236370

Parallel And Distributed Programming

HW2

Alon Ahuvi – 200878064

alonahuvi@gmail.com

Natan Preminger - 011887262

natan.preminger@gmail.com

תיאור הפתרון הרטוב:

<u>חלק ראשון:</u>

פונקציה ראשית - simple_parallel_walsh מריצה **לולאת for מקבילית** (באמצעות OpenMP) אשר בכל איטרציה כל חוט בונה את העמודה המתאימה במטריצה WHT ע"פ הנוסחה שהתקבלה בתרגיל (באמצעות פעולות על הביטים של אינדקס השורה והעמודה).

בניית העמודה נעשית באמצעות פונקציית העזר - generate_hadamard_matrix_column שמשתמשת בפונקציית עזר set_bits_num לחישוב ערך כל תא.

<u>חלק שני:</u>

פונקציה ראשית fast_parallel_walsh.

הפונקציה תחילה קוראת לפונקציית העזר fast_parallel_walsh_vec_generator. תפקידה של פונקציה זאת היא להפוך את בעיית הWHT שהתקבלה מסדר N ל-NumOfThreads בעיות מסדר N/NumOfThreads .

המימוש של פונקציה זאת מבוצע ב-3 לולאות for :

- לולאה ראשית – תפקידה לחלק כל פעם את את בעיית הWHT הנוכחית ל-2 בעיות נפרדות מסדר אחד נמוך יותר. מתבצע ע"י כך שגודל הווקטור עליו עובדים בכל שלב נחצה לשניים.
- לולאה שנייה – תפקידה לעבור על כלל תת-הווקטורים הקיימים כרגע בתוך הווקטור הראשי.
- לולאה שלישית – **רצה במקביל באמצעות OpenMP** – מחשבת את ערך כל תא בכל תת ווקטור באמצעות הנוסחה שהתקבלה בדף התרגיל.

לאחר סיום פונקציה זאת מתקבלים בווקטור הראשי NumOfThreads תתי ווקטורים מסדר נמוך יותר.

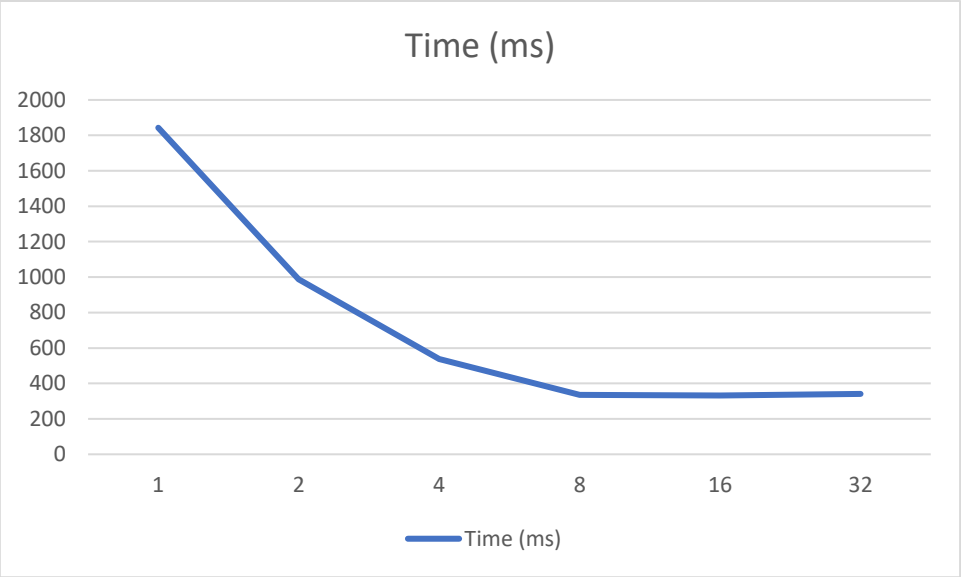
כעת חוזרים לפונקציה הראשית **שעוברת לקטע קוד מקבילי** שבו כל חוט קורא לפונקציית העזר serial_fast_walsh.

serial_fast_walsh היא פונקציה רקורסיבית שמחשבת את הווקטור באותו אופן ש fast_parallel_walsh_vec_generator חישבה כל תת-ווקטור. הפונקציה קוראת בכל פעם לעצמה באופן רקורסיבי עם 2 ווקטורים שכל אחד מהם הוא חצי מגודל ווקטור הבעיה הנוכחית.

כאשר כל החוטים סיימו את serial_fast_walsh סיימנו לפתור את הבעיה והתכנית הראשית יכולה להסתיים.

.1

a. ניתן להבחין שהחל מ-8 חוטים ויותר כבר לא נצפה שיפור בזמנים (אפילו נעשה קצת גרוע יותר).
לדעתנו, בגלל שניתן לראות מהגרף שהשיפור נעשה כמעט קבוע, ניתן להסיק שאמנם כמות החוטים עולה אך בפועל אנחנו מקבלים מספר ליבות מוגבל (ככל הנראה מקסימום 8) ולכן כאשר רצים 16 או 32 חוטים על 8 ליבות לא נצפה שיפור בביצועים.
בנוסף החוטים משתמשים בזיכרון מטמון משותף, כל חוט מביא חלקים שונים של הווקטור הראשי מהזיכרון למטמון וכל הבאה כזאת יכולה להוציא מהמטמון מידע אשר בשימוש של חוט אחר וכך החוטים פוסלים אחד לשני מידע במטמון.



.b

חוק אמדל:

$$speedup = \frac{T_{serial}}{T_{parallel}} = \frac{1}{(1-A)+\frac{A}{n}}$$

$$SpeedUp_{Threads=2} = 1.87$$
$$SpeedUp_{Threads=4} = 3.43$$
$$SpeedUp_{Threads=8} = 5.49$$
$$SpeedUp_{Threads=16} = 5.55$$
$$SpeedUp_{Threads=32} = 5.4$$

נחלק את תשובתנו לשני חלקים:
עבור החלק הראשון של הגרף (1-8 חוטים) ניתן לראות שיפור בביצועים שאינו לינארי ותואם לחוק אמדל. ניתן גם להבחין שככל שעולה כמות החוטים בתחום זה קיימת ירידה בשיפור כפי שחוק אמדל צופה, זה נובע מכך שגודל העבודה נשאר קבוע והוספת מעבדים משפרת אך ורק את החלק המקבילי.
עבור חלקו השני של הגרף בו הגרף כמעט קבוע, התוצאות אינן תואמות לחוק אמדל וזאת מהסיבות שפורטו בסעיף a.

.c

הסיבה העיקרית שהתכנית לא תשתפר באופן לינארי כאשר כמות החוטים עולה היא בגלל החלק הסדרתי שבה. כפי שחוק אמדל מתאר גם אם נמשיך ונגדיל את כמות החוטים עדיין ישנו חלק סדרתי שנשאר זהה.

בנוסף קימות סיבות נוספות שיכולות להיות לחוסר התאמה לינארית בין כמות החוטים לתוצאות הניסוי:

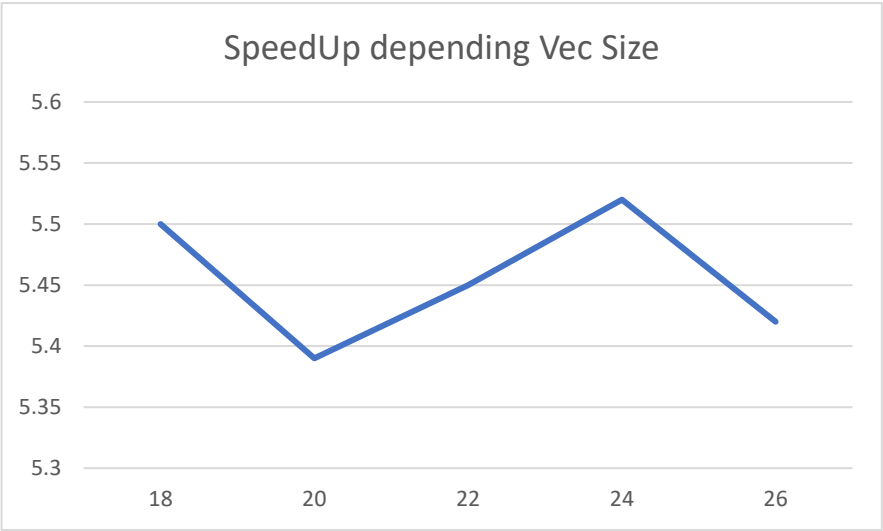
1. התנגשות במטמון – כפי שהוסבר בסעיף a, ככל שיש יותר חוטים ההסתברות להתנגשות במטמון המשותף גבוהה יותר.
2. עלות יצירת חוטים – מנגנון יצירת החוטים עלול להיות סדרתי (תלוי מימוש) ולכן להשפיע יותר ככל שנוסיף יותר חוטים.
3. ייתכן והשרת עליו אנחנו בודקים אינו מקצה מספר ליבות כמספר החוטים שהגדרנו ולכן בפועל לא כל החוטים רצים במקביל.

.d

- הסיבות לכך שב-16 ו- 32 חוטים יכולים להיות ביצועים פחות טובים מ-8 נובעות ממה שפורט בסעיף c (פורטו בסוף סעיף c שלוש סיבות). בפרט אם אכן איננו מקבלים את כמות הליבות שדרשנו, הנזק שנגרם לשני חוטים שרצים על אותה ליבה מהתנגשות במטמון המשותף הוא משמעותי וגורם לפעולות נוספות מול הזיכרון שגורמות לירידה בביצועים.
- סיבה לכך שב-16 ו-32 חוטים התוצאה תהיה זהה (או אפילו קצת פחות טובה) ל-8 היא אם בפועל אנחנו מקבלים רק 8 ליבות להרצת התכנית אז על כל ליבה רצים כמה חוטים במקביל.
  מכאן שיש יותר החלפות הקשר בין החוטים על כל מעבד פעולה אשר לוקחת גם כן כמה סייקלים וגורמת לירידה בביצועים.

.2

.a



חוק גוסטפון: $Speedup(N) = N + (1 - N)*s$

<u>ניסוי</u>

$$SpeedUp_{VecSize=2^{\wedge}18} = 5.5$$

$$SpeedUp_{VecSize=2^{\wedge}20} = 5.39$$

$$SpeedUp_{VecSize=2^{\wedge}22} = 5.45$$

$$SpeedUp_{VecSize=2^{\wedge}24} = 5.52$$

$$SpeedUp_{VecSize=2^{\wedge}26} = 5.42$$

ניתן להבחין בגרף המצורף שווקטור בגודל 2^{24} מספק את השיפור הטוב ביותר מבין הגדלים שנבדקו.
בנוסף ניתן להבחין שההבדלים בשיפור בין גדלי הווקטור השונים הם קטנים יחסית.

.b

חוק גוסטפון שאומר שכאשר אנחנו מגדילים את מספר המעבדים, אנו יכולים גם לבצע חישובים גדולים וארוכים יותר (שבדר"כ מתרחשים בחלק המקבילי של התכנה) וע"י כך גם להקטין את היחס של החלק הסדרתי בתכנה וכך להשיג SpeedUp גבוה יותר.
ע"פ גוסטפון, בתכנית כמו שלנו אנו מצפים שככל שגודל הווקטור עולה, משקל החלק המקבילי יגדל אל מול החלק הסדרתי (מפני שרוב החישובים מבוצעים במקביל).
ניתן לראות שהגרף אינו תואם לציפייה זאת. הסבר לכך יכול להיות שהתנגשות המטמונים כפי שפורט בשאלה הקודמת גורעת משיפור המקביליות.
ניתן להבחין בגרף שכאשר אנחנו מריצים 8 חוטים מתקבל ה SpeedUp הגבוה ביותר בגודל וקטור של 2^{24} – כלומר זהו ווקטור אשר מספק את היחס הטוב ביותר בין החלק הסדרתי למקבילי אך בנוסף מספיק קטן כדי לא ליצור הרבה התנגשויות במטמון.

.c

כפי שתואר קודם, השיפור/ירידה בSpeedUp תלוי בהיכן מבוצעים מרבית החישובים על הקלט ובמגבלות גודל המטמון.

- אם מרבית החישובים על הקלט נעשים בחלק המקבילי, הגדלת הקלט תשפר את הSpeedUp מכיוון שמשקל החלק המקבילי יעלה אל מול החלק הסדרתי עבור הקלט הנתון.
- אם מרבית החישובים על הקלט מבוצעים בחלק הסדרתי, הגדלת הקלט תיפגע בSpeedUp מכיוון שמשקל החלק הסדרתי יעלה אל מול משקל החלק המקבילי.
- מלבד חלק התכנית בו מתבצעים מרבית החישובים יש להתחשב במגבלת המטמון. עבור גודל מידע הנכנס כולו במטמון, ייתכן והגדלת המידע תגרור התנגשויות במטמון ומעבר לעבודה מול הזיכרון אשר איטי במספר סדרי גודל ולכן נקבל ביצועים טובים פחות.
  עבור קלט גדול מאוד אף תתכן עבודה מול הזיכרון המשני (דיסק). ולכן במעבר לקלטים גדולים מאוד תתכן התדרדרות נוספת.

.3

.a

חישוב החלק המקבילי:

הפונקציה העיקרית שרצה במקביל היא serial_fast_walsh .

זוהי פונקציה שרצה ברקורסיה בעומק $\log_2 N$ ובכל שלב של עומק הרקורסיה כל החוטים יחד עוברים על כל הווקטור (N) . בכל שלב כזה מתבצעות 6 פעולות אריתמטיות ולכן קיבלנו $6Nlog_2(N)$.

בנוסף בכל שלב ברקורסיה אנו מחשבים את גודל הווקטור. מספר הקריאות הכולל לפונקציה הינו $N\left(\frac{1}{2}+\frac{1}{4}+\cdots\right)=N$ ומבוצעות במהלך זה 2 פעולות אריתמטיות ולכן קיבלנו 2N

$$OpsParallel = 2N + 6Nlog_2(N)$$

חישוב החלק הסדרתי:

בתחילת התכנית עבור חלוק הווקטור אנו מבצעים מספר שלבים בהם החלק המקבילי מתבצע רק על חישוב שלב בווקטור. מכאן שביניהם ישנו חלק סדרתי התלוי במספר החוטים שייפתחו.

חישבנו באופן דומה את מספר הפעולות האריתמטיות בשלב זה וקיבלנו:

$$OpsSerial = 4 + 2\log(p) + p = 18$$

b. תחילה נחשב את יחס החלקים המקבילי והסדרתי:

$$A(n) = \frac{2n + 6nlog_2(n)}{18 + 2n + 6nlog_2(n)} \qquad S(n) = 1 - A(n)$$

$$2^{18}: \quad A = 0.9999993758 \quad S = 6.2422 * 10^{-7}$$
$$2^{20}: \quad A = 0.9999998593 \quad S = 1.407 * 10^{-7}$$
$$2^{22}: \quad A = 0.9999999968 \quad S = 3.2 * 10^{-8}$$

Amdahl:

$$speedup(2^{18}) = 7.99996$$
$$speedup(2^{20}) = 7.999992$$
$$speedup(2^{22}) = 7.999999$$

Gustafson:

$$speedup(2^{18}) = 7.9999956$$
$$speedup(2^{20}) = 7.999999$$
$$speedup(2^{22}) = 7.9999998$$

השיפורים מהניסויים המתאימים נרשמו בסעיף קודם.

c.

ניתן לראות שלא קיבלנו תוצאות זהות. השיפור שיצא לנו לא קרוב לשיפור האידיאלי שניתן לקבל.

הבדל משמעותי שקיים הוא שישנה ירידה ברמת השיפור במעבר בין 2^{18} ל 2^{20} . דבר שאינו צפוי.

d.

אנו חושבים שקיים מנגנון סנכרון בין החוטים (מאופן מימוש OpenMP) אשר מעכב את ביצוע התכנית וצריך להתחשב בו בחישוב היחס הסדרתי.
בחישובים שביצענו לא התחשבנו בהשפעה זאת ועל כן קיבלנו משקל כה קטן לחלק הסדרתי שהביא לחישוב של SpeedUp גבוה יחסית.
בנוסף עבור הירידה ברמת השיפור, ניתן לייחס חשיבות לגודל המטמון. מהרצת פקודה לבדיקת גודל המטמון קיבלנו:

```
cdp38@bl201:~/ex2-Natan$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    1
Core(s) per socket:    4
Socket(s):             2
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 23
Model name:            Intel(R) Xeon(R) CPU           E5420  @ 2.50GHz
Stepping:              6
CPU MHz:               2500.000
BogoMIPS:              5000.02
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              6144K
NUMA node0 CPU(s):     0-7
cdp38@bl201:~/ex2-Natan$
```

גודל L2 הינו 6MB.
גודל הווקטור שאנו מחשבים עבור 2^{18} איברים הוא כ-1MB ועבור 2^{20} הינו 4MB.
גדלים אלה הינם כסדר הגודל של גודל המטמון ולכן ייתכן שהירידה בשיפור נובעת מכך שלא כל איברי הווקטור נמצאים במטמון בו-זמנית ונדרשות גישות רבות לזיכרון הראשי דבר שלאו דווקא קורה בווקטורים קטנים יותר.

.4

.a

באופן כללי היינו מצפים ששרת של הקורס הזה המיועד להריץ הרבה משימות מקביליות יכיל מספר משמעותי של מעבדים בגודל של BCE 1 ויחד איתם מספר ליבות גדולות יותר שיתמודדו עם המשימות הסדרתיות.
סיבה למצב המתואר בשאלה יכולה להיות שבהינתן גודל השרת של הקורס ביחידות BCE (נניח שהוא יחסית גדול), במצב שגרתי לא מתקבלות מספיק משימות מקביליות ע"מ לפזר אותן בין כמות מעבדים מקבילה בגודל BCE 1 ולכן נוצר מצב של הרבה מעבדים לא פעילים. ועל כן ניצול מיטבי יותר של המשאבים יהיה שכלל המעבדים יהיו יותר גדולים ויבצעו עבודה יותר מהירה.

.b

ע"פ תרגול מס' 2.
נוסחת SpeedUp ל-Asymmetric Multicore Chips:

$$Speedup_{asymmetric}(A,n,r) = \frac{T_{BCE-serial} = 1}{\frac{1-A}{perf(r)} + \frac{A}{perf(r)+n-r}}$$

כאשר $perf(r) \sim \sqrt{r}$
ולכן עבור הנתונים בשאלה A=0.9, n=32 נקבל:

$$\frac{1}{\frac{0.1}{\sqrt{x}} + \frac{0.9}{\sqrt{x}+32-x}}$$

מחישוב בוולפראם מצאנו:

Local maximum:

$$\max\left\{\frac{1}{\frac{0.1}{\sqrt{x}} + \frac{0.9}{\sqrt{x}+32-x}}\right\} \approx 14.8915 \text{ at } x \approx 11.2749$$

ומפני ש-r הוא שלם השווינו בין r=11 לr=12 וקיבלנו תוצאה גבוהה יותר עבור r=11

$$SpeedUp(r = 11) = 14.8892$$

.c

ע"פ תרגול מס' 2.
נוסחת SpeedUp ל-Dynamic (Composed) Multicore Chips:

$$Speedup_{dynamic}(A,n,r) = \frac{T_{BCE-serial} = 1}{\frac{1-A}{perf(r)} + \frac{A}{n}}$$

זוהי פונק' מונוטונית עולה ממש ולכן נעדיף לבחור r מקסימלי, כלומר r=32 ונקבל:

$$SpeedUp(r = 32) = 21.8328$$