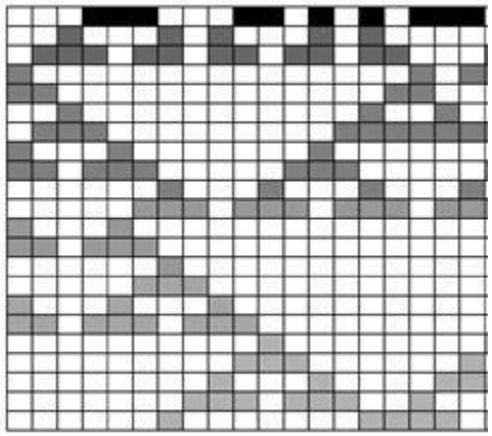


CELLULÁRIS AUTOMATA (SEJTAUTOMATA)

Számítások randevú típusú szinkronizálással

RÖVID ÖSSZEFOGLALÁS

A sejtautomata leggyakoribb formája: egy négyzetrácsban (a *sejttér*ben) helyezkedik el, a négyzetrácsok által közrefogott cellákat sejteknek nevezzük. A sejteknek különféle állapotaik lehetnek (véges sokféle). Ahogy az idő telik, a cellák változtatják állapotukat, általában saját és más sejtek, például néhány szomszédjuk előző időpillanatbeli állapotától függően.

Burian Sándor , AWXYHE

Szoftverfejlesztés párhuzamos architektúrákra

2020-2021, első félév

Óbudai Egyetem, Neumann János Informatikai kar

Tartalom

Bevezetés, a probléma bemutatása

“A feladatteret cellákra osztjuk. Minden cella véges állapotok közül pontosan egyet vesz fel. A cellákra a szomszédjai valamilyen hatással vannak bizonyos szabály szerint, és minden cella egy „generációs” szabállyal is rendelkezik, amely szimultán fut. A szabályok újra és újra alkalmazásra kerülnek minden generációs lépésben, így a cellák fejlődnek, vagy megváltoztatják állapotukat generációról generációra. A legismertebb celluláris automata John Horton Conway cambridge-i matematikus „Élet játéka” (“Game of Life”).” [1]

Az élet játéka

A táblás játék elméletben végtelen méretű kétdimenziós tömbökkel. Minden tömb cellákat tartalmaz. Tehát minden cellának 8 szomszédja van.

Pl:

	⋱	…	…	…	⋰
tömb	…	Cella	Cella	Cella	…
tömb	…	Cella	Cella	Cella	…
tömb	…	Cella	Cella	cella	…
	⋰	…	…	…	⋱

A cellákban amikben egy-egy organizmus lehet. Kezdetben néhány cella foglalt, a következő szabályok szerint, amiket Conway fektetett le [1]:

1. Minden organizmus aminek 2-3 szomszédja van életben marad a következő generációra.
2. Minden olyan organizmus amelynek 4+ szomszédja van meghal a túlnépesedés miatt.

[1] https://elearning.uni-obuda.hu/main/pluginfile.php/274206/mod_resource/content/1/P%C3%A1rhuzamos_programoz%C3%A1s_javasolt_feleves_feladatok.pdf hozzáférés dátuma: 2020 szeptember 28

3. Mindegyik olyan organizmus amelyiknek 2-nél kevesebb szomszédja van, tehát maximum egy, meghal az izoláltság miatt.
4. Minden üres cellában amelynek pontosan három nem üres szomszédja van egy új organizmus születik.

A példa mögé több mindent is beleláthatunk, a cellák tartalmai könnyen tovább gondolhatóak nyulakra és rókákra egy szigeten, vagy három dimenziós tömbökkel halakra és cápákra, az alábbiak szerint [1]:

Halak:

1. Ha van üres szomszédos cella odaúszik, ha több cella is üres akkor véletlenül választ egyet
2. Ha nincs üres szomszédos cella akkor helyben marad
3. Ha mozog és eléri egy természetes értékként megadott nemzési idejét akkor egy újabb hal kerül a szomszédos, üresen maradt cellába
4. Adott idő után a hal elpusztul

Cápák:

1. Ha szomszédos cellában hal van, akkor oda lép és megeszi, ha több szmszédos cellában is hal van akkor véletlen szerűen választ egyet.
2. Ha nincs szomszédos cellában hal akkor egy szomszédos üres cellába mozog, mint egy hal, ha több szomszédos cella is üres akkor véletlenül választva.
3. Ha eléri a nemzési idejét akkor akár egy hal új cápa kerül egy megüresedő cellába.
4. Ha a cápa adott generáción keresztül nem eszik akkor elpusztul.

Hasonlóan mögé lehet látni folyadék vagy gáz dinamikai folyamatokat, biológiai fejlődést, légáramlás vagy eróziós folyamatok modellezését.

Párhuzamosítás kérdése

Mivel egy nagy területen szükséges sokszor ugyanazt a műveletet elvégeznünk (*történetesen, hogy a különböző szomszédjaiban mik az állapotok egy-egy cellára nézve*) ezért a párhuzamosítás használata egyértelműen hasznos. Ugyanakkor az is egyértelmű, hogy időnként szinkronizációra van szükség. Elképzelésem szerint a teret felosztom nagyjából egyenlő részekre, annyira ahány magra optimalizálva számítom a feladatot[2]. Innentől csupán az egymással szomszédos területeket számoló folyamatok kell megvárják egymást, és csak azok kell szinkronizálódjanak, így, globális szinkronizációra nincs is igazán szükség, hiszen a terület két teljesen eltérő sarkában számolt értéknek nem sok értelme van, hogy egymásra várjon, mert nem sok közük van egymáshoz, nem befolyásolja egyik a másikat, a szinkronizálás pedig alapvetően egy költséges művelet [2], így a minél kevesebb szinkronizálás a praktikusabb a gyorsaság szempontjából.

"Az egyes folyamatok csak arra várnak, hogy az alapgráfbeli összes szomszédjuk *helyi_szint*-je legyen legalább *k*. Így ezt az összes folyamat maga felderítheti a *helyi_szint* minden egyes növekedése után a szomszédba vezető éleken küldött üzenettel. […] időbonyolultságának felső határa:

$$O(\mathrm{n}\log \mathrm{n}(\mathrm{l}+\mathrm{d}))$$

A kommunikációs bonyolultság azonban rosszabb a mindenegyes szinten használt szinkronizációs üzenetek miatt. Ez most $O(|\mathrm{E}|\log \mathrm{n})$ "[3]

Ezek persze csak a várt eredmények. A szükséges tárhely előre nem leszögezhető, hiszen a feladat elviekben végtelen tereket használ, így végtelen tárhelyre is volna szükség, ami persze jelenleg nem lehetséges. Előre tervezetten vagy a felhasználó által beadható, vagy a teljes memóriát hkihasználó tárhelyre lehetne felkészülni mint kényelmes-praktikus számítási modell.

[2] Ez akár dinamikusan is lekérhető, így a program konkrét processzortípustól függetlenül is optimalizált lehet [1]

Megvalósítás

A beadandómban csak a 2D szimulációt fogom figyelembe venni, halak-cápák felosztást tekintve. Az egy és a több szálas megvalósítás között sok közös vagy hasonló elem, függvény, osztály megtalálható, ami egyfelől az összevethetőséget támogatja, másfelől az újrahasznosítást elősegíti.

A program megvalósításához szükség lehet a főprogram osztályán kívűl még két osztályra. Az egyik a szimulációs tér mátrix absztrakciójának elemeit valósítja meg, azaz, hogy az adott mátrix elem hal, cápa vagy víz-e, és ennek megfelelően a szükséges segédfüggvények, segédváltozókat állítja be. A másik a szimulációs tér kezelésére egysegédobjektum, ami Descartesi koordinátaszerűen tudja kezelni a matrix elemeit, hogyazokra könnyebb legyen hivatkozni, például parameter átadás során.

A méréseket két processzoron is elvégeztem, melyeknek paraméterei az alábbbiak:

	Intel Core i7-4510U	AMD Ryzen 3 3200U
Socket Type	FCBGA1168	FP5
CPU Class	Laptop	Laptop
Clockspeed	2.0 GHz	2.6 GHz
Turbo Speed	Up to 3.1 GHz	Up to 3.5 GHz
# of Physical Cores	2 (Threads: 4)	2 (Threads: 4)
Max TDP	15W	15W
Yearly Running Cost	$2.74	$2.74
First Seen on Chart	Q2 2014	Q2 2019
# of Samples	3217	546
Cross-Platform Rating	5482	6494
Single Thread Rating	1557	1927
CPU Mark	2602	4101

Méresekhez használt processzorok adatai [4]

Egy szálas szimuláció megvalósítás

Alapvetően három fő funkció van implementálva a programba:

1) Futtathatjuk úgy, hogy a konzolra kirajzolja az aktuális futási állapotot, ez minden egyes lény módosításánál frissül, azaz, nem minden ciklus után, hanem mindn változás után rajzolja ki az állapotokat. Ez nyilván sokkal lassabb, de látványosabb szimulációs módszer.
2) Futtathatjuk úgy, hogy a képernyőre, a konzolra naplózza ki az eseményeket, ez a mérési módszer az előzőnél valamivel gyorsabb lesz, és a naplózás alapján elég pontosan követhetőek így is az események.
3) Végül szimulálhatunk úgy is, hogy statsiztikai adatokat szeretnénk látni, azaz fájlba mentse[3] a mérési eredményeket a kiinduló és a végső egyedszámokkal a futási sebességel együtt. Ezt megtehetjük a Conway féle szabályok alkalmazásával és azok elhagyásával is. Utóbbi esetben csak a kiinduló populáción múlik, hogy mi lesz a végeredmény, hiszen nem generálódnak úje elemek a szabad helyekre és nem halnak meg izoláltság miatt sem. A méréseket az utóbbi módszerrel végeztem.

Az implementálás szempontjából kulcskérdés, hogy szimulációs tér mátrixának frissítését hogyan oldjuk meg. Ez mind a tárhely mind a processzorkihasználtság szempontjából fontos kérdés, valamint mivel nem szeretnénk, csak nagyon kevés módosítást elvégezni a többszálúsításhoz, ohgy méréseink pontosabbak legyenek, ezért a többszálúsított programunk esetére is fel kell készülnünk, amennyire csak lehetséges.

Két irányba indulhatunk el. Egyfelől a szimulációs mátrixban minden egyes elem objektumánál külön-külön is eltárolhatjuk, hogy adott ciklusban lekezeltük-e már, vagy sem, ekkor viszont megnő a futási idő, mivel minden egyes alkalommal mikor az éppen vizsgált objektum (hal/cápa) körül levő többi cella objektumát (víz/hal/cápa) ellenőrizzük, nem csak azt kell figyelembe vennünk, hogy élőlény-e vagy sem, hanem, hogy ebben a fordulóban lépett-e vagy sem. Pl: ha van egy cápa ami egy egyel lentebbi sorba lép, akkor a ciklus később újra megvizsgálná, és újra léptetné, növelné a vemhessége idejét, ha tudná etetné, csökkentené a hátralevő életidejét stb. Ugyanígy egy hal esetében is. Más művelet esetén, pl ha azt vizsgáljuk, hogy van-e ehető hal a környéken, vagy mi a szabad pozíció, vagy hasonlók akkor nem okoz gondot, hiszen nem befolyásolja, ohgy az a hal már lépett, vagy lépni fog-e még, mert úgymond, “a jelenben kerül elfogyasztásra” a cápa által. Tehát, a futási költség az összesen annyival növekszik, hogy amikor megvizsgáljuk, hogy az adott elem amit vizságlunk az víz-e vagy sem, amennyiben nem víz úgy megvizsgáljuk, hogy lépett-e már vagy sem. Ugyanakkor minden módosítás(léptetés, szülés, evés, stb) után egy plusz művelet, hogy be kell állítani az aktuális ciklus cikluszszámlálóját mint utolsó módosítási ciklus jelölőt, de mivel ez vagylagos, ezért ez csak egy művelet pluszba minden ciklus esetén. A többi műveletbe ez a lépés nem kell

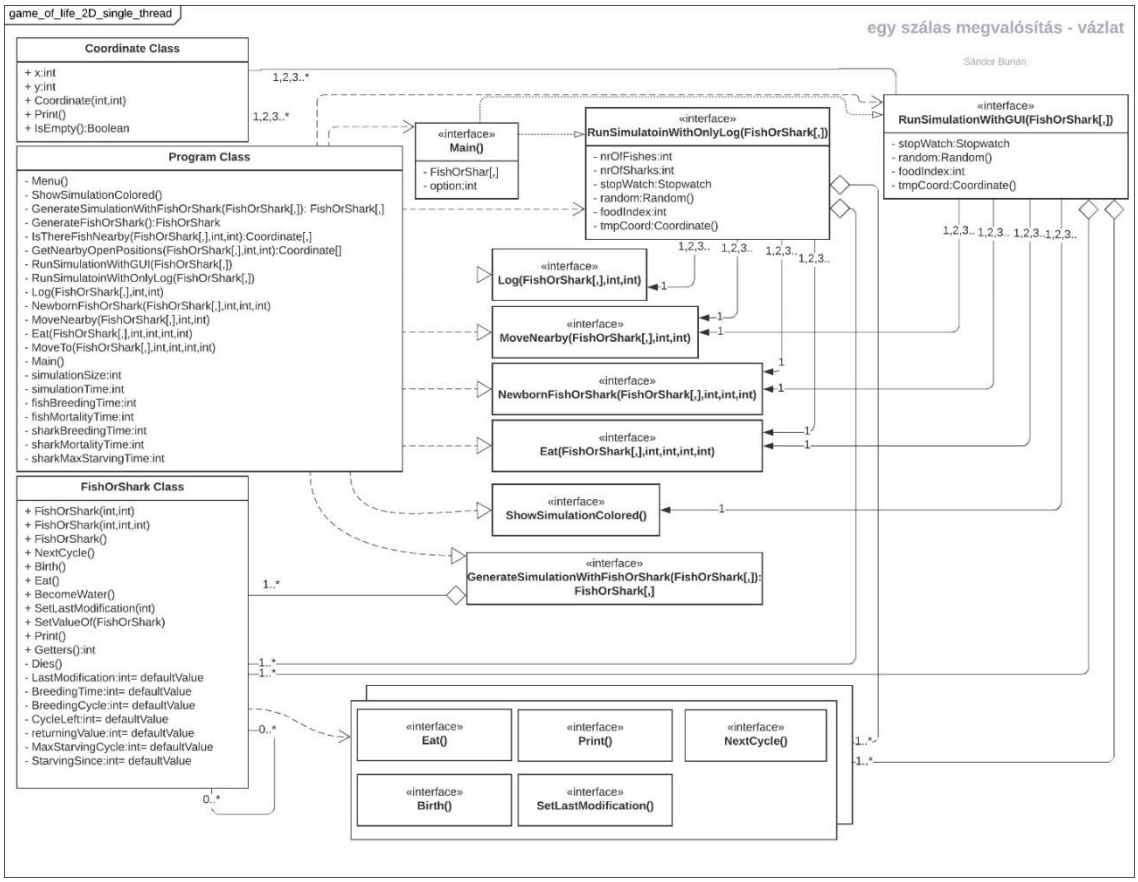
[3] Ekkor a ..*game_of_life_2D-single_thread**game_of_life_2D-single_thread**bin**Debugstat.txt* és a ..*game_of_life_2D-single_thread**game_of_life_2D-single_thread**bin**Debugstat_with_conway.txt* állományokról van szó.

beimplementálódjon, a fentebb kifejtettek okán. Tehát, összegzésképp összesen két plusz lépést kapunk, egy érték vizsgálatot és egy értékadást.

Másfelől készíthetünk egy másolatot a szimulációs mátrixról, amiben eltároljuk, az adott ciklusban már lekezelt elemeket. Ebben az esetben ahhoz, hogy csökkentsük a költségeket tehetünk úgy, hogy minden páros ciklusban a másolatot vizsgáljuk, és “léptetük”, majd a változtatott elemeket az “eredeti” szimulációs mátrixba másoljuk át, és minden páratlan ciklusban fordítva. Ugyanekkor ez később gond lehet, a léptetés megoldása, hiszen egyszerre két részmátrxban kell vizsgálni a szabad cellákat. Ez számszerűleg több futtatást ígényel, mivel két léptetős ciklus helyett négyet kell futtatnunk minden egyes esetben, és mindegyikben egy érték öszehasonlítást jelent pluszban.

Ez alapján arra következtethetünk, hogy jobb választás a háttértár szempontjából is és a futási sebesség szmepontjából is egy változóban tárolni az objektum oldalán, hogy az adott ciklusban vizsgáltuk-e már az adott objektumot.

Alább az egyszálú program UML diagrammjának vázlata látható, amiben csak a fontosabb részek kerültek megjelenítésre, a jobb átláthatóság végett.



Vázlatos UML diagram az egy szálú megvalósításhoz[4]

[4] Készült a https://lucid.app segítségével. A vázlat nem tartalmaz sok segédfüggvényt, és a nagyon hasonló működésű függvényeket sem!

Mérési eredmények[5]

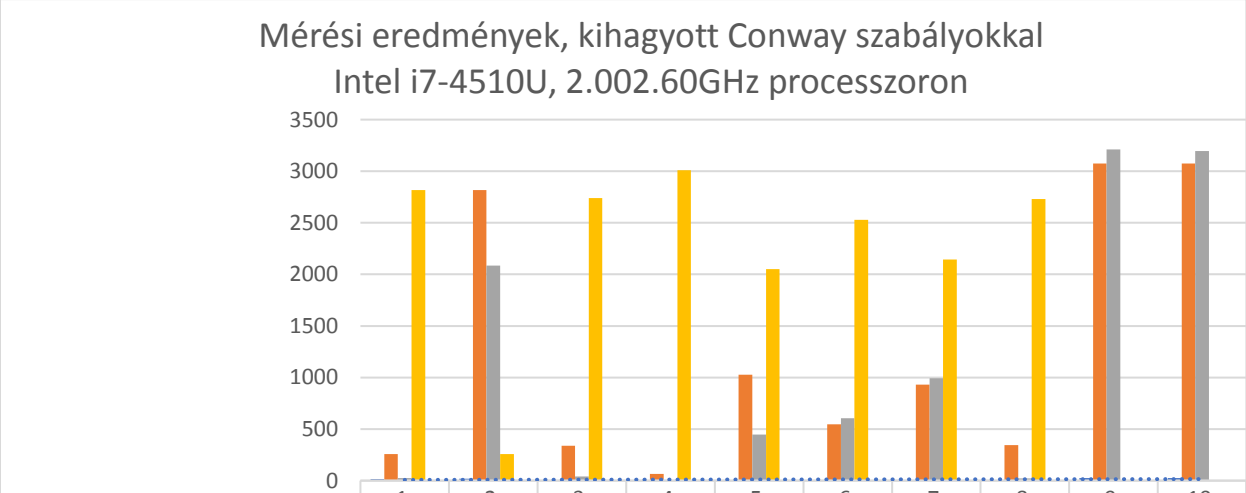
A kritériumfeltételek minden mérési esetben a következők:

Szimuláció ciklusainak száma	100
Szimuláció mérete *(nxn)*	100
Halak halálozási ideje	10
Halak nemzési ideje	5
Cápák halálozási ideje	15
Cápák nemzési ideje	10
Cápák maximum éhezési ideje	5

Megadott szimulációs paraméterek

Első esetben azt teszteltem le, hogy alakul a szimulációban a populáció, ha nem alkalmazzuk Conway eljárásából azt a kritériumot, hogy:

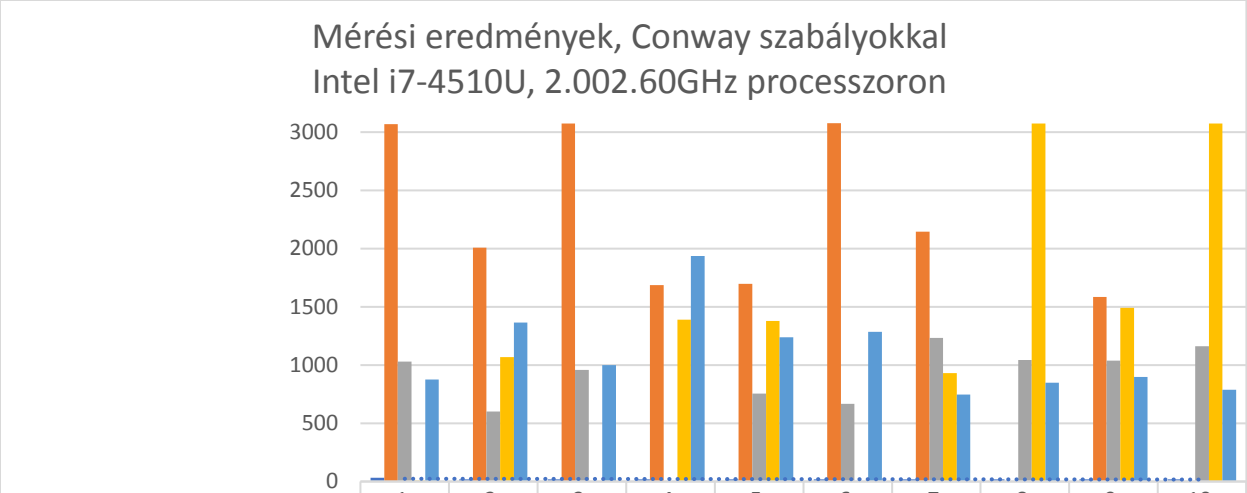
1. Minden organizmus aminek 2-3 szomszédja van életben marad a következő generációra.
2. Minden olyan organizmus amelynek 4+ szomszédja van meghal a túlnépesedés miatt.
3. Mindegyik olyan organizmus amelyiknek 2-nél kevesebb szomszédja van, tehát maximum egy, meghal az izoláltság miatt.
4. Minden üres cellában amelynek pontosan három nem üres szomszédja van egy új organizmus születik



	1. futtatás	2. futtatás	3. futtatás	4. futtatás	5. futtatás	6. futtatás	7. futtatás	8. futtatás	9. futtatás	10. futtatás
idő (s)	15.92	19.38	1.58	1.11	6.82	6.27	7.71	1.56	24.41	25.27
halak száma induláskor	259	2818	337	66	1026	546	932	345	3074	3075
halak száma szimuláció végén	24	2086	41	4	447	605	992	22	3211	3196
cápák száma induláskor	2817	258	2739	3010	2050	2530	2144	2731	2	1
cápák száma szimuláció végén	0	6	0	0	4	0	0	0	0	0

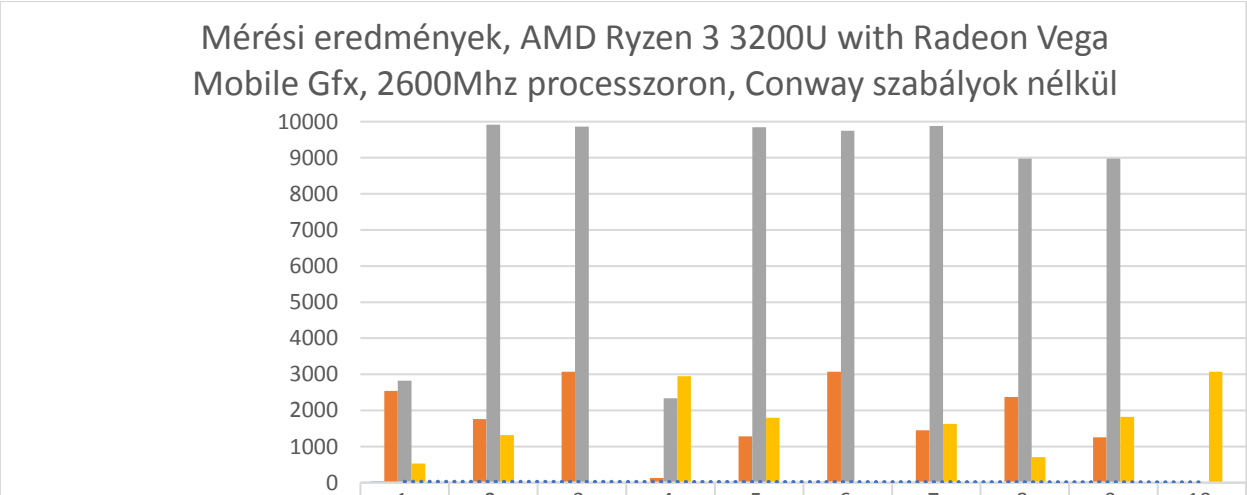
Mérési eredmények ábrázolása kihagyott Conway szabályokkal intel processzoron

[5] A mérések nem *debug mód*ban történtek, és csak a lényegi részt mértem, tehát pl a mátrixgenerálást nem!



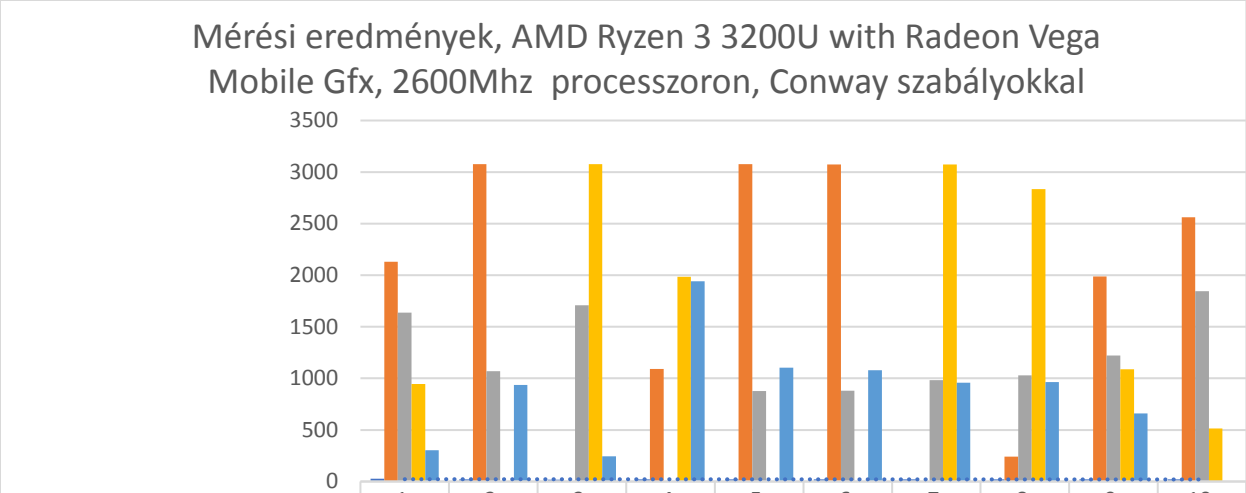
	1. futtatás	2. futtatás	3. futtatás	4. futtatás	5. futtatás	6. futtatás	7. futtatás	8. futtatás	9. futtatás	10. futtatás
idő (s)	33.46	19.02	19.06	19.01	19.04	18.83	19.14	19	19.54	19.04
halak száma induláskor	3068	2008	3075	1687	1698	3076	2146	3	1584	1
halak száma szimuláció végén	1029	601	958	0	756	667	1233	1044	1039	1163
cápák száma induláskor	8	1068	1	1389	1378	0	930	3073	1492	3075
cápák száma szimuláció végén	875	1364	1000	1938	1239	1286	748	848	897	788

Mérési eredmények Conway szabályok alkalmazásával, intel proceszoron



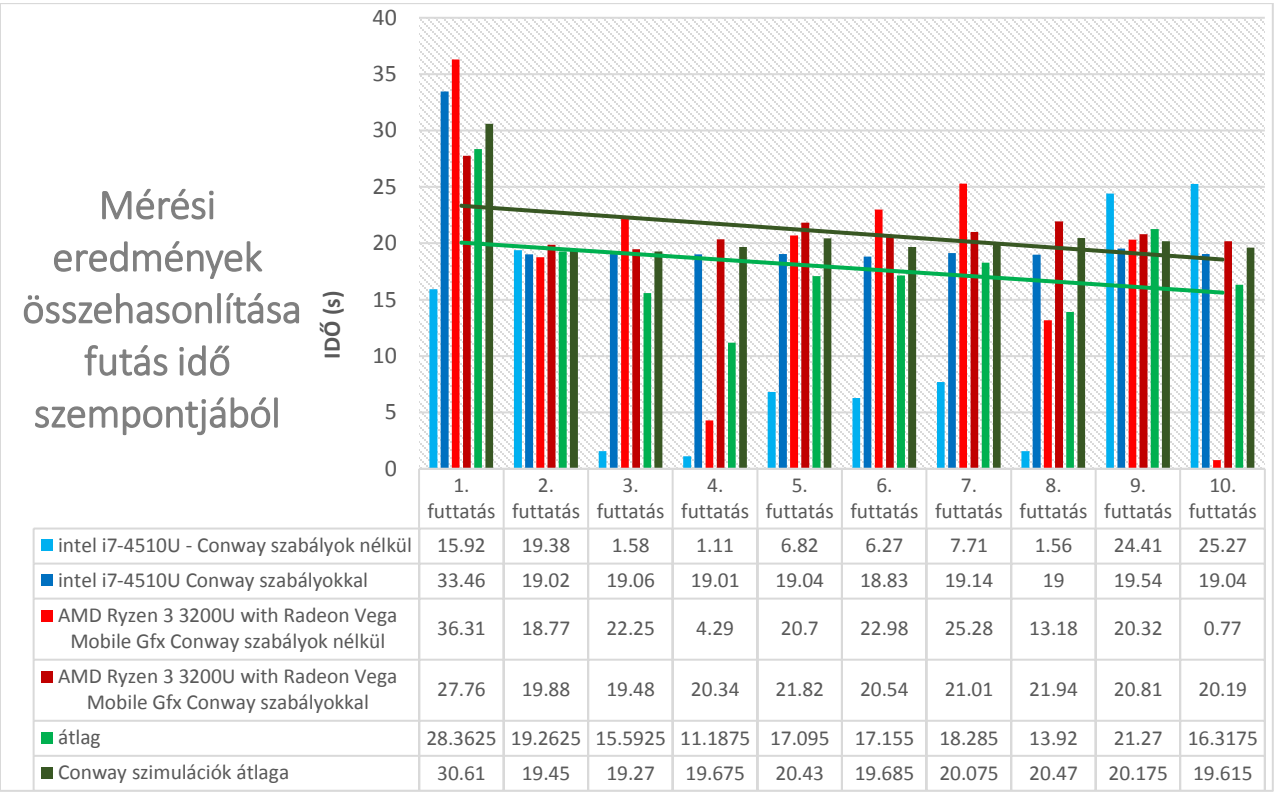
	1. futtatás	2. futtatás	3. futtatás	4. futtatás	5. futtatás	6. futtatás	7. futtatás	8. futtatás	9. futtatás	10. futtatás
idő (s)	36.31	18.77	22.25	4.29	20.7	22.98	25.28	13.18	20.32	0.77
halak száma induláskor	2545	1762	3076	130	1282	3076	1452	2369	1252	1
halak száma szimuláció végén	2822	9917	9865	2338	9849	9748	9878	8975	8974	0
cápák száma induláskor	531	1314	0	2946	1794	0	1624	707	1824	3075
cápák száma szimuláció végén	2	0	0	0	0	0	1	0	0	0

Mérési eredmények Conway szabályok alkalmazása nélkül AMD processzoron



	1. futtatás	2. futtatás	3. futtatás	4. futtatás	5. futtatás	6. futtatás	7. futtatás	8. futtatás	9. futtatás	10. futtatás
idő (s)	27.76	19.88	19.48	20.34	21.82	20.54	21.01	21.94	20.81	20.19
halak száma induláskor	2131	3076	0	1090	3076	3073	2	241	1987	2562
halak száma szimuláció végén	1637	1071	1708	0	876	880	983	1028	1221	1845
cápák száma induláskor	945	0	3076	1986	0	3	3074	2835	1089	514
cápák száma szimuláció végén	304	936	244	1941	1104	1079	959	964	659	0

Mérési eredmények Conway szabályok alkalmazásával AMD processzoron



	1. futtatás	2. futtatás	3. futtatás	4. futtatás	5. futtatás	6. futtatás	7. futtatás	8. futtatás	9. futtatás	10. futtatás
intel i7-4510U - Conway szabályok nélkül	15.92	19.38	1.58	1.11	6.82	6.27	7.71	1.56	24.41	25.27
intel i7-4510U Conway szabályokkal	33.46	19.02	19.06	19.01	19.04	18.83	19.14	19	19.54	19.04
AMD Ryzen 3 3200U with Radeon Vega Mobile Gfx Conway szabályok nélkül	36.31	18.77	22.25	4.29	20.7	22.98	25.28	13.18	20.32	0.77
AMD Ryzen 3 3200U with Radeon Vega Mobile Gfx Conway szabályokkal	27.76	19.88	19.48	20.34	21.82	20.54	21.01	21.94	20.81	20.19
átlag	28.3625	19.2625	15.5925	11.1875	17.095	17.155	18.285	13.92	21.27	16.3175
Conway szimulációk átlaga	30.61	19.45	19.27	19.675	20.43	19.685	20.075	20.47	20.175	19.615

Mérési eredmények futási idejének összehasonlítása 1 szálas futtatás esetén

Több szálas szimuláció megvalósítás

A több szálas futtatás több kérdést is felvet, a matrix kezelésével kapcsolatban. Például, mekkora területeket kezlejenek a szálak? Hány szál legyen? Egymás alatti vagy egymás melletti területeket is feldolgozzanak a szálak? Hogy oszzuk fel a szálak által feldolgozandó területet, úgy, hogy a legkevesebb szinkronizációs időre legyen szükség? És ehhez hasonlók.

Alább az egymás alatti szálakra bontással példa:

1. Szál
2. Szál
3. Szál
4. Szál

Elemek beszínezve azon szálak színével amivel fel lesznek dolgozva:

11	12	13	14	15	16	17	18
21	22	…					…
31	…	33	…				…
41	…		44	…			…
51	…			55	…		…
61	…				66	…	…
71	…					77	…
81	…						88

Ezzel a megközelítéssel nyilvánvalóan a legrosszabb esetben egy szál két másik szállal, legjobb esetben egy másik szállal kell szinkronizálon. Tehát négy szál esetén három szinkronizációs várakozásra van szükség. Ezzel szemben ha egymás mellé teszünk két szálat és ezt megimsételjük a következő sorban akkor már rögtön hat szinkronizációs pont szükséges ha átlósan is léphetnek a halak vagy cápák és négy ha nem léphetnek átlósan. *Megj: a fenti szimulációban engedélyezett az átlós lépés!*

Ugyanakkor ha eredetieleg egy 100x100-as mátrixról van szó akkor az előbbi esetben egy szál egy kb 20 x 100-as részmátrixon dolgozik, és egy (vagy két) 100 hosszú vektor szakaszon lehet érintett szinkronizációs kérdésben[6]. Az utóbbi esetben viszont egy 25 x 25-ös részmátrixon dolgozik egy szál, és egy 49[7] elemű vektor

[6] A teljes eredeti mátrix szélességén.
[7] 25 sorban, 25 oszlopban mínusz egy, hogy a sarkot ne számoljuk kétszer.

szakaszán lehet érintett szinkrnoizációs kérdésben, ugyanakkor ez nem egy teljes szakasz, hanem ha átlós lépéseket is engedélyezünk *(mint a szimulációnkban)* akkor három másik szállal kapcsolódó szinkronizációs pontok, amiből a legkritikusabb a sarokpont ami be kell várja mind a három másik szálat, hiszen mindegyikkel kapcsoaltban áll.

Ha egy négyszet alakba helyezzük el a szálakat, az előző módszert alkalmazva az alábbbi módon ábrázolhatjuk a második esetet:

11	12	13	14	15	16	17	18
21	22	…					…
31	…	33	…				…
41	…		44	…			…
51	…			55	…		…
61	…				66	…	…
71	…					77	…
81	…						88

Jól látszik tehát, hogy ezt a módszert használva konstans három másik szállal kell kommunikáljon egy-egy szál.

Ezek alapján azt figyelembe véve, hogy ajánlott, hogy ugyanakkora szeleteket dolgozzanak fel a szálak, ezért az előbbi verziót választottam megvalósításra, hiszen ahhoz szükséges kevesebb szinkronizációs idő, tehát úgy kaphatunk gyorsabb futást.

Azért, hogy a programot könnyebben lehessen tesztelésre használni négyüzemmódot építettem be a többszálú programba:

1. Szimuláció többszöri futtatási móddal a program által az adott CPU-n megtalált szálak számára optimalizálva, a Conway szabályok mellőzésével.
2. Szimuláció többszöri futtatási móddal a program által az adott CPU-n megtalált szálak számára optimalizálva, a Conway szabályok alkalmazásával.
3. Szimuláció többszöri futtatási móddal a felhasználó által megadott szálak számára optimalizálva, a Conway szabályok mellőzésével.
4. Szimuláció többszöri futtatási móddal a felhasználó által megadott szálak számára optimalizálva, a Conway szabályok alkalmazásával.

A szinkronizáció kérdése

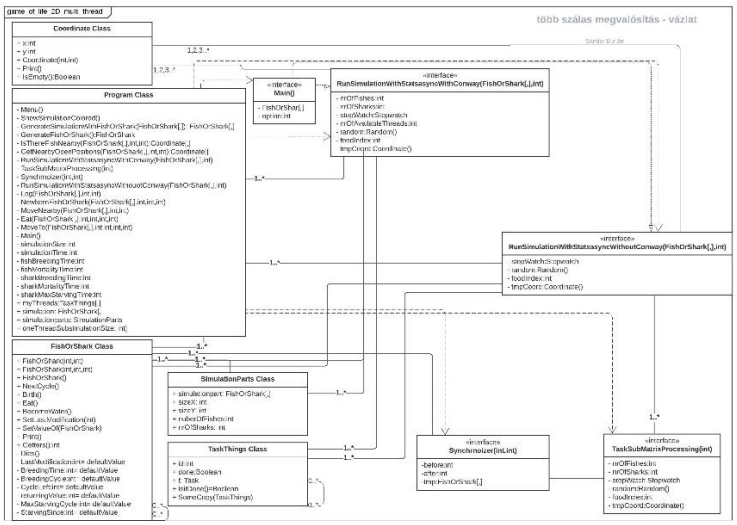
A program futtatása érdekében létrehozunk annyi taskot amennyi szálunk van. Ezek feloldása és lefoglalása fogja jelezni A szálakat egy segéd típusban kezelem. Létrehozok egy akkora tömböt mint a szálak mennyisége, a tomb minden eleme ez a segédobjektum típusú. Ha egy szál végzett a ciklusával, akkor könnyne szinkronizálhat az előtte-utána levő szálakkal mivel a töömbbeli srosázmukkal tudnka könnyen ihavatkozni egymásra.

Mérési eredmények[5]

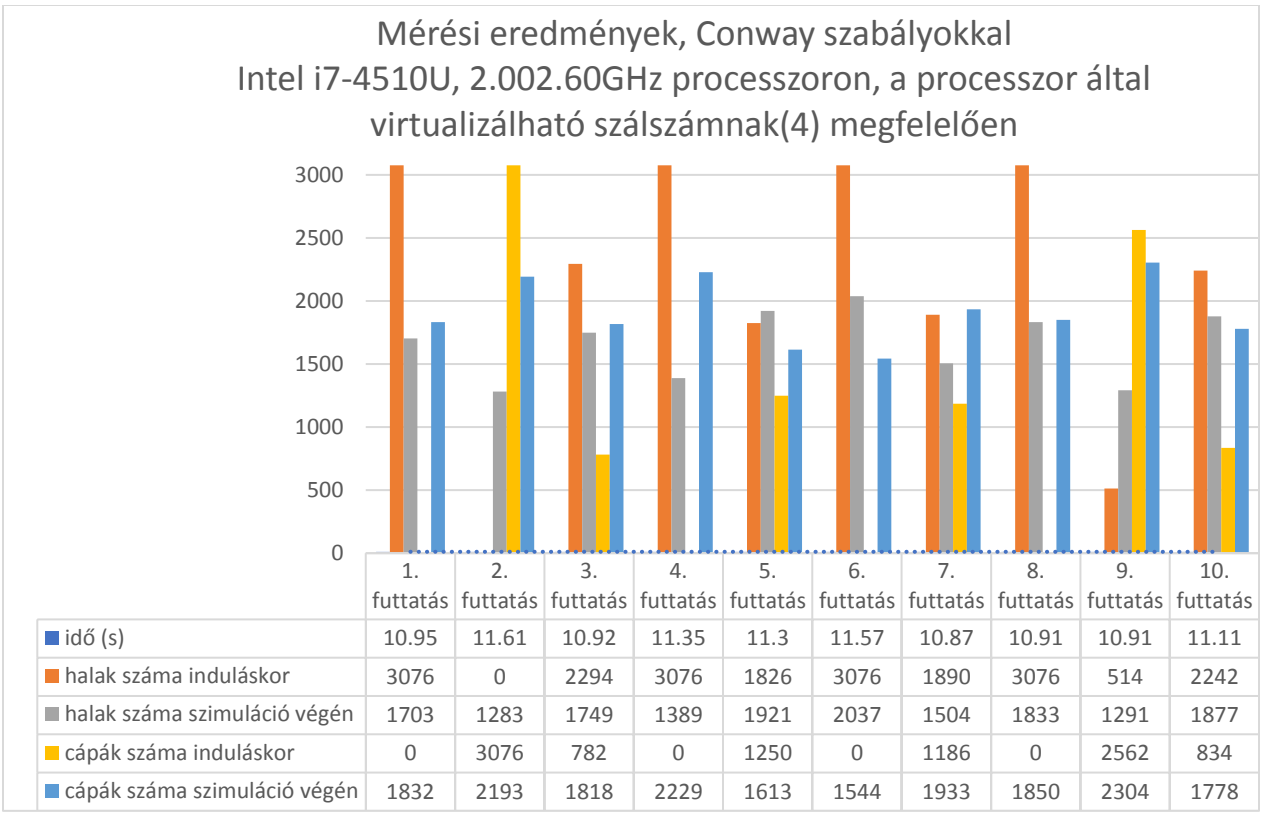
A méréseket a könnyebb összehasolíthatóság végett ugyanazokkal a paraméterekel futtattam:

Szimuláció ciklusainak száma	100
Szimuláció mérete *(nxn)*	100
Halak halálozási ideje	10
Halak nemzési ideje	5
Cápák halálozási ideje	15
Cápák nemzési ideje	10
Cápák maximum éhezési ideje	5

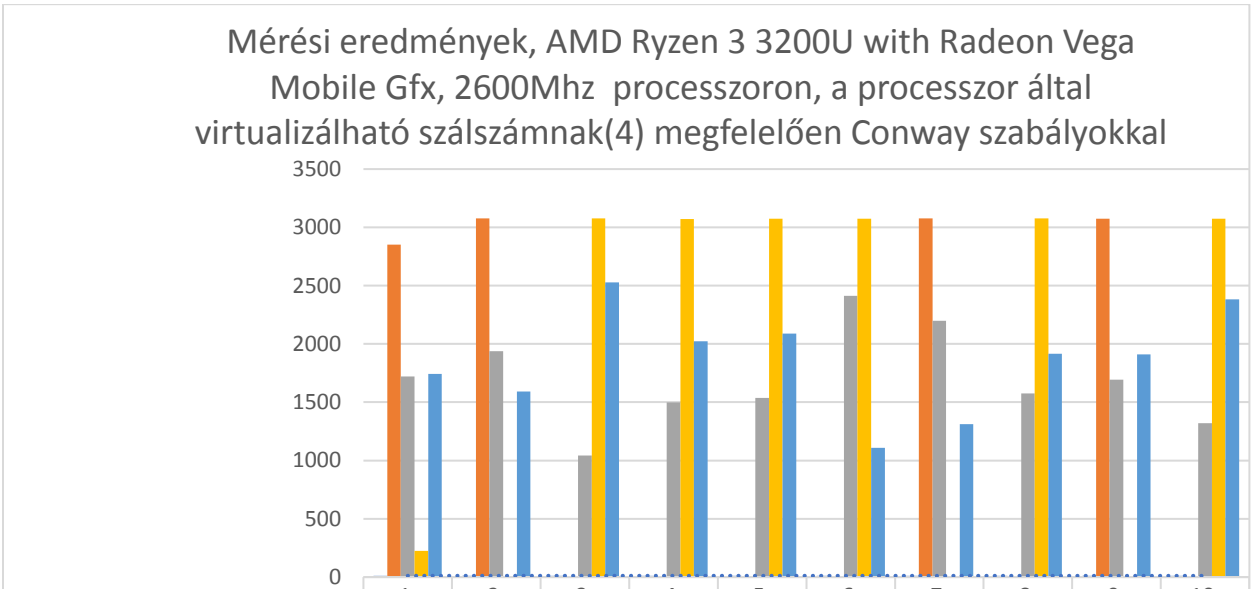
A méréseket jelen esetben csak Conway módszerrel végeztem, mivel eddig is tapasztaltuk, hogy mennyire jól tudja normalizálni a szimulációt. Felmerülhet a kérdés, hogy mit várhatunk ettő a megoldástól, egéyrtelműen több műveletet végzünk, de azt nem egy szálon, becsléseim szerint egy-egy szál legalább $sz * n + 2 * n$ objektum műveletet[8] végez, és legfeljebb $(sz * n) * 2 + n$ műveletet végez ahol *n* az eredeti matrix mérete, és *sz* a szálszám; a szinkronizációhoz pluszba az egy szálas megoldáshoz képest. Ugyanakkor a szinkronizációs műveletek egyszerre két szálat is érintenek, így várható, hogy felgyorsul a futás. Alább a program lényegi részéről készült UML vázlat:



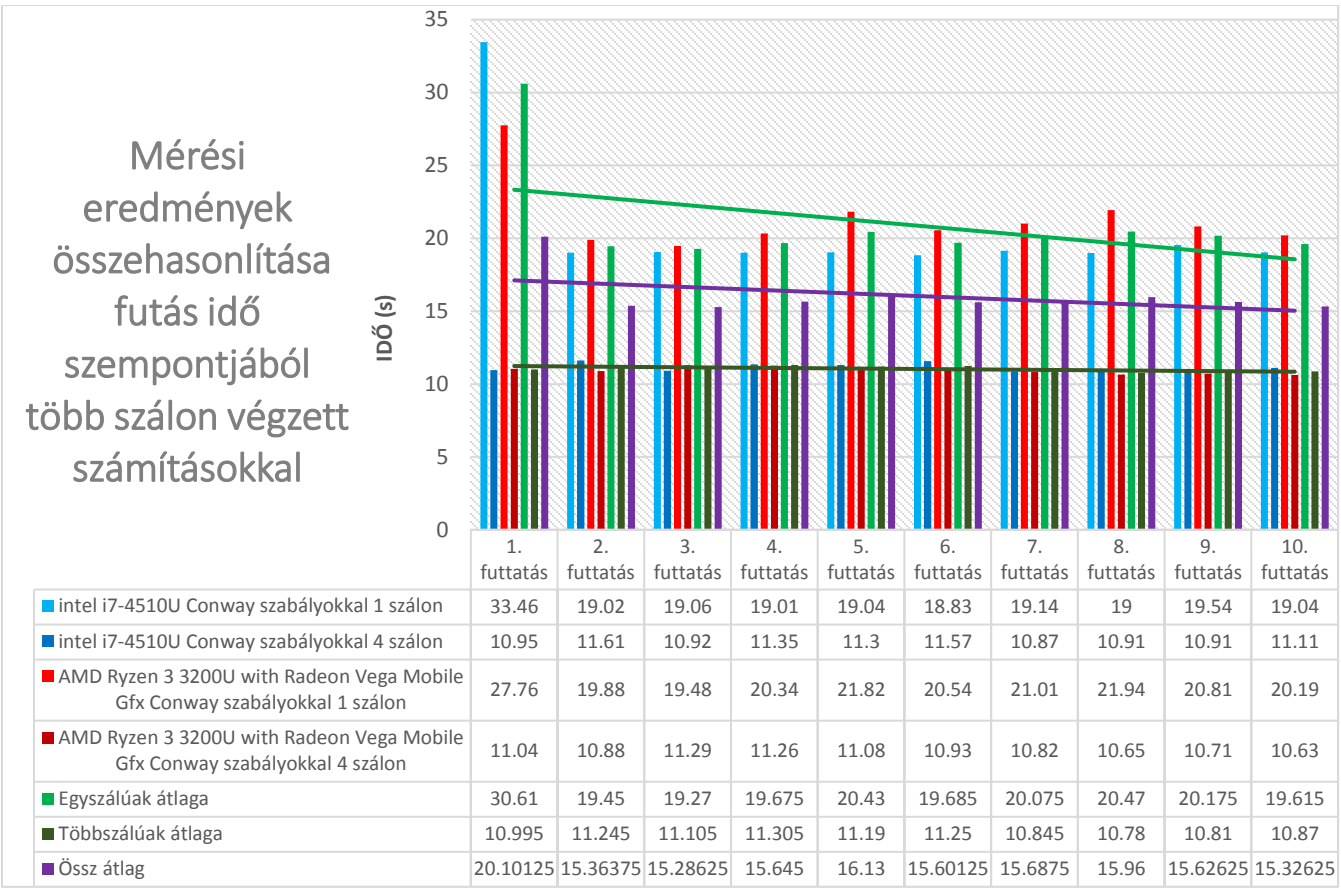
[8] Köztük komplexebb műveletek is, nem csak iteráció az objektumon, szűrés, keresés stb



	1. futtatás	2. futtatás	3. futtatás	4. futtatás	5. futtatás	6. futtatás	7. futtatás	8. futtatás	9. futtatás	10. futtatás
idő (s)	10.95	11.61	10.92	11.35	11.3	11.57	10.87	10.91	10.91	11.11
halak száma induláskor	3076	0	2294	3076	1826	3076	1890	3076	514	2242
halak száma szimuláció végén	1703	1283	1749	1389	1921	2037	1504	1833	1291	1877
cápák száma induláskor	0	3076	782	0	1250	0	1186	0	2562	834
cápák száma szimuláció végén	1832	2193	1818	2229	1613	1544	1933	1850	2304	1778



	1. futtatás	2. futtatás	3. futtatás	4. futtatás	5. futtatás	6. futtatás	7. futtatás	8. futtatás	9. futtatás	10. futtatás
idő (s)	11.04	10.88	11.29	11.26	11.08	10.93	10.82	10.65	10.71	10.63
halak száma induláskor	2851	3076	0	4	2	1	3076	0	3075	1
halak száma szimuláció végén	1722	1939	1043	1499	1538	2413	2198	1575	1694	1321
cápák száma induláskor	225	0	3076	3072	3074	3075	0	3076	1	3075
cápák száma szimuláció végén	1743	1593	2527	2022	2088	1108	1312	1917	1910	2383



	1. futtatás	2. futtatás	3. futtatás	4. futtatás	5. futtatás	6. futtatás	7. futtatás	8. futtatás	9. futtatás	10. futtatás
intel i7-4510U Conway szabályokkal 1 szálon	33.46	19.02	19.06	19.01	19.04	18.83	19.14	19	19.54	19.04
intel i7-4510U Conway szabályokkal 4 szálon	10.95	11.61	10.92	11.35	11.3	11.57	10.87	10.91	10.91	11.11
AMD Ryzen 3 3200U with Radeon Vega Mobile Gfx Conway szabályokkal 1 szálon	27.76	19.88	19.48	20.34	21.82	20.54	21.01	21.94	20.81	20.19
AMD Ryzen 3 3200U with Radeon Vega Mobile Gfx Conway szabályokkal 4 szálon	11.04	10.88	11.29	11.26	11.08	10.93	10.82	10.65	10.71	10.63
Egyszálúak átlaga	30.61	19.45	19.27	19.675	20.43	19.685	20.075	20.47	20.175	19.615
Többszálúak átlaga	10.995	11.245	11.105	11.305	11.19	11.25	10.845	10.78	10.81	10.87
Össz átlag	20.10125	15.36375	15.28625	15.645	16.13	15.60125	15.6875	15.96	15.62625	15.32625

Következtetés:

Hiába a szálak egyenként több művelete a szinkrnizációk miatt, a futásuk egyértelműen gyorsabb, már-már konstans idővel számolhatunk a futás időt tekintve.

Forrásjegyzet

Borítókép: [Online] Available: https://regi.tankonyvtar.hu/hu/tartalom/tamop412A/2011-0063_13_parhuzamos_algoritmusmodellek/images/fig3-9.jpg. [Hozzáférés dátuma: 2020 Szeptember 28]

[1] Environment.ProcessorCount Property [Online]. Available: https://docs.microsoft.com/en-us/dotnet/api/system.environment.processorcount?redirectedfrom=MSDN&view=netcore-3.1#System_Environment_ProcessorCount [Hozzáférés dátuma: 29 09 2020]

[2] TÖBBSZÁLÚ/TÖBBMAGOS PROCESSZORARCHITEKTÚRÁK PROGRAMOZÁSA [Online] Available: https://elearning.uni-obuda.hu/main/pluginfile.php/274151/mod_resource/content/1/0053_Tobbszalu_Tobbmagos_Processzorarchitekturak_Programozasa.pdf [Hozzáférés dátuma: 29 09 2020]

[3] Nancy Ann Lynch: Oszott algoritmusok, ISBN: 963-9301-03-5 Kiskapu kiadó, 2020 április 18, 479. Oldal

[4] Intel Core i7-4510U @ 2.00GHz vs AMD Ryzen 3 3200U [Online] Available: https://www.cpubenchmark.net/compare/Intel-i7-4510U-vs-AMD-Ryzen-3-3200U/2248vs3431 [Hozzáférés dátuma: 2 11 2020]

UML diagrammokat a https://lucid.app segítségével készítettem.