מהלך הניסוי – חלק א'

בחלק זה של הניסוי נבנה תוכנית פשוטה לסביבת מעבד ה-HAL, נריץ אותה בסימולאטור ונבדוק השפעה של פרמטרים שונים של ארכיטקטורת מעבד מרובה ליבות ומבנה התוכנית על ביצועי המערכת.

<u>שעת התחלת הניסוי</u>:

9:00

1. בניה והרצה בסימולאטור של שלד תוכנית בסיסית

היכנסו לסביבת Eclipse על ידי לחיצה על קיצור הדרך בשולחן העבודה:

פתחו את הפרויקט Part1 על ידי מקש-עכבר ימני Open Project

הפרויקט כולל שני קבצים: part1.c ו- part1.map.

תוכן הקובץ part1.map הינו:

```
regular task init ()
regular task evaluate (init/u)
regular task finish (evaluate/u)
```

1.1 ציירו את גרף התלויות של התוכנית

*הערה – ניתן (ורצוי) לצייר ידנית ולצרף תמונה

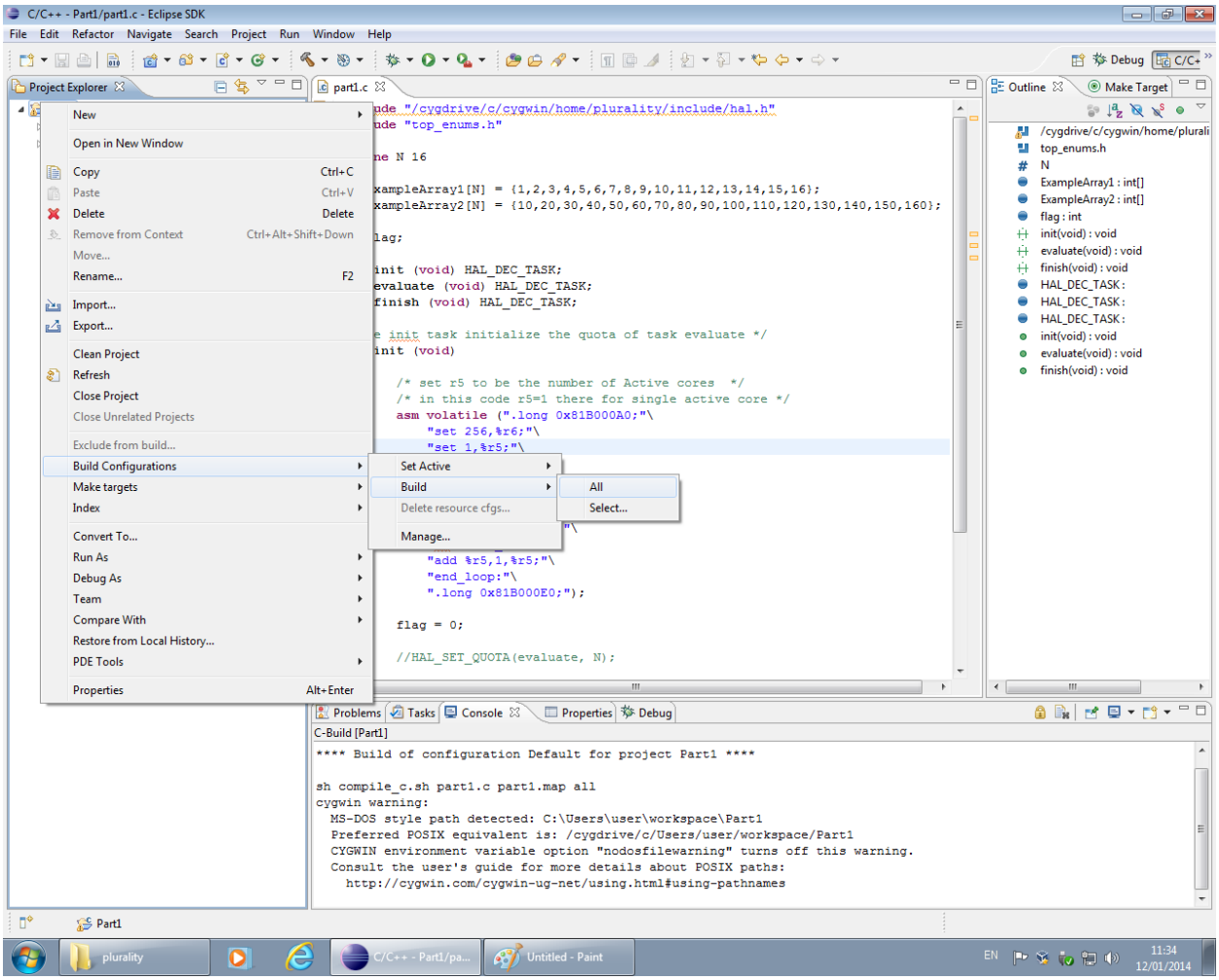
תוכן הקובץ part1.c הינו:

```
#include "/cygdrive/c/cygwin/home/plurality/include/hal.h"
#include "top_enums.h"

#define N 16

int ExampleArray1[N] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
int ExampleArray2[N] = {10,20,30,40,50,60,70,80,90,100,110,120,130,140,150,160};

int flag;

void init (void) HAL_DEC_TASK;
void evaluate (void) HAL_DEC_TASK;
void finish (void) HAL_DEC_TASK;




/* The init task initialize the quota of task evaluate */
void init (void)
{
		/* set r5 to be the number of Active cores  */
		/* in this code r5=1 there for single active core */
		asm volatile (".long 0x81B000A0;"\
			"set 256,%r6;"\
			"set 1,%r5;"\
			"start_loop:;"\
			"cmp %r5,%r6;"\
			"be end_loop;"\
			"nop;"\
			"sta %r0,[%r5]222;"\
			"ba start_loop;"\
			"add %r5,1,%r5;"\
			"end_loop:"\
			".long 0x81B000E0;");

		flag=0;

		//HAL_SET_QUOTA(evaluate, N);
}

/* The evaluate task evaluates the add of ExampleArray1[i] and ExampleArray2[i] */
void evaluate (void)
{

}
```

```
void finish (void)
{
	flag=1;
}
```

זהו שלד בסיסי של תוכנית במודל TOP. המשימה הראשונה init, מגדירה את כמות הליבות בסימולציה על ידי קביעת ערכו של האוגר %r5.

בנו את הפרויקט על ידי לחיצה **Ctrl-B** או על מקש עכבר ימני Build Configurations -> Build -> All



לאחר כל בנייה (קומפילציה של הפרוייקט) הקפידו להסתכל בפלט לשונית ה-Console על מנת לוודא הצלחת הבנייה.

הוסיפו שתי breakpoints בשורות 33 ו- 45 (flag = 0; flag = 1;)

על ידי מקש עכבר ימני Toggle Breakpoint:

```
/* The init task initialize the quota of task evaluate */
void init (void)
{
        /* set r5 to be the number of Active cores  */
        /* in this code r5=1 there for single active core */
        asm volatile (".long 0x81B000A0;"\
            "set 256,%r6;"\
            "set 1,%r5;"\
            "start_loop:;"\
            "cmp %r5,%r6;"\
            "be end_loop;"\
            "nop;"\
            "sta %r0,[%r5]222;"\
            "ba start_loop;"\
            "add %r5,1,%r5;"\
            "end_loop:"\
            ".long 0x81B000E0;");

        flag = 0;
```

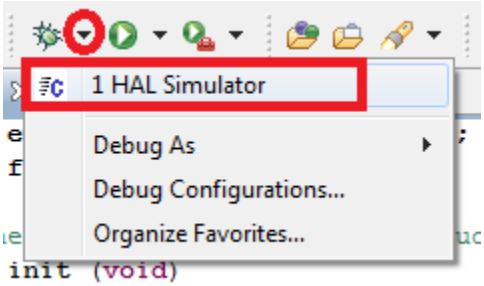
Toggle Breakpoint
Enable Breakpoint
Breakpoint Properties...

הנקודה הכחולה משמאל מראה שה-breakpoint נוסף בהצלחה.

```
        flag = 0;
```

אנו נשתמש ב-breakpoint על מנת לבדוק את מספר ה-cycles בכל שלב של ריצת התוכנית. לשם כך נקבע את ה-breakpoint הראשון בסוף משימת ה-init. כאשר נבדוק את מספר ה-cycles בנקודה זו נקבל את <u>זמן האתחול</u> של הסימולאטור. נרצה להחסיר זמן זה מכל מדידות ה-cycles שנבצע בהמשך התוכנית, לדוגמא במשימת ה-finish.

כעת, ננסה להריץ את התוכנית בסימולאטור. על מנת לעשות זאת לחצו על ה-icon של Debug:



ובחרו באפשרות HAL Simulator.

התוכנית תתחיל לרוץ ותעצור ב-breakpoint הראשון בשורה 33 , הקישו בשורת ה-Console :

```
info hal-simulation-cycles
```

```
Console   Tasks   Problems   Executables
HAL Simulator [Zylin Embedded debug (Cygwin)] hal-elf-gdb (11:57 12/01/14)
target hal
run
warning: while parsing target description: no element found
warning: Could not load XML target description; ignoring
[Switching to task 0 instance 0]

Breakpoint 1, end_loop () at part1.c:33
33                      flag = 0;
info hal-simulation-cycles
```

יתקבל מספר ה-cycles הנדרש לאתחול הסימולטור.

הריצו את התוכנית עד ה-breakpoint הבא על ידי הקשה על

(או על הכפתור F8).

הקישו בשורת ה-Console שוב info hal-simulation-cycles.

1.2 מהו ההפרש המתקבל? תוכלו להסביר ממה הוא נובע?

קיבלנו הפריש של $1871 - 1828 = 43$ סייקלים וזאת כיוון שבין המשימות יש צורך ב- schedule המשימה ל- core המתאים ושליחת ה- token על מנת ש- finish task יוכל להתחיל לרוץ לאחר evaluate task

ננסה כעת לבצע משימה פשוטה באופן סדרתי (למעבד בעל ליבה אחת). בפונקציה evaluate הוסיפו קוד המחשב סכום כולל של איברים במערכי התוכנית ExampleArray1 ו- ExampleArray2. תחילה חשבו חיבור ווקטורי של שני המערכים ולאחר מכן בצעו סכום של איברי הווקטור שהתקבל מתוצאת החיבור.

1.3 צרפו את קוד התוכנית שמימשתם. מהו הסכום שהתקבל?

```
void evaluate (void) {

    int sum = 0;

    for (int i=0 ; i<N ; i++)
        ExampleArray1[i] += ExampleArray2[i];

    for(int i=0 ; i<N ; i++)
        sum += ExampleArray1[i];
}
```

הסכום שהתקבל הינו 1496

1.4 כמה cycles נדרשו לצורך ביצוע התוכנית הסדרתית?

קיבלנו $2154 - 1828 = 326$

שנו את מספר הליבות בסימולאטור באופן הבא:

בפונקציה init שנו את ערכו של אוגר %r5 לערך 4.

```
asm volatile (".long 0x81B000A0;"\
        "set 256,%r6;"\
        "set 4,%r5;"\
        "start_loop:;"\
        "cmp %r5,%r6;"\
        "be end_loop;"\
        "nop;"\
        "sta %r0,[%r5]222;"\
        "ba start_loop;"\
        "add %r5,1,%r5;"\
        "end_loop:"\
        ".long 0x81B000E0;");
```

הריצו שוב את התוכנית ומדדו את כמות ה-cycles הנדרשו לביצוע.

1.5 האם התקבלה תוצאה שונה? מדוע?

מספר הסייקלים נשאר זהה, זאת כיוון שעדיין task מוקצה ל- core יחיד ולכן לא משנה כמה cores נקצה לתוכנית נקבל את אותם ביצועים.

השורות הבאות מסבירות באופן מדויק למה זה קורה, החלק המקבילי של התוכנית הוא 0 ולכן אין מה למקבל ומקבלים $speedup = 1$

נגדיר האצה באופן הבא: $A = \frac{t(1)}{t(P)}$, כאשר t(1) הוא הזמן שלוקח למשימה הכוללת על מעבד אחד ואילו t(P) הוא הזמן שלוקח למשימה הכוללת על P מעבדים. כזכור לכם מההכנה חוק Amdhal קובע את החסם להאצה המירבית, $\lim_{P\to\infty} \frac{1}{\frac{1-\alpha}{P}+\alpha} = \frac{1}{\alpha}$, כאשר α הינו החלק הסדרתי בתוכנית. הבהרה - α זו תכונה הנובעת מהקוד שכתבתם. התבוננו בקוד שלכם והבינו אילו חלקים ממנו מבוצעים (כרגע) במקביל ואילו חלקים מבוצעים סדרתית. הפרמטר α מתייחס לחלקים הסדרתיים.

1.6 נסו להעריך מהו החלק α בתוכנית שכתבתם? מהו החסם העליון של ההאצה על פי חוק Amdhal?

בתוכנית שלנו אין מקבול כלל ולכן $\alpha = 1$ ונקבל שהחסם העליון של ה- speedup לפי חוק Amdahl הינו 1

ננסה כעת לבצע במקביל את החיבור הווקטורי (איזה חלק ממנו ניתן לבצע במקביל?) הסירו את ההערה מהשורה HAL_SET_QUOTA(evaluate, N). בעזרת מאקרו זה קבענו שמספר המופעים של הפונקציה evaluate הוא N (כגודל הווקטור).

שנו את הקוד שלכם כך שיבצע את החיבור הווקטורי באופן מקבילי:

א. קבעו את הפונקציה evaluate להיות מסוג duplicable.
ב. השתמשו במאקרו HAL_TASK_INST על מנת לקבל את אינדקס המופע המקבילי של הפונקצייה.
ג. העבירו את החלק שמתבצע באופן סדרתי לפונקציה נוספת, זכרו לעדכן את part1.map בהתאם.

הריצו את התוכנית ובדקו שהתקבלה תוצאה נכונה. מדדו את כמות ה-cycles שנדרשו לביצוע.

מספר הסייקלים ירד ל- $2085 - 1807 = 278$, השיפור לא מאוד משמעותי כיוון שהחלק המקבילי נשאר קטן ובנוסף המערכים אינם גדולים ולכן ה- overhead של ה- scheduler אינו זניח ביחס לחישוב

1.7 חזרו על ההרצה עבור 1,4,8,16,32,64 ליבות. סכמו את התוצאות בטבלה.

# Processors	Run Time	SpeedUp
1	633	1
4	278	2.27
8	218	2.9
16	188	3.37
32	188	3.37
64	188	3.37

1.8 האם תוצאות המדידה הפרקטית שהתקבלו זהים להערכה התיאורטית? אם לא, אילו גורמים הוזנחו לדעתכם בנוסחת חישוב ההאצה המקבילית? בהתייחס לגרסת הקוד 'המשופרת' שבידכם – מהו החלק הסדרתי בתוכנית שכתבתם ואיזה חלק ניתן לבצע במקביל? מהו כעת החסם העליון של ההאצה על פי חוק Amdhal?

תוצאות המדידה הפרקטית יצאו שונים מהערכה התיאורטית.

בהערכה התיאורטית חשבנו ש- $\alpha = 0.5$ כיוון שמיקבלנו בדיוק חצי מהתכנית אך בפועל קיבלנו $\alpha = 0.25 < 0.5$ וזאת כיוון שה- scheduler מקצה tasks ל- cores ולכן כאשר ישנם x cores אז ה- scheduler מקצה כל פעם x tasks ל- x cores ולכן אנו מקבלים מקבול אפקטיבי הכולל בתוכו גם מקבול ה- overhead של ה- scheduler

לפי חוק אמדל נקבל חסם עליון $speedup \to \frac{1}{\alpha} \approx 4$

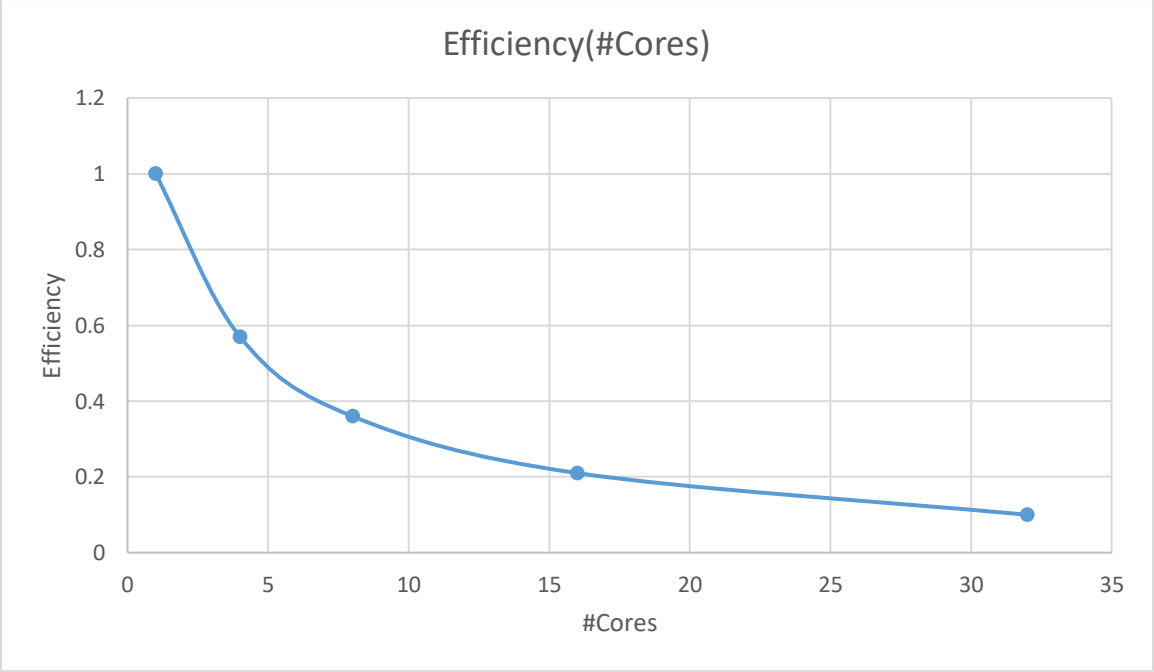
נגדיר **<u>יעילות</u>** באופן הבא:

$\varepsilon(P)=\frac{t(1)}{P\cdot t(P)}$, כאשר t(P) הוא הזמן לביצוע המשימה על P מעבדים. שימו לב כי זהו למעשה מיצוע של ההאצה.

כלומר, עד כמה אנחנו מנצלים את משאבי המערכת.

1.9 חשבו את היעילות עבור 1,4,8,16,32 ליבות והציגו את התוצאה בגרף.

# Processors	Run Time	SpeedUp	Efficiency
1	633	1	1
4	278	2.27	0.57
8	218	2.9	0.36
16	188	3.37	0.21
32	188	3.37	0.1



1.10 מהי היעילות המרבית? האם היעילות תשתנה אם מספר האיברים בווקטור ישתנה?

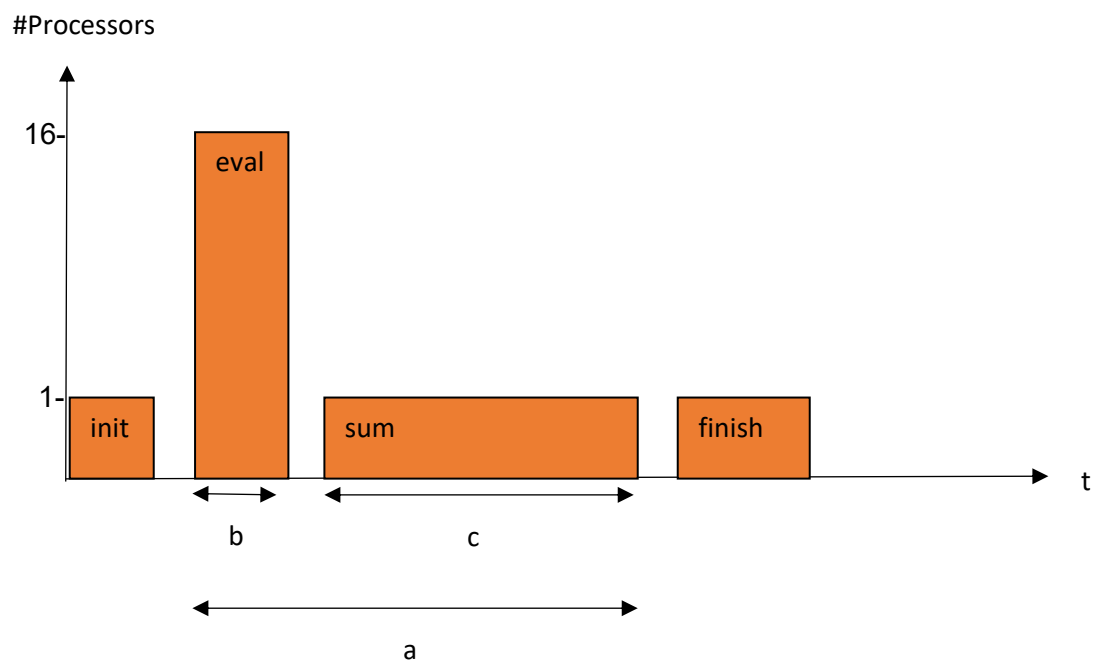
היעילות המקסימלית הינה 1

הגדלת מספר האיברים בווקטור אינה משנה את ה- speedup כיוון שה- a נשאר זהה (תיאורטית) ולכן גם לא משנה את היעילות כיוון שזה מיצוע אפקטיבי של ה- speedup

נגדיר **<u>נצילות</u>** באופן הבא:

$U = \frac{O(P)}{P \cdot T(P)}$, כאשר O(P) הוא זמן העבודה הכולל שמבצעים P מעבדים ו- T(P) הינו זמן הריצה של התוכנית עם P מעבדים.

לדוגמה, בהתייחס לדיאגרמת הבלוקים הבאה –



מתקיים:

$$O(p) = b \cdot 16 + c \cdot 1$$

1.11 $p \cdot T(p) = 16 \cdot a$ חשבו את הנצילות עבור 1,4,8,16,32 ליבות והציגו את התוצאה בגרף.

$$\Rightarrow U = \frac{16b + c}{16a}$$

בכדי למדוד את הזמן b מדדנו את הזמן מתחילת evaluate ועד לתחילת sum והחסרנו 22 סייקלים שהם מחצית מ-43 סייקלים של ה- scheduler שמדדנו בתחילת הדוח, לאחר מכן חישבנו את הנצילות לפי הנוסחה

# Processors	Run Time	SpeedUp	Efficiency	U
1	633	1	1	0.97
4	278	2.27	0.57	0.6
8	218	2.9	0.36	0.42
16	188	3.37	0.21	0.28
32	188	3.37	0.1	0.26

1.12 האם הייתם ממליצים להשתמש במעבד סדרתי בעל תדר גבוה יותר או במעבד מרובה ליבות? מדוע?

נעדיף מעבד סידרתי עם תדר גבוהה כיוון שמהמדידות הקודמות ניתן להבחין כי הנצילות שלנו יורדת עם הגדלת מספר הליבות כיוון ש- sum עדיין רצה כל core יחיד ולכן חלק זה פוגע בנצילות המעבד.

כעת נבחן כיצד ניתן למקבל עוד את התוכנית:

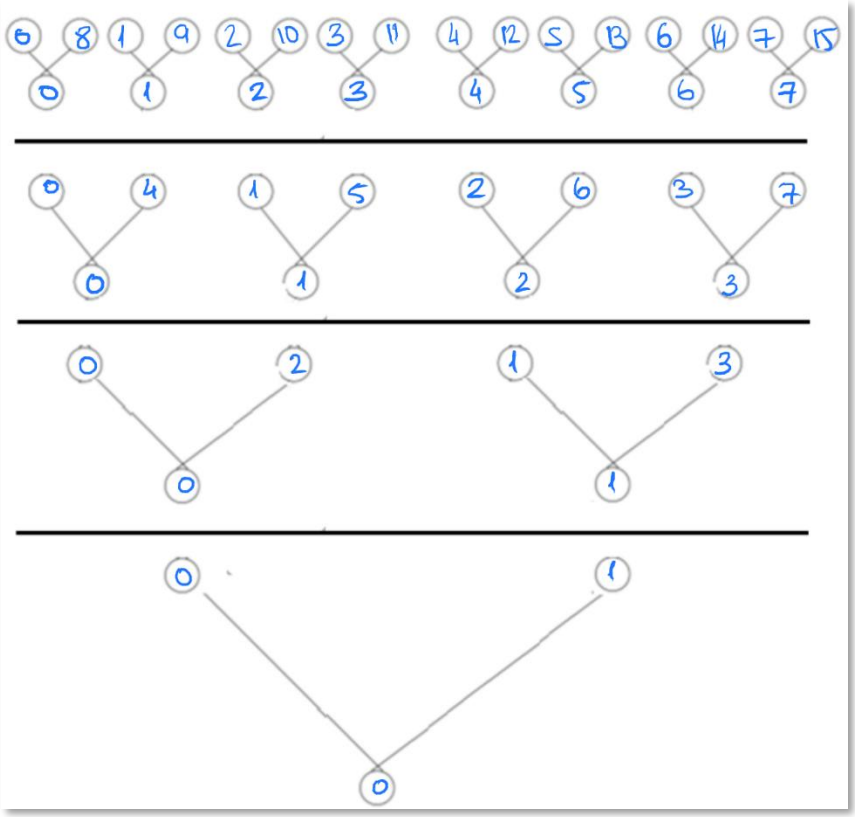
נראה כי ניתן לסכום את איברי ווקטור התוצאה במקביל גם כן. באיטרציה הראשונה נוכל לסכום במקביל 8 זוגות איברים בווקטור, באיטרציה השנייה נסכום 4 זוגות וכו'.

1.13 מהו מספר האיטרציות שירוצו במקביל כפונקציה של גודל הווקטור עד לקבלת סכום כלל איברי הווקטור?

עבור ווקטור בגודל N יהיו לנו $\frac{N}{2}$ איטראציות שירוצו במקביל, כל ליבה תסכום 2 איברים בהנחה ויש לפחות $\frac{N}{2}$ ליבות.

1.14 העזרו באחד האיורים הבאים. מלאו בעיגולים את אינדקסי הווקטור הרלוונטיים / ציירו קווים לסימול חיבור בין איברים:

*הערה – ניתן (ורצוי) לצייר ידנית ולצרף תמונה



1.15 ממשו את חיבורי איברי הווקטור באופן מקבילי.

```
regular task init ()
duplicable task evaluate (init/u)
regular task cond(evaluate | body)
duplicable task body (cond/1)
regular task finish (cond/0)
```

```
/* The init task initialize the quota of task evaluate */
void init (void) {
        /* set r5 to be the number of Active cores  */
        /* in this code r5=1 there for single active core */
        asm volatile (".long 0x81B000A0;"\
            "set 256,%r6;"\
            "set 32,%r5;"\ /* set the number of cores */
            "start_loop:;"\
            "cmp %r5,%r6;"\
            "be end_loop;"\
            "nop;"\
            "sta %r0,[%r5]222;"\
            "ba start_loop;"\
            "add %r5,1,%r5;"\
            "end_loop:"\
            ".long 0x81B000E0;");

        flag = 0;

        /* size update is done before the loop body so we start from 2N */
        size = 2 * N;

        HAL_SET_QUOTA(evaluate, N);
}
```

```c
/* The evaluate task evaluates the add of ExampleArray1[i] and ExampleArray2[i] */
void evaluate (void) {

    int id = HAL_TASK_INST;

    ExampleArray1[id] += ExampleArray2[id];
}

void cond(void) {

    if(size == 1) {
        HAL_TASK_RET_FALSE;
    } else {
        size /= 2;
        HAL_SET_QUOTA(body,size / 2);
        HAL_TASK_RET_TRUE;
    }
}

void body(void) {

    int id = HAL_TASK_INST;
    int stride = size / 2;

    ExampleArray1[id] += ExampleArray1[id + stride];
}

void finish (void) {
    flag = 1;
}
```

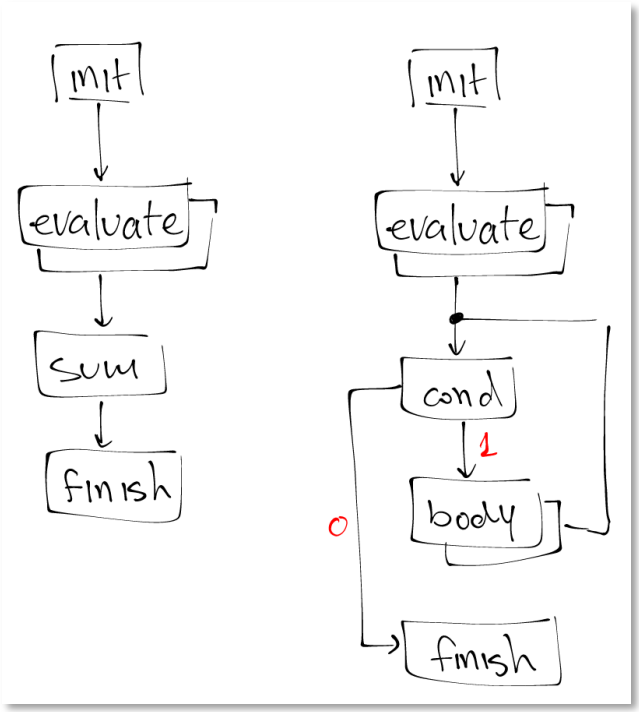
1.16 הריצו את המימוש שלכם ובדקו שהתקבלה תוצאה נכונה. מה זמני ביצוע המתקבלים עבור 1,4,8,16,32 ליבות? בנוסף, בהתייחס לגרסת הקוד המעודכנת, הסיקו מהו החסם העליון של ההאצה על פי חוק Amdhal.

#Processors	Run Time	SpeedUp	Efficiency
1	1439	1	1
4	681	2.11	0.53
8	581	2.48	0.31
16	549	2.62	0.16
32	549	2.62	0.08

בכדי לחשב את הגבול העליון של ה- speedup נצטרך להנדס גרסה מורחבת של Amdahl's law ולאחר משכן להשאיף את $\#Cores$ לאינסוף

1.17 סכמו את התוצאות שהתקבלו עבור המימושים השונים :

1.17.1 בגרף המתאר את דיאגרמת הבלוקים של כל אחד מהמימושים



1.17.2 וכן בגרף שמראה את ההאצה כתלות במספר המעבדים והסבירו את התוצאות שהתקבלו. האם נצפה שיפור מבחינת זמן הריצה עבור גרסת הקוד האחרונה? פרטו סיבות אפשריות לכך וציינו מתי הגרסה האחרונה אכן משפרת את זמן הריצה.

בגרסת הקוד האחרונה נצפה לשיפור מבחינת זמן הריצה בהנחה וה- overhead של ה- scheduler מאוד קטן כפי שמצוין בספר הניסוי כיוון שהוספנו מקביליות לחלק השני של האלגוריתם בשונה מגרסת הקוד הראשונה.

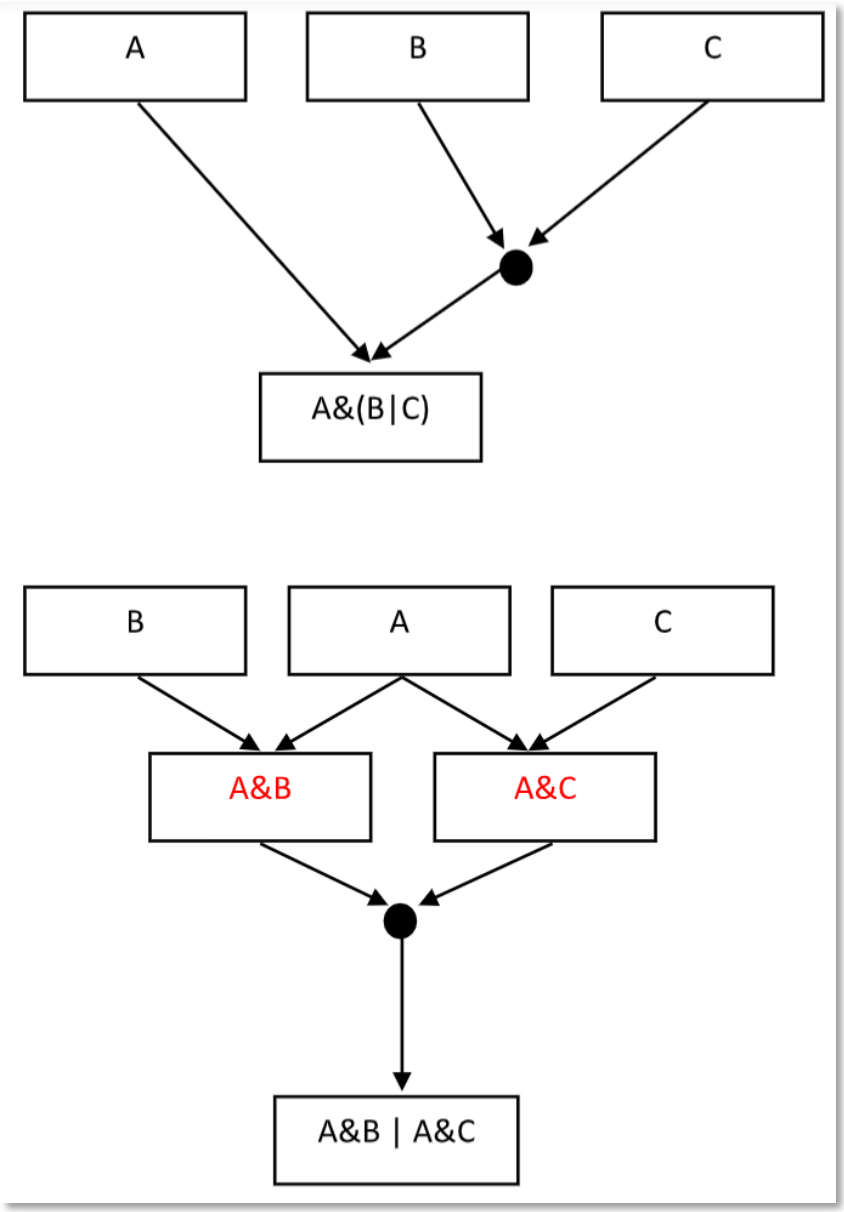
בהנחה וה- overhead של ה- scheduler לא יהיה זניח גרסת הקוד האחרונה תפגע בביצועים עם ריבוי משימות וניהולם.

ניתן להבחין כי ה- speedup בגרסת הקוד האחרונה קטן יותר מגרסת הקוד הראשונה וזאת כיוון שאמנם הביצועים אמורים להשתפר אך כעת יש לנו הרבה יותר משימות ולכן יותר overhead של ניהולם ולכן השיפור ב- speedup הינו פחות משמעותי.

שאלת סיכום:

1. לפי חוקי האלגברה הבוליאנית מתקיים A&(B|C) = A&B | A&C.
   א. ציירו את גרף התלויות עבור שתי הצורות. (העזרו ב-dummy tasks)

*הערה – ניתן (ורצוי) לצייר ידנית ולצרף תמונה



ב. הניחו כי המשימות הסתיימו לפי הסדר הבא: A, B, C (משאמל לימין). כתבו את סדר הריצה של המשימות בשני המקרים. העזרו בטבלת מעקב על בדיקת התנאים של המשימות.

$A \mathbin{\&} (B \mid C)$	$(A \mathbin{\&} B) \mid (A \mathbin{\&} C)$
$1 \mathbin{\&} (0 \mid 0) = 0$	$(1 \mathbin{\&} 0) \mid (1 \mathbin{\&} 0) = 0$
$1 \mathbin{\&} (1 \mid 0) = 1$	$(1 \mathbin{\&} 1) \mid (1 \mathbin{\&} 0) = 1$
$0 \mathbin{\&} (0 \mid 1) = 0$	$(0 \mathbin{\&} 0) \mid (1 \mathbin{\&} 1) = 1$

לא טוב

ג. מה מסקנתכם מתוצאת סעיף ב'.

המסקנה היא שאלגברה בוליאנית אינה מתקיימת כאן, כלומר הערכים אכן בוליאניים אך לא ניתן לשנות את הביטויים לפי אקסיומות בוליאניות בגלל מבנה החומרה אשר בודקת את התנאים.

<u>שעת סיום הניסוי</u>:

13:30

מהלך הניסוי – חלק ב'

בחלק זה של הניסוי נבדוק את השפעת מתודולוגיית התוכנה על מעבד מרובה-ליבות על ביצועי המערכת. בפרט נתמקד בנושא הגרנולאריות ותזמון/סנכרון, כפי שכבר יצא לכם לראות בחלק א'.

שעת התחלת הניסוי:

9:00

בחלק זה נתנסה במימוש אלגוריתם White Balance המשמש לאיזון צבעים בתמונה כך שצבעים נייטלרים יוצגו באופן תקין. האלגוריתם מבצע זאת על ידי שינוי העוצמות של הצבעים המרכיבים את התמונה (בדרך כלל אדום, ירוק וכחול) כך שהתמהיל הכולל שלהם ישתנה באופן שבו צבע נייטרלי יופיע באופן נכון או נעים יותר לעין.

Figure1 - Color balance example

באיור 28 ניתן לראות דוגמא לתמונה מקורית מצד שמאל והתמונה שמתקבלת אחרי ביצוע Color balance כך הפני השטח האפורים ייראו נייטרלים באותה תאורה.

לפניכם שלד תוכנית המבצעת White Balance לתמונה Lenna (זהו חלק מתמונת האמצע של המגזין Playboy מנובמבר 1972 המציג את פניה של הדוגמנית השבדית Lena Soderberg, תמונה זו משמשת לרוב לבדיקה של אלגוריתמי עיבוד תמונה שונים).

תוכן התמונה נתון ב-3 מטריצות בגודל 128x128 בשמות lennaR, lennaG, lennaB עבור צבעי האדום, ירוק וכחול בהתאמה. ערכי מטריצות אלה מוגדרות בקבצים lennaR.h, lennaG.h, lennaB.h.

```
/* Header file for all HAL specific macros */
#include "C:\cygwin\home\plurality\include\hal.h"
#include "top_enums.h"

/* source code for white balance */
#include "lennaR.h"
#include "lennaG.h"
#include "lennaB.h"
```

```
/** Tasks **/
void init(void) HAL_DEC_TASK;

void rbalance (void) HAL_DEC_TASK;
void gbalance (void) HAL_DEC_TASK;
void bbalance (void) HAL_DEC_TASK;

void finish (void) HAL_DEC_TASK;
/***********/

/** Defines **/
#define DIM 128

#define RFACTOR 1.017f
#define GFACTOR 1
#define BFACTOR 1.347f
/*************/

/** Globals **/
unsigned char rresult[DIM][DIM] = { 0 };
unsigned char gresult[DIM][DIM] = { 0 };
unsigned char bresult[DIM][DIM] = { 0 };

int finish_flag;
/*************/

void init(void) {
	/* set r5 to be the number of Active cores  */
	/* in this code r5=1 there for single active core */
	asm volatile (".long 0x81B000A0;"
				"set 256,%r6;"
				"set 1,%r5;"
				"start_loop:;"
				"cmp %r5,%r6;"
				"be end_loop;"
				"nop;"
				"sta %r0,[%r5]222;"
				"ba start_loop;"
				"add %r5,1,%r5;"
				"end_loop:"
				".long 0x81B000E0;");

	//HAL_SET_QUOTA(rbalance, DIM);
	//HAL_SET_QUOTA(gbalance, DIM);
	//HAL_SET_QUOTA(bbalance, DIM);

	finish_flag = 0;
}
```

```
void rbalance(void) {


}

void gbalance(void) {


}

void bbalance(void) {


}

void finish(void) {
        finish_flag = 1;
}
```

מימוש האלגוריתם הינו הכפלה של רכיב צבע במטריצה בגורם המתאים (RFACTOR, GFACTOR, BFACTOR).

2.1

א) בכדי לחשב את תוצאת המכפלה עלינו להחליט מה נעשה כשהתוצאה תחרוג מטווח המספרים הרלוונטי. קיימת אפשרות להשתמש בwrap around ואז נקבל את המספר. modulu255 של הבעייתיות עם שימוש ב wrap around זה שבמקום לקבל למשל 256 שמייצג צבע בהיר נקבל את הספרה 0 שמייצגת צבע כהה, ולכן אנו נשתמש בשיטת ה saturation, לפיה כאשר תוצאת המכפלה גבוהה מ255 נקבע את התוצאה להיות 255.

לפניכם שני קטעי קוד שמראים שימוש בsaturation -

```
Code A :

x=pixel*factor;
if (pixel*factor>255) x=255;
```

```
Code B :

tmp=pixel*factor;
if (tmp>255) {x=255;}
else{x=tmp;}
```

מה ההבדל בין שתי גרסאות הקוד? נסו לחשוב מה יהיה זמן הריצה בשימוש בכל אחת מהגרסאות והאם זמני הריצה ישתנו אם נחליף בין תמונות בגודל זהה.

במימוש B ביצוע פעולת הכפל מבוצע פעם אחת ואילו במימוש A מבצעים אותה פעמיים.

פעולת כפל היא פעולה מורכבת שצורכת מספר גדול של סייקלים לעומת פעולות פשוטות יותר ולכן מימוש B יהיה מהיר יותר.

בהנחה והשמה מתבצעת לרגיסטר ומפועפעת לזיכרון פעם אחת בסוף הריצה אז מספר ההשמות זניח לעומת פעולת הכפל.

מעתה היעזרו בגרסת הקוד השנייה עבור מימוש saturation.

ב) השלם את האלגוריתם הסדרתי.

נעשה

ג) צרף את הקוד המתקבל לדו"ח המסכם.

```
void rbalance(void) {

    int tmp, pixel;

    for (int i = 0; i < DIM; i++) {
        for (int j = 0; j < DIM; j++) {

            pixel = lennaR[i][j];
            tmp = pixel * RFACTOR ;

            if (tmp>255)
                x=255;
            else
                x=tmp;

            rresult[i][j] = x;
        }
    }
}

void gbalance(void) {

    int tmp, pixel;

        for (int i = 0; i < DIM; i++) {
            for (int j = 0; j < DIM; j++) {

                pixel = lennaG[i][j];
                tmp = pixel * GFACTOR ;

                if (tmp>255)
                    x=255;
                else
                    x=tmp;

                gresult[i][j] = x;
            }
        }
}
```

```
void bbalance(void) {

    int tmp, pixel;

        for (int i = 0; i < DIM; i++) {
            for (int j = 0; j < DIM; j++) {

                pixel = lennaB[i][j];
                tmp = pixel * BFACTOR ;

                if (tmp>255)
                    x=255;
                else
                    x=tmp;

                bresult[i][j] = x;
            }
        }
}
```

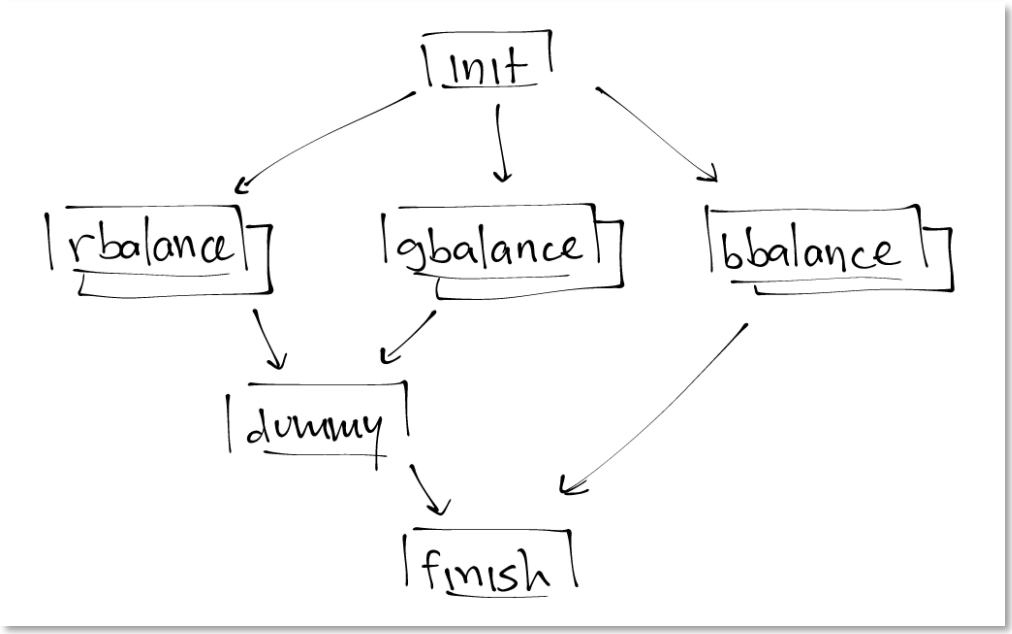
2.2 ממשו את האלגוריתם באופן מקבלי.

א) שנו את קובץ ה-part2.map בהתאם וצרפו אותו לדו"ח.

```
regular task init ()
duplicable task rbalance (init/u)
duplicable task gbalance (init/u)
duplicable task bbalance (init/u)
regular task dummy1 (rbalance & gbalance)
regular task finish (dummy1/u & bbalance)
```

ב) ציירו את גרף התלויות החדש.

*הערה – ניתן (ורצוי) לצייר ידנית ולצרף תמונה



ג) שנו את הקוד כך שיתבצע באופן מקבילי וצרפו אותו לדו"ח.

```c
void init(void) {
    /* set r5 to be the number of Active cores  */
    /* in this code r5=1 there for single active core */
    asm volatile (".long 0x81B000A0;"
            "set 256,%r6;"
            "set 128,%r5;"
            "start_loop:;"
            "cmp %r5,%r6;"
            "be end_loop;"
            "nop;"
            "sta %r0,[%r5]222;"
            "ba start_loop;"
            "add %r5,1,%r5;"
            "end_loop:"
            ".long 0x81B000E0;");

    HAL_SET_QUOTA(rbalance, DIM);
    HAL_SET_QUOTA(gbalance, DIM);
    HAL_SET_QUOTA(bbalance, DIM);

    finish_flag = 0;
}

void rbalance(void) {

    int i, tmp, pixel, x;
    int id = HAL_TASK_INST;


    for (i = 0; i < DIM; i++) {

        pixel = lennaR[id][i];
        tmp = pixel * RFACTOR ;

        if (tmp>255)
            x=255;
        else
            x=tmp;

        rresult[id][i]  = x;
    }
}
```

```c
void gbalance(void) {

    int i, tmp, pixel, x;
    int id = HAL_TASK_INST;

    for (i = 0; i < DIM; i++) {

        pixel = lennaG[id][i];
        tmp = pixel * GFACTOR ;

        if (tmp>255)
            x=255;
        else
            x=tmp;

        gresult[id][i]  = x;
    }
}

void bbalance(void) {

    int i, tmp, pixel, x;
    int id = HAL_TASK_INST;

    for (i = 0; i < DIM; i++) {

        pixel = lennaB[id][i];
        tmp = pixel * BFACTOR ;

        if (tmp>255)
            x=255;
        else
            x=tmp;

        bresult[id][i] = x;
    }
}
```

ד) הריצו את התוכנית. בדקו שהתקבלה תוצאה נכונה. צרפו את התמונה המקורית וזו שמתקבלת לאחר הרצת התוכנית.

על מנת להציג את תמונות (המקור והתוצאה) נבצע את הדברים הבאים:

בעת ריצת הסימולאטור כתבו את השורות הבאות:

```
dump value lennaB.dmp lennaB
dump value lennaR.dmp lennaR
dump value lennaG.dmp lennaG
```

פקודות אלו מורות לסימולאטור לשמור את תוכן המטריצות המכילות את צבעי היסוד של התמונה לתוך קבצים בינאריים (לדוגמא: המטריצה lennaB לתוך הקובץ הבינארי lennaB.dmp)

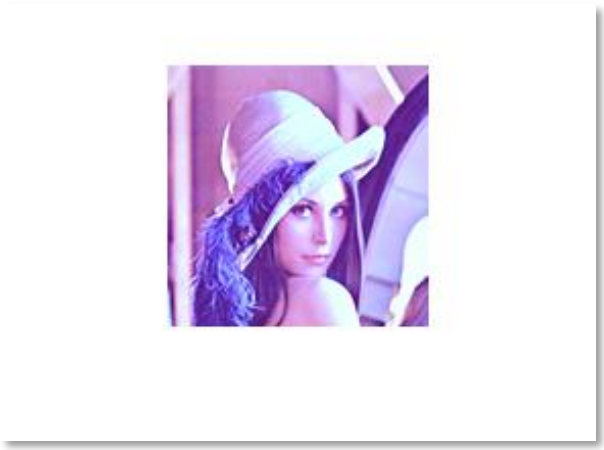
הקבצים יופיעו בתיקייה:

```
C:\Users\shmueli\workspace_new\zylin-cdt-debugging
```

פתחו Matlab, הכנסו לתיקייה והריצו את קוד ה-Matlab הבא:

```
dim = 128;
rfname = 'lennaR.dmp';
gfname = 'lennaG.dmp';
bfname = 'lennaB.dmp';

rfid = fopen(rfname);
img(:,:,1) = fread(rfid, [dim,dim]);
img(:,:,1) = img(:,:,1)';
rfid = fopen(gfname);
img(:,:,2) = fread(rfid, [dim,dim]);
img(:,:,2) = img(:,:,2)';
rfid = fopen(bfname);
img(:,:,3) = fread(rfid, [dim,dim]);
img(:,:,3) = img(:,:,3)';

img = uint8(img);
imshow(img);
```

קוד ה-Matlab נמצא בקובץ show_original.m. הריצו אותו והוסיפו את התמונה המקורית לדו"ח המסכם.

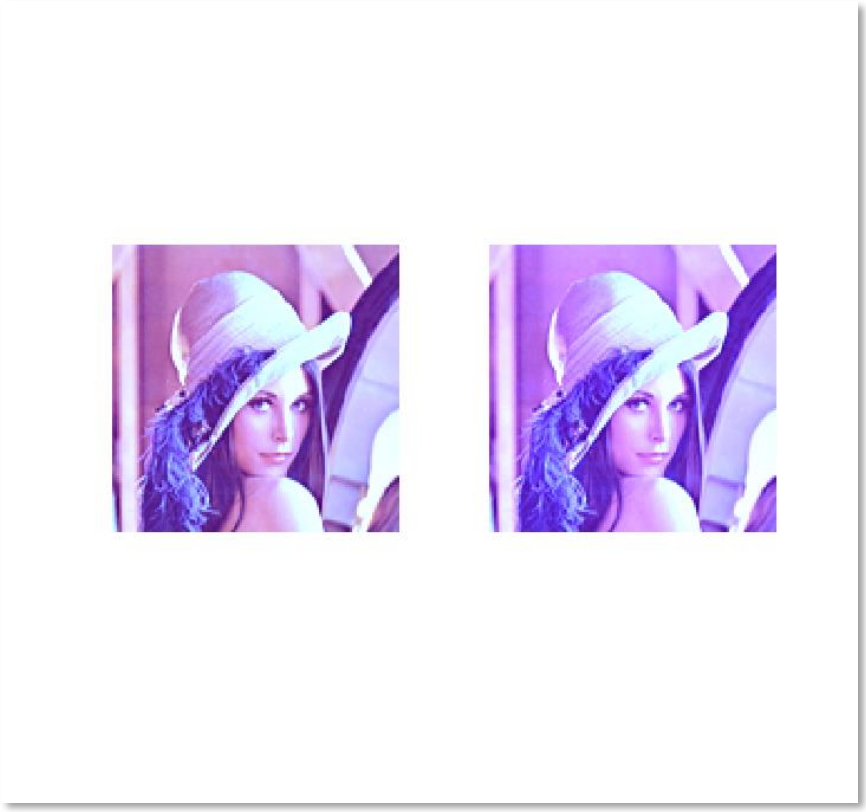


בצורה דומה נשמור לקבצים את המטריצות המתקבלות בסוף ריצת האלגוריתם:

```
dump value rresult.dmp result
dump value gresult.dmp gresult
dump value bresult.dmp bresult
```

קוד ה-Matlab להצגת תמונת התוצאה נמצא בקובץ show_result.m.

ה) הוסיפו את תמונת התוצאה לדו"ח המסכם.



הקפידו למחוק את קבצי ה-dump לפני כל שינוי שאתם מבצעים בקוד, זאת כדי להיות בטוחים שאתם מציגים את התוצאות הנוכחיות ולא את תוצאות הריצה הקודמת.

ו) מדדו את זמני הביצוע עבור 128 ליבות, וציירו דיאגרמת בלוקים מתאימה.

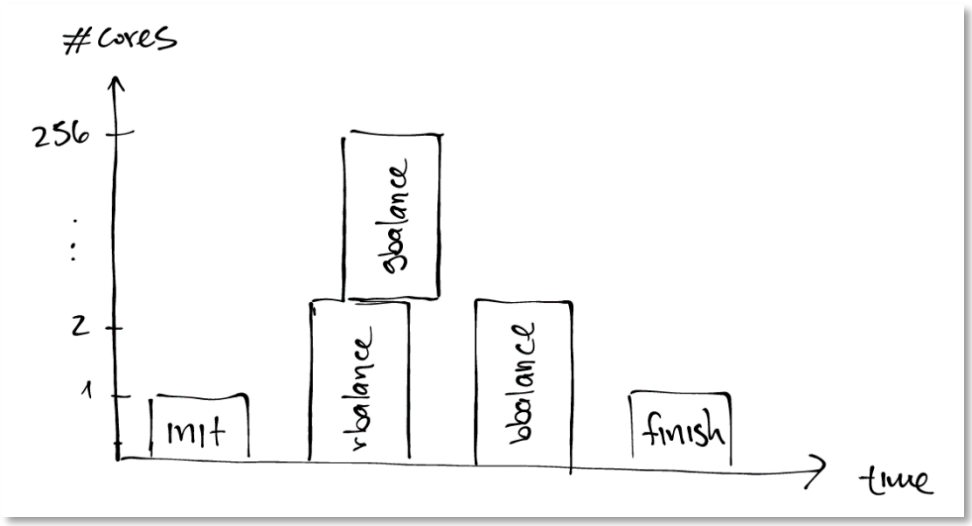
זמן הביצוע שקיבלנו הינו $10682 - 948 = 9734$

דיאגרמת הבלוקים תלויה ב- scheduler, ריצה **אפשרית** הינה הנ"ל בהנחה שמספר הליבות במעבד הינו לפחות 128.

כיצד תראה לדעתכם דיאגרמת הבלוקים עבור 256 ליבות? ציירו .

מהו הspeedup שתצפו לקבל במקרה זה

בהנחה וה- overhead של ה- scheduler וכן פעולות init, finish זניחות יחסית לחישוב של Xbalance יהיו לנו 2 יחידות זמן במקום 3 יחידות זמן עבור 128 ליבות ולכן נצפה לקבל $speedup = \frac{3}{2}$



ז) מדדו את זמני הביצוע עבור 256 ליבות.מהו ה speedup שהתקבל? האם זה תואם ל speedup שצפיתם בסעיף הקודם?

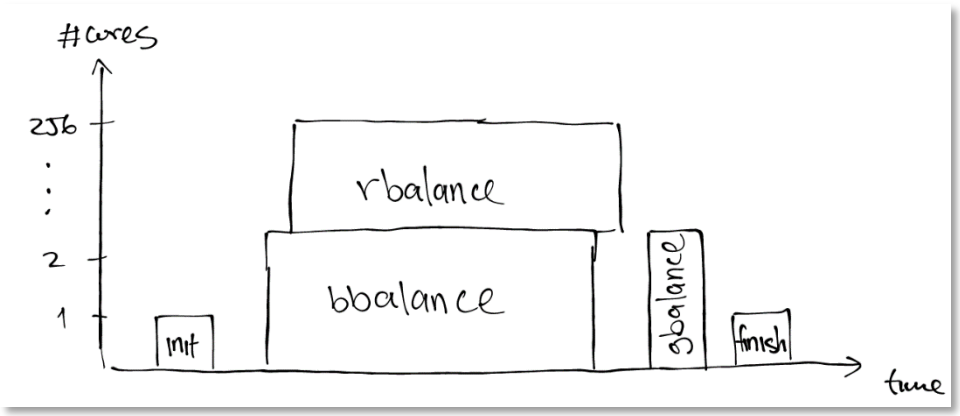
# Processors	Cycles
128	9734
256	4977

קיבלנו $speedup = 1.95$ וזה גדול משמעותית ממה שציפינו.

נשים לב ש- rbalance, bbalance מכפילים ב- floating point factor כלומר מתבצע ב- floating point aritmethic unit שלא בהכרח נמצא על כל מעבד ואילו gbalance מכפיל ב- integer factor כלומר מתבצע ב- ALU.

המסקנה היא שחישוב gbalance זניח ביחס ל-2 האחרים ולכן עבור 128 ליבות יש לנו 2 יחידות זמן (עם ההזנחות מהסעיף הקודם) ועבור 256 ליבות יש לנו יחידות זמן יחידה ולכן אנו מקבלים $speedup \approx 2$

דיאגרמת בלוקים קרובה יותר למציאות תראה כך:



נשים לב לעובדה שאנו מכפילים תמיד בפקטור זהה. ישנם סה"כ 256 גוונים אפשריים של כל צבע יסוד (גודל הייצוג הוא 8-bit). בפתרון הקודם שלנו אנו מבצעים 16384 = 128*128 פעולות כפל עבור כל צבע.

נוכל לצמצם באופן משמעותי את כמות הכפלים שאנו מבצעים על ידי בניית Lookup Table (LUT) לכל צבע המכיל עבור כל צבע את התוצאה המעודכנת.

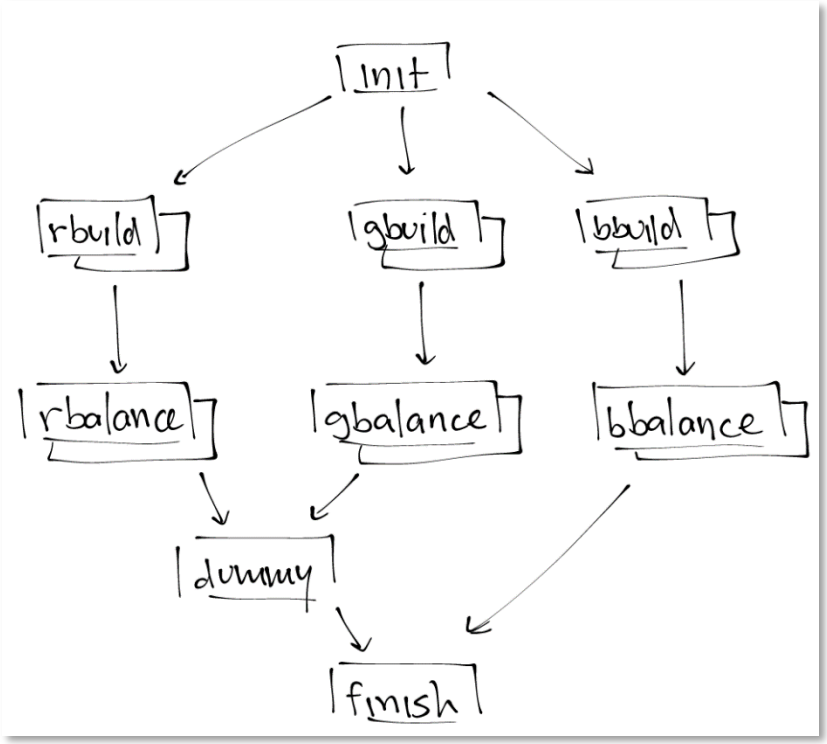
העזרו בשלד הקוד הבא:

```
#define COLORS 256

/** Globals **/
unsigned char rlut[COLORS] = {0};
unsigned char glut[COLORS] = {0};
unsigned char blut[COLORS] = {0};

void rbuildlut (void) HAL_DEC_TASK;
void gbuildlut (void) HAL_DEC_TASK;
void bbuildlut (void) HAL_DEC_TASK;
```

2.3 ממשו את הפתרון באמצעות Lookup Table.

א) ציירו את גרף התלויות החדש. עדכנו את קובץ ה-map בהתאם. צרפו אותם לדו"ח המסכם.

*הערה – ניתן (ורצוי) לצייר ידנית ולצרף תמונה




```
regular task init ()

duplicable task rbuildlut  (init/u)
duplicable task gbuildlut  (init/u)
duplicable task bbuildlut  (init/u)

duplicable task rbalance (rbuildlut)
duplicable task gbalance (gbuildlut)
duplicable task bbalance (bbuildlut)

regular task dummy1 (rbalance & gbalance)
regular task finish (dummy1/u & bbalance)
```

ב) עדכנו את הקוד לפתרון המוצע. בדקו את תקינות ריצתו. צרפו את הקוד לדו"ח המסכם.

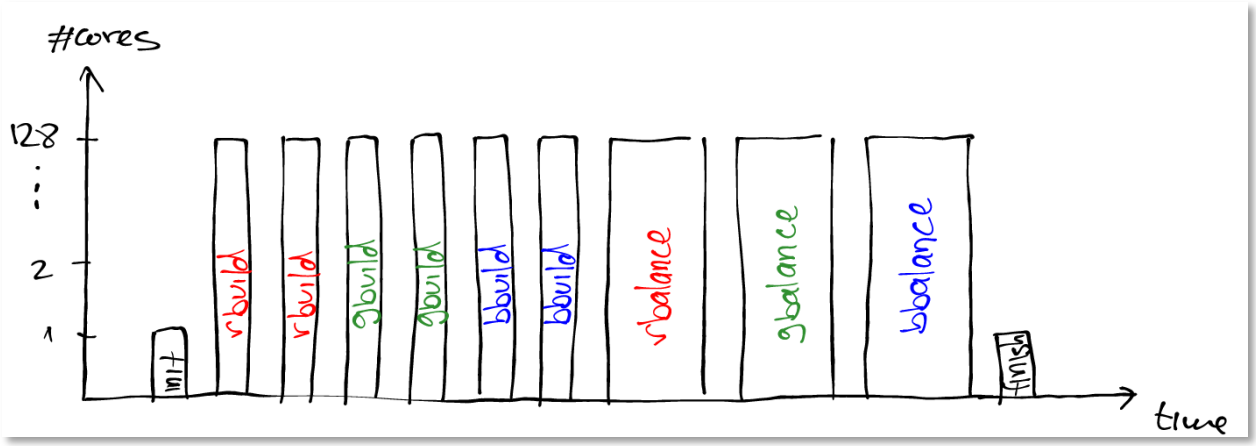
```
unsigned char rlut[COLORS] = {0};
unsigned char glut[COLORS] = {0};
unsigned char blut[COLORS] = {0};



int finish_flag;
/*************/

void dummy1 (void) {}

void init(void) {
    /* set r5 to be the number of Active cores  */
    /* in this code r5=1 there for single active core */
    asm volatile (".long 0x81B000A0;"
            "set 256,%r6;"
            "set 128,%r5;"
            "start_loop:;"
            "cmp %r5,%r6;"
            "be end_loop;"
            "nop;"
            "sta %r0,[%r5]222;"
            "ba start_loop;"
            "add %r5,1,%r5;"
            "end_loop:"
            ".long 0x81B000E0;");


    HAL_SET_QUOTA(rbuildlut, COLORS);
    HAL_SET_QUOTA(gbuildlut, COLORS);
    HAL_SET_QUOTA(bbuildlut, COLORS);

    HAL_SET_QUOTA(rbalance, DIM);
    HAL_SET_QUOTA(gbalance, DIM);
    HAL_SET_QUOTA(bbalance, DIM);

    finish_flag = 0;
}
```

```
void rbuildlut (void){

    int id = HAL_TASK_INST;
    int tmp = id * RFACTOR ;

    if (tmp>255)
        tmp=255;

    rlut[id] = tmp;
}

void gbuildlut (void){

    int id = HAL_TASK_INST;
    int tmp = id * GFACTOR ;

    if (tmp>255)
        tmp=255;

    glut[id] = tmp;
}

void bbuildlut (void){

    int id = HAL_TASK_INST;
    int tmp = id * BFACTOR ;

    if (tmp>255)
        tmp=255;

    blut[id] = tmp;
}
```

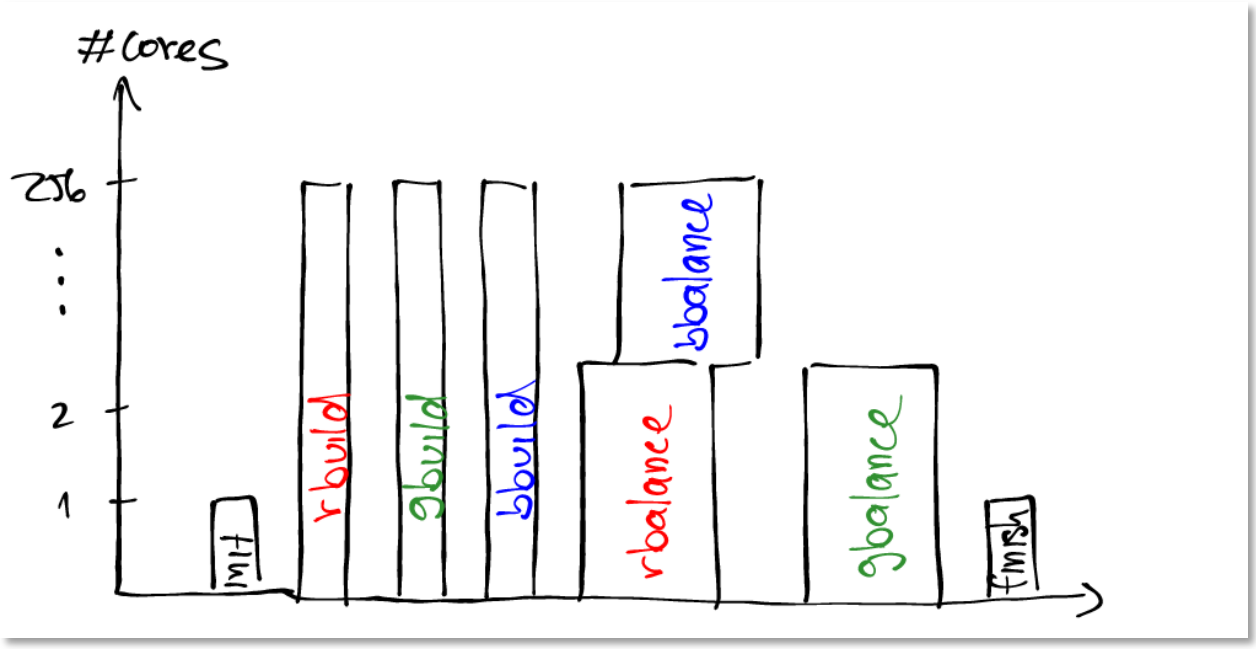
```
void rbalance(void) {

    int i, pixel;
    int id = HAL_TASK_INST;

    for (i = 0; i < DIM; i++) {
        pixel = lennaR[id][i];
        rresult[id][i]  = rlut[pixel];
    }
}

void gbalance(void) {

    int i, pixel;
    int id = HAL_TASK_INST;

    for (i = 0; i < DIM; i++) {
        pixel = lennaG[id][i];
        gresult[id][i]  = glut[pixel];
    }
}

void bbalance(void) {

    int i, pixel;
    int id = HAL_TASK_INST;


    for (i = 0; i < DIM; i++) {
        pixel = lennaB[id][i];
        bresult[id][i]  = blut[pixel];
    }
}
```

ג) מדדו את זמני הביצוע עבור 128 ליבות, וציירו דיאגרמת בלוקים מתאימה.

קיבלנו 6218 סייקלים עבור 128 ליבו



כיצד תראה לדעתכם דיאגרמת הבלוקים עבור 256 ליבות? ציירו .



מהו הspeedup שתצפו לקבל במקרה זה?

נצפה לקבל $\frac{3}{2} < speedup < 2$ כיוון ששלב בניית הטבלאות מתבצע פי 2 יותר מהר ושלב עיבוד התמונה פי $\frac{3}{2}$ יותר מהר

ד)מדדו את זמני הביצוע עבור 256 ליבות.מהו ה speedup שהתקבל? האם זה תואם ל speedup שצפיתם בסעיף הקודם? נסו להסביר את ההבדל בהאצה על סמך התוצאות שהתקבלו בסעיף זה ובסעיף ו בשאלה 2.2.

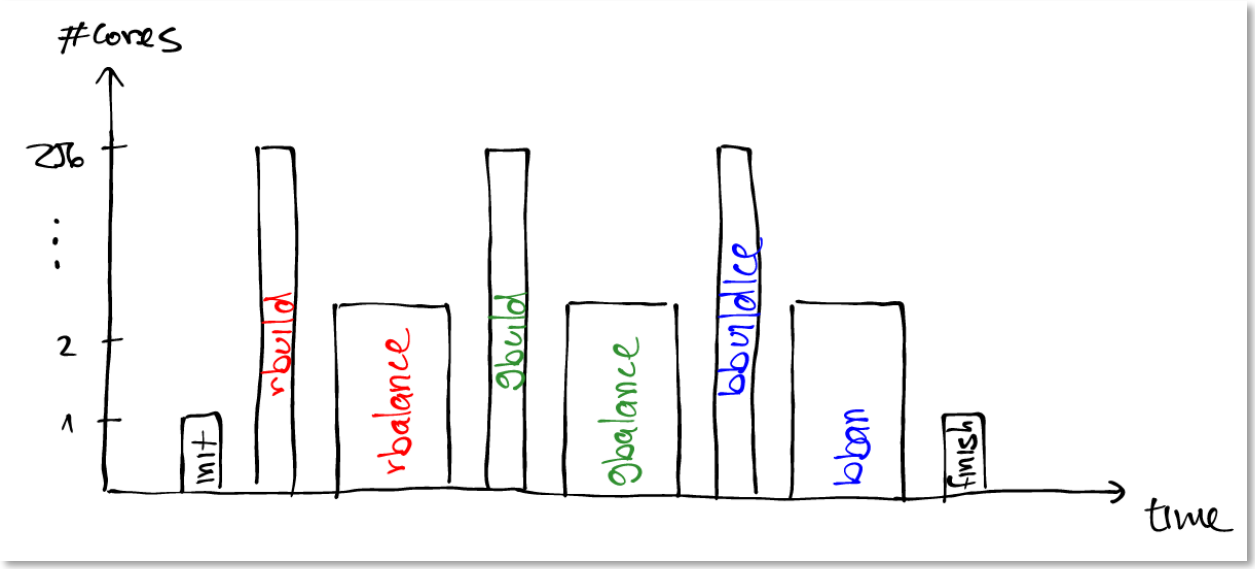
# Processors	Cycles
128	6218
256	4363

קיבלנו $speedup = 1.43$ וזה אינו תואם את הציפיות שלנו, כנראה שה- scheduler תזמן את המשימות בסדר אחר שאינו בהכרח אופטימלי.

ה) האם דיאגרמת הבלוקים שציירתם עבור 256 ליבות היא בהכרח מה שמתבצע בפועל? אם לא ציירו ווריאציות נוספות.

*הערה – ניתן (ורצוי) לצייר ידנית ולצרף תמונה

כפי שהסברנו בסעיף הקודם דיאגרמת הבלוקים של הסעיף הקודם אינה בהכרח מה שמתבצע בפועל, ווריאציה נוספת אפשרית הינה:



ו) בהסתמך על דיאגרמת הבלוקים של 256 ליבות, מה הייתם משנים בקוד כדי לשפר את הנצילות?

נשים לב כי זמן הריצה הכולל מושפע משני גורמים –

סינכרוניזציה – משפיע על משך הזמן בו המעבדים אינם מבצעים פקודות של התוכנית.

גרנולאריות – אורך ממוצע של משימה בתוכנית (משפיע על מספר ההחלפות של משימות פר מעבד).

תחילה נשנה את אופן בניית הטבלאות באופן כזה שאותו task מחשב את ערך הפיקסל עבור 3 הצבעים במקום עבור צבע יחיד ובכך אנחנו חוסכים זמן סינכרוניציה, בנוסף את עיבוד התמונה נחלק ל-256 משימות במקום 128 כך שכל משימה תטפל בחצי שורה במקום שורה שלמה ובכך ננצל את כל המעבדים במערכת.

אמנם השיפור השני מייצר לנו זמן סנכרון נוסף אך הוא גם מקטין לנו את הגרנולריות מה שיאפשר load balancing טוב יותר ובכך נשפר ביצועים ואת נצילות המעבד.

ז)

1) זהו בדיאגרמה שציירתם עבור 256 ליבות אלו זמנים מבוזבזים הם תוצאה של איזה אחד משני הגורמים.

הזמנים המתבזבזים בהחלפות הקשר הן הסנכרון והזמנים המתבזים בסוף כאשר המעבדים שסיימו ממתינים לאלו שעדיין רצים נובעת בגרנולריות גסה מדי

2) מהם האמצעים העומדים לרשותנו בתכנות מעבד ה-HAL על מנת לשלוט בגורמים שזיהיתם.

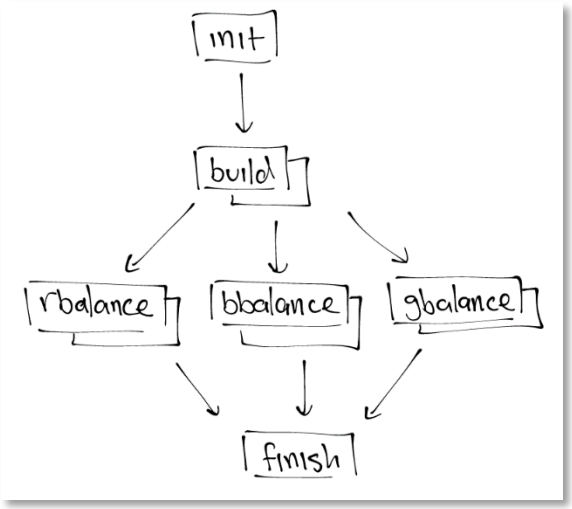
ניתן לשלוט במספר המופעים של כל משימה ובמספר הליבות (עד גבול מסוים) ובכך לשפר את הגרנולריות וניתן להרחיב את הפונקציונליות של משימות ובכך לצמצם את מספר המשימות ולחסוך בזמני סנכרון

2.4 דונו עם מדריך המעבדה על אפשרויות לשפר עוד את זמן הריצה על סמך התוצאות שקיבלתם בסעיף הקודם.

בצעו לפחות 3 מחזורי שיפור תוצאות. עבור כל שיפור יבוצע ניתוח נפרד.

2.4.1

א) ציירו את גרף התלויות החדש (במידה וקיים).



ב) עדכנו את הקוד ובדקו את תקינות ריצתו.

```
regular task init ()
duplicable task buildlut  (init/u)
duplicable task rbalance (buildlut)
duplicable task gbalance (buildlut)
duplicable task bbalance (buildlut)
regular task dummy1(rbalance & gbalance)
regular task finish (dummy1/u & bbalance)
```

```
void buildlut (void){

    int tmp1,tmp2,tmp3;
    int id = HAL_TASK_INST;

    tmp1 = id * RFACTOR ;
    tmp2 = id * GFACTOR ;
    tmp3 = id * BFACTOR ;

    if (tmp1>255) tmp1=255;
    if (tmp2>255) tmp2=255;
    if (tmp3>255) tmp3=255;

    rlut[id] = tmp1;
    glut[id] = tmp2;
    blut[id] = tmp3;

}
```

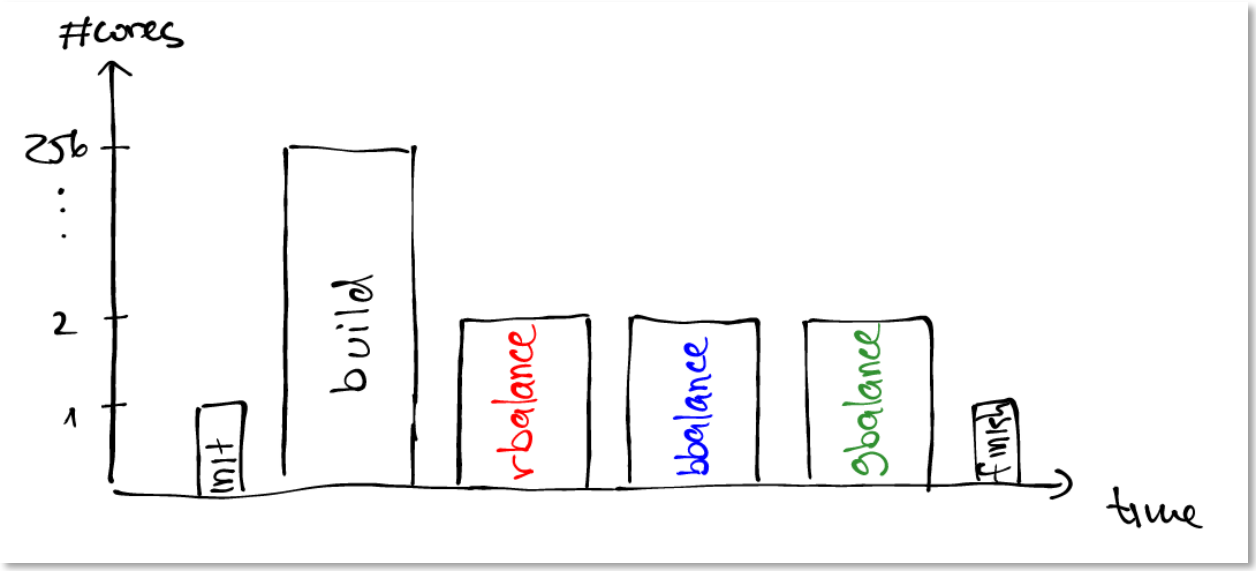
```
void rbalance(void) {

    int i, pixel;
    int id = HAL_TASK_INST;


    for (i = 0; i < DIM; i++) {
        pixel = lennaR[id][i];
        rresult[id][i]  = rlut[pixel];
    }
}

void gbalance(void) {

    int i, pixel;
    int id = HAL_TASK_INST;


    for (i = 0; i < DIM; i++) {
        pixel = lennaG[id][i];
        gresult[id][i]  = glut[pixel];
    }
}

void bbalance(void) {

    int i, pixel;
    int id = HAL_TASK_INST;


    for (i = 0; i < DIM; i++) {
        pixel = lennaB[id][i];
        bresult[id][i]  = blut[pixel];
    }
}
```

ג) מדדו את זמני הריצה עבור 256 ליבות.

# Processors	Cycles
256	4440

קצת פגענו בביצועים אבל הנ"ל ישתפר באיטראציות הבאות

ד) ציירו את דיאגרמת הבלוקים המתאימה.



ה) מהי הגרנולאריות שבא אתם עובדים?

בשלב ה- build יש לנו גרנולריות של 3 (אחד לכל צבע) ובשלב ה- xbalance גרנולריות של 128

2.4.2

א) ציירו את גרף התלויות החדש (במידה וקיים).

גרף התלויות לא השתנה

ב) עדכנו את הקוד ובדקו את תקינות ריצתו.

```
void rbalance(void) {

    int i, pixel, start;
    int id = HAL_TASK_INST;
    int row = id/2;

    if(id%2 == 0)
        start = 0;
    else
        start = DIM/2;

    for (i = start; i < start + DIM/2; i++) {
        pixel = lennaR[row][i];
        rresult[row][i]  = rlut[pixel];
    }
}

void gbalance(void) {

    int i, pixel, start;
    int id = HAL_TASK_INST;
    int row = id/2;

    if(id%2 == 0)
        start = 0;
    else
        start = DIM/2;

    for (i = start; i < start + DIM/2; i++) {
        pixel = lennaG[row][i];
        gresult[row][i]  = glut[pixel];
    }
}
```

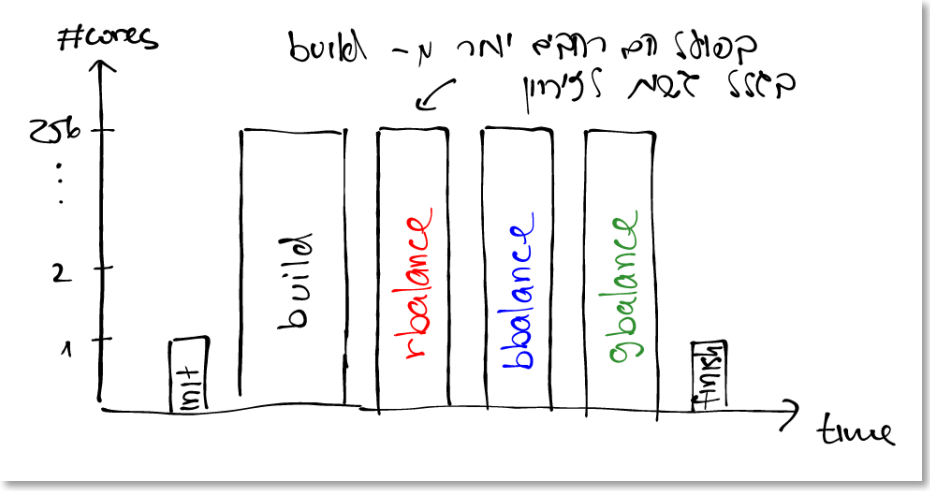
```
void bbalance(void) {

    int i, pixel, start;
    int id = HAL_TASK_INST;
    int row = id/2;

    if(id%2 == 0)
        start = 0;
    else
        start = DIM/2;

    for (i = start; i < start + DIM/2; i++) {
        pixel = lennaB[row][i];
        bresult[row][i]  = blut[pixel];
    }
}
```

```
HAL_SET_QUOTA(rbalance, DIM*2);
HAL_SET_QUOTA(gbalance, DIM*2);
HAL_SET_QUOTA(bbalance, DIM*2);
```

ג) מדדו את זמני הריצה עבור 256 ליבות.

# Processors	Cycles
256	3888

ד) ציירו את גרף הביצוע.

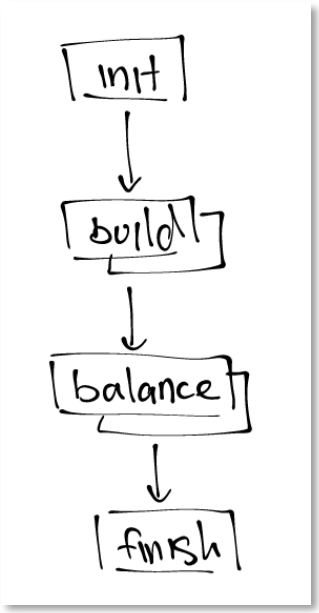


ה) מהי הגרנולאריות שבא אתם עובדים?

בשלב ה- build יש לנו גרנולריות של 3 (אחד לכל צבע) ובשלב ה- xbalance גרנולריות של 64

2.4.3

א) ציירו את גרף התלויות החדש (במידה וקיים).



ב) עדכנו את הקוד ובדקו את תקינות ריצתו.

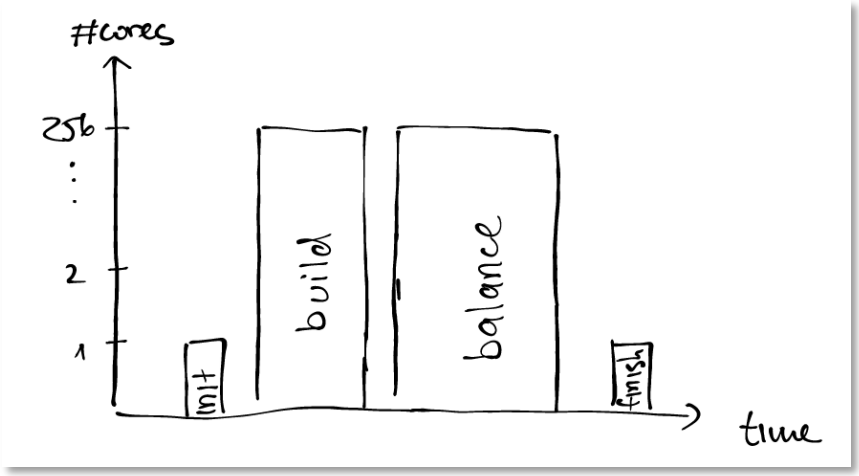
```
regular task init ()
duplicable task buildlut  (init/u)
duplicable task balance (buildlut)
regular task finish (balance)
```

```
void balance(void) {

    int i, pixel1,pixel2,pixel3;

    int start;

    int id = HAL_TASK_INST;
    int row = id2;

    if(id%2 == 0)
        start = 0;
    else
        start = DIM/2;

    for (i = start; i  start + DIM/2; i++) {
        pixel1 = lennaR[row][i];
        pixel2 = lennaG[row][i];
        pixel3 = lennaB[row][i];

        rresult[row][i]  = rlut[pixel1];
        gresult[row][i]  = glut[pixel2];
        bresult[row][i]  = blut[pixel3];
    }
}
```

ג) מדדו את זמני הריצה עבור 256 ליבות.

# Processors	Cycles
256	2881

ד) ציירו את דיאגרמת הבלוקים המתאימה.



ה) מהי הגרנולאריות שבה אתם עובדים?

בשלב ה- build יש לנו גרנולריות של 3 (אחד לכל צבע) ובשלב ה- xbalance גרנולריות של 192 (64 לכל צבע)

2.5 סכמו את תוצאות תהליך האופטימיזציה שביצעתם לאלגוריתם ואת מסקנותיכם בקשר ל-tradeoff שבין גודל המשימה לבין איזון העומסים – מתי נקבל זמן ריצה כולל אופטימלי? נצילות מירבית?

בשלב הראשון איחדנו את בניית 3 הטבלאות לתהליך יחיד (שהינו duplicable) ובכך קיבלנו גרנולריות גסה יותר אך כיוון שה- load balancing מושלם לא הייתה פגיעה בביצועים ובנוסף חסכנו את זמן הסנכרון בין 3 התהליכים השונים לבניית הטבלה בשלבים הקודמים.

בשלב ה-2 פיצלנו כל תהליך בעיבוד התמונה ל-2 מופעים של התהליך כך שכל אחד מהם מטפל בחצי שורה בתמונה במקום שורה שלמה. הוספנו סנכרון אך ניצלנו את כל המעבדים במערכת בצורה מתוזמנת יותר (תהליכים זהים רצים במקביל במקום תהליכים שונים במקביל) ובכך שיפרנו את ה- load balancing הכולל ובכך גם קיצרנו את זמן ההמתנה של מעבדים שסיימו למעבדים שעדיין רצים.

בשלב ה-3 איחדנו את עיבוד התמונה לתהליך יחיד (duplicable) באופן דומה לשלב 1 ולכן מסיבות דומות לשלב 1 קיבלנו שיפור בביצועים.

<u>שעת סיום הניסוי</u>:

14:30