Hazırlayan: **Mustafa Koçumoğlu 19360859062**

# 1 PROBLEMİN TANIMI VE AMACI

Projemin temel amacı verilen x ve y koordinat çiftlerinden oluşan bir poligon var. (Bu poligon dışbükey [convex] veya içbükey [concave] olabilir). Kontrol edilmesi gereken çok sayıda nokta kümesi var. Benim görevim, bu noktaların her birinin poligonun içinde mi yoksa dışında mı olduğunu paralel programlama yaklaşımlarını kullanarak belirlemek.

Bu tür bir problem, coğrafi bilgi sistemlerinde (GIS), bilgisayar grafiklerinde, oyun geliştirmede veya çarpışma tespiti gibi birçok alanda karşımıza çıkabiliyor. Özellikle çok sayıda noktanın hızlıca kontrol edilmesi gerektiğinde, paralel programlama büyük bir avantaj sağlıyor.

# 2 TEK BİR NOKTA İÇİN POLİGON KONTROLÜ: RAY CASTİNG ALGORİTMASI

Paralel programlamaya geçmeden önce, en temel soruyu cevaplamam gerekir: Tek bir noktanın tek bir poligonun içinde olup olmadığını nasıl belirleyebilirim? Araştırmalarım sonucunda Ray Casting (Işın Atma) Algoritması'nın bu iş için oldukça uygun ve anlaşılır olduğunu öğrendim.

## 2.1 RAY CASTİNG ALGORİTMASI NASIL ÇALIŞIR?

Algoritmanın mantığı oldukça basit:

1. Kontrol etmek istediğim noktadan (buna sorgu noktası diyelim) sağa doğru sonsuz uzunlukta hayali bir yatay ışın çiziyorum.
2. Bu ışının, poligonun kenarlarını kaç kez kestiğini sayıyorum.
3. Eğer kesişim sayısı tek ise, nokta poligonun içindedir.
4. Eğer kesişim sayısı çift ise, nokta poligonun dışındadır.

## 2.2 PYTHON UYGULAMASI (TEK NOKTA İÇİN)

Ray Casting algoritmasını Python'da uygulamak için, Point adında küçük bir sınıf oluşturdum ve poligonu bu Point nesnelerinin bir listesi olarak tanımladım. İşte temel *is_point_in_polygon* fonksiyonum:

C: > Users > musta > OneDrive > Masaüstü > Paralel_Programlama.py > ...

```python
import multiprocessing
from concurrent.futures import ProcessPoolExecutor
import time
import random
from functools import partial

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"Point({self.x}, {self.y})"

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

def is_point_in_polygon(point, polygon_vertices):

    n = len(polygon_vertices)
    if n < 3:
        return False

    inside = False

    for i in range(n):
        j = (i + 1) % n
        p1 = polygon_vertices[i]
        p2 = polygon_vertices[j]

        if ((p1.y > point.y) != (p2.y > point.y)):
            if p2.y - p1.y == 0:
                continue

            intersect_x = p1.x + (point.y - p1.y) * (p2.x - p1.x) / (p2.y - p1.y)

            if intersect_x > point.x:
                inside = not inside

    return inside
```

# 3 PARALEL PROGRAMLAMA İLE PERFORMANSI ARTIRMA

Tek bir noktanın kontrolü kolay olsa da projenin asıl zorluğu ve amacı çok sayıda noktayı paralel olarak kontrol etmek. Binlerce, hatta yüz binlerce noktayı tek tek kontrol etmek çok uzun sürebilir. İşte burada paralel programlama devreye giriyor.

## 3.1 NEDEN PARALEL PROGRAMLAMA?

Her bir noktanın poligon içinde olup olmadığını kontrol etme işlemi, diğer noktalardan bağımsızdır. Bu bağımsızlık, iş yükünü birden fazla işlemci çekirdeğine dağıtarak aynı anda birden fazla kontrol yapabilmemizi sağlıyor. Böylece toplam işlem süresini önemli ölçüde azaltabiliyor.

Python'da paralel programlama için threading ve multiprocessing modülleri bulunuyor. Threading aynı program içinde hafif iş parçacıkları oluştururken, Python'ın Global Interpreter Lock (GIL) kısıtlaması nedeniyle CPU yoğun işlerde gerçek paralel işlem (aynı anda birden fazla çekirdeği kullanma) sağlamaz. Benim problemim CPU yoğun olduğu için, multiprocessing modülünü kullanmaya karar verdim. multiprocessing, her biri kendi bellek alanına sahip ayrı süreçler (process) oluşturarak gerçek paralel işlem gücünden faydalanmamı sağlıyor.

## 3.2 PARALEL UYGULAMA YAKLAŞIMI

Uygulama için *multiprocessing.Pool* ve *concurrent.futures.ProcessPoolExecutor* yapılarını kullandım. Bu yapılar, belirli sayıda süreci yöneten bir "havuz" oluşturmamı ve görevleri (nokta kontrol işlemlerini) bu süreçlere otomatik olarak dağıtmamı sağlıyor.

*is_point_in_polygon* fonksiyonu, her bir noktanın kontrolünü yaparken poligon verisine de ihtiyaç duyuyordu. multiprocessing ile bu gibi sabit argümanları verimli bir şekilde iletmek için *functools.partial* yapısını kullandım. Bu yöntem, *is_point_in_polygon* fonksiyonunun poligon argümanını sabitleyerek, *executor.map* fonksiyonuna yalnızca kontrol edilecek noktaları göndermemi sağladı. Bu sayede her bir süreç, kendisine atanan noktalar için poligon kontrolünü bağımsız olarak gerçekleştirebildi.

```
53  def generate_random_points(count, x_min, x_max, y_min, y_max):
54      points = []
55      for _ in range(count):
56          x = random.uniform(x_min, x_max)
57          y = random.uniform(y_min, y_max)
58          points.append(Point(x, y))
59      return points
60
61  num_test_points = 100000
62  test_points = generate_random_points(num_test_points, -5, 15, -5, 15)
63
64
65  def check_points_parallel(points, polygon_vertices):
66
67      results = []
68
69      with ProcessPoolExecutor(max_workers=multiprocessing.cpu_count()) as executor:
70
71          func_with_polygon = partial(is_point_in_polygon, polygon_vertices=polygon_vertices)
72          results = list(executor.map(func_with_polygon, points))
73
74      return results
```

```python
# Ana Program
if __name__ == "__main__":
    print(f"Kullanılan CPU çekirdeği sayısı: {multiprocessing.cpu_count()}")
    print(f"Toplam kontrol edilecek nokta sayısı: {len(test_points)}")

    print("\n--- Paralel Kontrol Başlıyor ---")
    start_time_parallel = time.perf_counter()

    # Paralel fonksiyonu çağır
    parallel_results = check_points_parallel(test_points, u_shape_polygon)
    end_time_parallel = time.perf_counter()
    print(f"Paralel kontrol süresi: {end_time_parallel - start_time_parallel:.4f} saniye")

    # Sonuçları gösterme
    print("\nParalel Sonuçlardan Örnekler:")
    for i in range(min(5, len(test_points))):
        print(f"Nokta {test_points[i]}: {'İçinde' if parallel_results[i] else 'Dışında'}")
    if len(test_points) > 10:
        print("...")
        for i in range(max(0, len(test_points) - 5), len(test_points)):
            print(f"Nokta {test_points[i]}: {'İçinde' if parallel_results[i] else 'Dışında'}")

    # Tek iş parçacıklı (seri) performansı karşılaştırmak için:
    print("\n--- Seri Kontrol Başlıyor (Karşılaştırma İçin) ---")
    start_time_serial = time.perf_counter()
    serial_results = [is_point_in_polygon(p, u_shape_polygon) for p in test_points]
    end_time_serial = time.perf_counter()
    print(f"Seri kontrol süresi: {end_time_serial - start_time_serial:.4f} saniye")

    # Sonuçların doğruluğunu kontrol et (basit bir doğrulama)
    print(f"\nSonuçlar aynı mı? {parallel_results == serial_results}")
```

## 3.3 Elde Ettiğim Çıktı ve Analizi

Kodu çalıştırdığımda aşağıdaki gibi bir çıktı aldım:

```
Kullanılan CPU çekirdeği sayısı: 12
Toplam kontrol edilecek nokta sayısı: 100000

--- Paralel Kontrol Başlıyor ---
Paralel kontrol süresi: 46.8668 saniye

Paralel Sonuçlardan Örnekler:
Nokta Point(0.14761197572329898, -3.2496481031188007): Dışında
Nokta Point(-4.86980395485938, 5.799335855306907): Dışında
Nokta Point(0.990093943256463, 13.520494457639575): Dışında
Nokta Point(-4.351591119742211, 1.5045257468992679): Dışında
Nokta Point(12.243071812055447, 9.397483247720665): Dışında
...
Nokta Point(10.72665092901676, 8.582086016040584): Dışında
Nokta Point(-1.0600642612244116, 5.462355169940473): Dışında
Nokta Point(8.880430053123728, 10.328954026786459): Dışında
Nokta Point(7.940225110724539, 6.078683691598716): İçinde
Nokta Point(-1.5411800431049705, 8.36629939756021): Dışında

--- Seri Kontrol Başlıyor (Karşılaştırma İçin) ---
Seri kontrol süresi: 0.2708 saniye

Sonuçlar aynı mı? True
```

Çıktıyı incelediğimde, beklediğim gibi tüm işlemlerin doğru sonuçlandığını ve paralel ile seri sonuçların aynı olduğunu (True) gördüm. Bu, algoritmamın ve paralel uygulamanın matematiksel olarak doğru çalıştığını gösteriyor.

Ancak dikkatimi çeken bir şey oldu: Paralel kontrol süresi (46.8668 saniye), seri kontrol süresinden (0.2708 saniye) oldukça uzun.

Bu durumun nedenlerini araştırdım ve şöyle açıkladım:

1. **"Overhead" Maliyeti:** Her bir noktanın poligon içinde olup olmadığını kontrol etme işlemi, tek başına oldukça hızlı ve hafif bir işlemdir. Paralel programlamada, süreçleri başlatma, onlara verileri gönderme (pickle etme), işi yapıp bitirmelerini bekleme ve sonuçları ana programa geri getirme gibi ek bir "yönetim maliyeti" (overhead) vardır. Benim durumumda, bu yönetim maliyeti, her bir noktanın kendi içindeki hesaplama süresinden çok daha ağır bastı.
2. **Basit İşlem:** Eğer *is_point_in_polygon* fonksiyonu çok daha karmaşık ve uzun süren bir hesaplama olsaydı (örneğin, poligon binlerce kenara sahip olsaydı veya her nokta için ek karmaşık hesaplamalar yapılsaydı), paralel programlamanın faydaları kesinlikle ortaya çıkacaktı. 100.000 nokta benim için çok gibi görünse de her bir noktanın işlemi çok kısa sürdüğü için, 12 çekirdek arasında dağıtılan bu minik işlerin toplam yönetim yükü, tek çekirdekten gelen doğrudan hesaplamayı geçemedi.

# 4 SONUÇ VE GELECEK İYİLEŞTİRMELERİ

Bu projede, bir noktanın poligon içinde olup olmadığını tespit etmek için Ray Casting algoritmasını başarıyla uyguladım ve bu işlemi Python'ın multiprocessing modülü ile paralel hale getirdim. Karşılaştığım "pickle" hatasını functools.partial kullanarak aştım ve paralel programlamada karşılaşılabilecek yaygın bir sorunu nasıl çözebileceğimi öğrendim.

Performans beklentimin aksine seri versiyonun daha hızlı çıkması, beni şaşırtmış olsa da bu durum paralel programlamanın faydalarının iş yükünün niteliğine bağlı olduğunu gösteren değerli bir ders oldu. Bu tür durumlarda, daha az sayıda süreci daha büyük veri parçacıklarıyla beslemek (chunking) veya farklı paralel programlama paradigmalarını (örneğin, GPU tabanlı çözümler) düşünmek gerekebilir.

Gelecekte bu projeyi geliştirirken şu iyileştirmeleri yapabilirim:

1. **Daha Karmaşık Poligonlar:** Binlerce veya on binlerce kenarı olan poligonlar üzerinde Ray Casting performansını test etmek.
2. **Optimizasyonlar:** *is_point_in_polygon* fonksiyonu içindeki özel durumları (ışının kenara paralel olması, köşeden geçmesi vb.) daha sağlam bir şekilde ele almak.
3. **Farklı Paralelleştirme Stratejileri:** Görevleri süreçlere dağıtırken daha gelişmiş stratejiler (örneğin, veri yığınlarını daha akıllıca bölmek) denemek.
4. **Görselleştirme:** Noktaların ve poligonun interaktif bir görselleştirmesini ekleyerek algoritmanın ve paralel işlemin nasıl çalıştığını daha net göstermek.

# 5 LİNKLER

Sunum videosu linki:
https://drive.google.com/file/d/1AuMaj8Lswk8SHYkhePCVoi8Tr2wnh4TN/view?usp=sharing

Kodlarımın linki: https://drive.google.com/file/d/1BOxupvZYtbbOpwTG59le-fBchVE8HXa6/view?usp=sharing