Paralel Programlama Dersi

Dönem Projesi

BLM0461

Paralel Programlama ile Noktanın Poligon İçinde Olup Olmadığı Kontrolü

Proje Video Linki:

https://www.youtube.com/watch?v=gGL9bR2IA-Y

Github Repo:

https://github.com/emirhansesigur/Parallel-Programming-Project

Emirhan Ahmet Sesigür

20360859057

1. Problem Tanımı ve Amacı

Bu projede, konkav ya da konveks yapıda olabilen bir poligonun x ve y koordinat ikililerileriyle tanımlandığı bir ortamda, kullanıcı tarafından verilen bir noktanın bu poligonun içerisinde mi yoksa dışında mı olduğunu belirlemek amaçlanmaktadır.

Bu temel geometrik problemin çözümünde, hem seri (tek iş parçacıklı) hem de paralel (çok iş parçacıklı) algoritmalar kullanılmıştır. Uygulama sonunda, her iki yaklaşımın hem doğruluk hem de performans (çalışma süresi) açısından karşılaştırması yapılacak ve paralel programlamanın sağladığı hız avantajı analiz edilecektir.

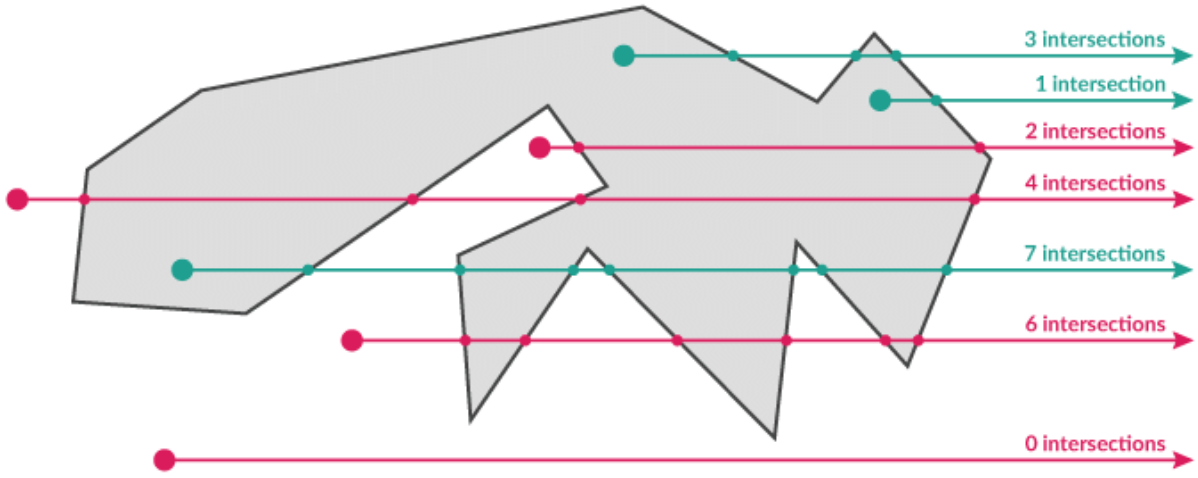
2. Kullanılan Algoritma: Ray Casting Yöntemi

Bu projede bir noktanın poligon içinde olup olmadığını belirlemek için en yaygın kullanılan yöntemlerden biri olan Ray Casting (Işın Yöntemi) kullanılmıştır. Bu yöntemin temel prensibi, test edilen noktadan sağa doğru yatay bir ışın gönderilmesi ve bu ışının poligonun kenarlarıyla kaç kez kesiştiğinin sayılmasıdır.

Yöntemin çalışma prensibi şu şekildedir:

- Noktadan x ekseni boyunca sonsuza giden bir ışın çizilir.
- Bu ışının poligonun kenarlarıyla kaç kere kesiştiği hesaplanır.
- Eğer kesişim sayısı tek sayı ise nokta poligonun içinde, çift sayı ise dışında kabul edilir.

Bu algoritma hem konkav hem de konveks poligonlar için geçerlidir ve sadece poligonun kenar bilgileriyle çalışır. Noktanın tam olarak bir kenar üzerinde bulunması durumu bu çalışmada kapsam dışı bırakılmıştır.



Şekil 1 – Noktalardan Çıkan Işınların Kenarları Kaç Kez Kestiğinin Gösterilmesi

Kod içerisinde bu algoritma **isInsideSerial** adlı fonksiyonla uygulanmıştır. Bu fonksiyonun işleyişi şu şekildedir:

```
public static boolean isInsideSerial(List<double[]> edges, double xp, double yp) {  1 usage
```

edges: Poligonun kenarlarını temsil eden [x1, y1, x2, y2] formatındaki kenar listesi.

xp, yp: İçeride olup olmadığı test edilen noktanın x ve y koordinatları.

```
int cnt = 0;
```

Işının kaç kenarla kesiştiği sayılacaktır.

```
for (double[] edge : edges) {
    double x1 = edge[0], y1 = edge[1], x2 = edge[2], y2 = edge[3];
```

Kenar, iki nokta (x1,y1)-(x2,y2) olarak alınır.

```
if ((yp < y1) != (yp < y2)
```

Bu koşul, ışının y-kesitiyle kenarın y-ekseninde kesişip kesişmediğini kontrol eder. Biri yukarıda biri aşağıda olmalıdır.

```
xp < x1 + ((yp - y1) / (y2 - y1)) * (x2 - x1))
```

Bu formül, ışının gerçekten kenarı geçtiği noktada solunda mı olduğunu kontrol eder.

```
cnt++;
```

Işın kenarı kesmişse sayacı artırır

```
return cnt % 2 == 1;
```

Eğer kesişim sayısı tekse true döner (nokta içerdedir), değilse false döner (nokta dışındadır).

3. Paralel Programlama Yaklaşımı

Ray Casting algoritmasının temel avantajlarından biri, her kenar ile yapılan kesişim kontrolünün diğer kenarlarla bağımsız olmasıdır. Bu özellik, algoritmanın paralel hale getirilmesini son derece uygun kılar. Bu projede Java programlama dilinin java.util.concurrent kütüphanesinden faydalanılarak çok iş parçacıklı bir yapı oluşturulmuştur.

Paralel uygulamada yapılan başlıca adımlar şunlardır:

Poligonun kenarları birer double[] dizisi ile temsil edilmiştir. Her kenar (edge), bir Callable<Integer> nesnesi olarak tanımlanmış ve ayrı bir iş olarak yorumlanmıştır.

Her Callable, kendi kenarının test noktasını kesip kesmediğini kontrol eder ve sonucu 0 veya 1 olarak döner.

Tüm bu Callable nesneleri bir ExecutorService havuzu içinde çalıştırılmış ve invokeAll() metodu ile eş zamanlı başlatılmıştır.

Son olarak, elde edilen tüm sonuçlar Future<Integer> nesneleri üzerinden toplanarak toplam kesişim sayısı hesaplanmıştır.

Bu yaklaşım, işlemci çekirdeklerinin sayısı kadar görevin eş zamanlı yürütülmesine olanak tanımış, böylece büyük sayıda kenardan oluşan poligonlarda ciddi performans artışı sağlanmıştır. Bu yapı sayesinde problem paralel hesaplama ile yapılabilmektedir.

4. Seri ve Paralel Yaklaşımların Karşılaştırılması

Bu projede, verilen bir noktanın belirli bir poligonun içinde yer alıp almadığını tespit etmek amacıyla Ray Casting (Işın Yöntemi) temelli bir algoritma uygulanmıştır. Bu algoritma hem seri (tek iş parçacıklı) hem de paralel (çok iş parçacıklı) olarak gerçekleştirilmiş ve bu iki yaklaşımın performansı karşılaştırılmıştır.

Seri Yöntem

Seri yöntemde poligonun tüm kenarları sırayla kontrol edilmiştir. Her kenar için, verilen noktanın o kenar ile sanal bir ışın kesiştirip kesiştirmediği hesaplanmış ve toplam kesişim sayısı üzerinden içeride/ dışarıda kararı verilmiştir.

- Tüm işlem tek bir thread üzerinde yürütüldüğü için herhangi bir iş parçacığı yönetim yükü yoktur.
- Uygulama küçük boyutlu poligonlarda oldukça hızlı çalışır.
- Basit ve okunabilir bir yapıya sahiptir.

Paralel Yöntem

Paralel yöntemde, her kenarın kontrolü ayrı bir iş olarak tanımlanmış ve çoklu iş parçacığı üzerinden eş zamanlı olarak yürütülmüştür. Bu amaçla Java'nın ExecutorService sınıfı kullanılarak Callable görevler bir thread pool'a atanmıştır.

- İşlem süresi potansiyel olarak azaltılabilir, özellikle kenar sayısının fazla olduğu büyük poligonlarda avantaj sağlar.
- Ancak küçük boyutlu örneklerde, thread havuzu oluşturma ve görev yönetimi gibi ek işlemler nedeniyle daha yavaş çalışabilir.
- Daha karmaşık ama ölçeklenebilir bir çözümdür.

5. Thread Pool Kullanımı

Bu projede paralel programlama yaklaşımında Java'nın çok iş parçacıklı (multithreading) programlama desteğinden yararlanılmış ve Thread sınıfı yerine modern ve daha kontrollü bir yapı olan ExecutorService kullanılmıştır. Bu tercih, sistem kaynaklarının daha verimli kullanılmasını ve görevlerin daha organize şekilde yürütülmesini sağlamaktadır.

ExecutorService Nedir?

ExecutorService, Java'nın java.util.concurrent paketinde yer alan ve çok sayıda görevin paralel olarak yürütülmesini sağlayan bir arabirimdir. Temel özellikleri:

Belirli sayıda iş parçacığına sahip bir thread havuzu (thread pool) oluşturur.

Yeni görevler için her seferinde yeni thread oluşturmak yerine, var olan thread'leri tekrar kullanarak performans maliyetini düşürür.

invokeAll() Metodu Nedir?

Paralel çözümde görevlerin çalıştırılması için invokeAll() metodu kullanılmıştır:

List<Future<Integer>> results = executor.invokeAll(tasks);

Bu metodun temel özellikleri:

- Parametre olarak birden fazla **Callable** görevi (task) alır.
- Bu görevleri paralel olarak yürütür ve her bir görevin sonucunu Future nesneleri olarak döner.

- Tüm görevlerin tamamlanmasını **engellemeden** bekler ve tamamlananları sırasıyla döndürür.
- İşlemler tamamlanmadan sonraki adım geçilmez, yani **senkron (blocking)** çalışır.

Bu yöntem sayesinde tüm kenarlar için yazılmış bağımsız kontrol görevleri paralel olarak başlatılmış ve her birinden dönüş değerleri toplanarak toplam kesişim sayısı elde edilmiştir.