

EGE ÜNİVERSİTESİ FEN BİLİMLERİ ENSTİTÜSÜ
(YÜKSEK LİSANS TEZİ)

**KÜMELEME ALGORİTMALARI VE
PARALELLEŞTİRİLMELERİ**

Can ATILGAN

Tez Danışmanı: Doç. Dr. Burak ORDİN

Matematik Anabilim Dalı

Bilim Dalı Kodu: 619.03.03
Sunuş Tarihi: 22.08.2014

Bornova-İZMİR

2014

Can ATILGAN tarafından yüksek lisans tezi olarak sunulan “Kümeleme Algoritmaları ve Paralelleştirilmeleri” başlıklı bu çalışma, E.Ü. Lisansüstü Eğitim ve Öğretim Yönetmeliği ile E.Ü. Fen Bilimleri Enstitüsü Eğitim ve Öğretim Yönergesi’nin ilgili hükümleri uyarınca tarafımızdan değerlendirilerek savunmaya değer bulunmuş ve 22.08.2014 tarihinde yapılan tez savunma sınavında aday oybirliği/~~oyçokluğu~~ ile başarılı bulunmuştur.

Jüri Üyeleri:

İmza

Jüri Başkanı : Doç. Dr. Burak ORDİN

Raportör Üye : Prof. Dr. Urfat NURİYEV

Üye : Prof. Dr. Efendi NASİBOĞLU

.....
.....
.....

EGE ÜNİVERSİTESİ FEN BİLİMLERİ ENSTİTÜSÜ

ETİK KURALLARA UYGUNLUK BEYANI

E.Ü. Lisansüstü Eğitim ve Öğretim Yönetmeliğinin ilgili hükümleri uyarınca Yüksek Lisans Tezi olarak sunduğum “Kümeleme Algoritmaları ve Paralleştirilmeleri” başlıklı bu tezin kendi çalışmam olduğunu, sunduğum tüm sonuç, doküman, bilgi ve belgeleri bizzat ve bu tez çalışması kapsamında elde ettiğimi, bu tez çalışmasıyla elde edilmeyen bütün bilgi ve yorumlara atıf yaptığımı ve bunları kaynaklar listesinde usulüne uygun olarak verdiğimi, tez çalışması ve yazımı sırasında patent ve telif haklarını ihlal edici bir davranışımın olmadığını, bu tezin herhangi bir bölümünü bu üniversite veya diğer bir üniversitede başka bir tez çalışması içinde sunmadığımı, bu tezin planlanmasından yazımına kadar bütün safhalarda bilimsel etik kurallarına uygun olarak davrandığımı ve aksinin ortaya çıkması durumunda her türlü yasal sonucu kabul edeceğimi beyan ederim.

22.08.2014

Can ATILGAN

ÖZET**KÜMELEME ALGORİTMALARI VE
PARALELLEŞTİRİLMELERİ**

ATILGAN, Can

Yüksek Lisans Tezi, Matematik Anabilim Dalı

Tez Danışmanı: Doç. Dr. Burak ORDİN

Ağustos 2014, 58 sayfa

Kümeleme (veya Küme Analizi), verilerin belirli bir benzerlik ölçütüne göre kümelere ayrılmasını amaçlayan bir veri analizi tekniğidir. Diğer bir deyişle, birbirine benzer verilerin bir arada yer aldığı homojen grupları belirleme işidir.

Kümelemede ele alınan veri setleri genellikle çok büyüktür. Bu yüzden, geliştirilen yöntemlerin hızı, kümelemedeki başarısı kadar –kimi zaman daha fazla– öneme sahiptir. Her iki yönden de başarılı yöntemler var olsa da, işin doğası nedeniyle potansiyel işlem yükü hala ağırdır. Temel kümeleme algoritmaları seri bir makine modeliyle geliştirilmiştir ve modern bir bilgisayardan kapasitesinin altında verim sağlayabilirler. Bu handikapı aşmanın yolu, yöntemleri paralel hesaplamaya uygun hale getirmektir.

Bu tezde, kümeleme yöntemleri ve paralel hesaplama araçları çeşitli açılardan incelenmiş, kümeleme başarısı yüksek iki algoritmanın paralel versiyonları önerilmiştir. Bu algoritmalar, C programlama dili ve OpenMP uygulama programlama arayüzü kullanılarak programlanmış ve alt programları barındıran bir kütüphane oluşturulmuştur. Önerilen yöntemler, teorik ve deneysel olarak analiz edilerek paralelleştirmenin etkisi gösterilmiştir.

Anahtar kelimeler: Kümeleme, paralel hesaplama, global k-ortalamlar, bulanık bağlantılı noktalar.

ABSTRACT**CLUSTERING ALGORITHMS AND THEIR PARALLELIZATION**

ATILGAN, Can

Supervisor: Assoc. Prof. Dr. Burak ORDİN

August 2014, 58 pages

Clustering (or Cluster Analysis) is a data analysis technique that intends to cluster data according to some similarity measure. In other words, it is a task of identifying homogeneous groups consisting of analogous data.

The data sets handled in clustering are usually very large. Thus, the speed of the developed methods is as important as –sometimes more important than– their clustering efficiency. Even though there are methods which are successful in both aspects, the potential work load remains heavy by the nature of the task. Fundamental clustering algorithms are developed using a serial machine model and can achieve a limited efficiency on a modern computer. The way to overcome this handicap is to tailor these methods for parallel computing.

In this thesis, clustering methods and parallel computing tools are studied from various aspects, and parallel versions of two clustering algorithms with high clustering efficiency are proposed. These algorithms are then programmed using C programming language and OpenMP application programming interface, and a library containing the subprograms is organized. The effect of the parallelization is presented by theoretically and experimentally analyzing the proposed methods.

Key words: Clustering, parallel computing, global k-means, fuzzy joint points.

TEŞEKKÜR

Bana bu yüksek lisans tezi konusunda çalışma olanağı tanıyan ve tez süresince yardımlarını esirgemeyen tez danışmanım Doç. Dr. Burak Ordin'e çok teşekkür ederim. Tez çalışmalarına bilgi ve yönlendirmeleriyle destek olan Prof. Dr. Urfat Nuriyev, Prof. Dr. Efendi Nasiboğlu ve Yrd. Doç. Dr. Murat Erşen Berberler'e de teşekkürü bir borç bilirim.

Tüm eğitim hayatım boyunca desteklerini hissettiğim aileme ve dostlarıma ayrıca teşekkür ederim.

İÇİNDEKİLER

Sayfa

ÖZET	vii
ABSTRACT.....	ix
TEŞEKÜR.....	xi
ŞEKİLLER DİZİNİ	xvi
ÇİZELGELER DİZİNİ	xviii
1. GİRİŞ	1
2. KÜMELEME.....	2
2.1. Uygulama Alanları	2
2.2. Kümeleme Problemlerinin Modelleri	3
2.2.1. Kesin kümeleme	3
2.2.2. Bulanık kümeleme.....	4
2.3. Kümeleme Yöntemleri	5
2.3.1. Hiyerarşik kümeleme	6
2.3.2. Bölümleme tabanlı kümeleme	8
2.3.3. Yoğunluk tabanlı kümeleme	12
3. PARALEL HESAPLAMA	16
3.1. Paralel Bilgisayar Mimarileri	16

İÇİNDEKİLER (devam)

Sayfa

3.1.1. Flynn sınıflandırması	17
3.1.2. Paylaşımlı bellek mimarisi	18
3.1.3. Dağıtık bellek mimarisi	19
3.1.4. Karma mimari.....	20
3.2. Paralel Programlama Modelleri	20
3.2.1. Bellek paylaşım modeli	20
3.2.2. İş parçacığı (thread) modeli	21
3.2.3. Mesaj geçme modeli	22
3.2.4. Karma model	23
3.3. Paralel Algoritma Tasarımı ve Performans Analizi.....	24
3.3.1. Soyut hesaplama modelleri.....	24
3.3.2. Teorik hızlanma limitleri	26
4. PARALEL KÜMELEME.....	28
4.1. Paralel Global k -ortalamalar Algoritması.....	28
4.2. Paralel α Seviye Tarama Algoritması	31
4.3. Geliştirilen Programlar	39
4.4. Hesaplama Denemeleri.....	41

İÇİNDEKİLER (devam)

	<u>Sayfa</u>
4.4.1. P-GKM deneme sonuçları.....	41
4.4.2. P- α Scan deneme sonuçları	46
5. SONUÇ	53
KAYNAKLAR DİZİNİ	54
ÖZGEÇMİŞ	58
EKLER	

ŞEKİLLER DİZİNİ

<u>Şekil</u>	<u>Sayfa</u>
2.1. A kümesinin üyelik fonksiyonunun grafiği	4
2.2. B kümesinin üyelik fonksiyonunun grafiği	5
2.4. k -ortalamalar algoritmasının örneği	10
2.5. DBSCAN yöntemine ait temel kavramlar	13
2.6. Çeşitli veri setleri üzerinde bölümlleme tabanlı ve yoğunluk tabanlı yöntemlerin sonuçlarının karşılaştırılması	14
3.1. Flynn Sınıflandırmasına göre paralel bilgisayar mimarileri	18
3.2. Paylaşımlı bellek mimarisi	19
3.3. Dağıtık bellek mimarisi	19
3.4. Karma mimari	20
3.5. İş parçacığı modeli	22
3.6. Mesaj geçme modeli	23
3.7. İş derinliği modelinde yönlü çizge gösterimi	25
4.1. Konik bulanık nokta $P = (p, r)$, $x, p \in \mathbb{R}^2$ ve α -seviye kümesi	32
4.2. FJP yöntemine ait bulanık kavramlar	34
4.3. IRIS veri seti üzerinde P-GKM ile elde edilen hızlanma	44
4.4. YEAST veri seti üzerinde P-GKM ile elde edilen hızlanma	45

ŞEKİLLER DİZİNİ (devam)

<u>Şekil</u>	<u>Sayfa</u>
4.5. PAGEBLOCK veri seti üzerinde P-GKM ile elde edilen hızlanma	45
4.6. PENDIGIT veri seti üzerinde P-GKM ile elde edilen hızlanma	46
4.7. α Scan ile kümeleme sonuçları	47
4.8. YAPAY1 veri seti üzerinde P- α Scan ile elde edilen hızlanma	49
4.9. YAPAY2 veri seti üzerinde P- α Scan ile elde edilen hızlanma	50
4.10. YAPAY3 veri seti üzerinde P- α Scan ile elde edilen hızlanma	50
4.11. YAPAY4 veri seti üzerinde P- α Scan ile elde edilen hızlanma	51
4.12. PENDIGIT veri seti üzerinde P- α Scan ile elde edilen hızlanma	51

ÇİZELGELER DİZİNİ

<u>Çizelge</u>	<u>Sayfa</u>
3.1. Flynn Sınıflandırması	17
4.1. Veri setlerine ait bilgiler	41
4.3. YEAST veri seti üzerinde P-GKM denemeleri	42
4.4. PAGEBLOCK veri seti üzerinde P-GKM denemeleri	43
4.5. PENDIGIT veri seti üzerinde P-GKM denemeleri	43
4.6. YAPAY1 veri seti üzerinde P- α Scan denemeleri	47
4.7. YAPAY2 veri seti üzerinde P- α Scan denemeleri	48
4.8. YAPAY3 veri seti üzerinde P- α Scan denemeleri	48
4.9. YAPAY4 veri seti üzerinde P- α Scan denemeleri	48
4.10. PENDIGIT veri seti üzerinde P- α Scan denemeleri	49

1. GİRİŞ

Her an ve her yerde, dolaylı veya doğrudan, verilerle etkileşim içinde olduğumuz bir dünyadayız. Canlıların karakteristik özelliklerini tanımlamak, doğa olaylarını tasvir etmek, bilimsel bir deneyin çıktılarını göstermek, bir üretim sistemine ait kayıtları tutmak vb. çok çeşitli amaçlar için verileri topluyor ve kullanıyoruz. Bunlardan daha önemlisi ise, verileri analiz ederek çıkarımlar yapabiliyor, bunlara dayanarak karar verebiliyor veya karar veren sistemler yaratabiliyoruz. Veri analizi yaklaşımlarından belki de en önemlisi, insanlığın en eski faaliyetlerinden biri olan veriyi belirli gruplara bölmek, yani sınıflandırmaktır (Xu and Wunsch, 2009; Anderberg'den, 1973 ve Everitt'den, 2001). Tyron ve Bailey (1970) bu konuda şöyle demektedir:

“Dünyamızı anlamak onu oluşturan varlıklar arasındaki benzerlikleri ve farklılıkları kavramsallaştırmayı gerektirir.”

Sınıflandırma gözetimli ya da gözetimsiz olabilir. Kısaca, eğer verilerin atanacağı sınıfların özelliği önceden belirli ise gözetimli, değilse gözetimsizdir denilebilir. Kümeleme (Clustering) veya Küme Analizi (Cluster Analysis), en genel gözetimsiz sınıflandırma yaklaşımlarından biridir.

Günümüze kadar biriken ve özellikle son yıllarda sayıları katlanarak artan veriler, veri analizini gittikçe zorlaştırmaktadır. Veri setlerinin büyüklüğü ve kümelemenin doğası gereği, bir kümeleme uygulamasının işlem yükü çok ağırdır. Buna karşın, hesaplamalı bilimlerin genelinde olduğu gibi, temel kümeleme algoritmaları seri (tek işlemcili) bir bilgisayarda çalışmak üzere tasarlanmıştır. Oysa, dizüstü bilgisayarlardan süper bilgisayarlara kadar, neredeyse tüm modern hesaplama sistemleri paraleldir. Yani, eş zamanlı işlem yapabilen işlemcilerden oluşmaktadırlar. Kümeleme uygulamalarının böylesi bilgisayarlardan en yüksek verim alabilmesi için, kümeleme algoritmalarının paralelleştirilmeleri gerekir.

Bu tez, bazı kümeleme başarısı yüksek algoritmaların paralelleştirilmeleri üzerinedir. Önce, kümeleme ve paralel hesaplama konuları sırasıyla, Bölüm 2 ve Bölüm 3'te incelenmiştir. Bölüm 4'te iki farklı kümeleme algoritmasının paralel versiyonları önerilmiştir. Önerilen paralel algoritmaların teorik performans incelemeleri yapılmış, algoritmaların uygulanması için geliştirilen kütüphane ve programların detayları verilmiş, ve hesaplama denemeleri sunulmuştur. Sonuç bölümü olan Bölüm 5'te tezin çıktıları değerlendirilmiştir.

2. KÜMELEME

Kümelemenin amacı, verilerin küme içi diğer verilere benzerliği en büyük ve küme dışı verilere benzerliği en küçük olacak biçimde kümeler belirlemektir. Kümelemenin karakteristiğini belirleyen benzerlik ölçütü, uygulamaya ve çözüm yaklaşımına göre değişkenlik gösterir. Verilerin benzerliği, basit bir uzaklık metriği ile ölçülebileceği gibi, yoğunluk, komşuluk, bağlantılılık vb. özel kavramlar da kullanılabilir.

Bu bölümün geri kalanında, önce kümelemenin uygulama alanlarından bahsedilmektedir. Daha sonra, kesin ve bulanık kümeleme problemlerinin modelleri sunulmuştur. Son olarak, kümeleme yöntemlerinin çeşitli kategoriler altında bir incelemesi verilmiştir.

2.1. Uygulama Alanları

Başta veri madenciliği olmak üzere çok boyutlu istatistiksel tahminleme, vektör niceleme, makine öğrenmesi vb. araştırma alanlarının konusu olan kümelemenin tıp, biyo-enformatik, ekonomi, mühendislik, astronomi ve yerbilim, sosyal bilimler gibi alanlarda önemli uygulamaları vardır (Xu and Wunsch, 2009; Berkhin, 2002; Savaş vd., 2012):

- Kümeleme uygulamalarının büyük bir kısmı tıp ve biyo-enformatik alanlarında yer almaktadır. Özellikle gen ifadesi, genotip belirleme ve homolog gen dizilerinin gruplanması gibi çalışmalarda, kümeleme çok önemli bir araçtır. Yine tıp alanında hastalık teşhisi, ilaç sınıflama, radyolojik görüntüleme vb. geniş bir uygulama yelpazesi vardır.
- Pazar araştırması yapılırken potansiyel müşterilerin belirlenmesi ve tüketim alışkanlıklarının ortaya çıkarılmasında, marka ve ürün gruplandırma kümelemeden yararlanılmaktadır. Ayrıca bankacılık sektöründe, bazı diğer veri madenciliği yöntemleriyle beraber, kredi verme, yatırım yönlendirme gibi işlerde karar verme aşamasında kümeleme yöntemleri kullanılmaktadır.
- Mühendislikte, genel olarak görüntü ve ses tanıma, sinyal analizi, veri sıkıştırma ve gürültü temizleme için kümeleme yöntemleri kullanılmaktadır. Yani, sensörlerin kullanıldığı bir mühendislik uygulamasında –ki bu hiç de nadir bir durum değildir– kümelemenin önemli bir rolü olması muhtemeldir.

- Gök cisimlerinin sınıflandırılması, yeryüzü şekillerinin tanımlanması, akarsu sistemlerinin analizi, uzay simülasyonlarında ölçeklendirme gibi konularda kümelemeden faydalanılmaktadır.
- Sosyal bilimlerde davranış analizi, kültür etkileşimlerinin ortaya çıkarılması, dillerin ortak özellikleri gibi ilginç kümeleme uygulamalarına rastlanmaktadır. Arkeolojik çalışmalarda ve yapıtların gruplandırılmasında, yine kümeleme kullanılmaktadır.

Tüm bu uygulamaların yanı sıra, başta fizik ve kimya olmak üzere temel bilimlerde de kümeleme uygulamaları mevcuttur (Akpolat vd., 2013).

2.2. Kümeleme Problemlerinin Modelleri

Kümeleme problemi denildiğinde kastedilen genellikle kesin kümeleme (hard clustering) problemidir. Kesin kümelemede her bir veri noktası tek bir kümeye ait olmalıdır. Kesin kümeleme probleminin çözüm yaklaşımlarında çeşitli modellere ve farklı amaç fonksiyonlarına rastlanasa da, verinin tek bir kümeye ait olması gerekliliği değişmez. Yumuşak/bulanık kümelemede (soft/fuzzy clustering) ise, veriler belirli üyelik dereceleriyle birden çok kümeye ait olabilirler (Bezdek, 1984).

2.2.1. Kesin kümeleme

Kümelenmek istenen veri seti $x_i \in \mathbb{R}^m$, $i = 1, \dots, n$ olmak üzere $X = \{x_1, \dots, x_n\}$ olsun. Kesin kümeleme probleminde amaç, X kümesini $X^j, j = 1, \dots, k$ gibi k adet ayrık alt kümeye aşağıdaki kurallara uygun olarak ayırmaktır (Bezdek et al., 1981; Bezdek, 1984; Bagirov and Mardaneh, 2006):

$$X^j \neq \emptyset, \quad j = 1, \dots, k, \quad (2.1)$$

$$X^j \cap X^t = \emptyset, \quad j, t = 1, \dots, k, \quad j \neq t, \quad (2.2)$$

$$\bigcup_{j=1}^k X^j = X. \quad (2.3)$$

Belirlenen her bir X^j ayrık alt kümesi, aradığımız kümelerden (cluster) birine karşılık gelir.

2.2.2. Bulanık kümeleme

Bulanık küme kavramı, L. Zadeh (1965) tarafından yapılan bir yayınlı bilimsel yazına kazandırılmıştır. Klasik küme kavramında bir eleman bir kümeye ya aittir ya da değildir. Bulanık küme kavramında ise bir eleman bir kümeye belirli bir dereceden ait olabilir.

Klasik mantığa göre daha esnek bir modelleme aracı olan bulanık mantık birçok araştırma alanında kendine yer bulmuştur. Kümeleme bunlardan bir tanesidir. Şimdi, yalnızca bulanık kümelemenin kavranması için bulanık kümelerle kısa bir giriş verilmiştir:

Tanım 2.1. (Bulanık küme) X evrensel kümesinde $F(X)$ bulanık kümesi $F = \{(x, \mu_F(x)) \mid x \in X\}$ şeklinde tanımlanır. Burada $\mu_F(x): X \rightarrow [0,1]$ fonksiyonu, F kümesinin üyelik fonksiyonudur ve her bir x 'in F kümesine üyelik derecesini belirler.

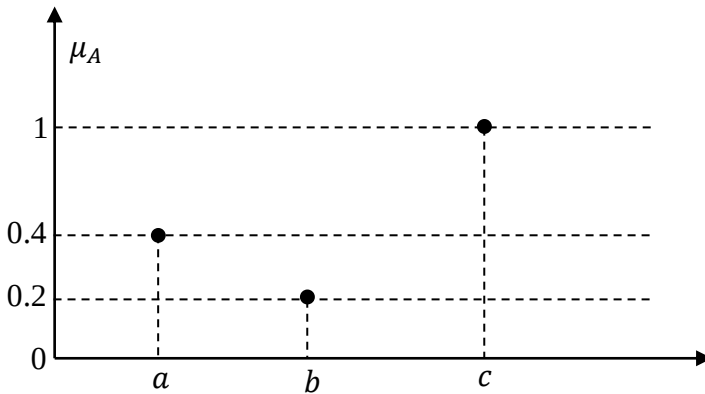
Bulanık kümelerle örnek olarak

$$A = \{(a, 0.4), (b, 0.2), (c, 1)\}$$

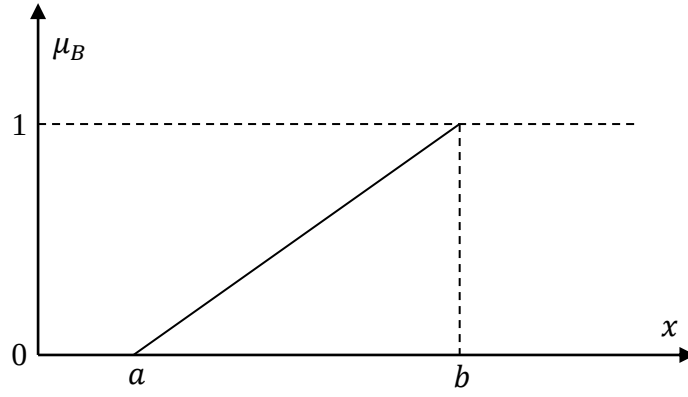
ve

$$B = \{(x, \mu_B(x)) \mid \mu_B(x) = \frac{x-a}{b-a}, x \in [a, b]\}$$

verilebilir.



Şekil 2.1. A kümesinin üyelik fonksiyonunun grafiği.



Şekil 2.2. B kümesinin üyelik fonksiyonunun grafiği.

Bir bulanık kümenin karakteristiğini tayin eden üyelik fonksiyonu, Tanım 2.1’de verildiği gibi, görüntü kümesi $[0,1]$ aralığında olan herhangi bir fonksiyon olabilir. Ancak bulanık uygulamalarda kullanılan genel birkaç parametrik fonksiyon türü vardır: Üçgensel (triangular) üyelik fonksiyonu, yamuksal (trapezoidal) üyelik fonksiyonu, Gauss üyelik fonksiyonu ve çan (bell) üyelik fonksiyonu (Jang et al., 1997).

Kümelenmek istenen veri seti $x_i \in \mathbb{R}^m$, $i = 1, \dots, n$ olmak üzere $X = \{x_1, \dots, x_n\}$ olsun. Bulanık kümeleme probleminde amaç, X kümesini $X^j, j = 1, \dots, k$ gibi k adet bulanık alt kümeye aşağıdaki kurallara uygun olarak ayırmaktır (Bezdek, 1981; Bezdek et. al., 1984):

$$\sum_{i=1}^n u_{ij} > 0, \quad j = 1, \dots, k, \quad (2.4)$$

$$\sum_{j=1}^k u_{ij} = 1, \quad i = 1, \dots, n. \quad (2.5)$$

Burada u_{ij} , x_i elemanının X^j kümesine üyelik derecesini gösterir. (2.4) koşulu, kesin kümeleme modelindeki (2.1)’e karşılık gelir. (2.5) ise kesin kümelemenin aksine, bir noktanın, toplamı 1 edecek şekilde çeşitli üyelik dereceleriyle farklı kümelerle ait olabileceğini ifade etmektedir.

2.3. Kümeleme Yöntemleri

Çözüm yöntemleri açısından kümeleme, hiyerarşik kümeleme, bölümlenme tabanlı (partitioning-based) kümeleme ve yoğunluk tabanlı (density-based) kümeleme şeklinde kategorize edilebilir. Farklı kategorizasyonlar da yapılabilir ve her bir kategori birçok alt kategoriye ayrılabilir (Bkz. Berkhin, 2002; Larose,

2005; Xu and Wunsch, 2009). Bu tezde incelenen yöntemler, bahsedilen üç kategori altında ele alınmıştır.

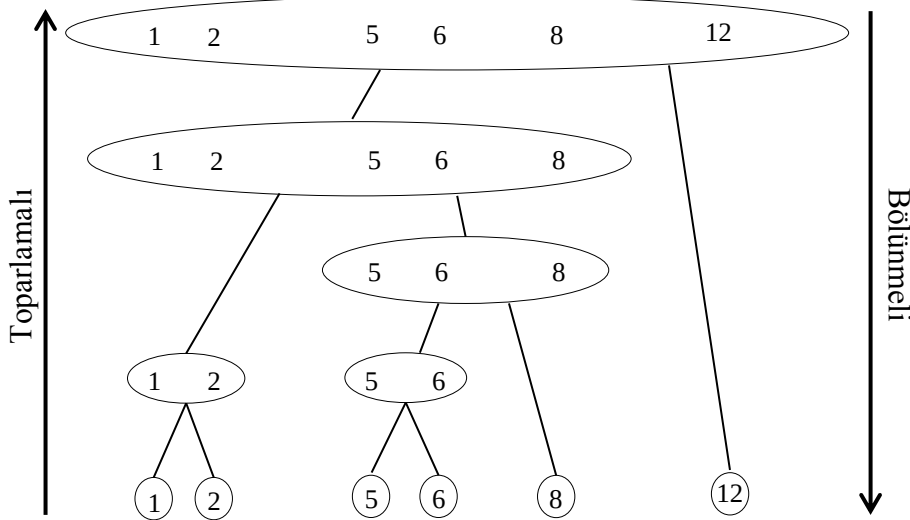
2.3.1. Hiyerarşik kümeleme

Hiyerarşik kümeleme, veri setini tek bir biçimde grupta yerine, bir dizi farklı kümeleme elde etme stratejisidir. Han ve Kamber (2006) hiyerarşik kümelemeyi şöyle açıklamıştır:

“Bir hiyerarşik kümeleme yöntemi, veri nesnelerinin bir kümeler ağacı içine gruplanması şeklinde çalışır.”

Hiyerarşik kümeleme yöntemleri tüm veri noktalarının birer küme olduğu durumdan tüm noktaların tek bir kümeye ait olduğu kümelemeye doğru ya da tam tersi yönde çalışır. İlk yönde ilerleyen yöntemler toparlamalı (agglomerative), tersine ilerleyen yöntemler ise bölünmeli (divisive) olarak adlandırılmaktadır.

Şekil 2.3 toparlamalı ve bölünmeli kümelemeyi aynı örnek üzerinde göstermektedir. $A = \{1, 2, 5, 6, 8, 12\}$ kümesinin elemanları arasındaki benzerlik ölçütü $a, b \in A$ için $d = |a - b|$ fonksiyonu ile belirlenmiştir. Örneklenen iki yöntem de “tekil bağlantı” yaklaşımını esas almaktadır. Tekil bağlantı yaklaşımında iki küme arasındaki uzaklık, kümelerin birbirine en yakın elemanları arasındaki uzaklığa eşittir. Toparlamalı yöntemde, her seviyede birbirine en yakın kümeler birleştirilir. Bölünmeli yöntemde ise, küme içindeki komşular içerisinde birbirine en uzak iki noktadan küme bölünür.



Şekil 2.3. $A = \{1, 2, 5, 6, 8, 12\}$ kümesinin tekil bağlantı yaklaşımı ile toparlamalı ve bölünmeli hiyerarşik kümeleme ile kümelenmesi.

Hiyerarşik kümeleme yöntemlerinde, kümeler arasındaki uzaklık ölçütüne göre 4 temel yaklaşım vardır. $\|x - y\|$ x ve y noktaları arasındaki uzaklık, m_i C_i kümesinin merkezi ve $|C_i|$ kümenin eleman sayısı olmak üzere, C_i ve C_j kümeleri arasında tanımlı 4 temel uzaklık ölçütü sırasıyla şunlardır (Han and Kamber, 2006):

$$d_{min}(C_i, C_j) = \min_{x \in C_i, y \in C_j} \|x - y\|, \quad (2.6)$$

$$d_{max}(C_i, C_j) = \max_{x \in C_i, y \in C_j} \|x - y\|, \quad (2.7)$$

$$d_m(C_i, C_j) = \|m_i - m_j\|, \quad (2.8)$$

$$d_{ort}(C_i, C_j) = \frac{1}{|C_i||C_j|} \sum_{x \in C_i} \sum_{y \in C_j} \|x - y\|. \quad (2.9)$$

Bu ölçütlerden (2.6) fonksiyonu tekil bağlantı (single-linkage) yaklaşımına, (2.7) fonksiyonu tam bağlantı (complete-linkage) yaklaşımına, (2.8) fonksiyonu merkez bağlantı (centroid-linkage) yaklaşımına ve (2.9) fonksiyonu ortalama bağlantı (average-linkage) yaklaşımına aittir.

Toparlamalı kümeleme için kümeler arasında bağlantısal bir metrik kullanmak yerine, bölümlene tabanlı kümelemede (Bkz. Bölüm 2.3.2) olduğu gibi, bir amaç fonksiyonunun değerindeki değişime göre kümeler birleştirilebilir. Bu yaklaşım Ward (1963) tarafından önerilmiştir.

Klasik hiyerarşik kümeleme yöntemlerinin yanı sıra, bazı daha gelişmiş sayılabilecek yöntemler de hiyerarşik kümeleme kategorisi altında incelenebilir. Bunlardan birisi BIRCH (Balanced Iterative Reducing and Clustering Using Hierarchies) algoritmasıdır. BIRCH algoritmasında tüm veriler, sıradışı durumlar dışında, yalnızca bir kez değerlendirilir. Dolayısıyla, oldukça hızlı bir algoritmadır ve klasik yöntemlere göre daha büyük ölçekli veri setleri üzerinde kullanılabilir. Bir diğer avantajı ise, önceki kümeleme seviyeleri üzerinde ayarlamalar yapabilmesidir. Ayrıca, algoritmayı öneren Zhang et al. (1996), BIRCH'ın gürültü (herhangi bir kümeye ait olmaması gereken, normal dışı veriler) ile baş edebilen ilk kümeleme algoritması olduğunu ifade etmiştir.

ROCK (RObust Clustering using linKs) algoritmasında, veriler arasında yeni bir bağ (link) kavramı tanımlanmıştır. Bu bağ kavramı özetle, iki veri arasındaki ortak komşuların sayısıdır. Komşuluk ise yine bir benzerlik ölçütüne

dayalıdır ve belirli bir eşik değerinden fazla benzerliğe sahip veriler komşu olarak kabul edilir. Veriler arasındaki bağların kuvvetine göre bir toparlamalı hiyerarşik kümeleme yapılır (Guha et al., 1999).

İki aşamalı ve toparlamalı bir hiyerarşik kümeleme algoritması olan Chameleon (Karypis et al., 1999), kümeleme problemini çizge teorisi ile ilişkilendirir. Veri noktaları çizgenin tepeleri ile temsil edilirler. Eğer bir veri, diğerine en çok benzeyen k adet veriden biriyse, onları temsil eden tepeler arasına bir ayrıt eklenir. Böylece bir k -en yakın komşu çizgesi oluşturulur. İlk aşamada bir parçalama prosedürü kullanarak alt çizgeler elde edilir. Bu alt çizgeler ikinci aşamadaki toparlamalı kümelemenin ilk seviyesidir. İkinci aşamada, kümeler arasındaki benzerliğin ve çizge üzerindeki bağlantılılığın kullanıldığı bir değerlendirme ile yeni seviyeler belirlenir.

2.3.2. Bölümleme tabanlı kümeleme

En yaygın olarak incelenen ve kullanılan kümeleme yöntemleri bölümleme tabanlı kümeleme kategorisine girer. Bu yöntemlerde, kümeleme tanımıyla örtüşen bir amaç fonksiyonunu optimize edecek şekilde tek bir bölümleme elde edilmeye çalışılmaktadır. Yerel optimuma ulaşmak için bir başlangıç çözümünden iteratif olarak iyileştirme stratejisi izlenir. Bölümleme tabanlı kümeleme, en doğal kümeleme yaklaşımıdır denilebilir.

Hiyerarşik yöntemlerde olduğu gibi her veri çifti arasındaki benzerliğin ele alınması, karesel bir hesaplama karmaşıklığı getirecektir. Bu güçlüğü aşmak için her küme için bir temsilci belirlenir ve verilerin bu temsilciyle olan benzerliğine dayanan bir amaç fonksiyonu kullanılır. Bu durumda, küme sayısının veri sayısından çok daha küçük olacağı öngörüsüne dayanılarak karmaşıklığın doğrusal olduğu söylenebilir (Xu and Wunsch, 2009).

Benzerlik ölçütü olarak $x_1, x_2 \in \mathbb{R}^m$ için $d(x_1, x_2)$ gibi bir uzaklık fonksiyonu kullanılabilir. $d(x_1, x_2)$ fonksiyonunun seçimi uygulama bağımlıdır. Bu tez çalışması boyunca, temel bir metrik olması ve geniş kullanım alanı sebebiyle Öklid uzaklığı (2.10) kullanılmıştır.

$$d(x_1, x_2) = \sqrt{\sum_{i=1}^m |x_{1i} - x_{2i}|^2}. \quad (2.10)$$

$d(x_1, x_2)$ ile benzerlik arasında ters bir ilişki vardır. Yani, $d(x_1, x_2)$ değeri ne kadar küçükse, x_1 noktası x_2 noktasına o kadar çok benzer.

Her bir küme $c_j \in \mathbb{R}^m$, $j = 1, \dots, k$ gibi bir merkez noktasıyla temsil edilirse, kümeleme problemi, aşağıdaki gibi bir optimizasyon problemi ile ifade edilebilir (Bagirov and Mardaneh, 2006; Spath'den, 1980 ve Bock'dan, 1998):

$$\min. f(c, w) = \frac{1}{n} \sum_{i=1}^n \sum_{j=1}^k w_{ij} d(c_j, x_i)^2 \quad (2.11)$$

şunlara göre:

$$c = (c_1, \dots, c_k), \quad (2.12)$$

$$c_j = \frac{\sum_{i=1}^n w_{ij} x_i}{\sum_{i=1}^n w_{ij}}, j = 1, \dots, k, \quad (2.13)$$

$$w_{ij} \in \{0,1\}, i = 1, \dots, n, j = 1, \dots, k, \quad (2.14)$$

$$\sum_{j=1}^k w_{ij} = 1, i = 1, \dots, n, \quad (2.15)$$

ve

$$w_{ij} = \begin{cases} 1, & \text{eğer } x_i \text{ elemanı } j \text{ kümesine atandıysa} \\ 0, & \text{aksi halde} \end{cases}. \quad (2.16)$$

Bu model kesin kümeleme probleminin çözümü için kurulmuştur. Yani, modelin bir uygun çözümü (2.1)–(2.3) koşullarını sağlar.

(2.11)–(2.16) ile verilen model karma tam sayılı doğrusal olmayan programlama modelidir ve karesel hatalar toplamı (sum of squared errors) modeli olarak da bilinmektedir. Tabii ki bölümlere tabanlı kümeleme, farklı koşul ve amaç fonksiyonları ile de modellenabilir (Bkz. Bagirov and Mardaneh, 2006; Xu and Wunsch, 2009).

Yalnızca bölümlere tabanlı kümeleme yöntemlerinin değil, bütün kümeleme yöntemlerinin içinde en popülerleri k -ortalamlar (k -means)

algoritmasıdır (MacQueen, 1967). Bölümleme tabanlı kümeleme yöntemlerinin çoğu k -ortalamalar algoritmasına dayanmaktadır. Kümelenmek istenen veri seti X ve küme sayısı k olmak üzere k -ortalamalar algoritmasının bir kaba kodu Algoritma 2.1 ile verilmiştir.

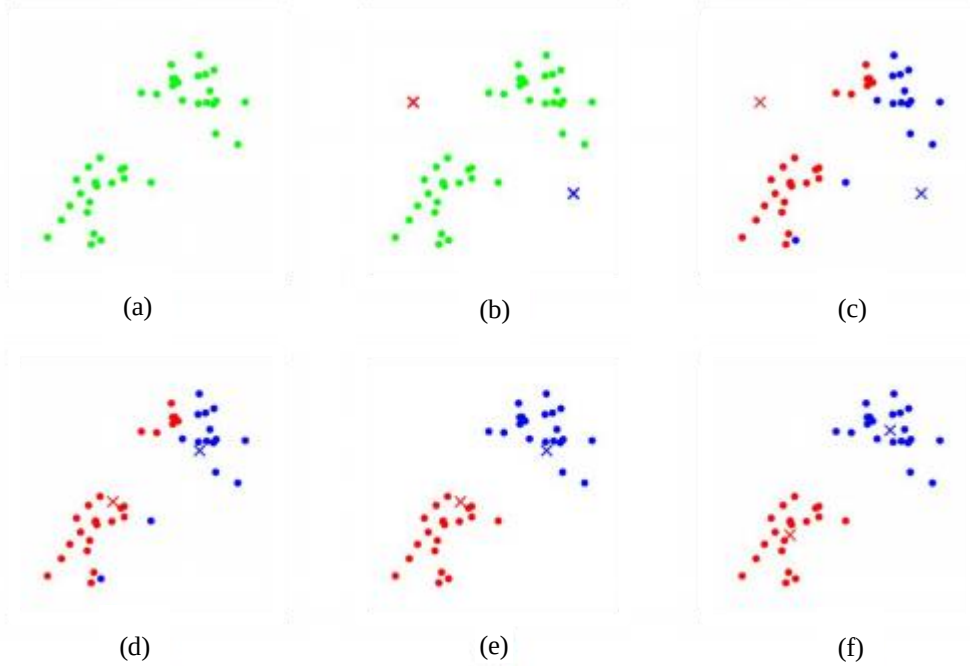
Algoritma 2.1. (k -ortalamalar)

Adım.1: k merkezli bir başlangıç çözümü belirle;

Adım.2: X kümesindeki verileri kendilerine en yakın merkezli kümeye ata.

Adım.3: Kümelerin merkezlerini belirle; Eğer hiçbir veri öncekinden farklı bir kümeye atanmadıysa dur, aksi halde Adım-2'ye git.

k -ortalamalar algoritmasının kolayca kavranmasını sağlayacak görsel bir örnek Şekil 2.4 ile verilmiştir.



Şekil 2.4. k -ortalamalar algoritmasının örneği. (a) Ham veri. (b) Başlangıç merkezleri. (c,d) İlk iterasyon. (e,f) İkinci iterasyon. Aynı renkteki noktalar aynı kümeye aittir. (Chris'ten, 2013)

k -ortalamalar algoritmasının her zaman yakınsayacağı kesin olsa da (Xu and Wunsch, 2009; Selim ve İsmail'den, 1980) başlangıç merkezlerinin seçimine bağlı olarak farklı yerel minimumlara ulaşabilir. Dahası, bu yerel minimumlar global minimumdan çok daha büyük olabilir. Bu sebeple, başlangıç merkezlerinin seçimi önemli bir alt problemdir.

En basit yol, başlangıç merkezlerini veri noktaları arasından rastgele seçmektir. Algoritmayı farklı rastgele başlangıç seçimleriyle birkaç defa çalıştırılması esasına dayanan bir araştırma (Jain and Dubes, 1988)'de verilmiştir.

Başlangıç çözümünün seçiminde temel yöntemlerden birisi, tek bir genel merkezden başlayarak her yeni merkezde o anki amaç fonksiyonu değeri minimum olacak şekilde artımlı olarak merkezler eklemektir (Kaufman and Rousseeuw, 1990).

Olasılıksal bir yöntem olan k -ortalamalar++ algoritmasında (Arthur and Vassilvitskii, 2007), seçilme olasılıkları uzaklıkların karesine oranlı olarak belirlenir ve artımlı olarak başlangıç merkezleri seçilir. Yazarlar çalışmada, yalnızca daha iyi bir kümeleme değil, daha hızlı bir yakınsama da elde edilebileceğini göstermişlerdir.

Yakın dönemde kombinatorik optimizasyon problemlerinin çözümünde kullanımları popülerleşen meta-sezgisel yöntemler, k -ortalamalar algoritmasının başlangıç seçimine olan hassasiyetini azaltmak için de uygulanmıştır. Genetik algoritma tabanlı iki yaklaşım (Babu and Murty, 1993; Krishna and Murty 1999) ile, ısıtım işlem yaklaşımıyla merkez seçimlerinin yapıldığı bir yöntem de (Brown and Huntley, 1991) ile verilmiştir.

Bulanık kümeleme modelini temel alan bulanık c -ortalamalar algoritması (Bezdek, 1984), bulanık temelli bir optimizasyon modelini çözerek bir kümeleme elde eder. Bulanık mantık modelinin esnekliği sayesinde, daha sağlam (robust), yani başlangıç çözümünün seçimi, veri setinin dağılımı vb. etkenlerden daha az etkilenen bir algoritmadır.

k -ortalamalar algoritmasının ve k -ortalamaların üzerine geliştirilmiş pek çok algoritmanın performansını etkileyen diğer bir güçlük, k sayısının belirlenmesidir. Bu güçlüğü aşmak için de tipik olarak farklı k değerleri için algoritma çalıştırılır ve elde edilen en iyi kümeleme sonuç kümelemesi olarak belirlenir. Merkez sayısı arttıkça, amaç fonksiyonunun değerinin artmayacağı kesindir. Bu nedenle, hangi kümelemenin sonuç kümelemesi olarak seçileceği de yine bir problemdir. Çözümü için ise, sayısal yöntemlerdeki gibi bir durdurma koşulu kullanılır: Ardışık k değerleri arasındaki iyileşmenin bir eşik değerinin altında olup olmadığı incelenir.

Yerel minimumları aşarak globale yakınsamak için geliştirilen etkin bir yöntem olan global k -ortalamalar algoritması (Likas et al., 2003) artımlı olarak merkezler ekleyerek en iyi kümelemeyi elde etmeyi amaçlar. Her bir adımda tüm veri seti içerisinde global bir arama yapılır. Global k -ortalamalar algoritmasının paralelleştirilerek hızlandırılması bu tezdeki çalışmalardan biridir ve detayları Bölüm 4.2’de verilmiştir.

2.3.3. Yoğunluk tabanlı kümeleme

Yoğunluk tabanlı kümeleme yöntemlerinin öne çıkan özelliği, farklı ve düzensiz şekillerdeki kümeleri belirleyebilmesidir. Bir kümeyi oluşturmak için, birbirine bağlantılı yoğun noktalar belirlenir. Yani bir küme, yoğunluğun yönlendirdiği herhangi bir yönde genişleyebilir. Böylece herhangi bir şekildeki küme oluşturulabilir. Yoğunluk değerlendirmesinin doğal bir etkisi olarak, yoğunluk tabanlı kümeleme yöntemleri gürültü (sıradışı veya anormal özelliklere sahip veri) ile baş edebilirler. Ayrıca, bu yaklaşımı benimseyen yöntemler genellikle düşük hesaplama karmaşıklığına sahiptirler. En belirgin dezavantaj ise, ortaya çıkan kümelerin nasıl değerlendirileceği ve yorumlanacağıdır.

Yoğunluk tabanlı kümelemenin temsilcisi DBSCAN (Density Based Spatial Clustering of Applications with Noise) algoritması kabul edilir (Berkhin, 2002; Han and Kamber, 2006). DBSCAN ε ve $MinPts$ diye adlandırılan iki parametre girdisine ihtiyaç duyar. Bu parametrelerin yardımıyla tanımlanan, yöntemin ve genel olarak yoğunluk tabanlı kümelemenin temel kavramları şunlardır:

Tanım 2.2. (ε -komşuluk) Bir x noktasının ε -komşuluğu, $d(x, y)$ yansımali ve simetrik bir uzaklık bağıntısı olmak üzere, $N_\varepsilon(x) = \{y \in X \mid d(x, y) \leq \varepsilon\}$ ile tanımlıdır.

Tanım 2.3. (*Çekirdek nokta*) En az $MinPts$ adet ε -komşusu olan bir veri noktası çekirdek noktadır.

Tanım 2.4. (*Yoğunluk ulaşılabilirlik*) x_1 çekirdek noktası ile y noktası arasında, ardışık noktalar birbirlerinin ε -komşusu olmak üzere $x_1, x_2, \dots, x_k$ gibi bir çekirdek noktalar dizisi varsa ve $y \in N_\varepsilon(x_k)$ ise, y noktası x_1 noktasından yoğunluk ulaşılabilir.

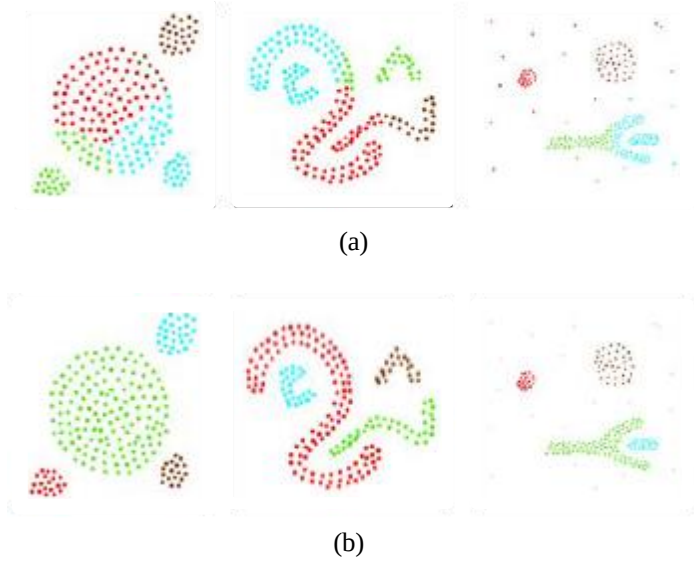
Tanım 2.5. (Yoğunluk bağlantılılık) x ve y gibi iki nokta aynı çekirdek noktadan yoğunluk ulaşılabilir ise, bu iki nokta birbirine yoğunluk bağlantılıdır.

DBSCAN veri noktalarını 3 kategoride ele alır. Yukarıda tanımı verilen *çekirdek nokta*, yeterince yoğun bölgedeki bir noktadır ve birbirlerine yoğunluk bağlantılı çekirdek noktalar aynı kümeye aittir. Kendisi bir çekirdek nokta olmayan, fakat bir çekirdek noktadan yoğunluk ulaşılabilir olan bir nokta *sınır noktası* olarak adlandırılır. Sınır noktaları, yoğunluk bağlantılı olduğu diğer noktalarla aynı kümeye aittir. Herhangi bir çekirdek nokta ile yoğunluk bağlantılı olmayan noktalar ise *gürültü noktası* olarak etiketlenir ve hiçbir kümeye ait olmazlar.



Şekil 2.5. DBSCAN yöntemine ait temel kavramlar. $MinPts = 4$ seçilmiştir. (a) x noktasının ϵ -komşuluğu. (b) Yoğunluk ulaşılabilirlik ve yoğunluk bağlantılılık ilişkisi. y noktası z noktasından yoğunluk ulaşılabilir, ancak z noktası y noktasından yoğunluk ulaşılabilir değildir. x , y ve z noktaları yoğunluk bağlantılıdır. x ve z noktaları çekirdek noktalardır; y ise sınır noktasıdır.

DBSCAN algoritmasının doğrudan bir uygulamasında, herhangi bir noktadan başlanır ve bu nokta eğer çekirdek noktaysa, noktanın tüm ϵ -komşuları bir kuyrukta saklanır. Kuyruktaki bir sonraki eleman da aynı şekilde değerlendirilir ve kuyruktan çıkarılır. Bu işlem, kuyruкта hiçbir eleman kalmayana kadar devam eder. Kuyruk boşaldığında, o ana kadar değerlendirilen ve gürültü noktası olmayan tüm noktalar aynı kümeye ait olur. Daha önce değerlendirilmeyen bir nokta varsa, yeni bir kuyruk oluşturularak aynı süreç tekrarlanır. Bir noktanın ϵ -komşularının sorgusu n adet verinin incelenmesini gerektirir. Bu sorgu her bir veri noktası için gerçekleştirileceğinden, algoritmanın zaman karmaşıklığı $O(n^2)$ dir. Eğer veri seti düşük boyutlu uzaysal (spatial) verilerden oluşuyorsa, R^* ağacı gibi bir veri yapısıyla ϵ -komşuluk sorgusu $O(\log n)$ zamanda yapılabilir (Beckmann et al., 1990). Bu durumda algoritmanın zaman karmaşıklığı $O(n \log n)$ olur.



Şekil 2.6. Çeşitli veri setleri üzerinde bölümlleme tabanlı ve yoğunluk tabanlı yöntemlerin sonuçlarının karşılaştırılması. (a) Bölümlleme tabanlı yöntem. (b) Yoğunluk tabanlı yöntem. (Ester et al.'den, 1996).

Yoğunluk tabanlı DBSCAN algoritmasının, uzaysal noktalardan oluşan büyük veri setleri için geliştirilmiş bölümlleme tabanlı CLARANS algoritmasıyla kümeleme performanslarının görsel bir karşılaştırılması Şekil 2.6 ile verilmiştir. Bu karşılaştırma, yoğunluk tabanlı kümeleme yaklaşımının düzensiz şekilli kümelerin belirlenmesindeki başarısını göstermektedir.

Kümelenmek istenen veriler noktasal değil, geometrik şekiller cinsinden tanımlı olabilir. Uzaysal veri tabanlarının pek çoğunda, poligon gibi, genişletilmiş veri nesneleri de bulunmaktadır. DBSCAN algoritmasını böylesi verilere uygun hale getirmek için, ϵ -komşuluğu, $d(x, y)$ gibi bir uzaklık bağıntısı yerine, yansıma ve simetri özelliklerine sahip farklı bir ölçüt (mesela poligonların kesişme durumu) ile tanımlanabilir. Ayrıca, yoğunluk değerlendirmesi komşu noktaların sayısı yerine değişik bir ölçütü de yapılabilir. Farklı yapıdaki verilere uygun GDBSCAN (Generalized DBSCAN) algoritması, DBSCAN algoritmasına bu iki genişletme uygulanarak geliştirilmiştir (Sander et al., 1998).

DBSCAN algoritması girdi parametrelerine oldukça duyarlıdır ve farklı yoğunluktaki kümeleri tespit etmede başarısızdır. OPTICS (Ordering Points To Identify Clustering Structures) algoritması (Mihaeli et al., 1999), yeni bir bağlantılılık özelliğine göre veri noktalarını akıllı biçimde sıralayarak bu problemleri aşmayı hedefler. Tek bir belirgin kümeleme elde etmek yerine, DBSCAN algoritmasının geniş bir parametre yelpazesine karşılık gelen, farklı yoğunluk değerlerine sahip bir kümelemeler dizisi oluşturur.

Parametre duyarlılığı sorununu çözmek ve daha sağlam bir yoğunluk tabanlı kümeleme algoritması tasarlamak için bulanık mantıktan faydalanılması şaşırtıcı değildir. Doğrudan DBSCAN algoritmasının bir genişletmesi olan FN-DBSCAN (Fuzzy Neighborhood DBSCAN), ε -komşuluğunun bulanık üyelik fonksiyonları yardımıyla belirlenmesi esasına dayanır (Nasibov and Ulutagay, 2009). Bu algorithmada da iki parametre girdisine ihtiyaç olmasına rağmen, DBSCAN algoritmasının aksine, parametrelerin çok geniş aralıkları için başarılı kümelemeler elde edilebilmektedir.

Yoğunluk tabanlı kümeleme ile bulanık mantığı birleştiren bir diğer algoritma olan FJP (Fuzzy Joint Points) algoritması (Nasibov and Ulutagay, 2007), gürültüye dayanıklılıktan ödün vermesine karşın, parametre bağımsız bir yöntemdir. Yoğunluk bağlantılılığı değerlendirmek yerine, noktalar arasında bir bulanık benzerlik matrisi oluşturularak matrisin geçişli kapanışı (transitive closure) hesaplanır. Bu geçişli kapanış matrisinin bağlantılı bileşenleri (connected components) birer kümeye karşılık gelir.

Bulanık bağlantılı noktalar (yani FJP), bu tez çalışmasının konusuna dahildir. Algoritmanın ilk önerildiği halinin doğrudan bir uygulaması çok yavaş çalışır ve ancak küçük veri setleri için kullanılabilir. FJP algoritması, türevleri ve paralelleştirilmesine ait detaylar Bölüm 4.3’de verilmiştir.

Bu bölümde bahsedilen yoğunluk tabanlı kümeleme algoritmaları, bulanık mantığa dayalı olanlar da dahil, kesin kümeleme modelini ele alırlar. Buna dayanarak şu yorum yapılabilir: Bulanık mantığın avantajlarından yararlanmak için, problemin bulanık olarak tanımlanmış olması şart değildir.

3. PARALEL HESAPLAMA

Klasik algoritmaların hemen hepsi (Bölüm 2’de sunulan algoritmalar da) seri hesaplama yapmak üzere geliştirilmiştir. Yani, aynı anda yalnızca tek bir işlem gerçekleştirileceği kabul edilmektedir. Paralel hesaplama ise, aynı anda birden fazla işlemin gerçekleştirilebildiği bir hesaplama modelidir.

İşlemci teknolojisi ilk elektronik bilgisayarlardan bu yana büyük bir hızla gelişerek günümüzdeki, 3 GHz üzerine çıkan saat vurum sıklığına (clock rate) sahip ve tek bir çevrimde (cycle) birden fazla işlem yapabilen mikro işlemcilerle ulaşmıştır. Bilgisayarların işlem gücünün arttırılmasına yönelik gelişimler işlemcilerin iyileştirilmesinden ibaret değildir. Özellikle son 20 yıl içerisinde, birden fazla işlemcilerin bir arada kullanıldığı paralel sistemler ön plana çıkmaktadır. Dahası, yalnızca işlem gücünü arttırdığı için değil, birçok gerçek hayat probleminin doğası gereği paralel olması nedeniyle, paralel hesaplama giderek popülerlik kazanmaktadır. Barney (2014a), paralelliğin geleceği hakkında şöyle yazmıştır:

“Geçtiğimiz 20+ yıl boyunca, her zamankinden hızlı ağlar, dağıtık sistemler ve çok işlemcili bilgisayar mimarileri (masaüstü bilgisayarlarda bile) ile belirlenen trendler, hesaplamanın geleceğinin paralellik olduğunu göstermektedir.”

Paralel hesaplama, karmaşık olayların simülasyonundan ticari hesaplama paketlerine, çevrimiçi arama motorlarından sanal gerçeklik uygulamalarına kadar geniş bir yelpazede büyük etki yaratmıştır. Uygulama seviyesinde paralelleştirmenin yanı sıra, doğrusal cebir, çizge teorisi, dinamik programlama, ayrık optimizasyon gibi alanlara ait temel algoritmaların paralel versiyonları, bir paralel algoritmalar literatürü oluşturmaya başlamıştır (Cormen et al., 2009).

Teoride, paralel bir algoritmanın, seri bir algoritmanın yaptığı işi daha kısa sürede gerçekleştireceği açıktır. Pratikte ise hızlanma sağlamak için dikkatli bir paralelleştirme uygulanmalıdır. Dolayısıyla, paralel bir algoritma uygulamadan bağımsız düşünülmemelidir. Paralel uygulamalar da, üzerinde çalışacağı paralel bilgisayar mimarisine bağımlıdır.

3.1. Paralel Bilgisayar Mimarileri

Birden fazla işlemcisi olan bir bilgisayar, bir paralel bilgisayar olarak kabul edilebilir. Bu durumda, modern kişisel bilgisayarların neredeyse tamamı birer paralel bilgisayardır. Genellikle yalnızca eğlence ve çoklu-medya özellikleri için

kullanılan akıllı telefon, tablet bilgisayar gibi ürünler dahi paralel mimariyle üretilmiştir.

Hesaplama camiasında, seri bilgisayarlar için rastgele erişimli bir mimari modeli üzerinde uzun zaman önce uzlaşmıştır. Fakat, paralel bilgisayar mimarileri için böylesi tekil bir model söz konusu değildir. Genelde belleğe erişimlerine göre sınıflandırılan paralel bilgisayar mimarilerinde, ya işlemcilerin doğrudan erişebildiği bir paylaşımlı bellek söz konusudur, ya da işlem birimlerinin kendilerine ait ayırık belleklerinin olduğu bir dağıtık bellek düzeni vardır (Cormen et al., 2009). Daha klasik bir sınıflandırma ise, emir-veri ilişkisine dayalı Flynn Sınıflandırmasıdır.

3.1.1. Flynn sınıflandırması

Paralel mimarilerin ilk sınıflandırılmasının Flynn tarafından 1966 yılında yapıldığı kabul edilmektedir (Flynn, 1966). Emir-veri ilişkisine göre yapılan bu klasik sınıflandırma Flynn Sınıflandırması olarak anılır.

Çizelge 3.1. Flynn Sınıflandırması

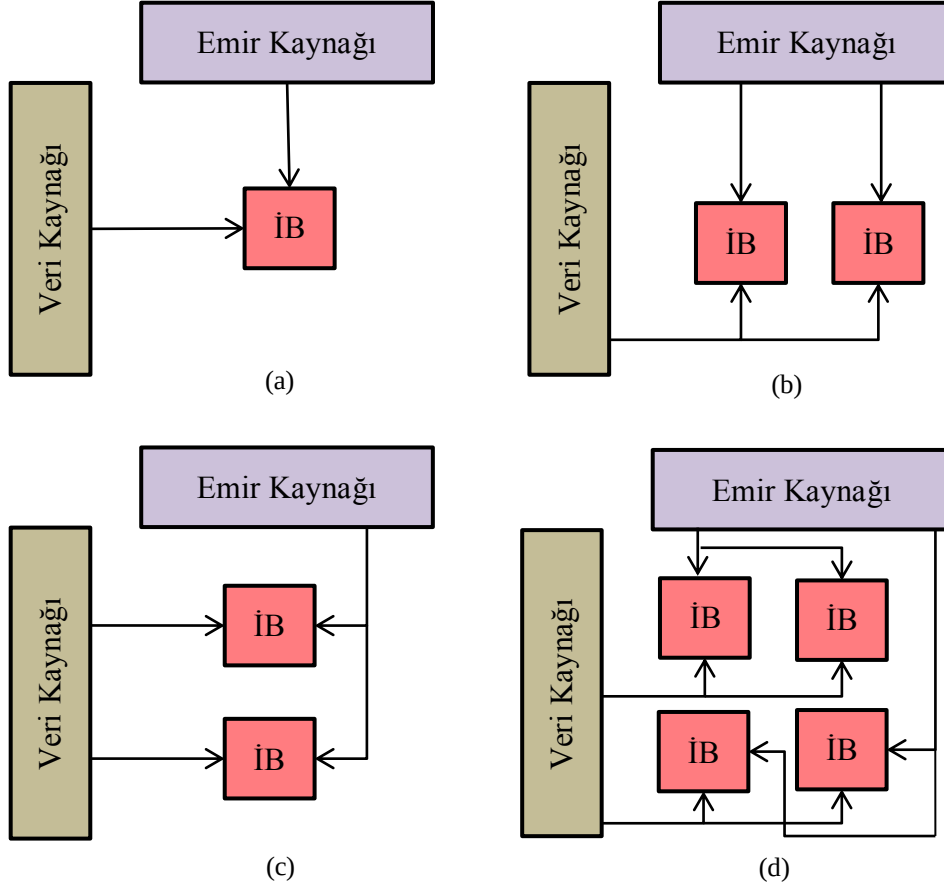
<i>Flynn Sınıflandırması</i>	Tek Veri	Çok Veri
Tek Emir	TETV (SISD)	ÇETV (MISD)
Çok Emir	TEÇV (SIMD)	ÇEÇV (MIMD)

Tek Emir Tek Veri, aslında bir seri bilgisayar mimarisidir. Bir veri üzerinde aynı anda yalnızca bir emir gerçekleştirilir. İlk elektronik bilgisayarlar ve tek çekirdekli kişisel bilgisayarlar bu sınıftandır.

Çok Emir Tek Veri, hesaplamanın doğasına uygun bir paralel mimari modeli değildir. Tek bir veri üzerinde, birden çok, farklı emir uygulanır. Bu sınıftan bir bilgisayara nadir rastlanır. Bir şifreyi farklı şekilde çözmeye çalışan bir dizi şifre kırıcı böyle bir mimaride çalışabilir.

Tek Emir Çok Veri, düzenli hesaplama gerektiren işlemlere uygun, paralel bir mimaridir. Birden çok veri üzerinde aynı anda aynı işlem yapılır. Özellikle grafik işlemlerine çok uygundur. Bu nedenle, grafik işlemcilerde yaygın olarak bu mimari kullanılır.

Çok Emir Çok Veri, neredeyse tüm modern paralel bilgisayarların mimarisidir. Birden çok veriye, çeşitli farklı emirler uygulanır. Çok çekirdekli kişisel bilgisayarlar, hesaplama kümeleri ve süper bilgisayarlar bu sınıftandır.

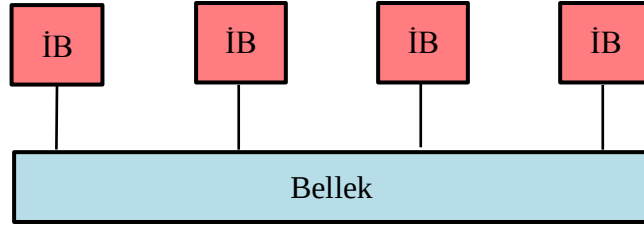


Şekil 3.1. Flynn Sınıflandırmasına göre paralel bilgisayar mimarileri. (a) Tek Veri Tek Emir. (b) Çok Emir Tek Veri. (c) Tek Emir Çok Veri. (d) Çok Emir Çok Veri.

3.1.2. Paylaşımlı bellek mimarisi

Paylaşımlı bellek mimarisinde, tüm işlem birimleri belleğin tamamına aynı adres uzayı üzerinden erişebilir. İşlem birimleri birbirlerinden bağımsız çalışabilir, ancak belleği ortak kullanırlar. Bir programın çalışması süresince işlemciler arasındaki iletişim bu sayede sağlanır.

Paylaşımlı bellek mimarisine genel olarak simetrik çoklu işlemcilerde (symmetric multi-processor) rastlanır. Bir simetrik çoklu işlemci, her biri tek başına komple bir işlem birimi olan, ortak bir belleğe erişimli birden çok *çekirdek* içeren bir işlemcidir. Tipik olarak, her bir çekirdeğin kendine özel bir ön belleği (cache) bulunur. Modern kişisel bilgisayarların işlemcileri simetrik çoklu işlemcilerdir.



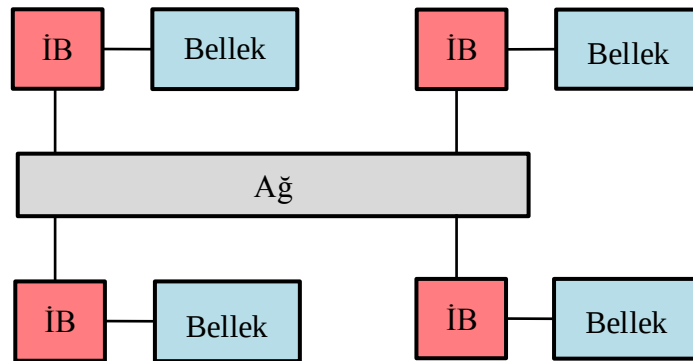
Şekil 3.2. Paylaşımlı bellek mimarisi.

Simetrik çoklu işlemciler gibi işlemciler, düzgün bellek erişimine (uniform memory access) sahiptir. İşlemcilerin belleğe erişim süreleri eşittir ve bir işlemcinin bellek üzerinde yaptığı değişiklik, diğer işlemciler tarafından eş zamanlı olarak görülür. Birden çok simetrik çoklu işlemcinin bir veri yoluyla bağlandığı bir işlemcide ise, düzgün olmayan bellek erişimi (non-uniform memory access) vardır. Bir çekirdek, farklı bir simetrik çoklu işlemcinin belleğindeki veriye doğrudan, fakat o simetrik çoklu işlemcideki çekirdeklerden daha yavaş erişir (Barney, 2014a).

3.1.3. Dağıtık bellek mimarisi

İşlem birimlerinin kendine özel belleklerinin olduğu, iletişimin bir ağ ile gerçekleştirildiği mimariler, dağıtık bellek mimarisi olarak adlandırılırlar. Bir işlemci, yalnızca kendi belleğindeki verilere doğrudan erişebilir ve bu bellekte yaptığı bir değişiklik yalnızca kendini etkiler.

Dağıtık bellek mimarisindeki bir sistemde, bellekleri ve dolayısıyla işlemcileri birbirlerine bağlayan ağın niteliği büyük önem taşır. Dağıtık bellekli bir sistemde, gelişmiş bir ağ sistemi söz konusu olabileceği gibi, basit bir ethernet ağı da kullanılabilir.

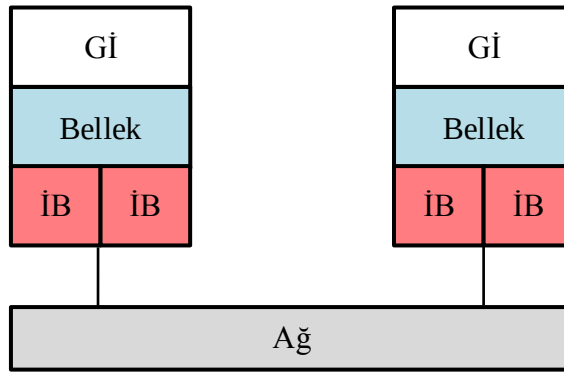


Şekil 3.3. Dağıtık bellek mimarisi.

3.1.4. Karma mimari

Paylaşımlı belleğe sahip paralel bilgisayarların, dağıtık bellek mimarisine benzer şekilde bir ağ üzerinden iletişimi sağladıkları mimaridir. Gelişmiş hesaplama kümeleri ve süper bilgisayarlar karma mimariye sahiptirler.

Paylaşımlı bellek bileşeni, çok çekirdekli bir merkezi işlem birimi ve/veya bir grafik işlemciden oluşabilir. Grafik işlemcilerin genel amaçlı hesaplamalarda kullanılması, son birkaç senede hızla yaygınlaşmıştır. NVIDIA gibi grafik işlemci üreticilerinin öncülük ettiği bu akıma, Intel de “hızlandırıcı” olarak adlandırdığı yeni nesil işlemci türüyle katılmıştır.



Şekil 3.4. Karma mimari.

3.2. Paralel Programlama Modelleri

Bir paralel programlama modeli, bir paralel programın temel özelliklerini ve parçalarını belirleyen bir programlama şablonudur. Paralel programlama modelleri teorik olarak mimariden tamamen bağımsızdırlar. Yani, herhangi bir paralel programlama modeli, herhangi bir paralel bilgisayar için uygulanabilir. Buna karşın, paralel programlama modelleri dolaylı olarak mimariye yönelik tasarlanmıştır ve bir paralel uygulamadan en iyi verimi alabilmek için, donanıma uygun bir model seçilmesi son derece önemlidir.

3.2.1. Bellek paylaşım modeli

Bellek paylaşım modelinde, görevler (task) asenkron okuma ve yazma yapabildikleri bir adres uzayını paylaşırlar. Her bir görevin farklı bir işlemcide

çalıştığı düşünülebilir. Ortak belleğe erişimi kontrol etmek için kilit ve semafor gibi mekanizmalar kullanılır.

Bir iletişim mekanizmasına gerek yoktur; ortak bellek üzerinden bilgi alışverişi yapılır. Verilere “sahip olma” söz konusu olmadığı için, her bir görevin yerel verilerini programcı idare etmek zorundadır. Bu kontrolün kavranması ve uygulanması zordur (Barney, 2014a).

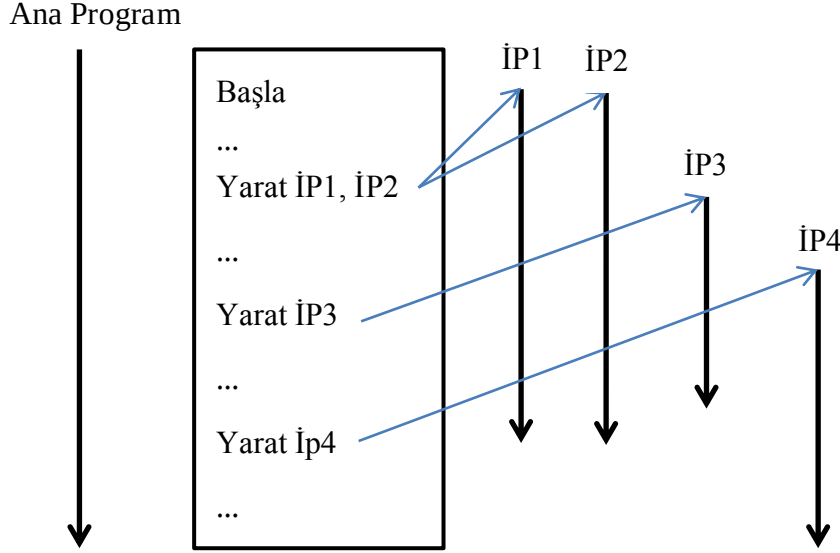
3.2.2. İş parçacığı (thread) modeli

İş parçacığı modelinde, ana görev küçük iş parçacıklarına bölünerek gerçekleştirilir. Bir iş parçacığı (thread), sanal bir işlemci soyutlaması olarak düşünülebilir. İş parçacığı modelinde de paylaşımlı bellek söz konusudur. Bellek paylaşımı modelinden en önemli fark ise, her iş parçasının yerel verileri olmasıdır. Tabii ki, tüm iş parçacıklarının ortak erişebildiği global bir bellek alanı da vardır.

İş parçacığı modeli statik veya dinamik olarak uygulanabilir. Statik iş parçacığı modelinde, her iş parçacığı bağımsız bir programı çalıştırır ve işletim sistemi bu iş parçacığını bir işlemciye yükler. Eğer bir iş parçasının yükleneceği müsait bir işlemci yoksa (örn. tek işlemcili bir bilgisayarda birden fazla iş parçası oluşturulduğunda), işletim sistemi diğer iş parçalarından birinin çalışmasını bir süreliğine durdurur ve diğer bir iş parçasını işlemciye yükler. İşletim sistemi aracılığıyla iş parçacığı açılıp kapatılabilse de, bu işlemler ciddi süre alır. Ayrıca, iş yükünü dengeli dağıtmak ve iş parçacıkları arasındaki iletişimi sağlamak tümüyle programcının sorumluluğundadır. En basit uygulamalarda bile, karmaşık iletişim protokolleri kullanılması gerekir. Dinamik iş parçacığı modelinde ise, uygulama aşamasında iş parçacıkları yaratılıp yok edilebilir. Bu modeli destekleyen bir platformda bir zaman planlayıcısı (scheduler) yardımıyla, basit komutlarla iş yükü dengesi kontrol edilebilir, ve hatta dengeleme otomatik olarak yapılabilir (Cormen et al., 2009). Tez içerisinde iş parçacığı modeli ile ilgili tüm anlatımlarda, dinamik iş parçacığı modeli ima edilmektedir.

İş parçacığı modelinin işleyişi, alt prosedürler çağıran tek bir programinkine benzetilebilir. Bu tek program, işletim sisteminden gerekli bütün kaynakları temin ederek çalışmaya başlar. Tipik olarak, seri bir işlem gerçekleştirir ve sonra, belirli sayıda iş parçacığı yaratır. Yaratılan iş parçacıkları eş zamanlı olarak çalıştırılır ve ana programın tüm kaynaklarını kullanabilirler. İş parçalarının yerel verileri olabilir. Ana programın ortak bellek alanı üzerinden iletişim sağlanır. Önemli olan

nokta, iş parçalarının aynı anda aynı adresteki bir veriyi değiştirmelerini engellemektir. Bunun için de, çeşitli senkronizasyon yapıları kullanılır (Barney, 2014a).



Şekil 3.5. İş parçacığı modeli.

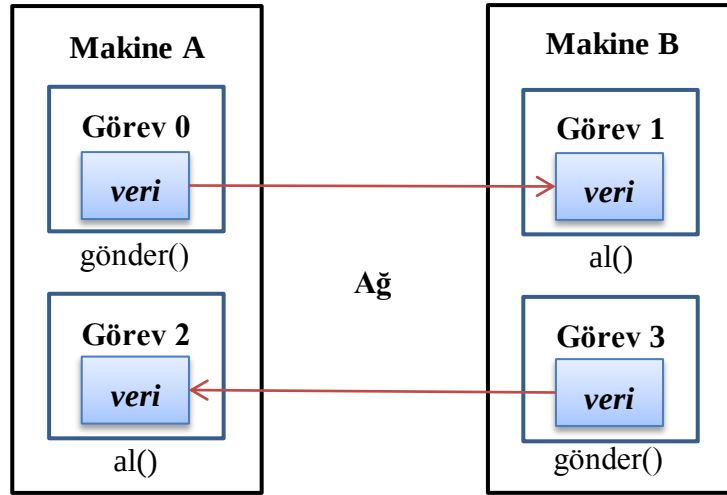
Şekil 3.5 iş parçacığı modelinin çalışma şeklini, iş parçalarını alt prosedürler gibi ele alarak göstermektedir. İş parçaları herhangi bir zamanda yaratılabilir ve ana program çalıştığı sürece herhangi bir zaman boyunca, birbirlerinden bağımsız ve eş zamanlı olarak çalışabilirler.

İş parçacığı modeli ile paralel program yazmak için en yaygın platformlar Pthreads ve OpenMP uygulama programlama arayüzleridir. Pthreads, C dili için bir dizi paralel fonksiyon ve bazı özel programlama yapılarının tanımlı olduğu bir kütüphane aracılığıyla kullanılır. OpenMP ise, kaynak koda eklenen derleyici komutlarına dayalıdır. C/C++ ve Fortran dillerinde OpenMP kullanılabilir. Ayrıca, grafik işlemcilerin programlanmasına olanak sağlayan CUDA ve OpenCL gibi uygulama programlama arayüzleri de iş parçacığı modelini temel alırlar.

3.2.3. Mesaj geçme modeli

Dağıtık bellek mimarisiyle örtüşen mesaj geçme modelinde, her görev kendine ait bir belleğe ve diğer sistem kaynaklarına sahiptir. Görevlerin ortak bir erişim alanı yoktur. Görevler iletişimi sağlamak için bir ağ üzerinden mesajlaşırlar.

Mesaj geme modelinde paralelliğın kontrolü tümüyle programcıdır. Görevlerin işlem yükünün paylaşımı, bir görevden mesaj gönderme ve o mesajın gönderildiğı görevlerin bu mesajı karşılama işlemleri, programlama aşamasında belirlenir. Ortak bir bellek bölgesi olmadığından, belleğe eş zamanlı yazma durumu söz konusu değildir. Öte yandan, mesajlaşma işi görece yavaş bir ağ üzerinden gerçekleştiğı için, iletişim maliyeti çok yüksektir. Bu nedenle, paralel programın performansını belirleyen en önemli unsur mesajlaşma maliyetidir.



Şekil 3.6. Mesaj geme modeli (Barney'den, 2014a).

Mesaj geme modeli doğrudan MPI (Message Passing Interface) ile bağdaştırılabilir. MPI uygulama programlama arayüzünde, mesajlaşma ve paralellikle ilgili diğer tüm işler için fonksiyonlar tanımlıdır. MPI ile paralel program geliştirmek için C/C++, Java, Fortran, Python, R vb. birçok dil kullanılabilir.

3.2.4. Karma model

Mevcut donanımdan en yüksek verimi almak amacıyla farklı paralel programlama modelleri bir arada kullanılabilir. Böylesi bir modele de karma model ismi verilir.

İşlem gücü çok yüksek olan modern hesaplama sistemleri, diğer bir deyişle süper bilgisayarlar, yüksek hızlı bir iletişim ağıyla birbirlerine bağlanmış çok işlemcili (çekirdekli) hesaplama düğümlerinden meydana gelmektedirler. Son birkaç sene içerisinde, süper bilgisayarlarda hızlandırıcılar (örn. grafik işlemciler)

da kullanılmaya başlanmıştır. Hesaplama düğümleri ve hızlandırıcılar paylaşımlı bellek mimarisine sahiptir. Düğümler arasında ise paylaşımsız bellek mimarisi söz konusudur. Kısaca, bu bir karma mimaridir. Dünyanın en “güçlü” bilgisayarlarının sıralamasına ve bu bilgisayarlar hakkında detaylı bilgilere TOP500 projesinden (<http://www.top500.org>) ulaşılabilir.

Karma mimariye sahip bir süper bilgisayarda koşturmak üzere yazılan bir programda, her bir hesaplama düğümünde çalışacak program parçası OpenMP ile iş parçacığı modeliyle yazılabilir. Hızlandırıcılar için kendilerine özel (örn. NVIDIA için CUDA) bir uygulama programlama arayüzü kullanılabilir. Düğümler arasında da MPI ile mesajlaşarak global bir paralellik sağlanabilir. Böyle bir geliştirme süreci tek bir model kullanmaya göre daha zahmetli ve daha zaman alıcı olacaktır. Fakat karma bir sistemden maksimum verim elde etmek, günümüzdeki araçlarla, ancak bu şekilde mümkün olabilir.

3.3. Paralel Algoritma Tasarımı ve Performans Analizi

Hesaplama dünyası seri algoritmaların tasarımı ve analizi konusunda bir fikir birliği sağlamıştır: Seri algoritmalar RAM (Random Access Machine) modeliyle tasarlanır. RAM modelinde, bir belleği yöneten tek bir işlemci vardır. İşlemci aritmetik işlemleri, mantıksal işlemleri ve bellek işlemlerini birim zamanda gerçekleştirir. Bu soyut model somut mimariyle oldukça uyumludur. Böylelikle, algoritmanın analizi oldukça kolay ve isabetli yapılabilir.

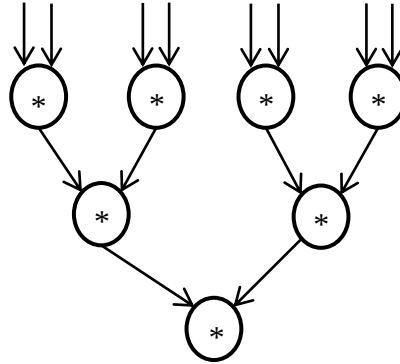
Paralel algoritmalarda ise durum daha karmaşıktır. Çünkü, daha önce belirtildiği gibi, paralel bilgisayar mimarileri ve programlama modelleri farklılık göstermektedir.

3.3.1. Soyut hesaplama modelleri

En çok kabul gören paralel algoritmik model, RAM (Random Access Machine) modelinin bir genelleştirmesi olan PRAM (Parallel Random Access Machine) modelidir. RAM modelinde olduğu gibi her bir işlem birim zamanda gerçekleştirilir. İşlemci sayısı ise p gibi bir pozitif tam sayıdır ve işlemciler eş zamanlı çalışırlar. Örneğin, $p = 4$ için, 4 nesne arasında yapılan bir arama 1 adım sürer. $p = 1$ durumu RAM modeline denk gelir. PRAM modeli de belleğe erişim yöntemine göre alt sınıflara ayrılır. Bunlar, aynı anda yalnızca tek bir işlemcinin bellekten okuma/yazma yapabildiği EREW (Exclusive Read Exclusive Write),

okumanın eş zamanlı yapılabildiği, yazmanın ise eş zamanlı yapılamadığı CREW (Concurrent Read Exclusive Write), eş zamanlı okuma/yazmanın yapılabildiği CRCW (Concurrent Read Concurrent Write) modelleridir. Eş zamanlı okuma ve ayırık zamanlı yazmanın yapılabildiği erişim modeli doğal bir paralel ortam kurgusu olmadığı için kullanılmamaktadır.

PRAM modeli soyut bir makine üzerine kurulmuştur. Bu nedenle, PRAM modelinin başarısı bir ölçüde uygulamaya ve mimariye bağlıdır. Daha soyut bir yaklaşım, işlemci hareketlerini değil, algoritma adımlarını değerlendirmektir. İş derinliği modelleri bu yaklaşıma dayanmaktadırlar. İş derinliği modellerinin en temel halinde çizgeye benzer tepe ve yönlü ayırıt gösterimi kullanılır. Bir tepe, gerçekleştirilen bir işlemi temsil eder. Bir işlemin girdi ve çıktıları da yönlü ayırıtlarla belirtilir. Birbirine bağlı işlem basamaklarının sayısı derinlik (D), toplam işlem sayısı ise iş (W) şeklinde tanımlanır. Bir algoritmanın yalın performansı derinlikle değerlendirilebilir. Paralellik ölçütü ise $P = \frac{W}{D}$ dir (Blelloch and Maggs, 2004).



Şekil 3.7. İş derinliği modelinde yönlü çizge gösterimi.

Şekil 3.7, yukarıda bahsedilen gösterim biçimi ile iş derinliği modelini örneklemektedir. Problemin 8 adet sayının çarpımı olduğu düşünülebilir. Örnekte, $D = 3$, $W = 7$ ve $P = 7/3$ tür. Genel durumda, veri sayısı n 2'nin bir kuvveti olmak üzere, $D(n) = O(\log_2 n)$ ve $W(n) = n - 1$ olacaktır. Ayrıca, seri algoritmanın ($p = 1$) çalışma zamanı fonksiyonu $T(n)$ olmak üzere, $W(n) = T(n)$ eşitliği dikkate değerdir.

3.3.2. Teorik hızlanma limitleri

Paralel algoritmelerde, seri algoritmelerdeki gibi yalnızca veri sayısına veya veri boyutuna bağlı Big-O gibi bir asimptotik notasyonla zaman karmaşıklığı ve üst limit belirlemek kullanışlı değildir. Çünkü, işlemci sayısı p 'yi, veri sayısı (veya problem boyutu) n cinsinden ifade etmek anlamsızdır. Dahası, pratikte her zaman $p \ll n$ dir. Örnek olarak, bir seri algoritmanın hiçbir fazladan maliyet olmadan tümüyle paralelleştirildiğini varsayalım. Seri algoritmanın karmaşıklığı $O(g(n))$ olmak üzere, paralel algoritmanın karmaşıklığı $O\left(\frac{g(n)}{p}\right) = O(g(n))$ olur. Dolayısıyla, bir algoritmadaki paralelleştirmenin etkisini bu yolla analiz etmek isabetli olmaz.

Bir hesaplamamanın zamanının o hesaplamamanın en yavaş parçası tarafından belirleneceğini söyleyen Amdahl Yasası (Amdahl, 1967), paralel algoritmalar için bir hızlanma limiti belirlemede önemli bir yol göstericidir. p işlemci sayısı, $S(p)$ paralellikle elde edilen hızlanma oranı, algoritmadaki seri hesaplama süresi T_s , paralel hesaplama süresi T_p , algoritmanın seri kısmı (seri işlemlerin sayısı) f_s , paralel kısmı f_p ve $f_s + f_p = 1$ olmak üzere, Amdahl Yasası şöyle ifade edilebilir:

$$S(p) = \frac{T_s}{T_p} = \frac{1}{f_s + \frac{f_p}{p}}. \quad (3.1)$$

Örneğin Amdahl Yasası'na göre, 2 işlemcili bir paralel hesaplamada algoritma %40 oranında paralelleştirilmişse, paralel program seri programdan en fazla $1/(0.6 + 0.2) = 1.25$ kat hızlı olabilir. Eğer algoritmanın tamamı paralelse, $S(p) = p$ olur. Bu en iyi durumda, doğrusal hızlanma elde edildiği söylenir. Seri bir algoritma için $f_s = 1$ ve $f_p = 0$ olduğundan, $S(1) = 1$ olur. Yani, bir hızlanma beklenmez.

Amdahl Yasası, belirli büyüklükteki bir veri seti üzerinde gerçekleştirilen hesaplamamanın işlemci sayısına bağlı hızlanma oranını belirtir. Gustafson Yasası (Gustafson, 1988) ise farklı bir bakış açısı ile hızlanmayı ele alır. Gustafson Yasası'nda, belirli bir sürede, işlemci sayısına bağlı olarak, işlenebilecek verilerin sayısındaki artış öngörülmektedir. Yine, p işlemci sayısı ve f_s algoritmanın seri kısmı olmak üzere, Gustafson Yasası şöyle ifade edilebilir:

$$S(p) = p - f_s * (p - 1). \quad (3.2)$$

Önceki örnek Gustafson Yasası'na göre değerlendirilirse, elde edilebilecek maksimum hızlanma $2 - 0.6 * (2 - 1) = 1.4$ olur. Bu hızlanma süre bakımından değil, işlenebilecek veri setinin büyüklüğü bakımındandır. Algoritmanın tümüyle paralel olması durumunda, yine doğrusal bir üst sınır ortaya çıkar. Seri bir algoritmada da, yine $S(1) = 1$ olur.

Pratikte durum daha kötüdür. En azından girdi/çıkı için programın bir kısmı seri çalışmalıdır. Asıl önemli güçlük ise, teoride ihmal edilen iletişim, senkronizasyon, iş parçacıklarının yaratılması, yük dengeleme prosedürleri vb. paralel programlama unsurlarının getirdiği fazladan yüklerdir.

Teorik analiz ile uygulama performansı arasında ilginç bir tutarsızlık daha gözlemlenebilir. Bu sıra dışı durumda, uygulama teorik hızlanma limitini aşabilir. Bunun sebebi, ön belleğin işin içine girmesidir. Birden fazla işlemci kullanılırken yalnızca işlem gücü değil, ön bellek kaynağı da artar. Ön bellek bellekten daha hızlı çalıştığından, sadece hesaplamalar dikkate alınarak belirlenen sınırdan daha yüksek bir hızlanma elde edilebilir. Fakat bu çok özel bir durumdur ve yalnızca küçük hesaplamalarda karşılaşılır. Hemen her zaman, paralelleştirme maliyetleri bu küçük hızlanmadan çok daha büyük olacaktır.

4. PARALEL KÜMELEME

Bir insanın genetik yapısına, uluslararası bir şirketin müşterilerinin alışverişlerine, yüksek çözünürlüklü bir dizi resime, bir quantum fiziği deneyindeki parçacıkların çeşitli anlardaki durumlarına ait veriler üzerinde başarılı bir küme analizi yapılmak istendiğinde çok ağır bir işlem yükü söz konusudur. Kümeleme başarısından mümkün olduğunca ödün vermeden böylesi büyük veri setlerinin kümelenmesi için, bilgisayarlardan yüksek verim alınması gerekir. Bu gereklilik, modern bilgisayarların yapısı göz önüne alındığında, açıkça paralel hesaplama işaret etmektedir.

Algoritmaların hızı artıkça kümeleme başarısı genellikle azalmaktadır. Bu tezde paralelleştirilen iki algoritma, kendi türlerinde yüksek kümeleme başarısı elde edebilen, görece yavaş algoritmalar. Bir tanesi, bölümlene tabanlı artımlı bir algoritma olan GKM algoritması, diğeri de FJP yaklaşımıyla geliştirilmiş, yoğunluk tabanlı bulanık bir algoritmadır.

Paralel hesaplama alanında kabul gören tek bir programlama modelinin olmayışı, araştırmacıyı bir tercihe itmektedir. Yine de, daha yaygın ve ulaşılabilir olan modelin paylaşımlı bellek modeli olduğu gözlemlenebilir. Cormen et al. (2009) çok çekirdekli teknolojinin ilerleyişiyle tüm yeni dizüstü ve masaüstü bilgisayarların aslında birer paylaşımlı bellekli paralel bilgisayar olduğunu ve akımın paylaşımlı bellek modeline doğru yöneldiğini söylemektedir. Bu tez çalışmasında, paylaşımlı bellek mimarisine uygun olan iş parçacığı modeli (Bkz. Bölüm 3.2.2) tercih edilmiştir.

4.1. Paralel Global k -ortalamalar Algoritması

Global k -ortalamalar algoritması, Bölüm 2.3.2’de bahsedildiği gibi, bölümlene tabanlı artımlı bir algoritmadır. k -ortalamalar algoritmasını temel alır ve tek bir merkez noktasından başlayarak her iterasyonda yeni bir merkez eklemek üzere nihai kümelemeyi elde eder. Kümelenmek istenen veri seti $X = \{x_1, \dots, x_n\}$ olmak üzere, global k -ortalamar algoritmasının bir kaba kodu Algoritma 4.1 ile verilmiştir.

Algoritma 4.1. (*Global k-ortalamlar: GKM*)

Adım.1: c_1 merkezini hesapla: $c_1 = \frac{1}{n} \sum_{i=1}^n x_i$; $q = 1$;

Adım.2: $q = q + 1$; Eğer $q > k$ ise dur.

Adım.3: $(c_1, \dots, c_{q-1}, x_i)$, $i = 1, \dots, n$ başlangıç çözümleri ile Algoritma 2.1 (k -ortalamlar) uygula ve en iyi kümelemeye karşılık gelen merkezleri sakla: $(c'_1, \dots, c'_q)$;

Adım.4: $c_i = c'_i$, $i = 1, \dots, q$; Adım-2'ye git.

Bu kaba kodda, durdurma koşulunun sınanması Adım.2'de gerçekleşmektedir. k değerinin otomatik olarak belirlendiği bir versiyonda f_i (2.11) denkleminde göre i adet merkezin olduğu iterasyondaki amaç fonksiyonunun değeri olmak üzere, (4.1) koşulu ile Adım-4'te kontrol edilebilir. Bu koşul sağlanırsa algoritma durur.

$$\frac{f_{q-1} - f_q}{f_1} < \varepsilon. \quad (4.1)$$

Global k -ortalamlar algoritması kümeleme başarısı yüksek bir algoritma olmasına karşın, her bir q değeri için $|X|$ defa k -ortalamlar uygulandığından, hesaplama yükü ağırdır. Bu durum, global k -ortalamlar algoritmasının büyük veri setlerinde uygulanmasına engel olmaktadır. Tüm veri noktaları yerine bazı özel seçilen merkezler ile k -ortalamlar uygulanarak bu dezavantaj aşılıp çalışılmıştır (Bagirov and Mardaneh, 2006; Shen and Hasegawa, 2006). Paralel global k -ortalamlar algoritmasında, kümeleme performansından ödün verilmeyen klasik versiyon temel alınmıştır.

Algoritmanın en çok zaman alan kısmının her bir iterasyonda n defa k -ortalamlar algoritmasının çalıştırılması olduğu açıktır. Yani, son küme sayısı K olmak üzere, $K*n$ defa k -ortalamlar çağrılır. İzlenebilecek 2 paralelleştirme stratejisi, doğrudan k -ortalamlar algoritmasını paralelleştirmek veya her iterasyondaki n adet k -ortalamlar çağrısını iş parçalarına bölmektir. İlk stratejide, her çağrı için iş parçaları yaratılacak ve sonlandırılacaktır. Bu fazladan yük, paralel uygulamanın performansını önemli ölçüde düşürecektir. İkinci stratejide bu dezavantaj yoktur. Diğer yandan, her bir iş parçasında belirlenen en iyi kümeleme, iş parçasının yerel belleğinde yer alacaktır. Bu kümelemelerden en iyisi, seri olarak değerlendirilerek en iyi merkezler sonraki iterasyona

aktarılmalıdır. Yine de, her iterasyonda yalnızca iş parçacığı sayısı olan p defalık bir seri işlem yapılacağından, maliyet yüksek olmayacaktır.

Kümelenecek veri seti $X = \{x_1, \dots, x_n\}$ olmak üzere, paralel global k -ortalamalar algoritmasının kaba kodu aşağıdaki gibidir.

Algoritma 4.2. (*Paralel global k -ortalamalar: P-GKM*)

Adım.1: c_1 merkezini hesapla: $c_1 = \frac{1}{n} \sum_{i=1}^n x_i$; $q = 1$;

Adım.2: $q = q + 1$; Eğer $q > k$ ise dur.

Adım.3: X veri setini p eşit (ya da kabaca eşit) parçaya böl: $X = \bigcup_{i=0}^{p-1} X^i$;

/ Paralel bölgenin başı */*

p adet iş parçacığı yarat ve indisler: $t = 0, \dots, p - 1$;

$(c_1, \dots, c_{q-1}, x_i^t)$, $x_i^t \in X^t$ başlangıç çözümleri ile k -ortalamalar uygula ve en iyi kümeleme fonksiyonu değeri f^t 'yi ve ona karşılık gelen merkezleri sakla: $(c_1^t, \dots, c_q^t)$;

Tümünün görevleri bittiğinde iş parçacıklarını sonlandır;

/ Paralel bölgenin sonu */*

Adım.4: $f^m = \max_{i=0, \dots, p-1} f^i$;

Adım.5: $c_i = c_i^m$, $i = 1, \dots, q$; Adım.2'ye git.

3. adımdaki paralel bölgede, her bir iş parçacığı kendine ait t indisine bağlı olarak eş zamanlı işlem yapar. t hariç tüm değişkenler ortak bellektedir. f^t ve $(c_1^t, \dots, c_q^t)$ üzerindeki işlemler t indisine bağlı olarak dağıtıldığı için, eş zamanlı yazma söz konusu değildir.

Paralel global k -ortalamalar algoritmasının performans analizini yapabilmek için, önce k -ortalamalar algoritmasının hesaplama karmaşıklığı incelenmelidir. Bu incelemeyi, tipik olarak yalnızca veri sayısı n cinsinden yapılması, isabetli olmaz. Uzaklık hesaplama işlemleri, veri boyutu olan m cinsinden ifade edilmelidir. Küme sayısı k olmak üzere, Algoritma 2.1'in 2. adımında, her bir noktanın her bir merkeze olan uzaklığı hesaplandığından $O(knm)$ zaman alır. 3. Adımda, verilere ait değerler bir kez toplanması gerekir. Yani bu adım $O(nm)$ zamanda çalışır. Kaç iterasyon sonunda yakınsanacağı bu değişkenler ile belirlenemez. Bu

durumda iterasyon sayısı I kabul edilirse, k -ortalamalar algoritmasının zaman karmaşıklığı $O(Ikmn)$ olur (Manning et al., 2008).

Global k -ortalamalar algoritması, maksimum küme sayısı olan K adet iterasyon yapar. Her bir iterasyonda, n kez k -ortalamalar çağırıldığından, hesaplama karmaşıklığı $O(K^2mn^2)$ olur.

Paralel global k -ortalamalar algoritmasının ilk adımında, $O(nm)$ işlem gerçekleştirilir. 2. adım ihmal edilebilir. 3. adım K defa çalışır ve her seferinde $O(Kmn^2)$ işlem yapılır. Bu adım tamamen paraleldir ve zamanı işlemci sayısı p ile bölünebilir. 4. Adım, paralel algoritmayla doğan serileştirme adımıdır ve karmaşıklığı $O(pm)$ dir. Hemen hemen her zaman, K ve p sayıları n ve m sayılarından çok küçüktür. Karşılaştırmayı kolaylaştırmak için iterasyon sayısı da ihmal edilirse –aslında bu durum paralelliğin aleyhinedir–, algoritmanın paralel kısmının $O(mn^2)$ zamanda, seri kısmının $O(m)$ zamanda çalışacağı söylenebilir. Sonuçta, algoritmanın seri parçası yalnızca $f_s = \frac{1}{O(n^2)+1}$ dir ve yeterince büyük bir n değeri için $f_s \ll f_p$ olduğu düşünüldüğünde, hem Amdahl Yasası (Bkz. denklem (3.1)), hem de Gustafson Yasası'na (Bkz. denklem (3.2)) göre $S(p) \cong p$ elde edilir. Yani, algoritmanın başarabileceği hızlanma, mümkün en iyi teorik üst sınıra çok yakındır.

4.2. Paralel α Seviye Tarama Algoritması

Yoğunluk tabanlı algoritmaların temsilcisi DBSCAN algoritmasının bulanık bir akrabası olan FJP algoritması (Bkz. Bölüm 2.3.3), herhangi bir girdi parametresine ihtiyaç duymayan, sağlam (robust) bir algoritmadır. İlk önerildiği haliyle $O(n^4)$ gibi kümeleme için çok yüksek bir karmaşıklığa sahip olan FJP algoritması, modifiye edilerek $O(n^3 \log n)$ zamanda çalışan MFJP algoritması (Ulutagay and Nasibov, 2013) geliştirilmiştir. Yine de bu iki algoritma, orta ve büyük veri setleri için kullanılabilir değildir. Bu tez çalışması dahilinde, FJP için optimal zaman karmaşıklığı olan $O(n^2)$ karmaşıklığa sahip OFJP algoritması önerilmiştir (Nasibov et al., basımda).

Paralel α seviye belirleme algoritmasının tanıtılabilmesi için, öncelikle FJP algoritmasının incelenmesi gerekir. FJP ve türevi algoritmaların temelini oluşturan bazı bulanık mantık kavramları Tanım 4.1-4.8 ile verilmiştir. Tanımlarda $d(x,y)$ ile ifade edilen uzaklık, (2.10) denklemi ile verilen Öklid uzaklığıdır.

Tanım 4.1. (*Konik bulanık nokta*) Bir konik bulanık nokta $P = (p, r)$ $x, p \in X$ olmak üzere, üyelik fonksiyonu (4.2) ile verilen bir bulanık kümedir (Bkz. Tanım 2.1).

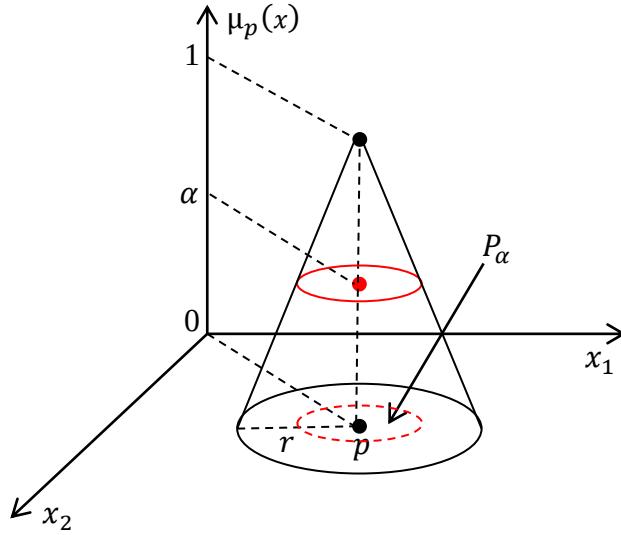
$$\mu_p(x) = \begin{cases} 1 - \frac{d(x,p)}{r}, & d(x,p) \leq r \\ 0, & \text{aksi halde} \end{cases} \quad (4.2)$$

Burada p , P konik bulanık noktasının merkezidir ve r , P noktasının destek kümesi $\text{supp } P = \{x \in X \mid \mu_p(x) > 0\}$ nin yarıçapıdır.

Tanım 4.2. (*α -seviye kümesi*) $\alpha \in [0,1]$ için bir P bulanık kümesinin α -seviye kümesi,

$$P_\alpha = \{x \in X \mid \mu_p(x) \geq \alpha\} \quad (4.3)$$

şeklinde tanımlanır. Dikkate değerdir ki P_α bir bulanık küme değil, bir klasik kümedir ve P kümesinin üyelik derecesi α dan küçük olmayan elemanlarından oluşur.



Şekil 4.1. Konik bulanık nokta $P = (p, r)$, $x, p \in \mathbb{R}^2$ ve α -seviye kümesi.

Tanım 4.3. (*Bulanık komşuluk bağıntısı*) $p_1, p_2 \in X$ noktaları sırasıyla P_1 ve P_2 konik bulanık noktalarının merkezleri olmak üzere, bulanık komşuluk bağıntısı $T: X \times X \rightarrow [0, 1]$ şöyle tanımlıdır:

$$T(P_1, P_2) = 1 - \frac{d(p_1, p_2)}{2r}. \quad (4.4)$$

FJP yöntemlerinde, tüm noktaların birbirlerine belirli bir dereceden komşu olabilmeleri için, r şöyle seçilmiştir: $r^* = \frac{\max d(p_i, p_j)}{2} = \frac{d_{max}}{2}$, $i, j = \overline{1, n}$. $r = r^*$ için bulanık komşuluk bağıntısı (4.5) denkleminde dönüşür.

$$T(P_1, P_2) = 1 - \frac{d(p_1, p_2)}{d_{max}}. \quad (4.5)$$

Tanım 4.4. (α -komşuluk) α -seviye kümesine benzer şekilde, bir P noktasının α -komşuluğu, bulanık komşuluk bağıntısı değeri bir α değerinden büyük olan noktaların kümesidir. Yani,

$$T(P_1, P_2) \geq \alpha, \quad \alpha \in (0, 1] \quad (4.6)$$

ise, P_1 ile P_2 birbirinin α -komşusudur ve $P_1 \sim_\alpha P_2$ ile gösterilir.

Tanım 4.5. (*Max-min bileşim*) $T: X \times X \rightarrow [0, 1]$ bir bulanık komşuluk bağıntısı olsun. Elemanları (4.7) denklemi ile hesaplanan $T' = (t'_{ij})_{n \times n}$ matrisi, $T = (t_{ij})_{n \times n}$ bağıntı matrisinin bir max-min bileşimidir ve $T \circ T$ ile gösterilir.

$$t'_{ij} = \max_k \min\{t_{ik}, t_{kj}\}, \quad i, j = 1, \dots, n. \quad (4.7)$$

Tanım 4.6. (*Max-min geçişli kapanış*) $T: X \times X \rightarrow [0, 1]$ bir bulanık komşuluk bağıntısı olsun. $T^k = T \circ T^{k-1}$, $k \in \mathbb{Z}$, $k \geq 2$ olmak üzere, (4.8) ile tanımlı $\hat{T}$ bağıntısına, T bağıntısının max-min geçişli kapanışı denir.

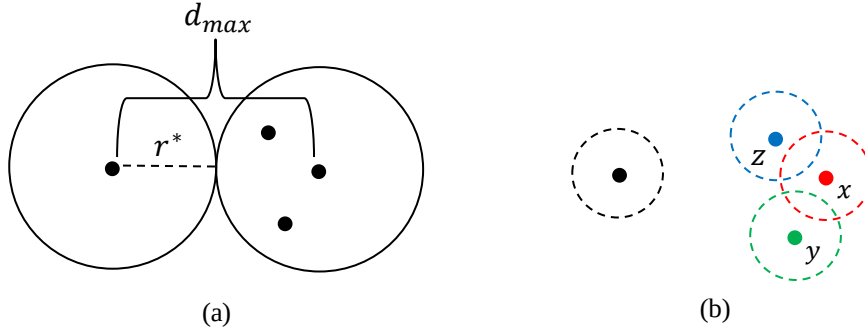
$$\hat{T} = T \cup T^2 \cup \dots \cup T^n \cup T^{n+1} \cup \dots \quad (4.8)$$

Tanım 4.7. (*Bulanık α -bağlantılı noktalar*) Eğer P_1 ve P_2 konik bulanık noktaları arasında (4.9) gibi bir α -komşular dizisi varsa, P_1 ve P_2 noktaları bulanık α -bağlantılıdır.

$$P_1 \sim_\alpha Y_1, Y_1 \sim_\alpha Y_2, \dots, Y_{k-1} \sim_\alpha Y_k, Y_k \sim_\alpha P_2, \quad k \in \mathbb{Z}, \quad k \geq 0. \quad (4.9)$$

Nasibov ve Ulutagay (2007) P_1 ve P_2 noktalarının ancak ve ancak (4.10) sağlandığında bulanık α -bağlantılı olduğunu göstermiştir.

$$\hat{T}(P_1, P_2) \geq \alpha. \quad (4.10)$$



Şekil 4.2. FJP yöntemine ait bulanık kavramlar. (a) r^* yarıçaplı konik bulanık noktalar. (b) Konik bulanık noktaların bir α seviyesi. $X = (x, r^*)$, $Y = (y, r^*)$ ve $Z = (z, r^*)$ α bağlantılı noktalardır. Y ve Z noktaları, X noktasının α komşularıdır. Y ile Z α komşu değildir.

Kümelenecek veri seti $X = \{x_1, \dots, x_n\}$ olmak üzere, Tanım 4.1–4.8 ile verilen kavramlar ışığında, FJP algoritmasının bir kaba kodu, Algoritma 4.3 ile verilmiştir.

Algoritma 4.3. (Bulanık bağlantılı noktalar: FJP)

Adım.1: Uzaklıkları hesapla $D: d_{ij} = d(x_i, x_j)$, $i, j = 1, \dots, n$;

$d_{max} = \max d_{ij}$, $i, j = 1, \dots, n$;

Adım.2: Bulanık komşulukları hesapla $T: t_{ij} = 1 - \frac{d_{ij}}{d_{max}}$, $i, j = 1, \dots, n$;

Adım.3: Geçişli kapanış matrisini hesapla $\hat{T}$;

Adım.4: $\hat{T} = (\hat{t}_{ij})_{n \times n}$ matrisinin elemanlarından oluşan, büyükten küçüğe sıralı $V = (a_i)_{i=1}^{n^2}$ dizisini oluştur;

$\Delta\alpha_i = \alpha_i - \alpha_{i+1}$, $i = 1, \dots, n^2 - 1$; $z = \arg \max_{i=1, \dots, n^2-1} \Delta\alpha_i$;

Adım.5: $\hat{T}$ geçişli kapanışının α_z -seviye kümesini hesapla ve kümelemeyi elde etmek için kümenin bağlantılı bileşenlerini belirle.

Algoritma 4.3, FJP algoritmasının ilk önerildiği hali değildir. Aslında, FJP tabanlı algoritmaların genel bir şeması niteliğindedir. FJP tabanlı bir algoritmanın en yavaş adımı, Algoritma 4.3'ün 4. adımı olan geçişli kapanışın hesaplanmasıdır. Doğrudan bir uygulamada, bulanık komşuluk matrisi n defa çarpılır. Bu durumda algoritma $O(n^4)$ gibi kümeleme işi için hayli yüksek bir zaman alır. MFJP algoritmasında, geçişli kapanışın hesaplanması için $\log_2 n$ defa çarpım yapmanın yeterli olduğu önermesine dayanarak, karmaşıklık $O(n^3 \log_2 n)$ a indirilmiştir. OFJP algoritmasında ise, geçişli kapanışın hesaplanması için Lee (2001)

tarafından önerilen, $O(n^2)$ zaman alan optimal zamanlı bir prosedür kullanılmaktadır. Bu prosedürün her adımı bir önceki adıma bağlı olduğu için, paralelleştirmeye uygun değildir. FJP algoritmasının bu en yavaş adımı paralelleştirilmeden, algoritmanın paralel bir versiyonunun önerilmesi de faydasız bir çalışma olacaktır.

Geçişli kapanış matrisinin elemanlarının büyükten küçüğe sıralı değerlerinden oluşan dizi $V = (a_i)_{i=1}^{n^2}$ olmak üzere, eşit olmayan α_i değerleri için elde edilen kümelemeler farklı, $[\alpha_{i+1}, \alpha_i)$ aralığındaki α değerleri için elde edilen kümelemeler aynı olur (Ulutagay and Nasibov, 2013). Bu özellik kullanılarak hiyerarşik kümeleme yöntemleri gibi bir dizi farklı kümeleme elde edilebilir. Dikkate değer bir durum, $\alpha = 0$ olduğunda tüm noktaların α bulanık bağlantılı olduğu, yani bir tekil kümeye ait olduklarıdır. Benzer şekilde, $\alpha = 1$ için, bir nokta yalnızca kendisine α bulanık bağlantılıdır ve n adet küme oluşur.

Algoritma 4.3'ün 4. adımında sonuç kümelemesi için V dizisinin en büyük aralığının seçilmesi ile, kümeler arası uzaklık maksimize edilmektedir. Dolayısıyla FJP algoritmasında α_z bulunarak, bu anlamda optimal bir kümeleme elde edilmektedir. α_z değerini bulmak için geçişli kapanışı hesaplamak yerine, belirli bir $\Delta\alpha$ birim farkıyla $[0,1]$ aralığında bir tarama yapılabilir. $\Delta\alpha$ yeterince küçük seçilirse, FJP algoritmasındaki gibi optimal bir α_z değeri elde edileceği açıktır. Bu sezgisel seviye tarama fikriyle geliştirilen α Scan algoritmasının kaba kodu, girdi parametresi $\Delta\alpha$ ve kümelenmek istenen veri seti $X = \{x_1, \dots, x_n\}$ için Algoritma 4.4 ile verilmiştir.

Algoritma 4.4. (α Seviye tarama: α Scan)

Adım.1: Uzaklıkları hesapla $D: d_{ij} = d(x_i, x_j), i, j = 1, \dots, n;$

$$d_{max} = \max d_{ij}, i, j = 1, \dots, n;$$

Adım.2: Bulanık komşulukları hesapla $T: t_{ij} = 1 - \frac{d_{ij}}{d_{max}}, i, j = 1, \dots, n;$

Adım.3: Güncel α seviyesi $\alpha_c = 1,$

Son farklı kümelemenin küme sayısı $k_l = 0,$

Güncel kümelemenin tekrar sayısı $r_c = 0,$

En çok tekrar eden kümelemenin tekrar sayısı $r_{max} = 0,$

Elde edilen en iyi kümeleme $S^* = \emptyset;$

Adım.4: T bulanık komşuluğunun α_c -seviye kümesini hesapla ve güncel kümelemeyi elde etmek için kümenin bağlantılı bileşenlerini belirle;

k = Güncel kümelemenin küme sayısı ve S = Kümeleme sonucu;

Adım.5: Eğer $k_l = k$ ise kümeleme aynı kalmıştır, $r_c = r_c + 1;$

Aksi halde,

Eğer $r_{max} < r_c$ ise bu kümeleme o ana kadarki en iyi kümelemedir,

$$r_{max} = r_c, \alpha_z = \alpha_c + \Delta\alpha, S^* = S;$$

$$r_c = 1, k_l = k;$$

Adım.6: $\alpha_c = \alpha_c - \Delta\alpha;$

Eğer $\alpha_c > 0$ ve $k > 1$ ise, Adım.4'e git;

Aksi halde, sonuç kümelemesi α_z -seviye kümesinden elde edilen S^* dir.

α Scan algoritmasının 4. ve 6. adımları arasındaki ana döngü en fazla $r = \left\lfloor \frac{1}{\Delta\alpha} \right\rfloor$ tekrar yapar ve $\Delta\alpha$ ne kadar küçük seçilirse, algoritma o kadar yavaş çalışır. Bu döngünün hızlandırılması, büyük veri setlerinde küçük $\Delta\alpha$ değerlerinin seçilerek en iyi kümelemenin elde edilmesine olanak sağlayacaktır.

α Scan algoritmasının paralelleştirilmesinde odak noktası, 4. ve 6. adımlar arasındaki döngüdür. Seri algoritmada, her bir kümelemenin bir öncekiyle karşılaştırılması nedeniyle, döngünün iterasyonları birbirlerine bağımlıdır. Bu bağımlılıktan kurtulmak için, her kümelemenin bilgisi ayrı ayrı tutularak en sonda

bunlar üzerinden bir değerlendirme yapılabilir. Dahası, değerlendirme için kümeleme bilgilerini saklamak yerine, yalnızca küme sayıları saklanabilir. Çünkü, her farklı kümelemenin küme sayısı da farklıdır ve eşit küme sayısına sahip kümelemeler aynıdır. Küme sayılarının saklandığı kümenin modu, yani en çok tekrar eden değeri, en iyi kümelemedeki küme sayısıdır ve buna karşılık gelen kümeleme sonuç kümelemesi olarak belirlenir. Yalnızca küme sayıları saklanarak belirli bir zaman karşılığında bellek alanında tasarruf edilir. Paralel α seviye tarama algoritmasının Algoritma 4.4 ile verilen kaba kodunda, uygulamada da olduğu gibi, bu bellek tasarruflu yöntem kullanılmıştır. Öte yandan bu yöntem ile kaybedilen zaman, paralel hızlanmanın teorik performans değerlendirmesinde bir dezavantaj yaratır. Bu konu, Algoritma 4.4 verildikten sonra tekrar tartışılacaktır.

α Scan algoritmasının paralelleştirmeye müsait diğer bir parçası, ilk iki adımında gerçekleşen bulanık komşulukların hesaplanmasıdır. Uzaklık matrisinin ve bulanık komşuluk matrisinin satırları, iş parçacıkları arasında paylaştırılarak paralellik sağlanır. Bu matrisler simetrik olduğundan, yalnızca üst üçgenlerinin hesaplanması yeterlidir. Uygulama seviyesindeki bu detay, sadelik ve anlaşılabilirlik adına, önceki seri algoritmalarda göz ardı edilmiştir. Fakat paralelleştirme aşamasında bu durum önem kazanır. Daha yüksek indisli satırlar için daha az işlem gerekecektir ve eğer satırlar bloklar halinde iş parçacıklarına dağıtılırsa, iş yükü dengesi sağlanamaz. Daha dengeli bir paylaşım için, satırlar iş parçacıklarına teker teker dağıtılabilir. Aslında, bir baştan bir sondan satır seçilerek biraz daha iyi bir paralelleştirme sağlanabilir. Fakat bu, zaten görece karmaşık olan algoritmayı daha da karmaşıktıracaktır. Üstelik iş parçacığı platformlarında, bir döngünün kaçır kaçır paylaştırılacağı ilgili fonksiyona veya derleyici komutuna bir parametre olarak kolayca geçirilebilir. Satırları baştan ve sondan seçmek için ise, programda fazladan bir iş yapılması gerekir.

Girdi parametresi $\Delta\alpha$ ve kümelenmek istenen veri seti $X = \{x_1, \dots, x_n\}$ için, paralel seviye tarama algoritmasının kaba kodu Algoritma 4.5 ile verilmiştir.

Algoritma 4.5. (Paralel α seviye tarama: $P\text{-}\alpha\text{Scan}$)

/* Paralel bölgenin başı */

Adım.1: p adet iş parçacığı yarat ve indisle: $t = 0, \dots, p - 1$; $I = \{i \mid i = t + wp, 0 \leq w \leq n, w \in \mathbb{Z}\}$ olmak üzere,Uzaklıkları hesapla D : $d_{ij} = d_{ji} = d(x_i, x_j)$, $i \in I, j = i + 1, \dots, n$; $d_{max}^t = \max d_{ij}$, $i \in I, j = 1, \dots, n$;

İş parçalarına özel en büyük uzaklıkları seri olarak indirge (reduction):

 $d_{max} = \max d_{max}^j$, $j = 0, \dots, p - 1$;**Adım.2:** $I = \{i \mid i = t + wp, 0 \leq w \leq n, w \in \mathbb{Z}\}$ olmak üzere,Bulanık komşulukları hesapla T : $t_{ij} = 1 - \frac{d_{ij}}{d_{max}}$, $i \in I, j = 1, \dots, n$;**Adım.3:** İş parçacıklarının yerel iterasyon indisi $l = t + 1$,İş parçacıklarının yerel güncel α seviyesi $\alpha_c^t = 1 - t\Delta\alpha$;**Adım.4:** T bulanık komşuluğunun α_c^t -seviye kümesini hesapla ve güncel kümelemeyi elde etmek için kümenin bağlantılı bileşenlerini belirle; k_l = Güncel kümelemenin sayısı;/* $r = \left\lfloor \frac{1}{\Delta\alpha} \right\rfloor$ olmak üzere $(k_i)_{i=1}^r$ ortak bellekte bir dizidir */**Adım.5:** $\alpha_c^t = \alpha_c^t - p\Delta\alpha$, $l = l + p$;Eğer $\alpha_c^t > 0$ ve $k_l > 1$ ise, Adım.4'e git;

Aksi halde, ilgili iş parçacığının görevi bitmiştir;

Tümünün görevleri bittiğinde iş parçacıklarını sonlandır;

/* Paralel bölgenin sonu */

Adım.6: $r = \left\lfloor \frac{1}{\Delta\alpha} \right\rfloor$ olmak üzere, $K = \{k \mid k > 1, k \in (k_i)_{i=1}^r\}$ kümesinin modunu, yani en çok tekrar eden değerini, k^* değişkenine ve $(k_i)_{i=1}^r$ dizisinde bu mod değerini saklayan en küçük indisi l^* değişkenine ata; $\alpha_z = 1 - l^*\Delta\alpha$;**Adım.7:** T bulanık komşuluğunun α_z -seviye kümesini hesapla ve sonuç kümelemesini elde etmek için kümenin bağlantılı bileşenlerini belirle.

1. ve 5. adımlar arasındaki paralel bölgede, her bir iş parçacığı kendine ait t indisine bağlı olarak eş zamanlı işlem yapar. t , l , α_c^t değişkenleri ile ilk iki adımdaki i ve j döngü sayaçları iş parçacıklarının yerel değişkenleridir. Küme sayıları ortak bellekte saklanıyor olsa da, iş parçaları, indislerine bağlı olarak farklı k_l değerlerini hesapladığından, eş zamanlı yazma durumu oluşmaz.

Paralel α seviye tarama algoritmasında, uzaklık matrisinin ve bu matristen de bulanık komşuluk matrisinin hesaplandığı ilk iki adım, m boyutlu n adet veri için $O(mn^2)$ zaman alır. İlk adımdaki indirgeme işlemi seri olarak gerçekleşse de, işlemci sayısı p olmak üzere yalnızca $O(p)$ zaman alacağından, ihmal edilebilir. Dolayısıyla, bulanık komşuluk matrisinin tamamen paralel olarak hesaplandığı kabul edilebilir. Seri algoritmadaki 4. ve 6. adımlar arasındaki ana döngü, paralel algoritmada 4. ve 5. adımlarda gerçekleşir. Bu döngünün zamanını, bağlantılı bileşenlerin hesaplanması belirler. Komşuluk matrisi üzerinde gerçekleştirilecek doğrudan bir uygulamada, n defa komşuluk sorgusu yapılır. Her bir komşuluk sorgusu da n veri üzerinde gerçekleştirileceğinden, karmaşıklığı $O(n^2)$ olur. Ana döngü en fazla $r = \left\lfloor \frac{1}{\Delta\alpha} \right\rfloor$ tekrar yapar. Bu durumda ana döngü $O(rn^2)$ zaman alır. Bağlantılı bileşenler, çizge dolaşma algoritmaları veya ayırık küme veri yapısı kullanılarak daha hızlı belirlenebilse de, n cinsinden zaman karmaşıklığı aynıdır. Algoritmanın seri çalıştığı 6. adım $O(r)$ zamanda çalışır. Son adımdaki bağlantılı bileşen belirleme işlemi ise yine $O(n^2)$ zaman alır. $O(rn)$ alan maliyeti ile her seviyedeki kümelemeler saklanarak seri çalışan bu $O(n^2)$ karmaşıklığındaki adımdan kurtulunabilir. Sonuçta, algoritmanın paralel kısmı $O(mn^2 + rn^2)$ zamanda çalışır. Seri kısmı ise, Algoritma 4.4'te verildiği gibi yalnızca küme sayıları saklanırsa $O(n^2 + r)$, kümelemeler de saklanırsa $O(r)$ zaman alır. Bu karmaşıklıklar dikkate alındığında, m , n ve r değerleri arttıkça $\frac{f_p}{f_s}$ de hızla artacaktır. Buna bağlı olarak, α seviye tarama algoritmasının problem boyutu büyüdükçe doğrusala yaklaşan bir hızlanma sağlayacağı öngörülmektedir.

4.3. Geliştirilen Programlar

Geliştirme ve hesaplamalar için kullanılan bilgisayar, üzerinde Linux Ubuntu 12.04 LTS Desktop işletim sistemi çalışan, 8 adet Intel Xeon 2.53 GHz işlemcisi ve 6 GB DDR3 RAM belleği olan bir Dell Precision T5500 iş istasyonudur. Tüm programlar ve kümeleme kütüphanesi C dilinde yazılmıştır. Derleyici olarak, OpenMP 3.1'i destekleyen GCC 4.7.3 kullanılmıştır (<https://gcc.gnu.org/wiki/openmp>). OpenMP, iş parçacığı modelinde standartlaşmış bir uygulama programlama arayüzüdür. Geliştirme sürecinde

(OpenMP version 3.1, 2011; Barney, 2014b) kaynaklarından yararlanılmıştır. C ve OpenMP kullanımı, programların taşınabilir olmasını sağlamıştır. Ayrıca bazı kümeleme sonuçlarının görselleştirilmesi için, Windows 8 işletim sistemine sahip bir kişisel bilgisayarda, C# ile bir program daha yazılmıştır.

Tüm programlarda kullanılan fonksiyonlar, “clusterlib” isimli kütüphanede toplanmıştır. Kaynak kodların şişmesini engellemek için, fonksiyon bildirimleri “clusterlib.h” dosyasına, fonksiyon ve veri tipi tanımlamaları “clusterlib.c” dosyasında yerleştirilmiştir. Kütüphanenin kullanılması için, “clusterlib.h” kaynak koda eklenmeli ve kod “clusterlib.c” ile beraber derlenmelidir. Bu iki kütüphane dosyasının içeriği Ek 1’de verilmiştir. Ek 2-6 sırasıyla k -ortalamlar, seri global k -ortalamlar, seri α seviye tarama, paralel global k -ortalamlar ve paralel α seviye tarama programlarının kaynak kodlarını içermektedir.

Kaynak kodların derlenebilmesi için, OpenMP 3.1 sürümünü destekleyen bir derleyiciye ihtiyaç vardır. Daha eski sürümler, kodlardaki bazı derleyici komutlarını desteklemeyebilir. Kodlar derlenirken -O3 optimizasyon parametresi kullanılmıştır. Örneğin GCC ile paralel global k -ortalamlar programı aşağıdaki gibi derlenir:

```
gcc clusterlib.c omp_gkm.c -lm -fopenmp -O3 -o omp_gkm
```

Global k -ortalamlar programı ilk argüman olarak okunacak dosyanın yolunu alır. İkinci argüman ise maksimum küme sayısıdır. Paralel versiyonda, iş parçacığı sayısının belirtildiği 3. bir argüman daha gereklidir. Seri ve paralel α seviye tarama programlarının ilk iki argümanı, dosya yolu ve $\Delta\alpha$ parametresidir. Paralel versiyonda, iş parçacığı sayısını 3. argüman olarak alınır. Yukarıdaki gibi derlenen paralel GKM programı, şöyle çalıştırılabilir:

```
./omp_gkm input.data 5 8
```

Veri sayısı n ve veri boyutu m olmak üzere, programların okuduğu metin dosyalarının formatı aşağıdaki gibidir.

$n \ m$

$x_{1,1} \ x_{1,2} \ \dots \ x_{1,m}$

...

$x_{n,1} \ x_{n,2} \ \dots \ x_{n,m}$

Programlarla ilgili bazı ilave açıklamalar, ekler bölümünde verilen kaynak kodların içerisinde yorum olarak yer almaktadır.

4.4. Hesaplama Denemeleri

Önerilen paralel algoritmaların programlarının hesaplama denemeleri için UCI Machine Learning Repository (<http://archive.ics.uci.edu/ml>) veri ambarından alınan, çeşitli boyutlarda 4 adet gerçek veri seti ve görselleştirmeye uygun bazı yapay veri setleri kullanılmıştır. Verilerin tüm özellikleri sayısalıdır. Veri setlerine dair bilgiler Çizelge 4.1’de yer almaktadır.

Çizelge 4.1. Veri setlerine ait bilgiler

Veri seti	Veri sayısı	Özellik sayısı
IRIS	150	4
YEAST	1484	4
PAGEBLOCK	5473	10
PENDIGIT	10992	16
YAPAY1	500	2
YAPAY2	2000	2
YAPAY3	4000	2
YAPAY4	7500	2

Kullanılan veri setlerinin hepsi, veri sayısı n ve özellik sayısı m olmak üzere (4.11) formülüyle normalize edilmiştir.

$$x_j^{max} = \max_{i=1,\dots,n} x_{ij}, \quad x_j^{min} = \min_{i=1,\dots,n} x_{ij}, \quad j = 1, \dots, m,$$

$$x'_{ij} = \frac{x_{ij} - x_j^{min}}{x_j^{max} - x_j^{min}}. \quad (4.11)$$

4.4.1. P-GKM deneme sonuçları

Sonuçları Çizelge 4.2-4.5’te sunulan, P-GKM programının hesaplama denemeleri sırasıyla IRIS, YEAST, PAGEBLOCK ve PENDIGIT veri setleri üzerinde gerçekleştirilmiştir. Çizelgelerde k küme sayısını, f_{km} ve f_{gkm} sırasıyla k -ortalamalar ve GKM algoritmalarından elde edilen amaç fonksiyonunun değerini, t saniye cinsinden süreyi ve p işlemci sayısını belirtmektedir. k -ortalamalar algoritmasının programı, veri noktaları arasından birbirinden farklı k tanesini başlangıç merkezi olarak seçecek şekilde yazılmıştır. Her farklı k değeri için program 10 defa çalıştırılıp en iyi sonuç kaydedilmiştir.

Çizelge 4.2. IRIS veri seti üzerinde P-GKM denemeleri

k	f_{km}	f_{gkm}	t (saniye)				
			GKM	P-GKM			
				$p = 2$	$p = 4$	$p = 6$	$p = 8$
2	0.248	0.248	0.00	0.00	0.00	0.00	0.00
3	0.195	0.195	0.00	0.01	0.01	0.00	0.00
4	0.174	0.174	0.01	0.01	0.01	0.00	0.00
5	0.163	0.157	0.02	0.02	0.01	0.01	0.01
6	0.153	0.146	0.03	0.02	0.01	0.01	0.01
7	0.151	0.137	0.04	0.03	0.01	0.01	0.01
8	0.146	0.129	0.05	0.03	0.02	0.01	0.01
9	0.143	0.122	0.06	0.04	0.02	0.02	0.02
10	0.139	0.118	0.08	0.05	0.03	0.02	0.02
11	0.133	0.113	0.09	0.05	0.03	0.02	0.02
12	0.118	0.109	0.10	0.06	0.03	0.02	0.02
13	0.116	0.105	0.12	0.07	0.04	0.03	0.03
14	0.107	0.102	0.13	0.07	0.04	0.03	0.03
15	0.103	0.099	0.14	0.08	0.04	0.03	0.03

Çizelge 4.3. YEAST veri seti üzerinde P-GKM denemeleri

k	f_{km}	f_{gkm}	t (saniye)				
			GKM	P-GKM			
				$p = 2$	$p = 4$	$p = 6$	$p = 8$
2	0.227	0.227	1.38	0.71	0.36	0.24	0.18
3	0.207	0.207	3.12	1.65	0.83	0.56	0.42
4	0.194	0.194	5.97	3.32	1.67	1.12	0.84
5	0.183	0.183	10.94	5.96	3.08	2.10	1.55
6	0.176	0.176	16.80	9.05	4.71	3.19	2.39
7	0.171	0.170	22.32	12.07	6.25	4.23	3.32
8	0.168	0.166	28.31	15.24	7.98	5.38	4.04
9	0.162	0.162	35.87	19.25	10.12	6.80	5.16
10	0.159	0.158	43.58	23.25	12.33	8.29	6.25
11	0.157	0.155	50.82	27.09	14.32	9.62	7.22
12	0.157	0.152	58.80	31.36	16.55	11.18	8.42
13	0.152	0.150	67.28	35.86	18.83	13.27	9.63
14	0.150	0.148	74.82	39.87	20.99	14.22	10.70
15	0.150	0.146	82.72	44.02	23.32	15.72	11.83

Çizelge 4.4. PAGEBLOCK veri seti üzerinde P-GKM denemeleri

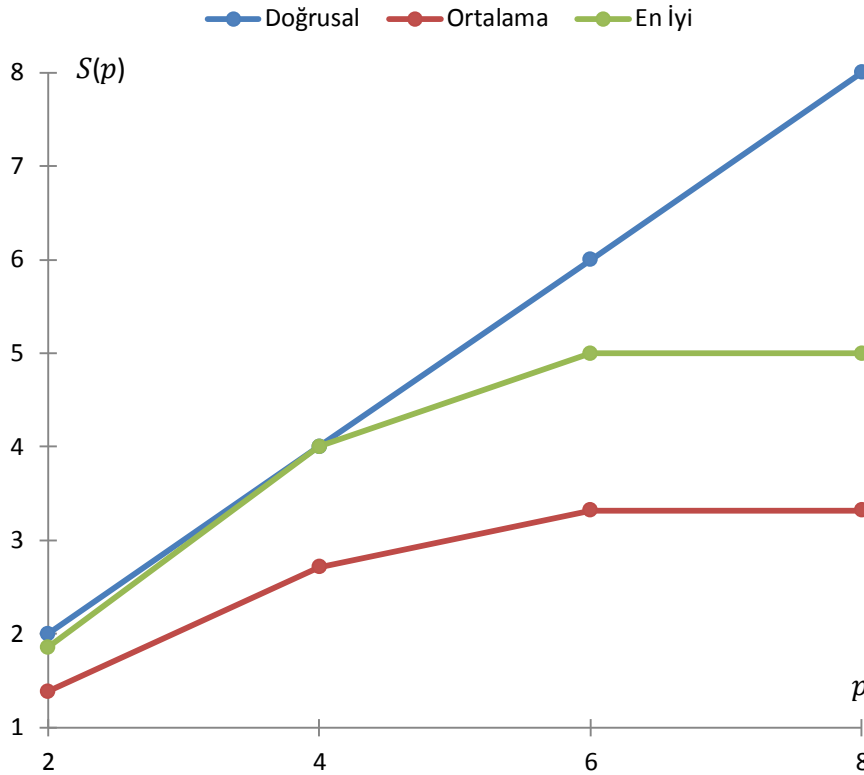
k	f_{km}	f_{gkm}	t (saniye)				
			GKM	P-GKM			
				$p = 2$	$p = 4$	$p = 6$	$p = 8$
2	0.236	0.236	36.02	18.93	9.83	6.90	5.18
3	0.201	0.201	85.97	44.07	23.56	16.28	12.25
4	0.176	0.176	141.64	74.67	39.28	27.72	21.54
5	0.166	0.164	245.51	129.76	67.27	48.10	36.73
6	0.154	0.154	355.27	186.84	97.35	68.12	52.12
7	0.151	0.145	495.38	260.07	133.01	94.60	72.87
8	0.146	0.138	661.85	349.27	180.38	125.00	95.37
9	0.135	0.132	831.39	434.00	229.32	157.51	120.81
10	0.132	0.128	1050.83	551.81	293.58	198.62	154.28
11	0.127	0.124	1347.87	723.36	393.61	269.11	205.46
12	0.123	0.121	1636.23	873.07	472.57	323.24	253.69
13	0.122	0.118	1917.26	1026.18	560.12	381.70	293.70
14	0.117	0.115	2237.83	1199.18	658.85	449.56	345.02
15	0.114	0.112	2260.31	1419.75	773.72	525.57	405.91

Çizelge 4.5. PENDIGIT veri seti üzerinde P-GKM denemeleri

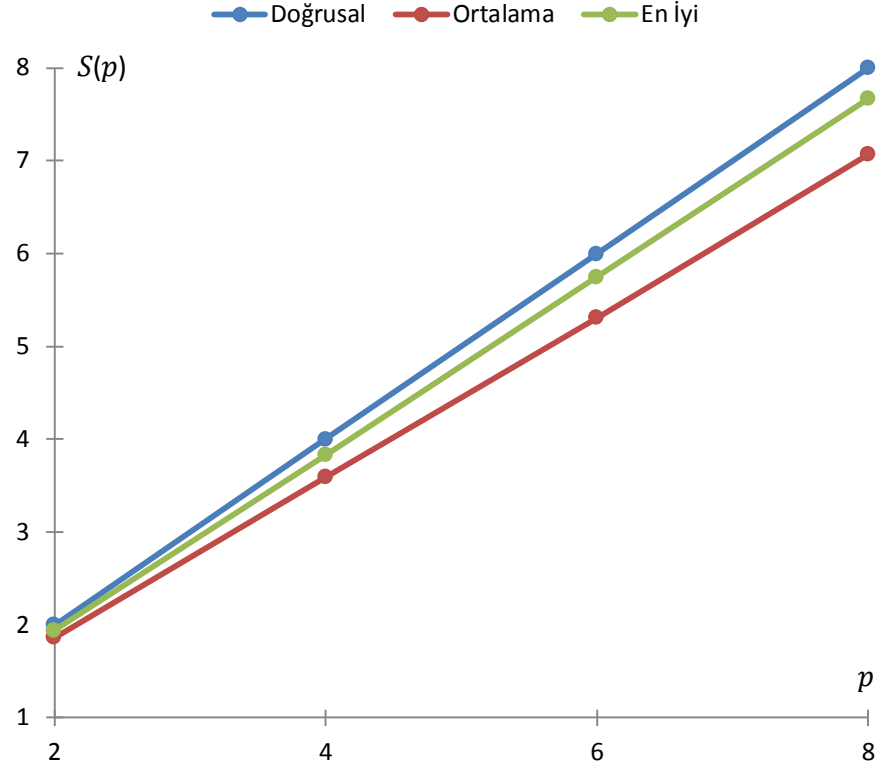
k	f_{km}	f_{gkm}	t (saniye)				
			GKM	P-GKM			
				$p = 2$	$p = 4$	$p = 6$	$p = 8$
2	1.051	1.051	127.04	63.17	33.56	22.37	16.95
3	0.932	0.932	324.96	159.74	83.94	56.38	42.61
4	0.873	0.849	683.69	340.11	177.763	118.95	89.66
5	0.794	0.793	1139.22	568.43	298.45	199.75	150.63
6	0.738	0.738	1711.42	855.43	448.66	300.93	226.24
7	0.698	0.698	2251.48	1124.67	590.72	394.94	297.54
8	0.674	0.674	2916.51	1455.23	764.75	511.02	384.76
9	0.661	0.652	3615.03	1804.75	948.02	634.14	477.18
10	0.639	0.630	4402.58	2199.50	1153.23	773.63	583.50
11	0.620	0.609	5331.05	2665.03	1396.72	938.26	708.07
12	0.605	0.592	6157.01	3079.61	1612.51	1085.15	817.508
13	0.590	0.578	7003.91	3606.42	1835.38	1234.93	929.98
14	0.590	0.567	8030.76	4021.38	2104.48	1414.28	1067.97
15	0.561	0.557	9489.86	4759.62	2495.23	1674.04	1265.31

Öncelikle kümeleme performansı açısından değerlendirme yapıldığında, global k -ortalamalar algoritmasının, tüm durumlarda en az k -ortalamalar algoritması kadar başarılı olduğu görülmektedir. k -ortalamalar algoritması 10 defa çalıştırılarak çoklu başlangıç seçimi yapılmış olmasına rağmen, özellikle küme sayısı arttığında global k -ortalamalar algoritmasının amaç fonksiyonu değerlerinin daha iyi olduğu gözlenmiştir.

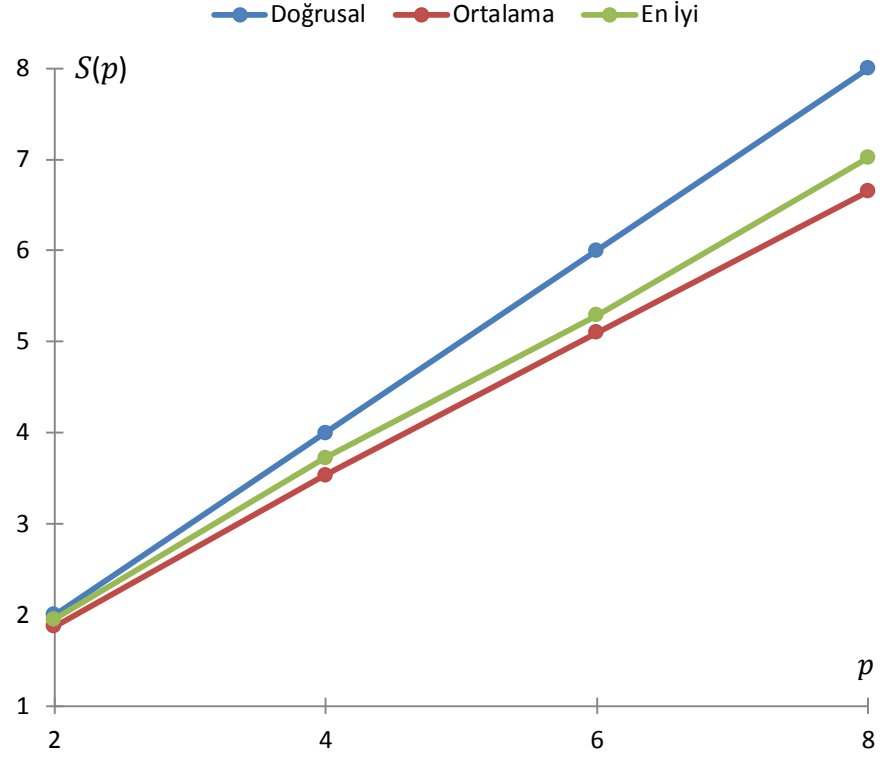
Paralel global k -ortalamalar ile elde edilen hızlanmayı değerlendirmek için, her bir veri seti üzerindeki denemelerde farklı k değerleri ile elde edilen hızlanmaları gösteren grafikler oluşturulmuştur. Şekil 4.3–4.6 ile verilen grafiklerde, P-GKM algoritması ile elde edilen ortalama hızlanma ve en iyi hızlanma gösterilmiştir. Yatay eksen iş parçacığı sayısı p , dikey eksen p 'ye bağlı hızlanma oranı $\frac{T_s}{T_p} = S(p)$ yer almaktadır. Doğrusal hızlanma çizgisi, referans olması için eklenmiştir.



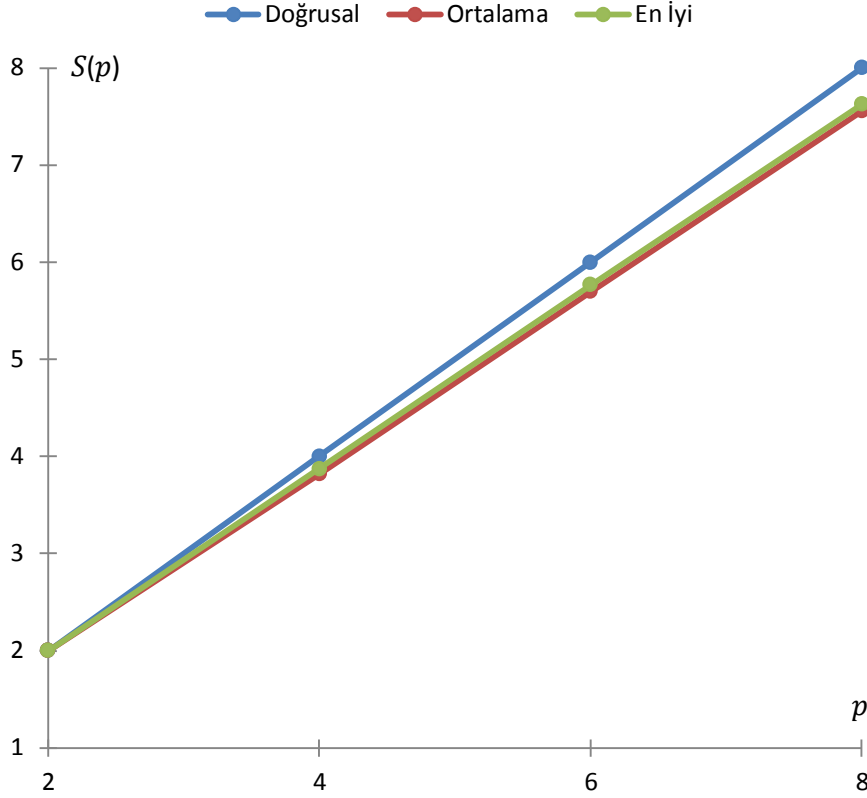
Şekil 4.3. IRIS veri seti üzerinde P-GKM ile elde edilen hızlanma.



Şekil 4.4. YEAST veri seti üzerinde P-GKM ile elde edilen hızlanma.



Şekil 4.5. PAGEBLOCK veri seti üzerinde P-GKM ile elde edilen hızlanma.

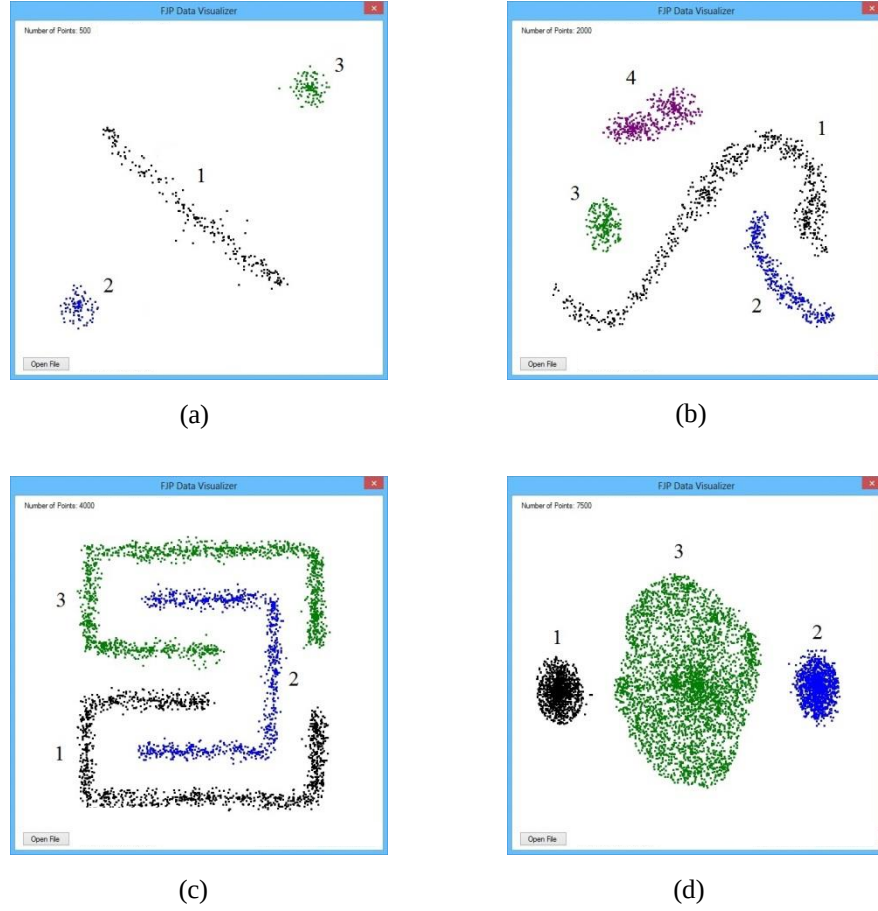


Şekil 4.6. PENDIGIT veri seti üzerinde P-GKM ile elde edilen hızlanma.

P-GKM algoritmasının hızlanma değerlerinde ilk dikkat çekici olan, veri ve özellik sayısı düşük olan IRIS veri seti için hızlanmanın belirli bir seviyeye kadar sağlanabilmiş olmasıdır. Ayrıca grafiklere yansımayan bir detay, küme sayısının düşük olduğu bazı durumlarda paralel programların seri programdan daha uzun sürmüş olmasıdır (Bkz. Çizelge 4.2). Ancak bu, P-GKM algoritmasına özgü bir durum değildir. Düşük işlem yükü olduğunda, paralelleştirme maliyeti nedeniyle paralel programların verimi sınırlı olur ve hatta negatif etki görülebilir. YEAST, PAGEBLOCK ve PENDIGIT veri setlerinde hızlanma iş parçacığı sayısı ile orantılı olarak artmaktadır. Özellikle işlem yükü arttıkça, teorik analizle uyumlu olarak doğrusala yakın hızlanma gözlenmiştir.

4.4.2. P- α Scan deneme sonuçları

α Scan algoritması için bir amaç fonksiyonu söz konusu olmadığından, kümeleme performansının değerlendirilmesi için 2 boyutlu yapay veri setleri üzerinde denemeler yapılarak, sonuçlar görselleştirilmiştir (Bkz. Şekil 4.7). Yoğunluk tabanlı kümeleme yöntemlerinin ortak özelliği olan düzensiz şekilli kümeleri belirleme avantajı da açıkça görülmektedir.



Şekil 4.7. α Scan ile kümeleme sonuçları. (a) YAPAY1 (b) YAPAY2 (c) YAPAY3 (d) YAPAY4

Yüksek boyutlu verilerde deneme yapabilmek için, görselleştirmeye uygun yapay veri setlerinin yanında PENDIGIT de kullanılmıştır. Çizelge 4.6–4.10’te sonuçları sunulan P- α Scan programının hesaplama denemeleri, sırasıyla YAPAY1, YAPAY2, YAPAY3, YAPAY4 ve PENDIGIT veri setleri üzerinde gerçekleştirilmiştir. Çizelgelerde k küme sayısını, $\Delta\alpha$ girdi paramatresini, t saniye cinsinden süreyi ve p işlemci sayısını belirtmektedir.

Çizelge 4.6. YAPAY1 veri seti üzerinde P- α Scan denemeleri

k	$\Delta\alpha$	t (saniye)				
		α Scan	P- α Scan			
			$p = 2$	$p = 4$	$p = 6$	$p = 8$
3	0.1	0.00	0.00	0.00	0.00	0.00
3	0.01	0.01	0.01	0.01	0.01	0.01
3	0.001	0.10	0.06	0.03	0.02	0.03
3	0.0001	0.97	0.49	0.25	0.17	0.24
3	0.00001	8.93	4.82	2.50	1.69	2.09

Çizelge 4.7. YAPAY2 veri seti üzerinde P- α Scan denemeleri

k	$\Delta\alpha$	t (saniye)				
		α Scan	P - α Scan			
			$p = 2$	$p = 4$	$p = 6$	$p = 8$
1674	0.1	0.06	0.05	0.04	0.03	0.03
4	0.01	0.11	0.07	0.04	0.03	0.03
4	0.001	0.51	0.28	0.15	0.10	0.08
4	0.0001	4.42	2.42	1.24	0.82	0.61
4	0.00001	45.49	22.31	12.00	7.92	6.00

Çizelge 4.8. YAPAY3 veri seti üzerinde P- α Scan denemeleri

k	$\Delta\alpha$	t (saniye)				
		α Scan	P - α Scan			
			$p = 2$	$p = 4$	$p = 6$	$p = 8$
3670	0.1	0.08	0.08	0.07	0.06	0.06
3	0.01	0.22	0.13	0.08	0.08	0.06
3	0.001	1.28	0.65	0.34	0.29	0.25
3	0.0001	10.43	6.02	3.03	2.18	1.74
3	0.00001	112.25	57.64	33.22	19.75	15.88

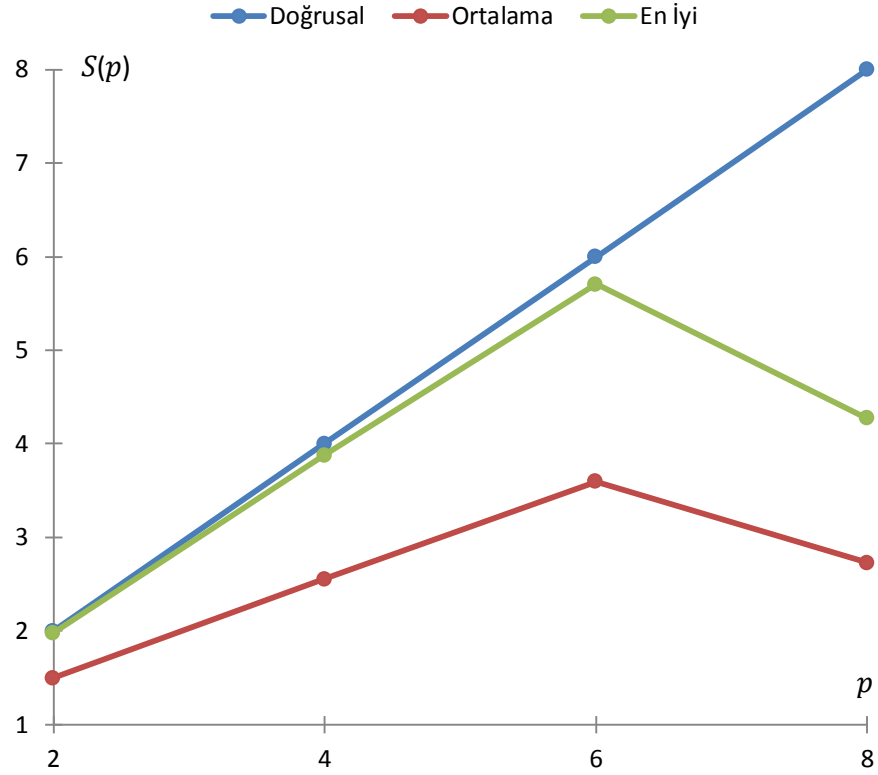
Çizelge 4.9. YAPAY4 veri seti üzerinde P- α Scan denemeleri

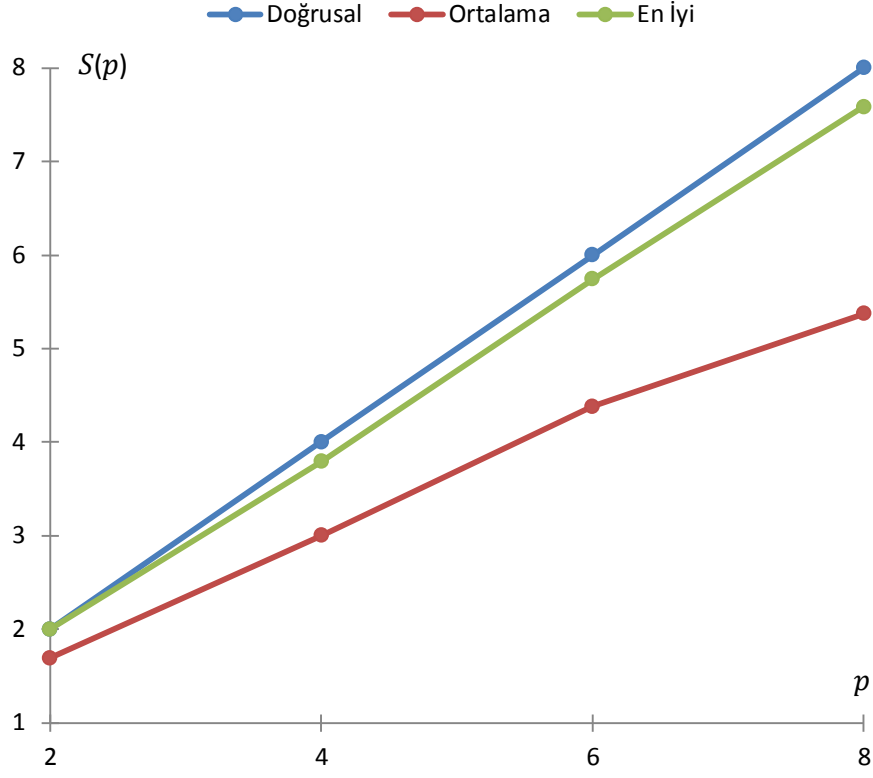
k	$\Delta\alpha$	t (saniye)				
		α Scan	P - α Scan			
			$p = 2$	$p = 4$	$p = 6$	$p = 8$
6135	0.1	1.34	0.97	0.79	0.57	0.48
3	0.01	1.90	1.66	0.94	0.69	0.66
3	0.001	6.36	3.51	2.52	2.00	1.48
3	0.0001	61.24	31.48	17.61	11.99	10.39
3	0.00001	572.91	307.95	167.49	121.30	88.50

Çizelge 4.10. PENDIGIT veri seti üzerinde P- α Scan denemeleri

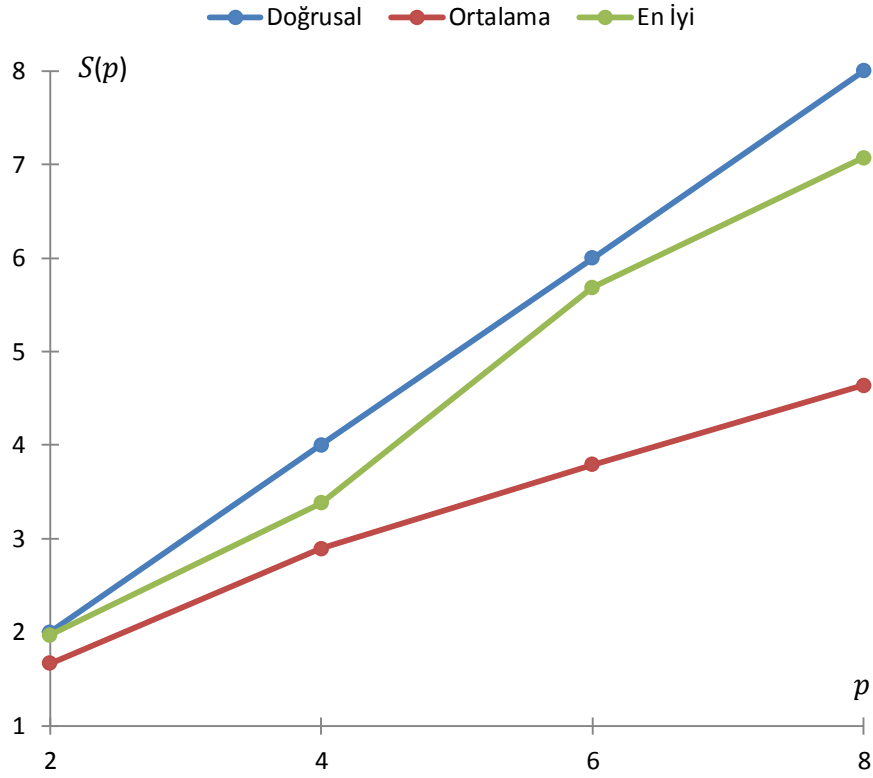
k	$\Delta\alpha$	t (saniye)				
		α Scan	P- α Scan			
			$p = 2$	$p = 4$	$p = 6$	$p = 8$
10992	0.1	5.36	4.09	2.90	2.17	1.75
4	0.01	10.25	6.82	4.34	3.56	3.02
4	0.001	56.37	32.97	20.02	17.57	14.15
4	0.0001	549.45	291.07	179.76	139.00	106.17
4	0.00001	5643.26	2910.34	1555.27	1150.98	1008.08

Paralel α Scan algoritması ile elde edilen hızlanmayı değerlendirmek için yine, her bir veri seti üzerindeki denemelerde farklı $\Delta\alpha$ değerleri ile elde edilen hızlanmaları gösteren grafikler oluşturulmuştur. Şekil 4.8–4.12 ile verilen grafiklerde, P- α Scan algoritması ile elde edilen ortalama hızlanma ve en iyi hızlanma gösterilmiştir. Yatay eksende iş parçacığı sayısı p , dikey eksen p 'ye bağlı hızlanma oranı $\frac{T_s}{T_p} = S(p)$ yer almaktadır. Doğrusal hızlanma çizgisi, referans olması için eklenmiştir.

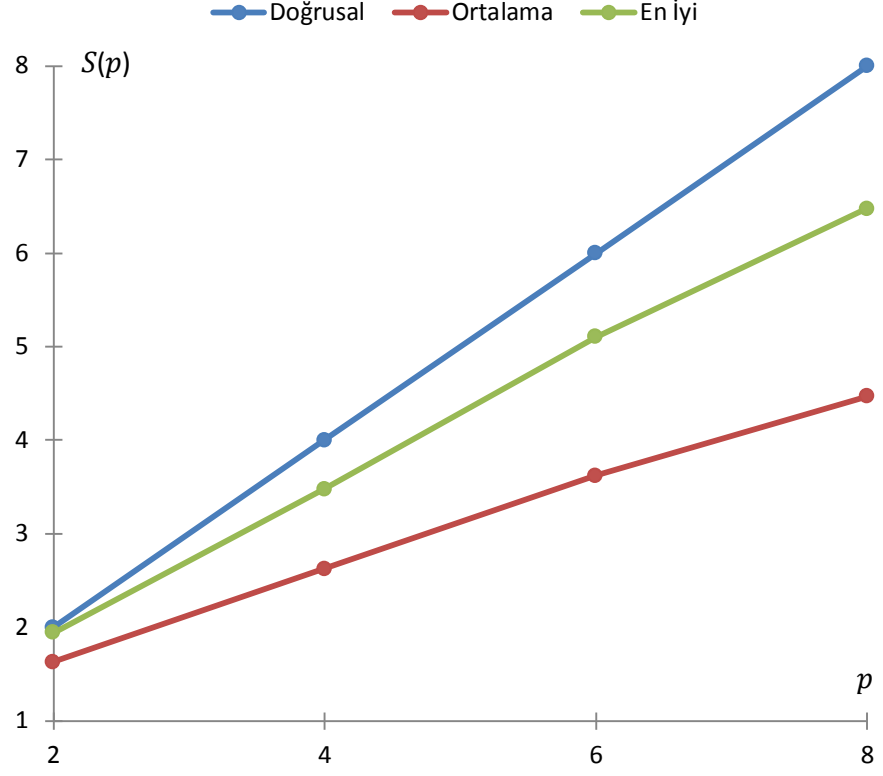
Şekil 4.8. YAPAY1 veri seti üzerinde P- α Scan ile elde edilen hızlanma.



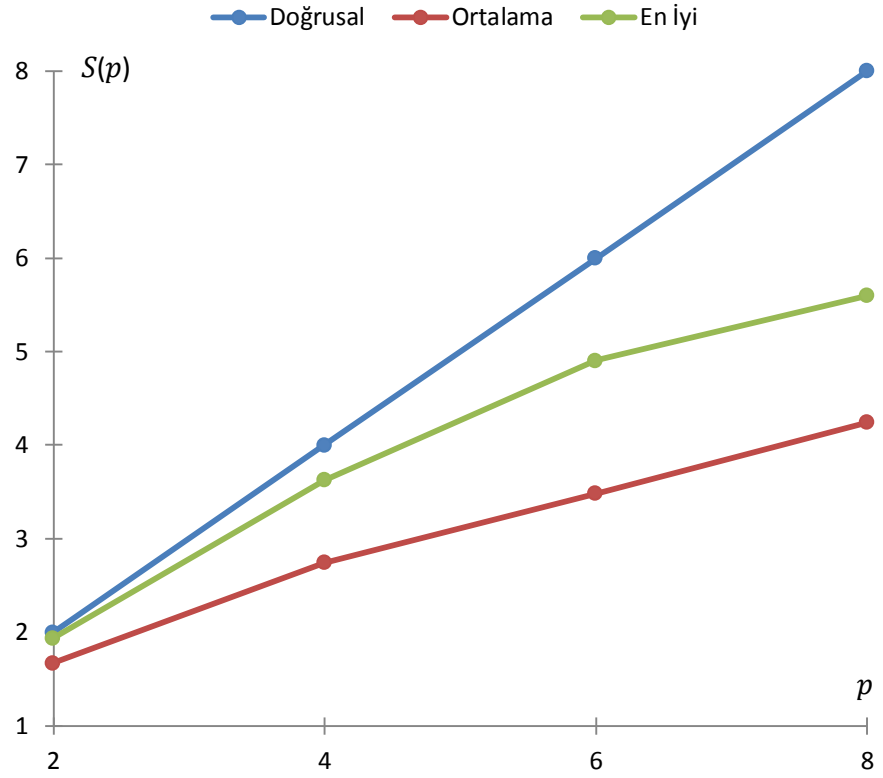
Şekil 4.9. YAPAY2 veri seti üzerinde P- α Scan ile elde edilen hızlanma.



Şekil 4.10. YAPAY3 veri seti üzerinde P- α Scan ile elde edilen hızlanma.



Şekil 4.11. YAPAY4 veri seti üzerinde P- α Scan ile elde edilen hızlanma



Şekil 4.12. PENDIGIT veri seti üzerinde P- α Scan ile elde edilen hızlanma

P- α Scan algoritmasıyla elde edilen hızlanmalarda, P-GKM denemelerinde de olduğu gibi, düşük işlem yükü sebebiyle verimsiz paralelleştirme olduğu durumlar gözlenmiştir (Bkz. Çizelge 4.6 ve 4.7). Aynı duruma paralel olarak Şekil 4.8'deki grafikte, iş parçacığı sayısı arttığında hızlanmada azalma olduğu görülmektedir. YAPAY3, YAPAY4 ve PENDIGIT veri setleri üzerindeki denemelerde, beklendiği gibi, ortalama hızlanma iş parçacığı sayısı ile orantılıdır (Bkz. Şekil 4.10–4.12). Öte yandan, iş parçacığı sayısı artarken verim çok az da olsa düşmüştür. YAPAY2 veri seti için, ortalama hızlanma dikkate alındığında, 8 iş parçacığının 6 iş parçacığından daha az verimli olduğu görülmektedir. En ilginç durum ise, YAPAY3 veri setinin en iyi hızlanmaları arasındaki ilişkide meydana gelmiştir. YAPAY3 veri seti ile denemelerde, $p = 6$ için en iyi durum, en yüksek $\Delta\alpha$ değerinde ortaya çıkmıştır ve ortalama durumdan çok daha yüksek bir hızlanma gözlenmiştir.

P- α Scan ile denemelerde karşılaşılan tüm beklenmedik durumların ve az da olsa görülen verim kaybının bir gerçek, bir de yanıltıcı sebebi olduğu düşünülmektedir. Gerçek sebep, iterasyon sayısının yalnızca $\Delta\alpha$ değerine bağlı olmamasıdır (küme sayısı 1 olduğunda da α taramasının sonlandırıldığını hatırlayın). Verilerin dağılımına bağlı olarak, tekil kümeye daha hızlı erişilebilir. Bu durumda işlem yükü beklendiği kadar yüksek olmaz ve paralel programın verimi düşer. Yanıltıcı sebep ise, P- α Scan denemelerinin görece dar bir parametre aralığında gerçekleştirilmiş olmasıdır. P-GKM ile 2 ile 15 arasında değişen küme sayıları için 14 farklı deneme yapılırken, P- α Scan ile her bir veri setinde 5 farklı deneme yapılmıştır. Büyük $\Delta\alpha$ değerleri için işlem yükü çok düşük olduğundan, bu denemelerde hızlanmalar da düşüktür. Bu da, toplam deneme sayısının az olması sebebiyle ortalama hızlanmaları olduğundan düşük göstermektedir.

Hesaplama denemelerinin sonuçlarına dayanarak α Scan algoritmasıyla ilgili yapılabilecek genel bir yorum, seri halde dahi çok hızlı bir algoritma olduğudur. Denemelerin tamamında, 0.01 gibi büyük bir tarama parametresi için doğru kümelemeler elde edilmiştir. Buna dayanarak, çok büyük olmayan veri setleri için paralel hesaplama ihtiyacı olmadığı düşünülebilir. Yine de, PENDIGITS veri setinde 0.01 parametresiyle bile, 8 iş parçacığı için 3.4 kat hızlanma elde edilmiştir. Ayrıca, α Scan algoritmasının ne kadar büyük tarama parametreleriyle doğru sonuç elde edebileceği muğlaktır ve kendi başına bir araştırma konusudur.

5. SONUÇ

Halihazırda çok büyük olan ve büyüme hızı katlanarak artan veri setlerinin, veri analizini gittikçe zorlaştırıyor olması bu tezin çıkış noktasıdır. En yaygın veri analizi tekniklerinden biri olan kümelemede, işlem yükü özellikle çok ağırdır ve günümüz bilgisayar teknolojileri göz önüne alındığında, paralel kümeleme yöntemlerinin geliştirilmesi açık bir ihtiyaçtır. Tez çalışmasının odak noktası da bu ihtiyaç olmuştur. Kümeleme yöntemleri ve paralel hesaplama teknikleri incelenerek, farklı yaklaşımlara sahip, kümeleme başarısı yüksek iki algoritmanın paralel versiyonları önerilmiştir.

Önerilen paralel algoritmalarından biri olan P-GKM, klasik k -ortalamalar algoritmasının daha önce önerilen bir geliştirmesinin paralel versiyonudur. Diğer algoritma, bulanık mantığa dayalı FJP algoritmasından türetilmiştir. Öncelikle seri algoritmanın hızlandırılması için bir çalışma yapılmış ve bu çalışma sonucunda ortaya çıkan sezgisel α Scan algoritması paralelleştirilerek P- α Scan sunulmuştur.

P-GKM ve P- α Scan algoritmaları programlanarak gerçek ve yapay veri setleri üzerinde hesaplama denemeleri gerçekleştirilmiştir. Denemeler sonucunda, P-GKM'nin mümkün en iyi hızlanma olan doğrusala yakın bir hızlanma sağladığı ortaya konmuştur. P- α Scan ile daha düşük bir verime ulaşılabilmiş olsa da, yine iş parçacığı sayısı ile orantılı bir hızlanma başarılmıştır. Deney sonuçları, teorik analizle uyum göstermektedir.

Gelecekte, algoritmaların ne kadar iyi ölçeklenebildiğinin ortaya çıkarılması için, daha fazla işlemciye sahip sistemlerde denemeler yapılması faydalı olacaktır. Algoritmaların dağıtık bellekli bilgisayarlara ve grafik işlemcilere göre düzenlemeleri de yapılabilir. Bir diğer ilgi çekici çalışma konusu, α Scan algoritmasının parametre girdisine olan duyarlılığının araştırılmasıdır.

KAYNAKLAR DİZİNİ

- Akpolat, O., Odabaş, S.Ç., Özevci, G. ve İpteş, N.**, 2013, Kümeleme tekniklerinin temel bilimlerde kullanımı, *Akademik Bilişim 2013*, <http://ab.org.tr/ab13/bildiri/185.pdf> (Erişim tarihi: 25 Haziran 2014)
- Amdahl, G.M.**, 1967, Validity of the single processor approach to achieving large scale computing capabilities, *Proceedings of the Spring Joint Computer Conference*, 483–485 p.
- Arthur, D. and Vassilvitskii, S.**, 2007, K-means++: the advantages of careful seeding, *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 1027–1035 p.
- Babu , G. and Murty , M.**, 1993, A near-optimal initial seed value selection in k-means algorithm using a genetic algorithm, *Pattern Recognition Letters*, 14: 763–769 p.
- Bagirov, A.M. and Mardaneh, K.**, 2006, Modified global k-means algorithm for clustering in gene expression data sets, *Proceedings of the 2006 workshop on Intelligent systems for bioinformatics*, 73: 23–28 p.
- Barney, B.**, 2014a, “Introduction to Parallel Computing”, Lawrence Livermore National Laboratory, https://computing.llnl.gov/tutorials/parallel_comp/ (Erişim tarihi: 25 Haziran 2014)
- Barney, B.**, 2014b, “OpenMp”, Lawrence Livermore National Laboratory, <https://computing.llnl.gov/tutorials/openMP/> (Erişim tarihi: 25 Haziran 2014)
- Beckmann, N., Kriegel, H.P., Schneider, R. and Seeger, B.**, 1990, The R*-tree: an efficient access method for points and rectangles, *Proceedings of International Conference on Geographic Information Systems*, 19: 322–331 p.
- Berkhin, P.**, 2002, Survey of clustering data mining techniques, Accrue Software Technical Report, 56 p.
- Bezdek., J.C.**, 1981, Pattern Recognition with Fuzzy Objective Function Algorithms, Plenum Press, New York, 256 p.
- Bezdek., J.C., Ehrlich, R. and Full, W.**, 1984, FCM: The fuzzy c-means clustering algorithm, *Computers & Geosciences*, 10: 191–203 p.

KAYNAKLAR DİZİNİ (devam)

- Blelloch, G.E. and Maggs, B.M.**, 2004, “Parallel Algorithms”, Computer Science Handbook (2nd Edition) <http://www.cs.cmu.edu/~guyb/papers/BM04.pdf> (Erişim tarihi: 25 Haziran 2014).
- Brown, D. and Huntley, C.**, 1991, A practical application of simulated annealing to clustering,, University of Virginia Technical Report.
- Chris, P.**, 2013, “K Means”, Stanford University, <http://stanford.edu/~cpiech/cs221/handouts/kmeans.html> (Erişim tarihi: 25 Haziran 2014).
- Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C.**, 2009, Introduction to Algorithms (3rd Edition), The MIT Press, USA, 1292 p.
- Ester, M., Kriegel H.P., Sander, J. and Xu, X.**, 1996, A density-based algorithm for discovering clusters in large spatial databases with noise, *Proceedings of 2nd International Conference on Knowledge Discovery and Data Mining*, 226–231 p.
- Flynn, M.J.**, 1966, Very high-speed computing systems, *Proceedings of the IEEE*, 54:1901–1909 p.
- Some computer organizations and their effectiveness, *IEEE Transactions on Computers*, 21: 948–960 p.
- Grama, A., Gupta, A., Karypis, G. and Kumar, V.**, 2003, Introduction to Parallel Computing (Second Edition), Addison Wesley, Edinburgh Gate, 856 p.
- Guha, S., Rastogi, R. and Shim, K.**, 1999, ROCK: A robust clustering algorithm for categorical attributes, *Proceedings of 1999 International Conference on Data Engineering*, 512–521 p.
- Gustafson, J.L.**, 1988, Reevaluating Amdahl's Law, *Communications of the ACM*, 31: 532–533 p.
- Han, J. and Kamber, M.**, 2006, Data Mining: Concepts and Techniques (Second Edition), Morgan Kaufmann Publishers, San Fransisco, 743 p.
- Jain, A.K. and Dubes, R.C.**, 1988, Algorithms for Clustering Data, Prentice Hall, New Jersey, 320 p.
- Jang, J.R., Sun and C., Mizutani, E.**, 1997, Neuro-Fuzzy and Sofy Computing, Prentice-Hall, New Jersey, 614 p.

KAYNAKLAR DİZİNİ (devam)

- Karypis, G., Han, E.H. and Kumar, V.,** 1999, CHAMELEON: A hierarchical clustering algorithm using dynamic modeling, *IEEE Computer*, 32: 68–75 p.
- Kaufman, L. and Rousseeuw, P.J.,** 1990, Finding Groups in Data: An Introduction to Cluster Analysis, John Wiley & Sons, 368 p.
- Krishna, K. and Murty, M.,** 1999, Genetic k-means algorithm, *IEEE Transactions on Systems, Man, and Cybernetics Part B: Cybernetics*, 29, 433–439 p.
- Larose, D. T.,** 2005, Discovering Knowledge in Data, Wiley Press, New Jersey, 222 p.
- Lee, H.S.,** 2001, An optimal algorithm for computing the max-min transitive closure of a fuzzy similarity matrix, *Fuzzy Sets and Systems*, 123: 129–136 p.
- Likas, A., Vlassis, N. and Verbeek, J.J.,** 2003, The global k-means clustering algorithm, *Pattern Recognition*, 36: 451–461 p.
- MacQueen, J.B.,** 1967, Some Methods for classification and analysis of multivariate observations, *Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability*, 281–297 p.
- Manning, C.D., Raghavan, P. and Schütze, H.,** 2008, Introduction to Information Retrieval, Cambridge University Press, 544 p.
- Mihaeli A., Breunig, M.M., Kriegel H.P. and Sander, J.,** 1999, OPTICS: ordering points to identify the clustering structure, *ACM SIGMOD International Conference on Management of Data*, 49–60 p.
- Nasibov, E. and Ulutagay, G.,** 2007, A new unsupervised approach for fuzzy clustering, *Fuzzy Sets and Systems*, 158: 2118–2133 p.
- Nasibov, E. and Ulutagay, G.,** 2009, Robustness of density-based clustering methods with various neighborhood relations, *Fuzzy Sets and Systems*, 160: 3601–3615 p.
- Nasibov E., Atilgan, C. and Berberler, M.E.,** 2014, Fuzzy joint points based clustering algorithms for large data sets, *Fuzzy Sets and Systems* (in press).
- Ng, R. and Han, J.,** 1994, Efficient and effective clustering method for spatial data mining, *Proceedings of the 20th VLDB Conference*, 144–155 p.

KAYNAKLAR DİZİNİ (devam)

- Sander, J., Ester, M., Kriegel, H.P. and Xu, X.,** 1998, Density-based clustering in spatial databases: the algorithm GDBSCAN and its applications, *Data Mining and Knowledge Discovery*, 2: 169–194 p.
- Savaş, S., Topaloğlu, N. ve Yılmaz, M.,** 2012, Veri madenciliği ve Türkiye’deki uygulama örnekleri, *İstanbul Ticaret Üniversitesi Fen Bilimleri Dergisi*, 21: 1–23 s.
- Shen, F. and Hasegawa, O.,** 2006, An adaptive incremental LBG for vector quantization, *Neural Networks*, 19: 694–704 p.
- Tyron, R.C. and Bailey, D.E.,** 1970, Cluster Analysis, McGraw-Hill, 347 p.
- Ulutagay, G. and Nasibov, E.,** 2013, On fuzzy neighborhood based clustering algorithm with low complexity, *Iranian Journal of Fuzzy Systems*, 10: 1–20 p.
- Ward, J.H.,** 1963, Hierarchical grouping to optimize an objective function, *Journal of the American Statistical Association*, 58: 236–244 p.
- Xu, R. and Wunsch, D.C.,** 2009, Clustering, IEEE Press, New Jersey, 358 p.
- Zadeh, L. A.,** 1956, Fuzzy sets, *Information And Control*, 8: 338-353 p.
- Zhang, T., Ramakrishnan, R. and Livny, M.,** 1996, BIRCH: an efficient data clustering method for very large databases, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, 103–114 p.
- “OpenMP Application Program Interface Version 3.1”, 2011, <http://www.openmp.org/mp-documents/OpenMP3.1.pdf> (Erişim tarihi: 25 Haziran 2014).

ÖZGEÇMİŞ

Can Atılğan, 27 Ağustos 1988 tarihinde İzmir’de doğmuştur ve Türkiye Cumhuriyeti vatandaşıdır. Liseyi 2002–2006 yılları arasında Bornova Suphi Koyuncuoğlu Lisesi’nde okumuştur. Şubat 2012’de Ege Üniversitesi Matematik Bölümü’nün Bilgisayar Bilimleri Ağırlıklı programından mezun olmuştur. Eylül 2012’de Ege Üniversitesi Fen Bilimleri Enstitüsü Matematik Anabilim Dalı’nda yüksek lisans eğitime başlamıştır. Şubat 2013’de Dokuz Eylül Üniversitesi Bilgisayar Bilimleri Bölümü’nde Araştırma Görevlisi olarak göreve başlamıştır.

EKLER

Ek 1 Kümeleme Kütüphanesinin Kaynak Kodu

Ek 2 k-Ortalamlar Programının Kaynak Kodu

Ek 3 Global k-Ortalamlar Programının Kaynak Kodu

Ek 4 α Scan Programının Kaynak Kodu

Ek 5 Paralel Global k-Ortalamlar Programının Kaynak Kodu

Ek 6 Paralel α Scan Programının Kaynak Kodu

Ek 1 Kümleme Kütüphanesinin Kaynak Kodu – “clusterlib.h”

```
#ifndef CLUSTERLIB_H
#define CLUSTERLIB_H

typedef unsigned char logical;

typedef unsigned long ulong;

float euclid(float *data1, float *data2, ulong dim);

logical determineClusters (ulong *cluster, float **data, float **center, ulong n, ulong k, ulong
dim);

logical determineClustersWithDistanceToCenter (ulong *cluster, float **data, float **center, float
*distanceToCenter, ulong n, ulong k, ulong dim);

void determineCenters (float **center, float **data, ulong *cluster, ulong n, ulong k, ulong dim);

void evaluateClustering (float *clusterDist, ulong *clusterCard, float **data, float **center, ulong
*cluster, ulong n, ulong k, ulong dim);

void findCenterWithMaxAvg(float **center, float **data, ulong *cluster, ulong id, ulong n, ulong
k, ulong dim);

void locateLocalCenter (float **data, float *localCenter, float *distanceToCenter, float
*distanceVector, ulong n, ulong dim);

float calculateClusterFunction (float **data, float **center, float *distanceToCenter, float
*localCenter, ulong n, ulong dim);

ulong connectedComponents(float **T, ulong *v, float alfa, ulong n);

#endif
```

Ek 1 Kümeleme Kütüphanesinin Kaynak Kodu – “clusterlib.c”

```
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "omp.h"
#include "clusterlib.h"

float euclid(float *data1, float *data2, ulong dim) {
    float dist = 0;
    ulong i;

    for (i = 0; i < dim; i++)
        dist += (data1[i] - data2[i]) * (data1[i] - data2[i]);

    return sqrt(dist);
}

logical determineClusters (ulong *cluster, float **data, float **center, ulong n, ulong k, ulong
dim) {
    ulong i, j, prev;
    logical control = 0;
    float minDist, dist;

    for (i = 0; i < n; i++) {
        minDist = euclid(data[i], center[0], dim);
        prev = cluster[i];
        cluster[i] = 0;

        for (j = 1; j < k; j++) {
            dist = euclid(data[i], center[j], dim);
            if (dist < minDist) {
                minDist = dist;
                cluster[i] = j;
            }
        }

        if (prev != cluster[i])
            control = 1;
    }

    return control;
}

logical determineClustersWithDistanceToCenter (ulong *cluster, float **data, float **center, float
*distanceToCenter, ulong n, ulong k, ulong dim) {
    ulong i, j, prev;
    logical control = 0;
    float minDist, dist;

    for (i = 0; i < n; i++) {
        minDist = euclid(data[i], center[0], dim);
        prev = cluster[i];
        cluster[i] = 0;
```

Ek 1 Kümeleme Kütüphanesinin Kaynak Kodu – “clusterlib.c” (devam)

```
        for (j = 1; j < k; j++) {
            dist = euclid(data[i], center[j], dim);
            if (dist < minDist) {
                minDist = dist;
                cluster[i] = j;
            }
        }

        if (prev != cluster[i]) {
            control = 1;
        }

        distanceToCenter[i] = minDist;
    }

    return control;
}

void determineCenters (float **center, float **data, ulong *cluster, ulong n, ulong k, ulong dim) {
    ulong i, j, *clusterCard;

    clusterCard = (ulong*) malloc (k * sizeof(ulong));
    memset(clusterCard, 0, k * sizeof(ulong));

    for (i = 0; i < k; i++)
        memset(center[i], 0, dim * sizeof(float));

    for (i = 0; i < n; i++) {
        clusterCard[cluster[i]]++;

        for (j = 0; j < dim; j++)
            center[cluster[i]][j] += data[i][j];
    }

    for (i = 0; i < k; i++)
        for (j = 0; j < dim; j++)
            center[i][j] /= clusterCard[i];

    free(clusterCard);

    return;
}

void evaluateClustering (float *clusterDist, ulong *clusterCard, float **data, float **center, ulong
*cluster, ulong n, ulong k, ulong dim) {
    ulong i;

    memset(clusterDist, 0, k * sizeof(float));

    if (clusterCard == NULL) {
        for (i = 0; i < n; i++)
            clusterDist[cluster[i]] += euclid(data[i], center[cluster[i]], dim);
    }
    else
    {
        memset(clusterCard, 0, k * sizeof(ulong));
    }
}
```

Ek 1 Kümeleme Kütüphanesinin Kaynak Kodu – “clusterlib.c” (devam)

```
        for (i = 0; i < n; i++) {
            clusterDist[cluster[i]] += euclid(data[i], center[cluster[i]], dim);
            clusterCard[cluster[i]]++;
        }

    return;
}

void findCenterWithMaxAvg(float **center, float **data, ulong *cluster, ulong id, ulong n, ulong
k, ulong dim) {
    ulong i, ci;
    float dist, maxDist = 0;

    for (i = 0; i < n; i++)
        if (cluster[i] == id) {
            dist = euclid(data[i], center[id], dim);
            if (dist > maxDist) {
                maxDist = dist;
                ci = i;
            }
        }

    memcpy(center[k-1], data[ci], dim * sizeof(float));

    return;
}

void locateLocalCenter (float **data, float *localCenter, float *distanceToCenter, float
*distanceVector, ulong n, ulong dim) {
    ulong i, j, prevCard, card = 0;

    logical *v = (logical*) malloc (n * sizeof(logical));
    memset(v, 0, n * sizeof(logical));

    float *sumCenter = (float*) malloc (dim * sizeof(float));
    memset(sumCenter, 0, dim * sizeof(float));

    for (i = 0; i < n; i++)
        if ( distanceVector[i] < distanceToCenter[i] ) {
            v[i] = 1;
            card++;
            for (j = 0; j < dim; j++)
                sumCenter[j] += data[i][j];
        }

    for (i = 0; i < dim; i++)
        localCenter[i] = sumCenter[i] / card;

    do {

        prevCard = card;

        for (i = 0; i < n; i++)
            if (!v[i])
```

Ek 1 Kümeleme Kütüphanesinin Kaynak Kodu – “clusterlib.c” (devam)

```
        if ( euclid(data[i], localCenter, dim) < distanceToCenter[i] ) {
            v[i] = 1;
            card++;
            for (j = 0; j < dim; j++)
                sumCenter[j] += data[i][j];
        }

    if (prevCard == card)
        break;

    for (i = 0; i < dim; i++)
        localCenter[i] = sumCenter[i] / card;

} while (1);

free(v);
free(sumCenter);

return;
}

float calculateClusterFunction (float **data, float **center, float *distanceToCenter, float
*localCenter, ulong n, ulong dim) {
    ulong i;
    float dist, f = 0;

    if (localCenter == NULL)
        for (i = 0; i < n; i++)
            f += distanceToCenter[i];
    else
        for (i = 0; i < n; i++) {
            dist = euclid(data[i], localCenter, dim);

            if (dist < distanceToCenter[i])
                f += dist;
            else
                f += distanceToCenter[i];
        }

    return f;
}

ulong connectedComponents(float **T, ulong *v, float alfa, ulong n) {
    ulong i, j, k, kk, cc, *t;

    t = (ulong*) malloc (n * sizeof(ulong));

    memset(v, 0, n * sizeof(ulong));

    i = cc = 0;

    do {

        k = 0;
        kk = 0;

        cc++;
```

Ek 1 Kümeleme Kütüphanesinin Kaynak Kodu – “clusterlib.c” (devam)

```
t[k] = i;
v[i] = cc;

do {

    for(j = 0; j < n; j++)
        if(T[i][j] >= alfa && v[j] == 0) {
            k++;
            t[k] = j;
            v[j] = cc;
        }

        if (kk < k) {
            kk++;
            i = t[kk];
        } else
            break;

    } while (1);

    for(i = 0; i < n && v[i] != 0; i++);

} while (i < n);

return cc;
}
```

Ek 2 k-Ortalamlar Programının Kaynak Kodu

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include "clusterlib.h" // logical ve ulong tipleri burada tanimli

int main(int argc, char *argv[]) { // 1. argüman dosya yolu, 2. argüman k sayısı

    logical control;
    ulong i, j, n, dim, k;
    ulong *cluster;
    float **data, **center, totalDist;

    char *fileName = argv[1];

    FILE *f = fopen(fileName, "r");

    if (f == NULL) {
        printf("Dosya bulunamadi!\n");
        return 1;
    }

    k = strtoul(argv[2], NULL, 10);

    fscanf(f, "%lu %lu", &n, &dim);

    data = (float**) malloc (n * sizeof(float*));

    for (i = 0; i < n; i++)
        data[i] = (float*) malloc (dim * sizeof(float));

    center = (float**) malloc (k * sizeof(float*));

    for (i = 0; i < k; i++)
        center[i] = (float*) malloc (dim * sizeof(float));

    cluster = (ulong*) malloc (n * sizeof(ulong));

    for (i = 0; i < n; i++)
        for (j = 0; j < dim; j++)
            fscanf(f, "%f", &data[i][j]);

    fclose(f);

    clock_t start, stop;
    start = clock();

    // ilk merkezlerin aynı olmayacak şekilde rastgele seçilmesi
    srand(time(NULL));
    ulong *initCenterIds = (ulong*) malloc (k * sizeof(ulong));

    initCenterIds[0] = rand() % n;

    for (i = 1; i < k; i++) {

        do {
```

Ek 2 k-Ortalamlar Programının Kaynak Kodu (devam)

```
        initCenterIds[i] = rand() % n;

        for (j = 0; j < i; j++)
            if (initCenterIds[i] == initCenterIds[j])
                break;

    } while (j < i);
}

for (i = 0; i < k; i++)
    memcpy(center[i], data[initCenterIds[i]], dim * sizeof(float));
free(initCenterIds);

// k-means
do {

    control = determineClusters(cluster, data, center, n, k, dim);

    if (!control)
        break;

    determineCenters(center, data, cluster, n, k, dim);

} while(1);

stop = clock();

float *clusterDist = (float*) malloc (k * sizeof(float));
evaluateClustering(clusterDist, NULL, data, center, cluster, n, k, dim);
totalDist = 0;
for (i = 0; i < k; i++)
    totalDist += clusterDist[i];

strcpy(strchr(fileName, '.'), "_kmeans_sonuc.txt.");
f = fopen(fileName, "a");

fprintf(f, "k = %lu\tSüre = %.3f sn.\tf = %.3f\n", k, (float)(stop -
start)/CLOCKS_PER_SEC, totalDist/n);

fclose(f);

// sonuç kümelemesi cluster değişkeninde

return 0;
}
```

Ek 3 Global k-Ortalamalar Programının Kaynak Kodu

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include <float.h>
#include "clusterlib.h" // logical ve ulong tipleri burada tanimli

int main(int argc, char *argv[]) { // 1. argüman dosya yolu, 2. argüman K sayisi

    logical control;
    ulong i, j, n, dim, k, K;
    ulong *cluster;
    float **data, **center, **newCenter, **prevCenter, *clusterDist, firstDist, prevTotalDist,
    tempTotalDist, totalDist;

    // denemelerde kullanilmadi, K yerine durdurma kosulu olarak kullanilabilir
    float TH = 0.00;

    char *fileName = argv[1];

    FILE *f = fopen(fileName, "r");

    if (f == NULL) {
        printf("Dosya bulunamadi!\n");
        return 1;
    }

    K = strtoul(argv[2], NULL, 10);

    fscanf(f, "%lu %lu", &n, &dim);

    data = (float**) malloc (n * sizeof(float*));

    for (i = 0; i < n; i++)
        data[i] = (float*) malloc (dim * sizeof(float));

    center = (float**) malloc (K * sizeof(float*));
    newCenter = (float**) malloc (K * sizeof(float*));
    prevCenter = (float**) malloc (K * sizeof(float*));

    for (i = 0; i < K; i++) {
        center[i] = (float*) malloc (dim * sizeof(float));
        newCenter[i] = (float*) malloc (dim * sizeof(float));
        prevCenter[i] = (float*) malloc (dim * sizeof(float));
    }

    cluster = (ulong*) malloc (n * sizeof(ulong));

    for (i = 0; i < n; i++)
        for (j = 0; j < dim; j++)
            fscanf(f, "%f", &data[i][j]);

    clusterDist = (float*) malloc (K * sizeof(float));

    fclose(f);
```

Ek 3 Global k-Ortalamlar Programının Kaynak Kodu (devam)

```
clock_t start, stop;
start = clock();

// ilk merkezin belirle
k = 1;
memset(cluster, 0, n * sizeof(ulong));
determineCenters(prevCenter, data, cluster, n, k, dim);

// ilk toplam uzakligi bul
evaluateClustering(clusterDist, NULL, data, prevCenter, cluster, n, k, dim);
firstDist = totalDist = clusterDist[0];

// ana döngü (gkm)
do {

    k++;

    prevTotalDist = totalDist;
    totalDist = FLT_MAX;

    for (i = 0; i < n; i++) {

        // önceki merkezleri hesaplanacak merkezlere ata
        for (j = 0; j < k - 1; j++)
            memcpy(center[j], prevCenter[j], dim * sizeof(float));

        // hesaplanacak merkezlerin yeni noktası sıradaki veri olacak
        memcpy(center[k-1], data[i], dim * sizeof(float));

        // k-means
        do {

            control = determineClusters(cluster, data, center, n, k, dim);

            if (!control)
                break;

            determineCenters(center, data, cluster, n, k, dim);

        } while(1);

        // hesaplanan merkezlere göre toplam uzaklığı bul
        evaluateClustering(clusterDist, NULL, data, center, cluster, n, k, dim);
        tempTotalDist = 0;
        for (j = 0; j < k; j++)
            tempTotalDist += clusterDist[j];

        // hesaplanan merkez iterasyonun en iyisiye kaydet
        if (tempTotalDist < totalDist) {
            totalDist = tempTotalDist;

            for (j = 0; j < k; j++)
                memcpy(newCenter[j], center[j], dim * sizeof(float));
        }

    }

}
```

Ek 3 Global k-Ortalamlar Programının Kaynak Kodu (devam)

```
// iterasyonda kaydedilen en iyi merkezleri önceki merkezlere ata
for (j = 0; j < k; j++)
    memcpy(prevCenter[j], newCenter[j], dim * sizeof(float));

} while((prevTotalDist - totalDist) / firstDist > TH && k < K); // durdurma koşulları

// küme id lerini en son küme merkezlerine göre ayarla
determineClusters(cluster, data, newCenter, n, k, dim);

stop = clock();

ulong *clusterCard = (ulong*) malloc (k * sizeof(ulong));
evaluateClustering(clusterDist, clusterCard, data, newCenter, cluster, n, k, dim);
totalDist = 0;
for (i = 0; i < k; i++)
    totalDist += clusterDist[i];

strcpy(strrchr(fileName, '.'), "_gkm_sonuc.txt");
f = fopen(fileName, "a");

fprintf(f, "k = %lu\\t\\tSüre = %.3f sn.\\t\\tf = %.3f\\n", K, (float)(stop -
start)/CLOCKS_PER_SEC, totalDist/n);

fclose(f);

// sonuç kümelemesi cluster değişkeninde

return 0;

}
```

Ek 4 aScan Programının Kaynak Kodu

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <string.h>
#include "clusterlib.h" // logical ve ulong tipleri burada tanimli

int main(int argc, char *argv[]) // 1. argüman dosya yolu, 2. argüman delta alfa
{
    ulong dim, n, i, j, k, kk, cc, pcc, rep, maxRep, rekor, *bestv, *pv, *v, *t;
    float R, alfa, kalfa, **data, **T;

    char *fileName = argv[1];

    FILE *f = fopen(fileName, "r");

    if (f == NULL) {
        printf("Dosya bulunamadi!\n");
        return 1;
    }

    float dalfa = strtod(argv[2], NULL);

    fscanf(f, "%lu %lu", &n, &dim);

    data = (float**) malloc (n * sizeof(float*));
    T = (float**) malloc (n * sizeof(float*));

    for (i = 0; i < n; i++) {
        data[i] = (float*) malloc (dim * sizeof(float));
        T[i] = (float*) malloc (n * sizeof(float));
    }

    for (i = 0; i < n; i++)
        for (j = 0; j < dim; j++)
            fscanf(f, "%f", &data[i][j]);

    fclose(f);

    strcpy(strchr(fileName, '.'), "_ascan_sonuc.txt");
    f = fopen(fileName, "a");

    clock_t start, end;
    start = clock();

    // uzaklik tablosunun olusturulmasi ve R nin bulunmasi
    R = 0;

    for (i = 0; i < n - 1; i++)
        for (j = i + 1; j < n; j++) {
            T[i][j] = euclid(data[i], data[j], dim);

            if (T[i][j] > R)
                R = T[i][j];
        }
}
```

Ek 4 aScan Programının Kaynak Kodu (devam)

```
// R /= 2; (*)

for(i = 0; i < n; i++) {
    T[i][i] = 1;          // reflexive

    for(j = i + 1; j < n; j++)
        T[i][j] = T[j][i] = 1 - T[i][j] / R; // 2*R (*)
}

bestv = (ulong*) malloc (n * sizeof(ulong));
v = (ulong*) malloc (n * sizeof(ulong));
pv = (ulong*) malloc (n * sizeof(ulong));
t = (ulong*) malloc (n * sizeof(ulong));

// alfa döngüsü
pcc = maxRep = 0;
rep = 0;
alfa = 1;

do {

    cc = connectedComponents(T, v, alfa, n);

    if(cc == pcc) // yeni küme sayısı eskisine esit mi?
        rep++; // tekrar sayısı
    else {
        if (maxRep < rep) {
            maxRep = rep; // en çok tekrar sayısı saklanıyor
            rekor = pcc; // en çok tekrardaki küme sayısı saklanıyor
            kalfa = alfa + dalfa; // en çok tekrardaki son alfa saklanıyor
            // en çok tekrardaki kümeleme saklanıyor
            memcpy(bestv, pv, sizeof(pv));
        }

        rep = 1;
        memcpy(pv, v, sizeof(v)); // en son (yeni) kümeleme saklanıyor
        pcc = cc; // yeni küme sayısı eski küme sayısına ataniyor
    }

    alfa -= dalfa;

} while(cc > 1 && alfa > 0);

end = clock();

fprintf(f, "Süre = %.3f\ttk = %lu\ttdalfa = %f\n", (float) (end - start) /
CLOCKS_PER_SEC, rekor, dalfa);

fclose(f);

// sonuç kümelemesi bestv değişkeninde

return 0;

}
```

Ek 5 Paralel Global k-Ortalamlar Programının Kaynak Kodu

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include <float.h>
#include "omp.h"
#include "clusterlib.h"    // logical ve ulong tipleri burada tanımlı

int main(int argc, char *argv[]) {    // 1. arg. dosya yolu, 2. arg. K sayısı, 3. arg. thread sayısı

    logical control;
    ulong i, j, n, dim, k, tid, maxid, K;
    ulong *cluster;
    float **data, ***newCenter, **prevCenter, *clusterDist, firstDist, prevTotalDist,
    tempTotalDist, *totalDist;

    // denemelerde kullanılmadı, K yerine durdurma koşulu olarak kullanılabilir
    float TH = 0.0;

    char *fileName = argv[1];

    FILE *f = fopen(fileName, "r");

    if (f == NULL) {
        printf("Dosya bulunamadı!\n");
        return 1;
    }

    K = strtoul(argv[2], NULL, 10);

    ulong tnum = strtoul(argv[3], NULL, 10);
    omp_set_dynamic(0);
    omp_set_num_threads(tnum);

    fscanf(f, "%lu %lu", &n, &dim);

    // data[n][dim]
    data = (float**) malloc (n * sizeof(float*));

    for (i = 0; i < n; i++)
        data[i] = (float*) malloc (dim * sizeof(float));

    // prevCenter[K][dim]
    prevCenter = (float**) malloc (K * sizeof(float*));

    for (i = 0; i < K; i++)
        prevCenter[i] = (float*) malloc (dim * sizeof(float));

    // newCenter[tnum][K][dim]
    newCenter = (float***) malloc (tnum * sizeof(float**));

    for (i = 0; i < tnum; i++) {
        newCenter[i] = (float**) malloc (K * sizeof(float*));
        for (j = 0; j < K; j++)
            newCenter[i][j] = (float*) malloc (dim * sizeof(float));
    }
}
```

Ek 5 Paralel Global k-Ortalamlar Programının Kaynak Kodu (devam)

```
// totalDist[tnum]
totalDist = (float*) malloc (tnum * sizeof(float));

for (i = 0; i < n; i++)
    for (j = 0; j < dim; j++)
        fscanf(f, "%f", &data[i][j]);

fclose(f);

double start, stop;
start = omp_get_wtime();

// ilk merkezin belirle
// cluster[n]
cluster = (ulong*) malloc (n * sizeof(ulong));
k = 1;
memset(cluster, 0, n * sizeof(ulong));
determineCenters(prevCenter, data, cluster, n, k, dim);

// ilk toplam uzakligi bul
// clusterDist[1]
clusterDist = (float*) malloc (1 * sizeof(float));
evaluateClustering(clusterDist, NULL, data, prevCenter, cluster, n, k, dim);
maxid = 0;
firstDist = totalDist[maxid] = clusterDist[0];

free(cluster);
free(clusterDist);

do {

    k++;

    prevTotalDist = totalDist[maxid];

    // tüm noktalarla k-means uygula (global search)
#pragma omp parallel default(shared) private(i, j, control, tempTotalDist, clusterDist, cluster, tid)
    {
        tid = omp_get_thread_num();

        // center[k][dim] (her thread'e özel)
        float **center = (float**) malloc (k * sizeof(float*));
        for (i = 0; i < k; i++)
            center[i] = (float*) malloc (dim * sizeof(float));

        // clusterDist[k] (her thread'e özel)
        clusterDist = (float*) malloc (k * sizeof(float));

        // cluster[n] (her thread'e özel)
        cluster = (ulong*) malloc (n * sizeof(ulong));

        totalDist[tid] = FLT_MAX;

#pragma omp for schedule(static)
        for (i = 0; i < n; i++) {
```

Ek 5 Paralel Global k-Ortalamalar Programının Kaynak Kodu (devam)

```
// önceki merkezleri hesaplanacak merkezlere ata
for (j = 0; j < k - 1; j++)
    memcpy(center[j], prevCenter[j], dim * sizeof(float));

// hesaplanacak merkezlerin yeni noktası siradaki veri olacak
memcpy(center[k-1], data[i], dim * sizeof(float));

// k-means
do {

    control = determineClusters(cluster, data, center, n, k,
dim);

    if (!control)
        break;

    determineCenters(center, data, cluster, n, k, dim);

} while(1);

// hesaplanan merkezlere göre toplam uzaklığı bul
evaluateClustering(clusterDist, NULL, data, center, cluster, n,
k, dim);
tempTotalDist = 0;
for (j = 0; j < k; j++)
    tempTotalDist += clusterDist[j];

// hesaplanan merkez iterasyonun en iyisiyse kaydet
if (tempTotalDist < totalDist[tid]) {
    totalDist[tid] = tempTotalDist;

    for (j = 0; j < k; j++)
        memcpy(newCenter[tid][j], center[j], dim *
sizeof(float));
}

}

// her thread kendi evinin önünü temizlemeli
for (i = 0; i < k; i++)
    free(center[i]);
free(center);
free(clusterDist);
free(cluster);

} // end parallel

maxid = 0;

for (i = 1; i < tnum; i++)
    if (totalDist[i] < totalDist[maxid])
        maxid = i;

// iterasyonda kaydedilen en iyi merkezleri önceki merkezlere ata
for (j = 0; j < k; j++)
    memcpy(prevCenter[j], newCenter[maxid][j], dim * sizeof(float));
```

Ek 5 Paralel Global k-Ortalamlar Programının Kaynak Kodu (devam)

```
} while((prevTotalDist - totalDist[maxid]) / firstDist > TH && k < K); // durma kosullari

// küme id lerini en son küme merkezlerine göre ayarla
cluster = (ulong*) malloc (n * sizeof(ulong));
determineClusters(cluster, data, prevCenter, n, k, dim);

stop = omp_get_wtime();

ulong *clusterCard = (ulong*) malloc (k * sizeof(ulong));
clusterDist = (float*) malloc (k * sizeof(float));
evaluateClustering(clusterDist, clusterCard, data, prevCenter, cluster, n, k, dim);
totalDist[0] = 0;
for (i = 0; i < k; i++)

strcpy(strchr(fileName, '.'), "_omp_gkm_sonuc.txt");

f = fopen(fileName, "a");

fprintf(f, "k = %lu\tSüre = %.3f sn.\ttf = %.3f\n", K, stop - start, totalDist[0]/n);

fclose(f);

// sonuç kümelemesi cluster değişkeninde

return 0;
}
```

Ek 6 Paralel α Scan Programının Kaynak Kodu

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <string.h>
#include "omp.h"
#include "clusterlib.h"    // logical ve ulong tipleri burada tanimli

int main(int argc, char *argv[]) {    // 1. arg. dosya yolu, 2. arg. delta alfa, 3. arg. thread sayisi

    ulong dim, n, i, j, rep, maxRep, rekor, iter, maxIter, *v;
    float R, alfa, kalfa, **data, **T;

    char *fileName = argv[1];

    FILE *f = fopen(fileName, "r");

    if (f == NULL) {
        printf("Dosya bulunamadi!\n");
        return 1;
    }

    float dalfa = strtod(argv[2], NULL);

    ulong tnum = strtoul(argv[3], NULL, 10);
    omp_set_dynamic(0);
    omp_set_num_threads(tnum);

    fscanf(f, "%lu %lu", &n, &dim);

    data = (float**) malloc (n * sizeof(float*));
    T = (float**) malloc (n * sizeof(float*));

    for (i = 0; i < n; i++) {
        data[i] = (float*) malloc (dim * sizeof(float));
        T[i] = (float*) malloc (n * sizeof(float));
    }

    for (i = 0; i < n; i++)
        for (j = 0; j < dim; j++)
            fscanf(f, "%f", &data[i][j]);

    fclose(f);

    double start, stop;
    start = omp_get_wtime();

    // uzaklik tablosunun olusturulmasi ve R nin bulunmasi
    R = 0;
```

Ek 6 Paralel α Scan Programının Kaynak Kodu (devam)

```
#pragma omp parallel default(shared) private(i, j)
{
    #pragma omp for reduction(max:R) schedule(static, 1)
    for (i = 0; i < n - 1; i++)
        for (j = i + 1; j < n; j++) {
            T[i][j] = euclid(data[i], data[j], dim);

            if (T[i][j] > R)
                R = T[i][j];
        }

    // R /= 2; (*)

    #pragma omp for schedule(static, 1)
    for(i = 0; i < n; i++) {
        T[i][i] = 1;          // reflexive

        for(j = i + 1; j < n; j++)
            T[i][j] = T[j][i] = 1 - T[i][j] / R; // 2*R (*)
    }
} // end parallel region

maxIter = 1 / dalfa;

ulong *clusterNum = (ulong*) malloc (maxIter * sizeof(ulong));
ulong tid;

// alfa döngüsü
#pragma omp parallel default(shared) private(v, alfa, iter, tid)
{
    tid = omp_get_thread_num();

    v = (ulong*) malloc (n * sizeof(ulong));

    for (iter = tid; iter < maxIter; iter += tnum) {

        alfa = 1 - iter * dalfa;

        clusterNum[iter] = connectedComponents(T, v, alfa, n);

        if (clusterNum[iter] == 1)
            break;
    }
} // end parallel region

maxRep = 0;
rep = 1;
i = 0;
while (clusterNum[i] > 1 && i + 1 < maxIter) {
    if (clusterNum[i] == clusterNum[i + 1])
        rep++;
    else {
        if (maxRep < rep) {
            maxRep = rep;
        }
    }
}
```

Ek 6 Paralel α Scan Programının Kaynak Kodu (devam)

```
        kalfa = 1 - i * dalfa;
    }

    rep = 1;
}

    i++;
}

rekor = connectedComponents(T, v, kalfa, n);

stop = omp_get_wtime();

strcpy(strchr(fileName, '.'), "_omp_ascan_sonuc.txt");

f = fopen(fileName, "a");

fprintf(f, "Süre = %.3f\tk = %lu\t\tdalfa = %f\n", stop - start, rekor, dalfa);

fclose(f);

// sonuç kümelemesi v değişkeninde

return 0;
}
```