רטוב 1 - תכנון משולב חומרה/תוכנה

אמיר זועבי - 212606222

ניר שיף - 212980395

**הערה: לא נעשה שימוש בAI ביצירת הבעיה, פתרונה, ואיפטומה. השימוש ב AI נעשה רק ביצירת סקריפט הקימפול והריצה לנוחיותכם, בשם "run_particles.sh".*

1. הגישה הכללית שלך ותהליך החשיבה שלך.

התוכנית מחשבת את התנועה של חלקיקים במרחב. התכנית תקבל את מיקומם של מספר חלקיקים במרחב ביחס לציר תלת מימדי. התכנית תחזיר את מיקומו החדש של כל חלקיק לאחר זמן מסויים, בעקבות השפעת החלקיקים האחרים.

$$\vec{F}_{ij} = -\frac{\vec{r}_j - \vec{r}_i}{\left|\vec{r}_j - \vec{r}_i\right|^3} \qquad v_{x_i} = dt \cdot F_{x_i} \qquad x_i = dt \cdot v_{x_i}$$

החישוב מתבצע על ידי הנוסחאות:

תהליך החשיבה שלנו כלל מחשבה על כיצד ניתן לנצל את אי התלות של החישובים בכל נקודה על מנת לאפטם את התכנית. בנוסף, היות ובתכנית קיים הרבה מאוד מידע אליו אנו ניגשים באופן קבוע, הסקנו שעל מנת לקבל שיפור משמעותי בביצועים, עלינו לנצל את מנגנוני ה caching וvectorization הקיימים בחומרה שלנו על מנת לסחוט מקסימום ביצועים.

2. תיאור של **המימוש הלא אופטימלי** שלך:

- מדוע בחרת דווקא בו? בחרנו במימוש הפשוט והמיידי הזה, מכיוון שהוא מהווה פתרון פשוט וברור, הוא מטפל בכל חלקיק בנפרד ומבצע את כל החישובים הדרושים עבורו באופן ישיר. זהו הפתרון הראשוני והנאיבי ביותר שחושבים עליו כאשר מתחילים לממש תוכנית מהסוג הזה.

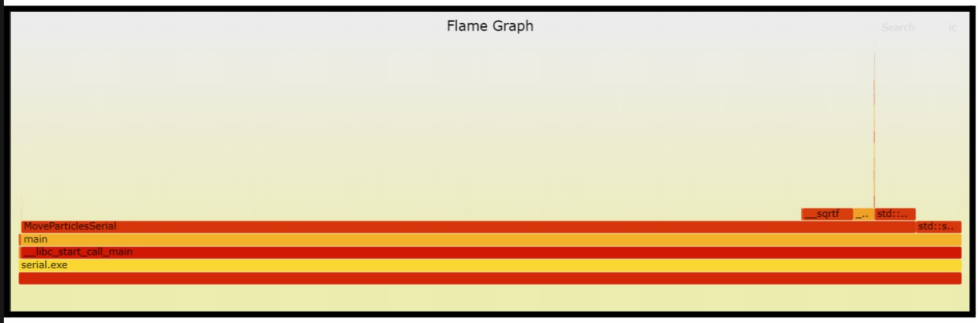
  עם זאת, מה שמייחד את התכנית הזו הוא הפוטנציאל הגדול לשיפור. היא מהווה בסיס מצוין להמחשת הדרכים שבהן ניתן לנצל בצורה מיטבית את מאפייני החומרה והתוכנה, כפי שנראה בהמשך.

- מה התוכנית עושה? התוכנית מבצעת את החישוב על כל חלקיק בנפרד. עבור כל חלקיק, היא מחשבת וסוכמת את הכוח הפועל עליו מכל חלקיק אחר בנפרד, ולאחר מכן מחשבת את תוספת המהירות כתוצאה מכל אחד מהכוחות הללו. חישובים אלה מתבצעים עבור כל אחד מכיווני המרחב בנפרד (x,y,z). לאחר חישוב המהירויות החדשות של כל החלקיקים, מעודכנים מיקומם במרחב בהתאם.

- מדוע היא לא אופטימלית? החישוב הנוכחי אינו אופטימלי ממספר סיבות. ראשית, חישוב הכוחות עבור כל חלקיק מתבצע באופן עצמאי, ללא תלות בתוצאות החישוב של חלקיקים אחרים. לכן, ניתן היה לנצל את עצמאות החישובים ולבצע אותם בצורה מקבילית (embarrassingly parallel), מה שיכול לשפר משמעותית את ביצועי המערכת.

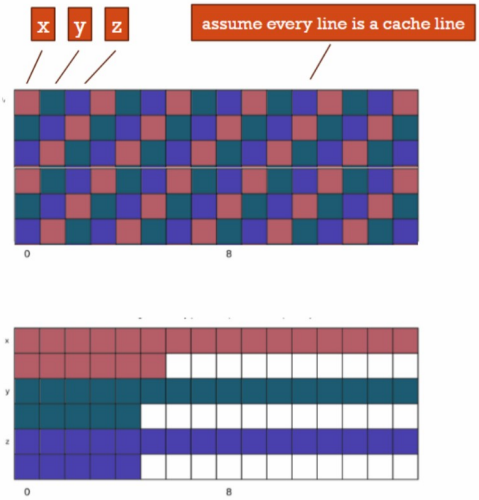
- תוצאות פרופיילינג (למשל: זמן ריצה, cache misses, מחזורי CPU וכו')

```
amirzuabi@tangerine:~/N-Body/SOFTWARE-HARDWARE-CODESIGN$ perf stat ./serial.exe 5

 ---  Starting Serial Run ---

 ---  Initializing Particles ---

 --- Serial Step 1 ---
 Serial step time: 18545 ms

 --- Serial Step 2 ---
 Serial step time: 18550 ms

 --- Serial Step 3 ---
 Serial step time: 18549 ms

 --- Serial Step 4 ---
 Serial step time: 18546 ms

 --- Serial Step 5 ---
 Serial step time: 18544 ms

  === Serial saving time time: 167 ms ===

 Serial simulation complete. Results saved to serial_result.txt

  Performance counter stats for './serial.exe 5':

          92,866.90 msec task-clock                #    1.000 CPUs utilized
               268      context-switches          #    2.886 /sec
                 5      cpu-migrations            #    0.054 /sec
               673      page-faults               #    7.247 /sec
   241,414,460,022      cycles                    #    2.600 GHz
   553,642,355,486      instructions              #    2.29  insn per cycle
    43,075,947,853      branches                  #  463.846 M/sec
         1,307,841      branch-misses             #    0.00% of all branches

       92.906798994 seconds time elapsed

       92.868126000 seconds user
        0.000000000 seconds sys
```



3. תיאור של **האופטימיזציה** שלך:

- איזו שינוי ביצעת? השינוי המרכזי שביצענו הוא ביצוע מקבילי של החישוב. בהינתן שהמעבד יכול להתמודד עם מספר x של threads, ושיש לנו n נקודות, כל thread יהיה אחרי ל n/x נקודות. בצורה כזו אנו מבצעים במקביל את החישוב של מספר נקודות. בנוסף את החישוב עצמו על כל נקודה אנחנו מבצעים באמצעות וקטורים. כלומר, כל פעולה חישובית אנו מבצעים בו זמנית על שמונה ערכים במקביל ובכך אנו חוסכים איטרציות מיותרות של הלולאה. יתר על כן, עברנו לצורה של SoA במקום AoS, דבר שחשוב מאוד על מנת לשמר את ה cache locality כאשר משתמשים בוקטוריזציה.



SoA - Struct of arrays
AoS - Array of Structs

```
// Global arrays for particle
float global_X[nParticles];
float global_Y[nParticles];
float global_Z[nParticles];
float global_Vx[nParticles];
float global_Vy[nParticles];
float global_Vz[nParticles];
```

```
// This is a AoS - Array of Structs
struct ParticleType
{
    float x, y, z;
    float vx, vy, vz;
    float trash1, trash2;
};

ParticleType serialParticles[nParticles];
```

- מדוע בחרת באופטימיזציה הזו דווקא? בחרנו באופטימיזציה הזו היות והיא מנצלת את אחת התכונות העיקריות של הבעיה הנתונה (אי התלות בין החישובים) וכמובן התמיכה בAVX ו SMT.

```
amirzuabi@tangerine:~$ lscpu | grep CPU
CPU op-mode(s):                       32-bit, 64-bit
CPU(s):                               56
On-line CPU(s) list:                  0-55
Model name:                           Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz
CPU family:                           6
CPU max MHz:                          3500.0000
CPU min MHz:                          1200.0000
NUMA node0 CPU(s):                    0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,46,48,50,52,54
NUMA node1 CPU(s):                    1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39,41,43,45,47,49,51,53,55
Vulnerability Mds:                    Mitigation; Clear CPU buffers; SMT vulnerable
Vulnerability Mmio stale data:        Mitigation; Clear CPU buffers; SMT vulnerable
Vulnerability Tsx async abort:        Mitigation; Clear CPU buffers; SMT vulnerable
amirzuabi@tangerine:~$ lscpu | grep Thread
Thread(s) per core:                   2
amirzuabi@tangerine:~$
```

```
amirzuabi@tangerine:~$ lscpu | grep -G avx
Flags:                                fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx p
dpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fm
a cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb cat_l3 cdp_l3 invpcid_s
ingle pti ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm cqm rdt_a rdseed adx smap intel_p
t xsaveopt cqm_llc cqm_occup_llc cqm_mbm_total cqm_mbm_local dtherm ida arat pln pts md_clear flush_l1d
amirzuabi@tangerine:~$
```

- איזו תובנה ברמת מערכת/חומרה הובילה אותך לכך? ראשית, מאחר ואין תלות בין החישובים השונים שמבוצעים, ניתן להריץ אותם במקביל מבלי לחשוש לנכונות. מאחר שישנה תמיכה בAVX ו SMT במעבד שלנו, החלטנו לנצל תכונה זו ולחלק את העבודה למספר threads. בנוסף לכך, לווקטר את כל החישובים על מנת להאיץ את זמן הריצה.

יחד עם זאת, חשוב לציין כי אם יוצרים יותר מדי threads, נוצר עומס על המערכת - בעיקר בגלל הזמן והמשאבים הדרושים לניהול, תזמון, והקשר ביניהם. בנוסף, אם יש יותר מדי threads ביחס לכמות הליבות הפיזיות הקיימות, רבים מה threads ישנים ולא מנצלים את המעבד בצורה יעילה. נמנע מבעיה זו בכך שאנו לוקחים את כמות הthreads של המערכת בעזרת syscall, ועקב גודל הdata הגרנולריות של הטאסקים תהיה טובה. כלומר:

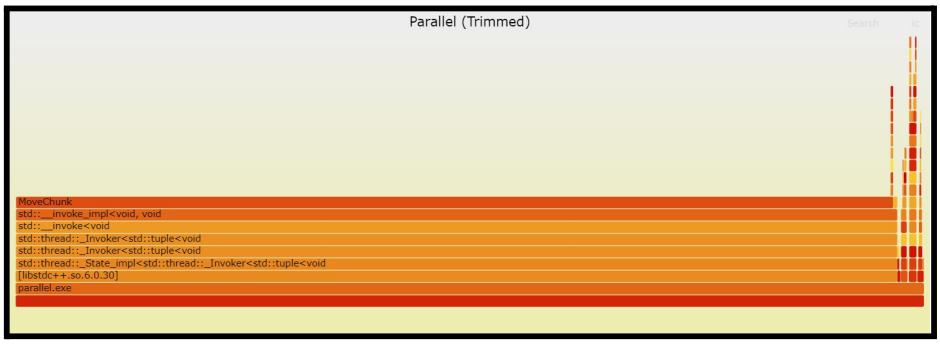
$$\frac{32{,}000\ particles}{56\ Threads} \approx 571\ Particle/Thread$$

ויתר על כן, לכל חלקיק יש כמות גדולה של פעולות וקטוריות.
$(32000 \cdot 20 \cdot 570 \approx 384{,}000{,}000\ vec.\ ops.\ /\ step)$

לכן בחרנו להקצות רק את מספר ה threads שהמערכת מסוגלת לשאת ביעילות. בחירה זו אפשרה לנו לבצע פיצול עבודה חכם שמנצל את היתרונות של מקביליות. נוסף לכך, חשוב גם לסדר את ה data בצורת SoA כפי שצוין קודם בשביל להרוויח מה- cache locality עבור הווקטורים.

```
Cache trashed successfully.
amirzuabi@tangerine:~/N-Body/SOFTWARE-HARDWARE-CODESIGN$ perf stat ./parallel.exe 5

---  Starting Parallel Run ---

---  Initializing Particles ---

--- Parallel Step 1 ---
Parallel step time: 338 ms

--- Parallel Step 2 ---
Parallel step time: 343 ms

--- Parallel Step 3 ---
Parallel step time: 308 ms

--- Parallel Step 4 ---
Parallel step time: 337 ms

--- Parallel Step 5 ---
Parallel step time: 343 ms

 === Parallel saving time time: 162 ms ===

Parallel simulation complete. Results saved to parallel_result.txt

 Performance counter stats for './parallel.exe 5':

         64,479.88 msec task-clock                #   35.079 CPUs utilized
               956      context-switches          #   14.826 /sec
                19      cpu-migrations            #    0.295 /sec
             1,388      page-faults               #   21.526 /sec
   167,632,609,076      cycles                    #    2.600 GHz
   101,355,595,566      instructions              #    0.60  insn per cycle
       800,787,301      branches                  #   12.419 M/sec
         1,366,407      branch-misses             #    0.17% of all branches

       1.838126222 seconds time elapsed

      64.445117000 seconds user
       0.043981000 seconds sys
```

- מה היו תוצאות הפרופיילינג?



- האם התוצאות היו צפויות או מפתיעות? צפויות, מאחר והמשימה היא מאוד compute heavy, צפוי שנבלה את רוב זמן התוכניות בפונקציות MoveChunk ו MoveParticleSerial. השינוי המשמעותי שנצפה לו יהיה בזמן הריצה של התוכנית.

4. **השוואה** בין תוצאות הפרופיילינג של שני המימושים:

Parallel vs Serial Execution Comparison

**ran for 5 steps*

Metric	Serial Execution	Parallel Execution
Execution Time (s)	92.9	1.83 (50x speedup)
System Time (s)	0.0	0.043981
CPU Utilization	1.000	35.079
Context Switches	268	956
CPU Migrations	5	19
Page Faults	673	1388
Cycles	241,414,460,022	167,632,609,076
Instructions	553,642,355,486	101,355,595,566
IPC (Instructions per Cycle)	2.29	0.6
Branches	43,075,947,853	800,787,301
Branch Misses	1,307,841	1,366,407

- ניתן לראות שישנו בx50 שיפור במהירות ריצת התוכנית, עקב השיפור המשמעותי של step time, ירדנו מ 18500ms לצעד ל330ms לצעד בממוצע.
- שיפור זה משמעותי ביותר והכל בזכות ניצול כמות הליבות, הוקטוריציה והניצול היעיל של זיכרון המטמון. השיפור משתקף בזמן הריצה הכולל, ובניצולת המעבדים, שבריצה המקבילית הוא גבוה בהרבה יותר בזכות SMT והmulticore.
- ניתן לראות שישנם גם הרבה יותר Context Switches, שעל פנוי זה אל דבר טוב, אך מצופה מאחר ואנחנו עובדים עם הרבה יותר חוטים. לדבר זה אין השפעה רבה על התוכנית מאחר והשינויים בקונטקס זניחים לעומת כמות החישובים שכל חוט מבצע (יש גם יותר migrations אבל זה לא מפתיע מאותם הסברים).
- ניתן לראות שגם ישנם יותר page faults, את התנהגות זו אנו לא יודעים בוודאות להסביר, אך אנחנו מניחים שקצב עיבוד המידע מהיר מאוד ביחס למימוש הסיריאלי, ולכן אולי נדרש יותר ממערכת ההפעלה באותו פרק זמן.
- ניתן לראות כי ה IPC ירד משמעותית, עם זאת, כעת כל אופרציה עובדת על 16 איברים (2 ווקטורים של 8), ולכן כנראה המדד הזה לא לוקח בחשבון נכון את הפעולות הווקטוריות ולא משקף נכון את השיפור בביצועים.

5. **כמו כן, מומלץ לתאר גם גישות שניסית שלא הצליחו** או אופטימיזציות ששקלת אך נטשת. זה משקף חקירה וחשיבה ביקורתית. אחת הגישות ששקלנו הינה לבנות thread עבור כל נקודה. הסיבה שהחלטנו לא לבצע את הרעיון הנ"ל הינה שבניית threads הינה פעולה יקרה מבחינת ניצול מקום וזמן. בנוסף, כאשר ישנה כמות גדולה מאוד של threads הדבר יכול להעמיס על המעבד, ולבזבז זמן ריצה על ביצוע context switch. יתר על כן, אין צורך בבניית threads מיותרים אשר לא ירוצו בעקבות העומס. לכן, במקום לבנות thread שלם בשביל נקודה בודדת, התאמנו את כמות ה threads לכמות אותה המעבד יכול להריץ, ואת הנקודות חילקנו שווה בשווה בין ה threads בכדי לקבל תפוקה מקסימלית.