיק שקיימות 4-5 ליבות.