**Szoftverfejlesztés párhuzamos architektúrákra**

**Sipos Levente - D985ET**

**Beadandó feladat**

**Paraméteres görbék síkbeli megjelenítése párhuzamosan**

Az alap algoritmust C#-on belül egy DLL-en belül valósítottam meg, melynek metódusait egy WinForm-os alkalmazáson belül hívtam meg. Melynek a kódja a github linken keresztül elérhető. (https://github.com/D985ET/Beadando\_par\_D985ET)

Háromféle megoldást készítettem a párhuzamosításra. Minden módszer egy adott stratégiát alkalmaz a számítások párhuzamosítására, különböző szinkronizálási és teljesítmény-optimalizációs technikákkal.

**1. Párhuzamos Metódus: Szálankénti Felosztás Async/Task alapokon**

**Megoldás jellemzői:**

* A teljes intervallumot **threadCount** számú szálra osztja fel.
* Minden szál a saját tartományában számolja ki és rajzolja meg a vonalakat.
* **Task.Run()** segítségével aszinkron módon indítja a szálakat.
* A rajzolási műveletek közvetlenül a párhuzamos szálakból történnek egy **lock(g)** zárolással.
* Színváltozást egy közös változó alapján végez.

### Kiemelkedő különbség:

Az egyes szálak közvetlenül rajzolnak a Graphics objektumra, emiatt szükség van a lock(g) zárolásra.

**Előnyök:**

Könnyen implementálható és kihasználja az aszinkron műveletek előnyeit.

**Task.Run()** és **await Task.WhenAll()** biztosítja a könnyen kezelhető párhuzamosítást.

Skálázható több szálra.

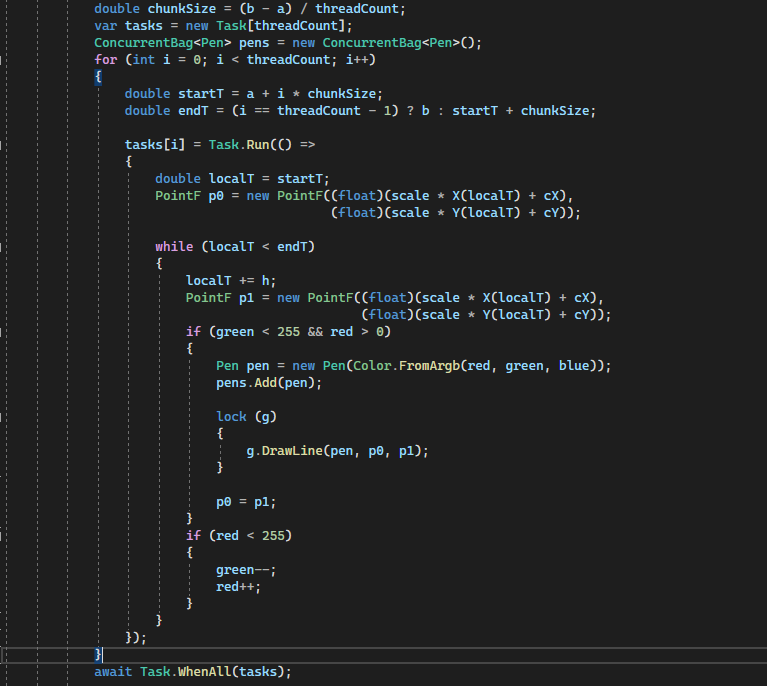
**Hátrányok:**

A **Graphics g** objektumot zárolni kell a párhuzamos szálak között, ami teljesítménycsökkenést okozhat.

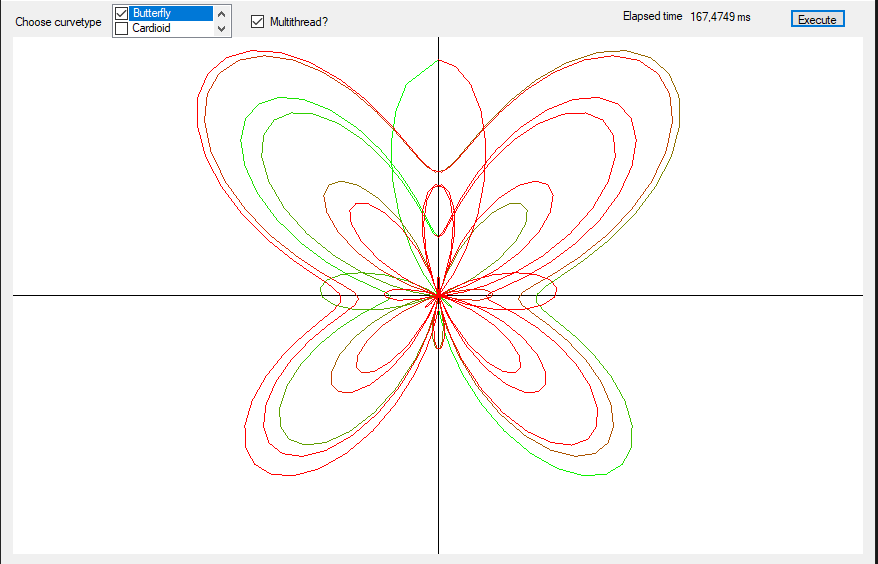
A **red/green/blue** változók megosztottak, és szinkronizációs problémák léphetnek fel.

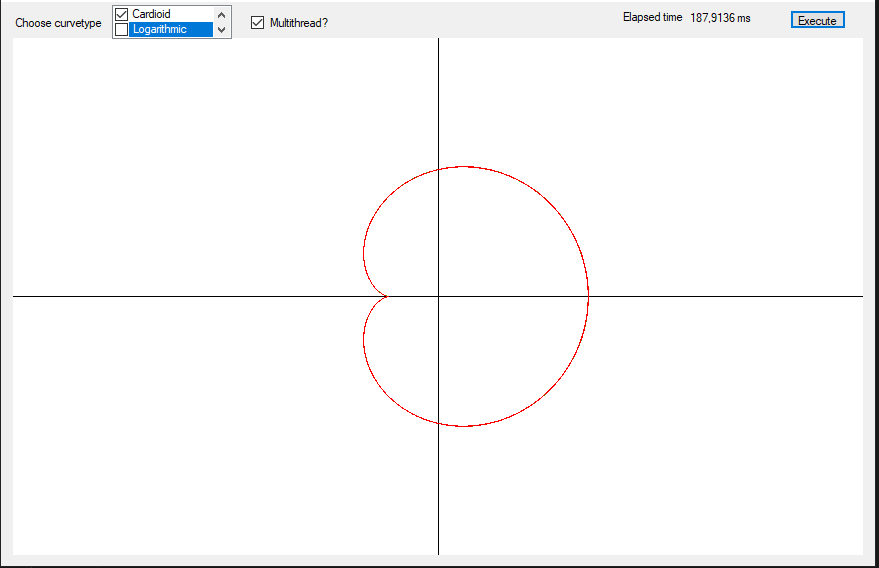
A **lock(g)** művelet a kritikus szakasz miatt csökkentheti a tényleges párhuzamos végrehajtást.

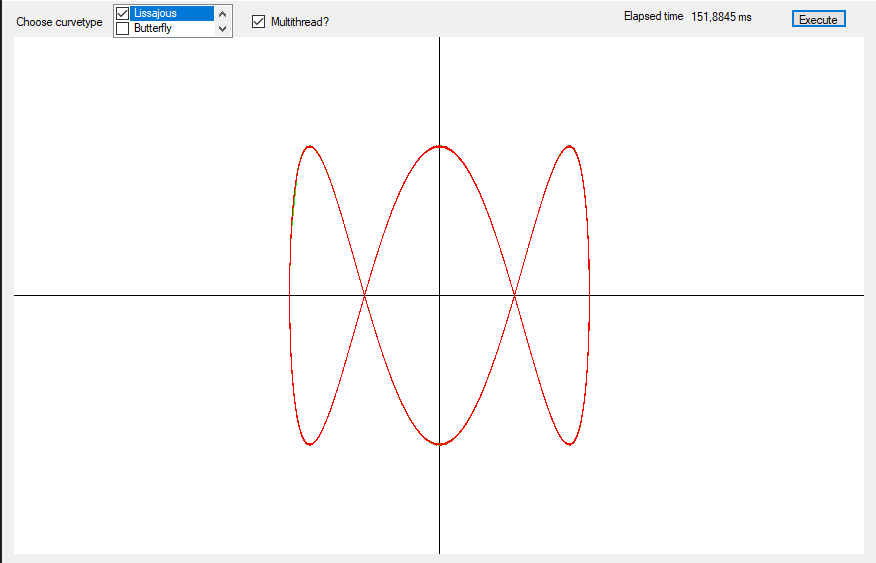
Fő kódbeli változtatás:

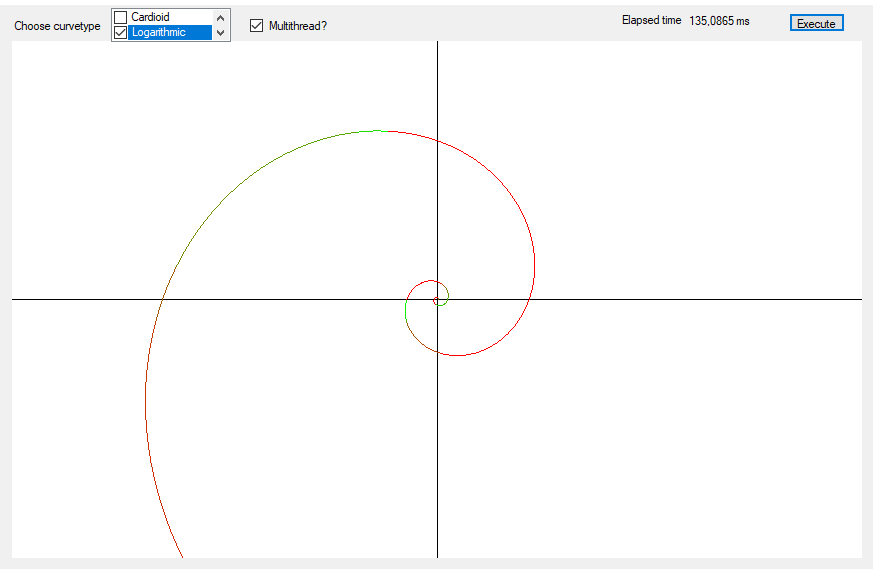


Futási ideje: 130 ~ 190 ms









**2. Párhuzamos Metódus: Szálankénti Előszámítás és Utólagos Rajzolás Thread**

**Megoldás jellemzői:**

* A számításokat **threadCount** számú szál párhuzamosan végzi el.
* Minden szál a saját részintervallumát dolgozza fel és eltárolja a rajzolási műveleteket egy **ConcurrentBag** listába.
* A tényleges rajzolás **egy külön szálon,** történik a végén.
* A **ManualResetEventSlim** segíti a szinkronizációt és az összes szál befejeződésének várakozását.
* Színkezelés minden szálban lokális, nincs ütközés.

### Kiemelkedő különbség:

Az egyes szálak **csak előszámítást végeznek**, a rajzolást pedig egyetlen szál hajtja végre, így nincs szükség zárolásra (lock).

**Előnyök:**

**Race condition mentes**, mert a rajzolás egy külön szálon történik az adatok előkészítése után.

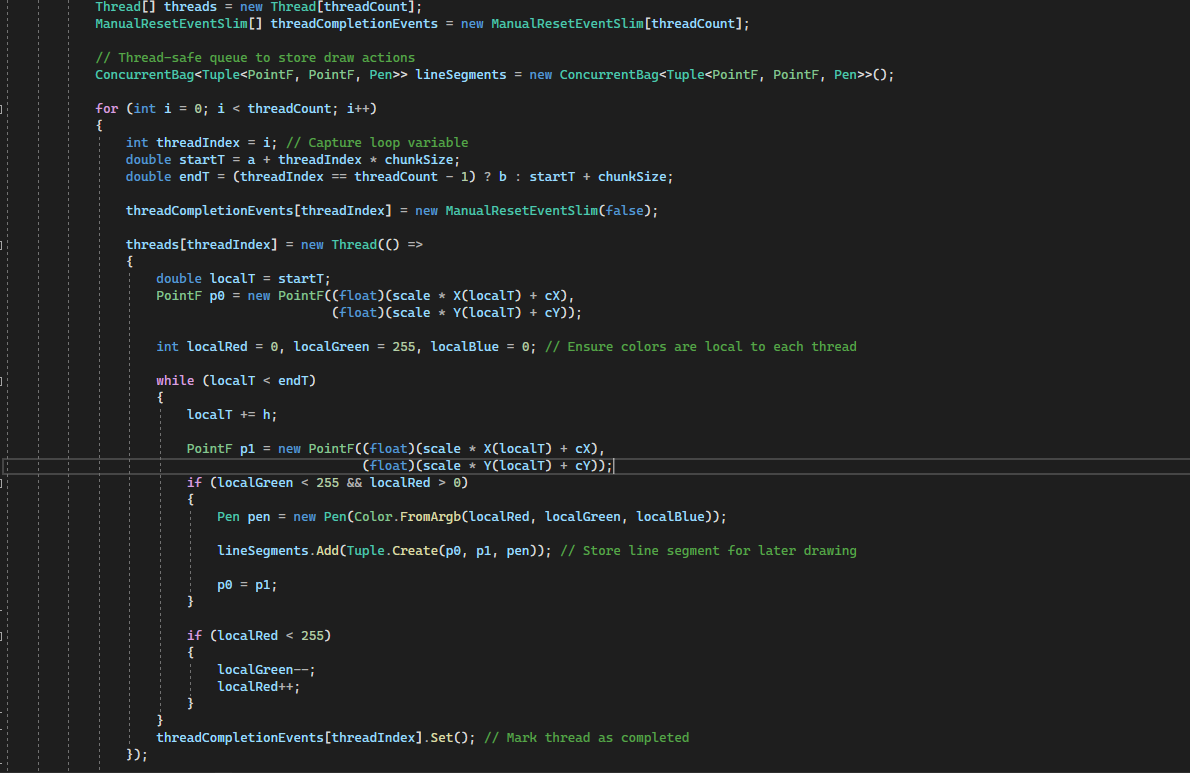
Nincs szükség lock(g)-re a párhuzamos szálak között, mivel a rajzolás utólag történik.

A memóriahasználat optimalizált, mert csak a szükséges adatokat tárolja el.

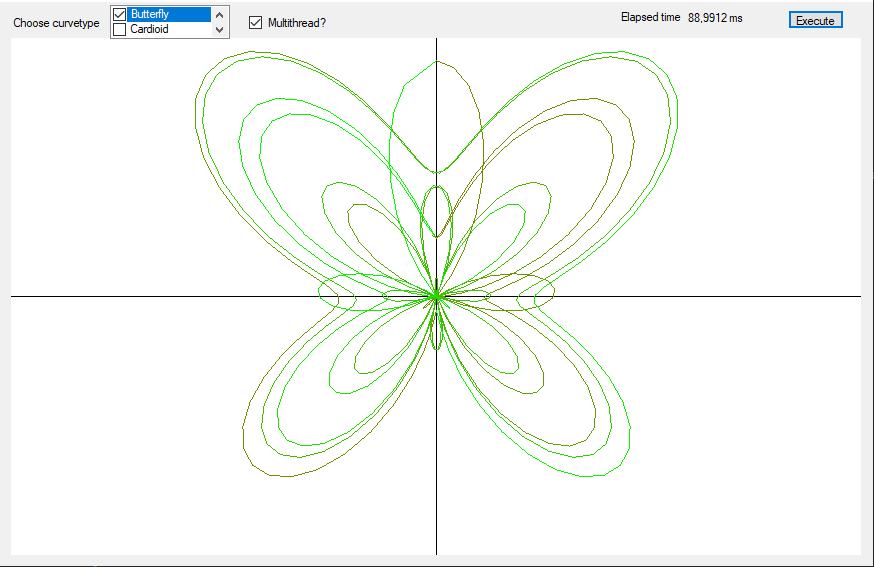
**Hátrányok:**

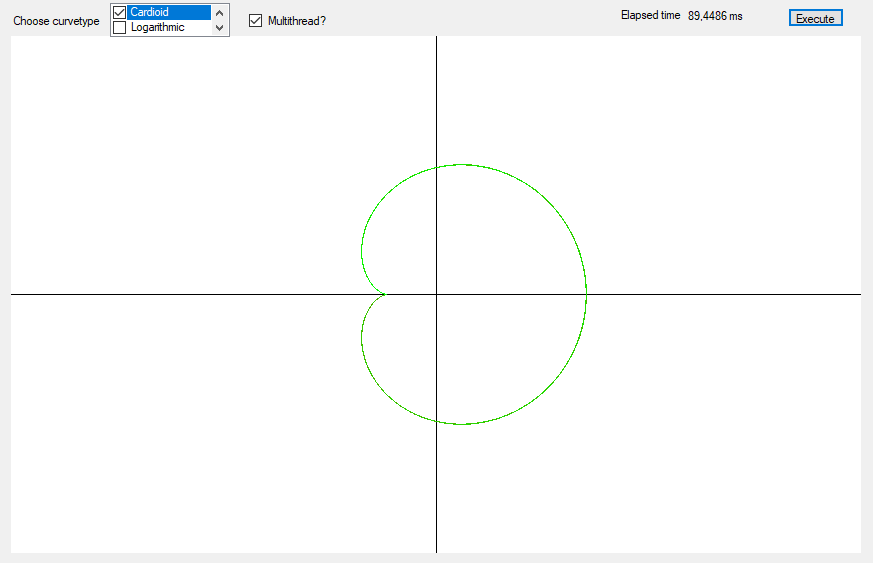
Kétszer kell bejárni az adatokat (előszámítás + rajzolás), ami némi plusz időt igényel.

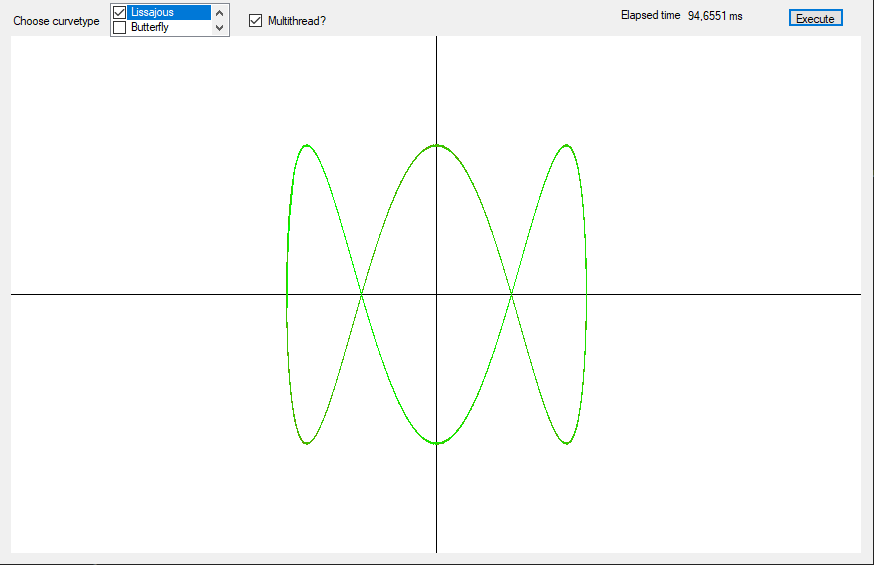
Nagy számú vonal esetén a **ConcurrentBag** kezelése is CPU overheadet jelenthet.

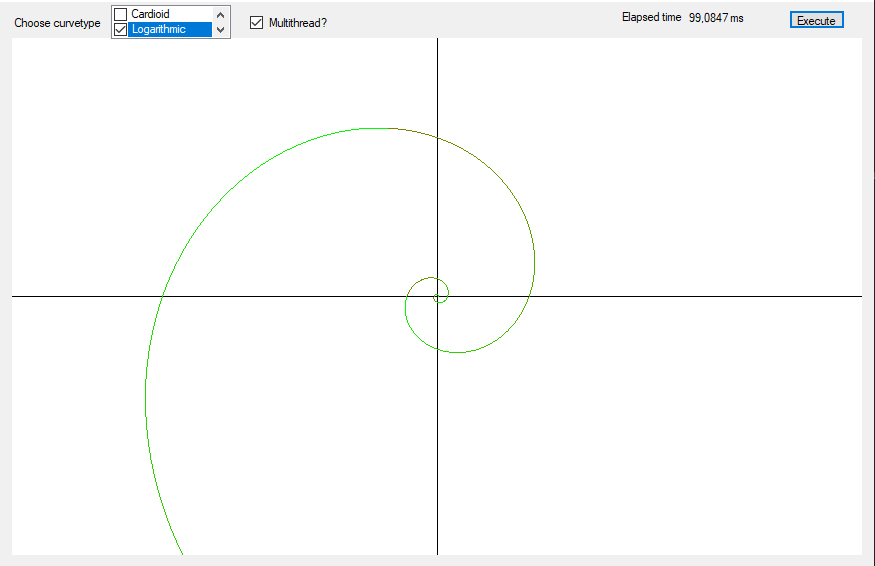
Fő kódbeli változtatás: 

Futási ideje: 88 ~ 99 ms









**3. Párhuzamos Metódus: Parallel.For alapú Végrehajtás**

**Megoldás jellemzői:**

* **Parallel.For()** használata a szálak kezelésére a natív **Thread** vagy **Task** helyett.
* Az egyes szálak saját részinformációikat előállítják és **ConcurrentBag** struktúrába mentik.
* A végső rajzolási művelet **sorosan történik**, így nincs szükség rajzolási zárolásra.

**Előnyök:**

**Parallel.For()** jobban optimalizált, mint a kézi szálkezelés, mivel automatikusan kezeli a szálakat és az erőforrásokat.

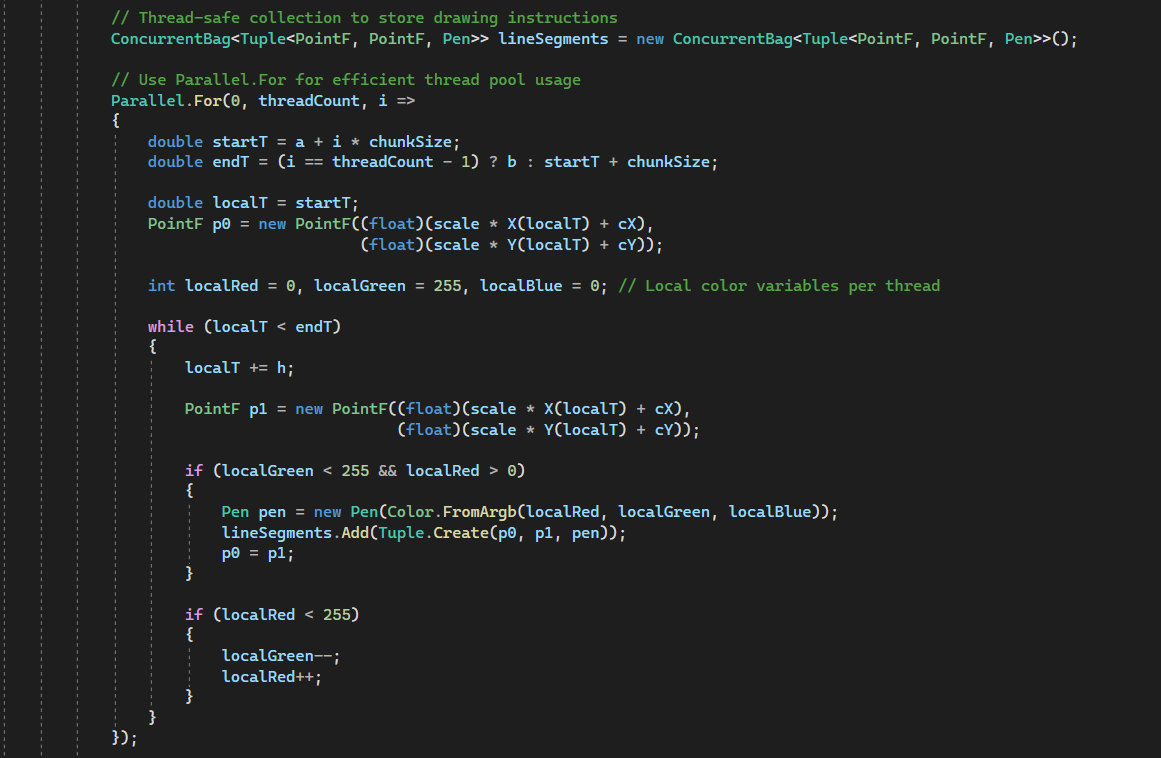
Nincs **lock(g)** szükségesség, mivel az adatok külön gyűjtődnek, majd egy külön szálon rajzolódnak ki.

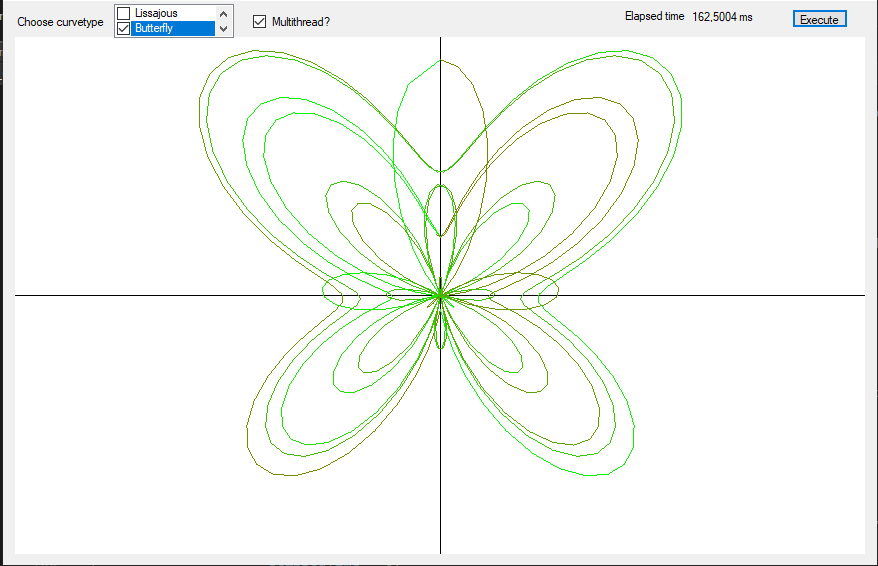
Jobb CPU-terhelés elosztás, mivel a .NET ThreadPool kezeli a szálakat.

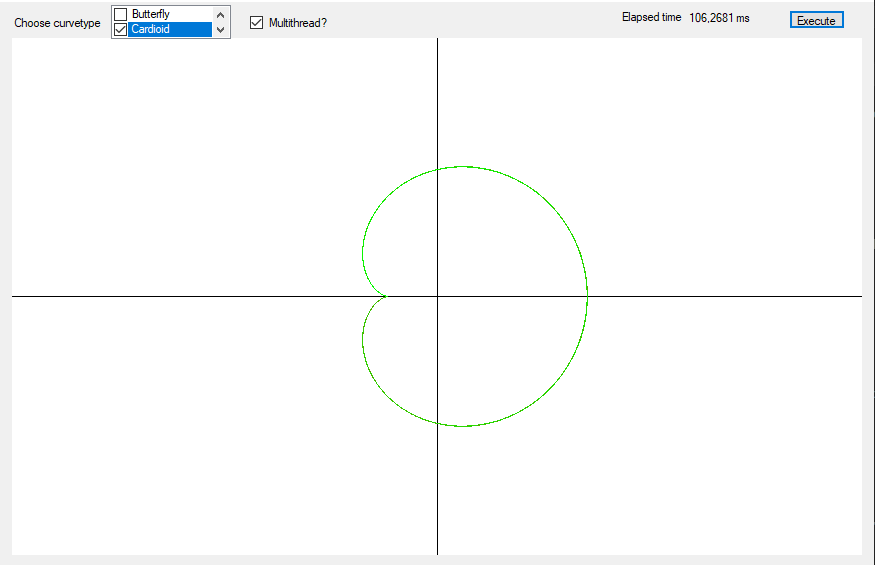
**Hátrányok:**

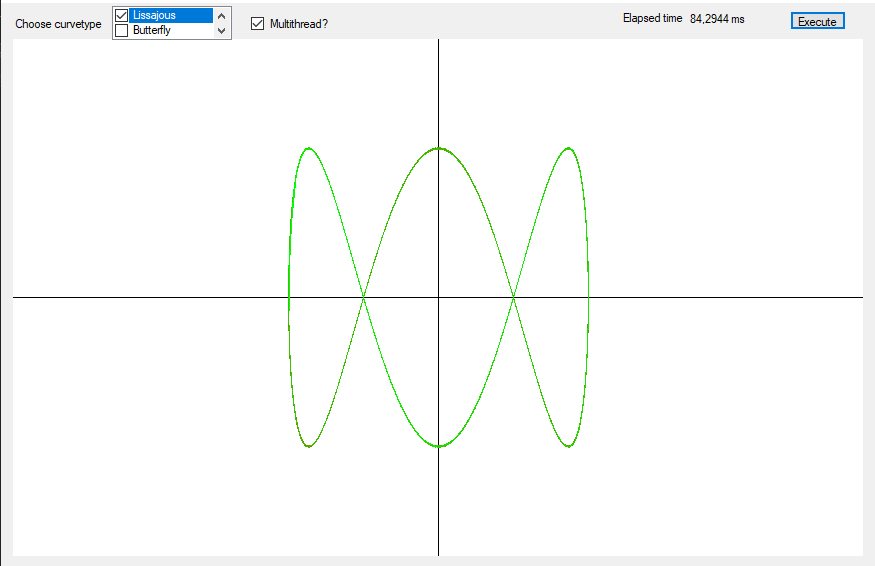
**Parallel.For()** a .NET menedzselt szálkezelésére támaszkodik, így **ritkán előfordulhat**, hogy nem skálázódik jól alacsony szálhasználat esetén. **ConcurrentBag** használata extra memóriahasználatot eredményezhet nagyobb adatok esetén.

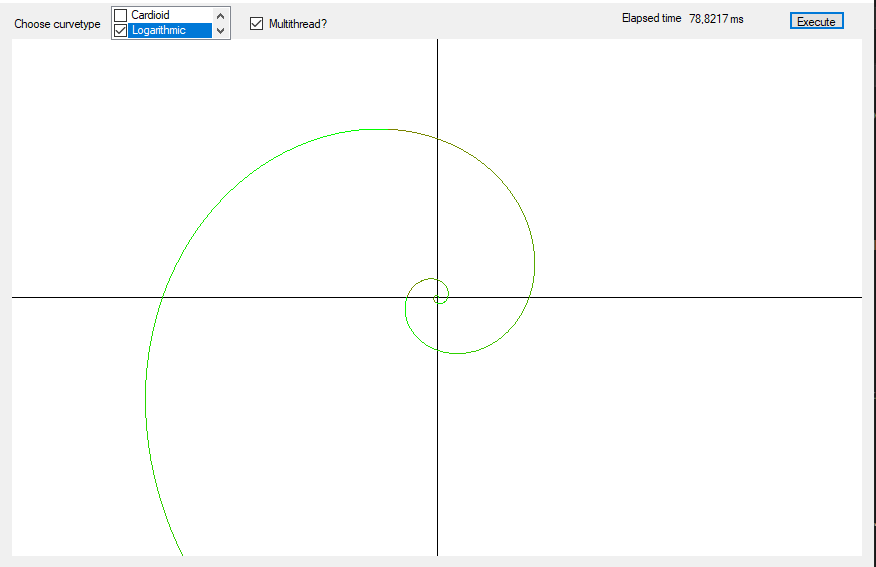
Fő kódbeli változtatás:

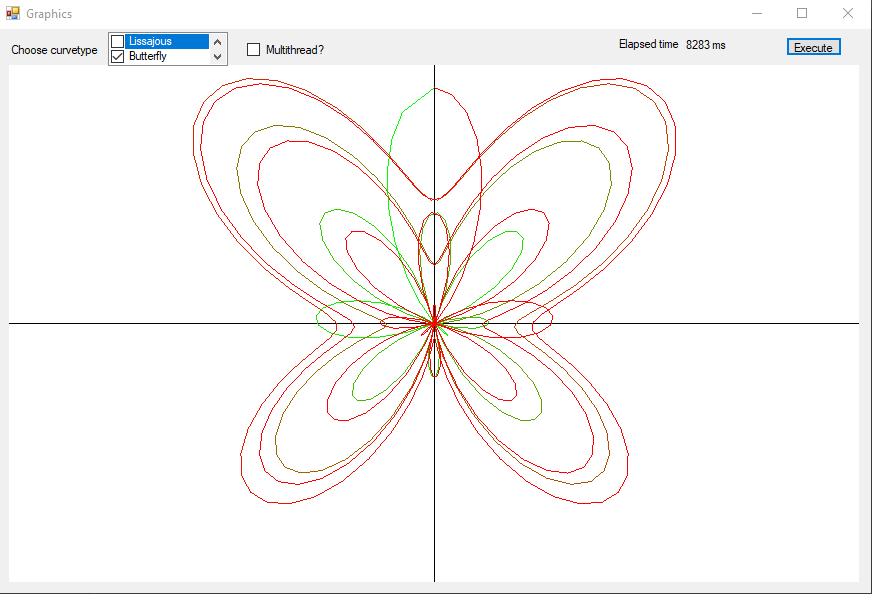


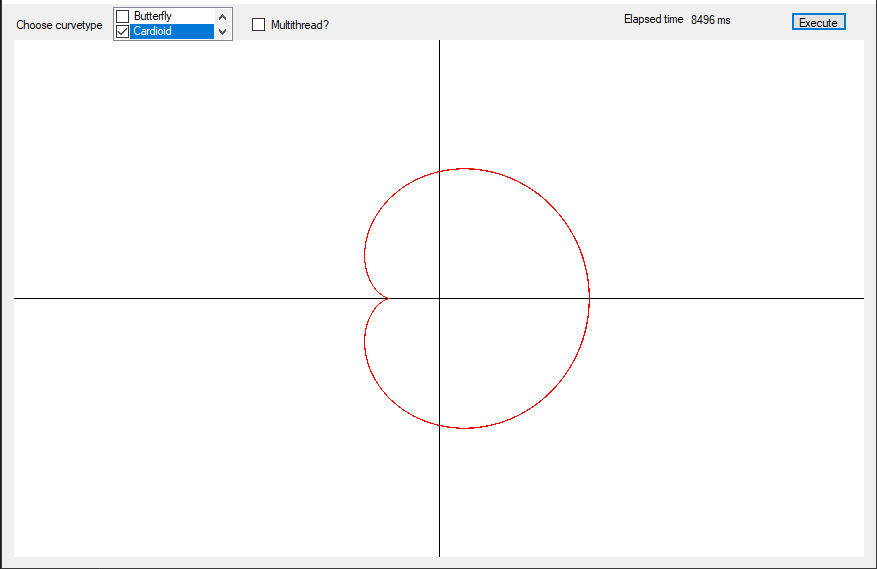
Futási ideje: 75 ~ 163 ms

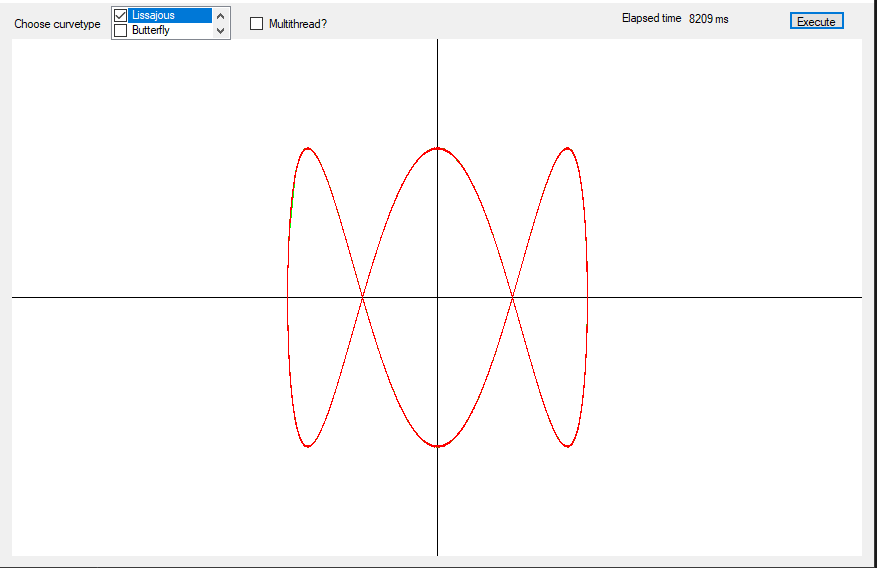


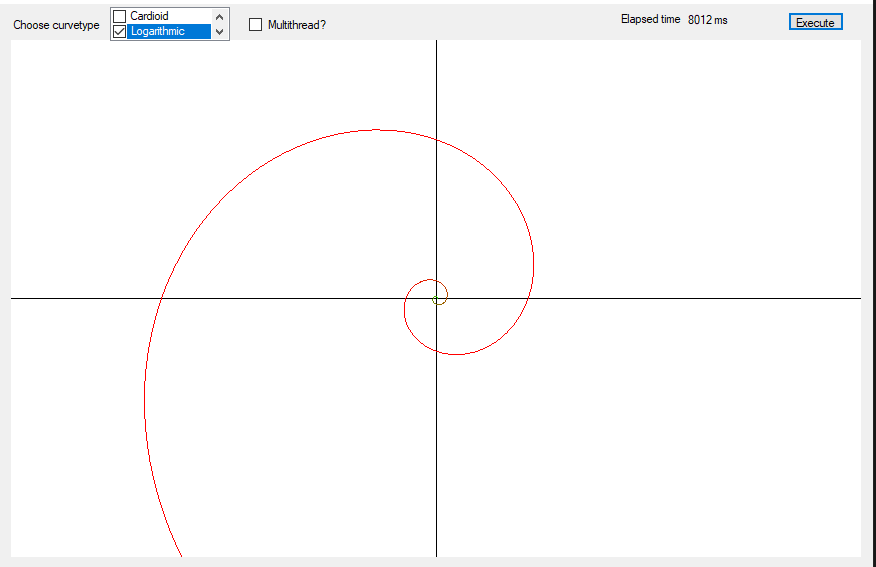




Szekvenciális működés futási ideje: 8000 ~ 8300 ms 







**Összegzés:**

Futási idő alapján a **2. megoldás (Szálankénti előszámítás és utólagos rajzolás)** tűnik a legoptimálisabbnak, mivel a futási ideje nemcsak közel van a legalacsonyabbhoz, hanem **stabilan és kis szórással teljesít** különböző adatmennyiségeknél és szálbeállítások mellett is. Ez részben annak köszönhető, hogy a számítás és a rajzolás **szét van választva**, így a szálak csak a saját részfeladataikra koncentrálnak, és nincs szükség lock() használatára. A lock elhagyása jelentős teljesítménynyereséget hoz, mivel elkerülhetőek az erőforrás-zárolásból eredő várakozási idők.

Ez a megközelítés **kiegyensúlyozott kompromisszumot kínál** a párhuzamos végrehajtás előnyei és a szinkronizációs bonyodalmak elkerülése között. További előnye, hogy könnyen kontrollálható, és a vizuális kimenet is konzisztens marad, mivel a rajzolás egyetlen, egységes szálon történik, meghatározott sorrendben.

A **3. megoldás (Parallel.For)** szintén kiváló választás lehet, különösen akkor, ha a **legrövidebb futási idő elérése** a cél. A Parallel.For belsőleg optimalizálja a szálkezelést, és automatikusan szétosztja a terhelést a rendelkezésre álló processzormagok között. Ez általában **jobb CPU-kihasználtságot** és rövidebb teljes időt eredményez. Ugyanakkor ez a megoldás kissé kevésbé kiszámítható, mivel a Parallel.For működése és ütemezése nem mindig teljesen determinisztikus, különösen ha eltérő környezetben (pl. gyengébb gépen vagy multitask alatt) futtatjuk.

Összességében elmondható, hogy a 2. megoldás ajánlott, ha **konzisztens teljesítményre és biztonságos működésre** törekszünk, míg a 3. megoldás akkor javasolt, ha a **maximum sebesség** az elsődleges szempont, és nem okoz gondot a némi plusz memóriahasználat vagy komplexitás.