# ÖBEK BİLGİSAYARLARDA PARALEL FP-GROWTH GERÇEKLEŞTİRİMİ

**GÜLİSTAN ÖZDEMİR ÖZDOĞAN**

**YÜKSEK LİSANS TEZİ**

**BİLGİSAYAR MÜHENDİSLİĞİ**

**TOBB EKONOMİ VE TEKNOLOJİ ÜNİVERSİTESİ**

**FEN BİLİMLERİ ENSTİTÜSÜ**

**OCAK 2010**

**ANKARA**

Fen Bilimleri Enstitü onayı

_____

Prof. Dr. Ünver KAYNAK

Müdür

Bu tezin Yüksek Lisans derecesinin tüm gereksinimlerini sağladığını onaylarım.

_____

Doç. Dr. Erdoğan Doğdu

Anabilim Dalı Başkanı

Gülistan ÖZDEMİR ÖZDOĞAN tarafından hazırlanan ÖBEK BİLGİSAYARLARDA PARALEL FP-GROWTH GERÇEKLEŞTİRİMİ adlı bu tezin Yüksek Lisans tezi olarak uygun olduğunu onaylarım.

_____

Yrd. Doç. Dr. Osman ABUL

Tez Danışmanı

Tez Jüri Üyeleri

Başkan : Doç. Dr. Erdoğan DOĞDU _____

Üye : Yrd. Doç. Dr. Nilay SEZER UZOL _____

Üye : Yrd. Doç. Dr. Osman ABUL _____

## TEZ BİLDİRİMİ

Tez içindeki bütün bilgilerin etik davranış ve akademik kurallar çerçevesinde elde edilerek sunulduğunu, ayrıca tez yazım kurallarına uygun olarak hazırlanan bu çalışmada orijinal olmayan her türlü kaynağa eksiksiz atıf yapıldığını bildiririm.

Gülistan ÖZDEMİR ÖZDOĞAN

**Üniversitesi** : **TOBB Ekonomi ve Teknoloji Üniversitesi**
**Enstitüsü** : **Fen Bilimleri**
**Anabilim Dalı** : **Bilgisayar Mühendisliği**
**Tez Danışmanı** : **Yrd. Doç. Dr. Osman ABUL**
**Tez Türü ve Tarihi** : **Yüksek Lisans – Ocak 2010**

**Gülistan ÖZDEMİR ÖZDOĞAN**

# ÖBEK BİLGİSAYARLARDA PARALEL FP-GROWTH GERÇEKLEŞTİRİMİ

## ÖZET

Teknolojinin gelişmesiyle verilerin miktarında ve çeşitliliğinde bir artış gözlemlenmiştir. Bu artış, veri işleme yöntemlerinde yeni ihtiyaçları gündeme getirmiştir. Veri madenciliği bu ihtiyaçlara cevap verebilmek amacıyla geliştirilen büyük veri ambarlarından yararlı bilgi elde etme sürecidir. Veri madenciliği kapsamında, çoklukla gereksinim duyulan ve araştırılan konulardan biri sık öge küme madenciliğidir. Sık öge küme madenciliği için farklı algoritmalar üretilmekte ve incelenmektedir. Ancak, fiziksel kapasitelerin sınırlı olmasından dolayı seri algoritmalar artan veri miktarını karşılamaya yetmemektedir. Bu durum paralel hesaplamayı gündeme getirir. Paralel algoritmalarla, zamandan ve kaynaktan (depolama, hafıza gibi) tasarruf edebilmek amaçlanır. FP-Growth, sık öge kümelerini bulmak için geliştirilen bir veri madenciliği algoritmasıdır. Literatürde FP-Growth, işlenecek veritabanını sadece iki kez tarama ve sık ögeleri bulurken kullandığı sık örüntü ağacı özelliği ile bilinir. Kullandığı bu ağaç yapısı ile daha verimli bir algoritma sunar. Bu çalışma içerisinde FP-Growth algoritmasına, paralel programlar yazmak için geliştirilen bir kütüphane olan MPI ile kodlanan üç farklı paralel yaklaşım önerilmektedir. Bu yaklaşımlarda temel alınan şey, paralel uygulama sırasında düğümlerin tümünde veritabanının mevcut olup olmadığıdır. Ek olarak, paralellik için görev dağılımı esas alındığından görev dağılımının statik ya da dinamik olması da diğer bir parametredir. Bu yaklaşımların iki farklı öbek bilgisayar üzerinde performans analizi gerçekleştirilmiştir. Çıkan sonuçlara göre, veritabanının mevcut olduğu iki yaklaşımın ağaç gönderimi esasına dayanan diğer yaklaşıma göre daha iyi sonuç verdiği gözlemlenmektedir. Her ikisi için de, paralel algoritma iki işlemciyle çalıştığı durumda bile çalışma zamanını dörtte bir oranında azaltmaktadır. Buna ek olarak, daha iyi sonuç veren iki algoritma içerisinde görev dağılımının dinamik olarak gerçekleştiği durum statik görev dağılımlı yaklaşımdan daha iyi sonuç vermektedir.

**Anahtar Kelimeler:** FP-Growth, Paralel Hesaplama, Sık Öge Küme Madenciliği, Veri Madenciliği.

**University : TOBB Economics and Technology University**
**Institute : Institute of Natural and Applied Sciences**
**Science Programme : Computer Engineering**
**Supervisor : Asst. Prof. Dr. Osman ABUL**
**Degree Awarded and Date : M.Sc. – January 2010**

**Gülistan ÖZDEMİR ÖZDOĞAN**

# IMPLEMENTATION OF PARALLEL FP-GROWTH ALGORITHM ON CLUSTER COMPUTERS

## ABSTRACT

With the development of technology, there is an increase in size and complexity of data. The increase bring about new needs in the process of data processing. Data mining, aimed to satisfy these needs, is the process of acquiring useful information from large data warehouses. In data mining, the purpose is usually to find the frequent items which is called frequent itemset mining (FIM). Different algorithms are developed and studied for this purpose. Because of the limited physical capacity, serial algorithms aren't sufficient when solving large data problems. So, paralel computing becomes important. With parallel algorithms, it is aimed to save on time and sources (like storage and memory). FP-Growth is a data mining algorithm that is developed to find frequent itemsets. FP-Growth is known with performing only two scans while reading database and using frequent pattern tree to find frequent items. Because of this data structure, it becomes more efficient than other frequent itemset mining algorithms. In the study, there are three different parallel approaches implemented in MPI, a library that is used to write parallel programs. The first basis thing for the approaches is whether there is a database on all of the nodes or not. The second thing is how task parallelism is done. For task, there are static and dynamical approaches. These are tested and analysed in two different clusters. According to the results, two of them are better than the other one which is sent database (tree) to the other processors. For both of them, the running time of the parallel algorithms is decreased by one forth by using two nodes. In addition to that, dynamical task parallelism is better than the static one in better algorithms.

**Keywords:** FP-Growth, Parallel Computing, Frequent Itemset Mining, Data Mining.

# TEŞEKKÜR

Çalışmalarım boyunca değerli yardım ve katkılarıyla beni yönlendiren tez danışmanım Yrd. Doç. Dr. Osman Abul'a, yine kıymetli tecrübelerinden faydalandığım Atılım Üniversitesi Yazılım Mühendisliği Bölümü öğretim üyesi Prof. Dr. Ali Yazıcı'ya, TOBB Ekonomi ve Teknoloji Üniversitesi Makina Mühendisliği Bölümü öğretim üyesi Yrd. Doç. Dr. Nilay Sezer Uzol'a, TOBB Ekonomi ve Teknoloji Üniversitesi Fizik Bölümü öğretim üyesi Prof. Dr. Turgut Baştuğ'a, çalışmam boyunca bana aktardığı tecrübeleri ve manevi destekten dolayı eşim ve aynı zamanda Çankaya Üniversitesi Bilgisayar Mühendisliği bölümü öğretim üyesi Doç. Dr. Cem Özdoğan'a ve hayatım boyunca bana vermiş oldukları manevi destekten dolayı aileme teşekkürü bir borç bilirim.

# İÇİNDEKİLER

# ÇİZELGELERİN LİSTESİ

**Çizelge** **Sayfa**

# ŞEKİLLERİN LİSTESİ

**Şekil** **Sayfa**

## KISALTMALAR

| Kısaltmalar | Açıklama |
|---|---|
| AMD | Advanced Micro Devices |
| ARM | Association Rule Mining (Birliktelik Kuralı Madenciliği) |
| FIM | Frequent Itemset Mining (Sık Ögekümesi Madenciliği) |
| FP-Growth | Frequent Pattern (Sık Örüntü) Growth |
| FP-Tree | Frequent Pattern Tree (Sık Örüntü Ağacı) |
| IBM | International Business Machines |
| MİB | Merkezi İşlem Birimi |
| MISD | Multiple Instruction Single Data |
| MIMD | Multiple Instruction Multiple Data |
| MLFPT | Multiple Local Frequent Pattern Tree |
| MPI | Message Passing Interface |
| NUMA | Non-Uniform Memory Access |
| PFPTC | Parallel FP-Tree Constructing |
| SISD | Single Instruction Single Data |
| SIMD | Single Instruction Multiple Data |
| SMP | Symmetric Multiprocessor |
| TOBB ETÜ | Türkiye Odalar ve Borsalar Birliği Ekonomi ve Teknoloji Üniversitesi |
| UMA | Uniform Memory Access |

## SEMBOL LİSTESİ

| Simgeler | Açıklama |
|---|---|
| B | Birliktelik Kuralı Madenciliği Problemi |
| D | Veritabanı (veri kümesi) |
| E | Verimlilik |
| F | Sık Öge Kümeleri |
| $f$ | Seri Kod Dilimi |
| I | Toplam Öge Kümesi |
| $i_d$ | d. Öge |
| p | İşlemci Sayısı |
| S(p) | Hızlanma Faktörü |
| T | Hareket |
| $t_s$ | Seri Çalışma Zamanı |
| $t_p$ | Paralel Çalışma Zamanı |
| X,Y | Öge kümeleri (k adet ögeden oluşan) |
| $\sigma$ | Destek Eşik Değeri |
| $\gamma$ | Güvenilirlik Derecesi Eşik Değeri |

# BÖLÜM 1

# GİRİŞ

Yakın bir geçmişe kadar üzerinde işlem yapılabilecek veriler sınırlıydı. Ancak, teknolojinin gelişmesiyle hem verinin miktarında hem de karmaşıklığında bir artış olmuştur. Bu da yeni teknolojileri gündeme getirmiştir. Çünkü veri olarak söz ettiğimiz şey aslında ham bilgiyi içerir. Bunun bize bir şey ifade edebilmesi için anlamlandırılması gerekir. Veri madenciliği tam olarak bu noktada devreye girer. Elde ham bir şekilde bulunan veriden anlamlandırılabilir ve kullanılabilir bilgi elde edilmesini sağlar. Bunun için temel olarak üç aşama mevcuttur. Elde olan veri ilk olarak bir ön işlemden geçirilir. Daha sonra bu veri üzerinde veri madenciliği yapılır. Son olarak ise, madencilik yapılan veri üzerinde bir son işlem yapılır. Bu şekilde yararlı bilgiye ulaşılmış olur.

Veri madenciliği, birçok disiplinin birleşmesinden oluşur. Bunlardan bazıları, istatistik, yapay zeka, makine öğrenme, örüntü tanıma, paralel hesaplama, dağıtık hesaplama ve veritabanı teknolojileridir [1].

Veri madenciliği günümüzde birçok alanda kullanılır. Bunlardan en yaygın bilineni pazarlama alanıdır. Müşterilerin en sık aldığı ürünler, müşteri profilleri, satış politikaları üzerinde veri madenciliği yapılmış satış veritabanından çıkarılabilir. Bunun dışında veri madenciliği bilimsel çalışmalarda, tıpta, finansal hesaplamalar gibi alanlarda da kullanılır.

Verilerin sayısında ve karmaşıklığında meydana gelen artış tek başına seri çözümlerin yeterli olmadığını göstermiştir. Bu durumda, mevcut veri madenciliği algoritmalarının paralelleştirilmesi yoluna gidilmiştir. Paralel uygulamalarla seri çözümlerin ulaşamayacağı sınırlar denenebilir. Bu şekilde zamandan tasarruf edilebilir.

Bu çalışmada, sık öge kümelerini bulmak için geliştirilen bir algoritma olan FP-Growth algoritmasının paralel uygulaması üzerinde çalışılmıştır. FP-Growth algoritması diğer sık öge küme madenciliği algoritmalarından daha verimli bir şekilde çalışır. Çünkü diğer birçok algoritma veritabanı üzerinde birçok kez tarama yapmasına rağmen FP-Growth algoritması sadece iki kez veritabanını tarar. Bu çalışma kapsamında, paralel FP-Growth için 3 farklı metot önerilmiştir. Bunlar (i) statik paralel öbek FP-Growth, (ii) dinamik paralel öbek FP-Growth ve (iii) ağaç gönderimli dinamik paralel öbek FP-Growth'dur. Literatürde geçen paralel FP-Growth algoritmaları incelendiğinde genelde hepsinin veri paralel yaklaşımı benimsediği görülmüştür. Bu çalışma onlardan farklı olarak görev paralel yaklaşım benimsenerek oluşturulmuştur. Önerilen ilk yaklaşım olan statik paralel öbek FP-Growth'da görev dağılımı statik bir şekilde yapılır. Buna ek olarak her işlemci üzerinde veritabanının mevcut olduğu düşünülür. Her bir işlemci belli sayıdaki sık öge için algoritmayı çalıştırarak sık öge kümelerini bulur. Dinamik paralel öbek FP-Growth'da da işlemcilerin tümünde veritabanı mevcuttur. Ancak buradaki fark, görev dağılımının dinamik gerçekleştirilmesidir. Yani, işlemcilerin görevlerini tamamladıkça yeni iş istemelerini sağlayarak bekleme sürelerini azaltmak hedeflenir. Son yaklaşım olan ağaç gönderimli dinamik paralel öbek FP-Growth'da da görev dağılımı dinamik gerçekleştirilir. Buradaki fark veritabanından kaynaklanır. Bu algoritmada yalnızca tek bir işlemci üzerinde veritabanının mevcut olduğu varsayılır. Günlük hayatta bazı durumlarda verinin tamamının herkes tarafından ulaşılır olması istenmez. Bu durumda veri yalnızca bir makine üzerinde tutularak o veri üzerinde işlem yapılır. Bu algoritma bu noktadan hareket edilerek ortaya çıkmıştır. Gerçekleştirilen uygulamalar iki farklı öbek *(cluster)* üzerinde test edilerek sonuçları analiz edilmiştir.

Bu doküman aşağıdaki gibi organize edilmiştir:
Bölüm 2'de paralel hesaplama anlatılacaktır. Paralel hesaplama içerisindeki temel kavramlar ve programların nasıl paralel yapılacağı üzerinde durulacaktır. Bölüm 3'de veri madenciliği görevlerinden biri olan birliktelikten, literatürde geçen birliktelik algoritmalarından ve bunların paralel uygulamalarından bahsedilecektir.

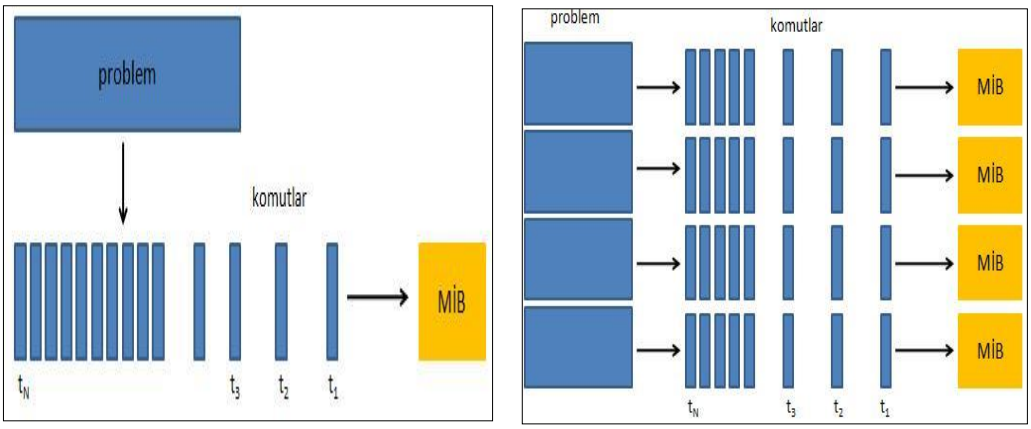
Bölüm 4’de seri FP-Growth algoritması anlatılacaktır. Aynı zamanda literatürde geçen paralel FP-Growth algoritmalarından bahsedilecektir. Bölüm 5’de ise bu çalışma içerisinde gerçekleştirilen üç farklı paralel FP-Growth algoritmasının uygulaması anlatılacak olup, geliştirme ve test ortamından bahsedilerek test sonuçları analiz edilecektir. Bölüm 6’de ise çalışma kapsamında geliştirilen üç yaklaşımın karşılaştırmalı sonuçları anlatılacaktır. Aynı zamanda gelecekteki çalışmalar için önerilere yer verilecektir.

# BÖLÜM 2

# PARALEL HESAPLAMA

Paralel hesaplama bir işlemin belli sayıda parçaya ayrılıp her bir parçanın birden fazla işlemci üzerinde aynı anda çalıştırılmasıdır. Özellikle bilimsel çalışmalarda büyük hesaplamalara ihtiyaç duyulmaktadır. Örneğin, tek bir makine üzerinde hava tahmini yapmak uzun ve zor bir iştir. Bu makinenin fiziksel özellikleri arttırıldığında daha iyi sonuç alınır. Ancak fiziksel özelliklerin ulaşacağı yerler sınırlıdır. Bundan dolayı başka çözümler aranmıştır. Sonuç olarak paralel hesaplama gündeme gelmiştir. Paralel hesaplamayla hız artarken çalışma zamanı azalır [2].

Şekil 2.1'de bir problemin seri ve paralel yaklaşımda nasıl çözüldüğü gösterilmiştir [3]. Şekil 2.1 a'da problem her bir zaman aralığında bir komut olmak üzere MİB (Merkezi İşlem Birimi)'ne gönderilir. Şekil 2.1 b'de ise problem önce belli sayıda parçaya ayrılır. Örneğin, burada problem dört parçaya ayrılmıştır. Bundan sonra her bir problem parçası bir zaman aralığında bir komut olmak üzere farklı MİB'ne gönderilir.




a) Seri hesaplama b) Paralel Hesaplama

Şekil 2. 1: Seri ve Paralel Hesaplama

## 2.1 Flynn Sınıflandırması

Paralel bilgisayarlar sınıflandırılırken, farklı parametreler kullanılarak farklı sınıflandırmalar yapılmıştır. Ancak, bunlardan en yaygın bilineni 1966'dan beri kullanılan ve Flynn Taksonomi olarak bilinenidir [3]. Michael J. Flynn, bu sınıflandırmayı yaparken parametre olarak komut *(instruction)* ve veriyi *(data)* seçmiştir. Bu bağlamda, Flynn Taksonomi'si şu şekildedir:

- SISD - Tek Komut Tek Veri (*Single Instruction, Single Data)*: Seri bilgisayardır. Tek komut tek veri üzerinde işlem yapar. Örnekleri, eski nesil anabilgisayarlar, mini bilgisayarlar, iş istasyonları ve kişisel bilgisayarlardır *(PC's)* (Şekil 2.2).

a) UNIVAC 1

b) IBM 360

Şekil 2. 2: SISD Örnekleri

- SIMD – Tek Komut Çoklu Veri *(Single Instruction, Multiple Data)*: Birden fazla veri kümesi üzerinde aynı işlemi (tek komut işlemi) yapan bilgisayarlardır. Sinyal işleme uygulamaları bu şekilde çalışır. Örnekleri, ILLIAC IV, Cray X-MP, Y-MP'dir (Şekil 2.3).

a) ILLIAC IV b) Cray Y-MP

Şekil 2. 3: SIMD Örnekleri

- MISD – Çoklu Komut Tek Veri *(Multiple Instruction, Single Data)*: Çok az örneği vardır. Bunlardan biri deneysel Carnegie-Mellon C.mmp bilgisayarıdır.

- MIMD – Çoklu Komut Çoklu Veri *(Multiple Instruction, Multiple Data)*: Günümüzde birçok bilgisayar bu türdendir. Örnekleri, yeni süper bilgisayarlar, bilgisayar öbekleri*(clusters)*, grid sistemler ve çok işlemcili bilgisayarlardır (Şekil 2.4).

a) IBM POWER5

b) HP/Compaq Alphaserver

c) INTEL IA32

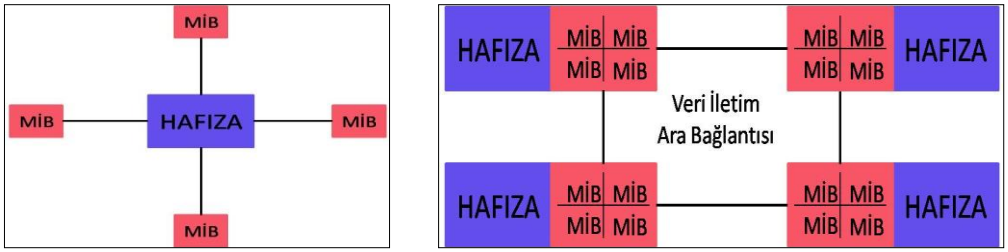
d) AMD Opteron

Şekil 2. 4: MIMD Örnekleri

## 2.2 Paralel Bilgisayar Hafıza Yapısı

Paralel bilgisayar hafıza yapısı genel olarak üç şekilde yapılandırılır [3]. Bunlar, paylaşımlı, dağıtık, melez paylaşımlı-dağıtık yapılardır. Paylaşımlı bellek yapıda işlemciler bağımsız olarak çalışabilir, ancak ortak hafızayı paylaşırlar. Paylaşımlı

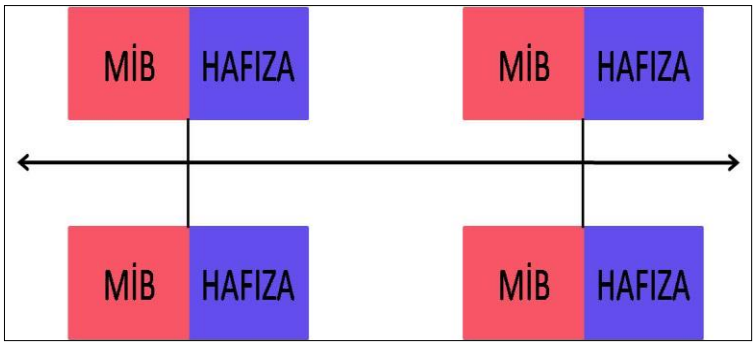
bellek makineler iki temel sınıfa ayrılır: UMA *(Uniform Memory Access)* ve NUMA *(Non-Uniform Memory Access)*'dır. UMA makinelerde tamamen eş işlemciler kullanılır (Şekil 2.5 a). Dolayısıyla hem hafızaya erişmeleri hem de erişim süreleri eşittir. NUMA ise iki ya da daha fazla SMP'nin *(Symmetric Multiprocessor)* fiziksel olarak birbirine bağlanmasıyla oluşturulur (Şekil 2.5 b). Bu durumda herhangi bir SMP diğerinin hafızasına direk olarak ulaşabilir. Bu yapıda hafızaya erişim diğerine göre daha yavaştır.




a)Paylaşımlı Bellek (UMA) b)Paylaşımlı Bellek (NUMA)

Şekil 2. 5: Paylaşımlı Bellek Şematik Gösterim

Dağıtık bellekli sistemlerde ise her makine kendi yerel hafızasına sahiptir ve makineler birbirlerine bir iletişim ağıyla bağlıdır (Şekil 2.6). Hepsi bağımsız olarak işlem yapabilir. Yerel hafızada yapılan değişiklikler diğerlerini etkilemez. Dağıtık bellekli sistemde düşünülmesi gereken şey, bir işlemcinin diğer işlemci üzerindeki veriye ihtiyaç duyabileceğidir. Bunun için çözüm mesaj iletimidir. Dağıtık bellekli sistemin avantajı, işlemci sayısı arttırılarak kullanıcı için gerekli olan hafıza miktarının arttırılabilmesidir.




Şekil 2. 6: Dağıtık Bellek Şematik Gösterim

Melez dağıtık-paylaşımlı sistemler ise günümüzdeki en büyük ve en hızlı sistemler (süper bilgisayarlar) için tasarlanmıştır (Şekil 2.7).


Şekil 2. 7: Melez Dağıtık-Paylaşımlı Bellek Şematik Gösterim

## 2.3 Dağıtık Hesaplama

Dağıtık hesaplama, büyük hesaplama problemlerinin parçalara ayrılıp, her parçanın birbirlerine bir bilgisayar ağıyla bağlı olan bir sistemdeki makineler üzerinde çözülmesidir [4]. Dağıtık bir sistemde her makinenin kendi yerel hafızası vardır. Bölüm 2.4 ve 2.5'de anlatılan öbek hesaplama ve grid hesaplama da dağıtık hesaplamadır.

## 2.4 Öbek Hesaplama

Öbek bilgisayarlar, normalden daha iyi bir performans verebilmek için birbirine yakın olan bilgisayarların bir araya getirilmesiyle oluşturulan sistemlerdir. Yerel ağ içerisindeki bilgisayarlar birbirine bağlanarak tek bir makineymiş gibi davranırlar. Bu şekilde, tek bir makine üzerinde gerçekleştirilemeyecek işlemler çalışılan kodun paralelleştirilmesiyle öbek bilgisayarlar üzerinde kolaylıkla gerçekleştirilebilir.

Öbek bilgisayarlar şu şekilde sınıflandırılır [5]:

- Yüksek Yararlanılabilir Öbekler (*High Availability Clusters*): Bunlar, öbeklerden sağlanan yararı arttırmak için oluşturulmuşlardır. Bu tür öbeklerde, bazı sistem bileşenleri çöktüğünde devreye boş düğümler *(nodes)* girer.
- Yük Dengeleyici Öbekler (*Load-balancing Clusters*): Performansı arttırmak için oluşturulmuşlardır. Bu öbekler, bir veya daha fazla yük dengeleyici ön uçtan gelen iş yükünü arka uç sunucularına dağıtırlar.
- Yüksek Performanslı Öbekler (*Compute Clusters*): Bu öbekler, düğümlerden gelen işi bölümlere ayırarak performans artışı sağlarlar. En bilinen yüksek performanslı öbek, Beowulf öbeğidir (Şekil 2.8).

Şekil 2. 8: Beowulf Öbeği

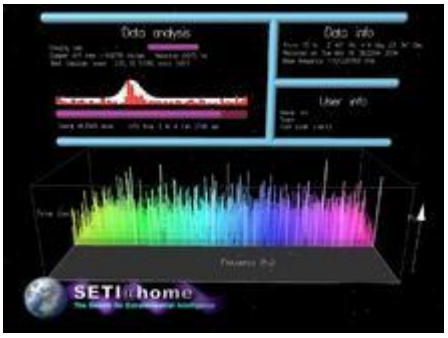
- Grid Hesaplama (*Grid Computing*): Detaylı olarak bölüm 2.5'de anlatılacaktır.

## 2.5 Grid Hesaplama

Grid, dünyanın farklı yerlerinde bulunan küme bilgisayarların, süper bilgisayarların veya kişisel bilgisayarların birbirlerine ağ ile bağlanması ile yüksek performanslı hesaplama yapabilmek için oluşturulmuş yapıya verilen isimdir [6]. Öbek bilgisayarlardan temel farkı coğrafi olarak dağıtık olması ve heterojen bir yapıda olmasıdır. Bu sistem ile bilgisayarlar, yazılımlar, veritabanları paylaşılabilir. Grid kullanımıyla daha büyük ölçekli problemler çözülebilir. Kaynaklar daha verimli bir şekilde kullanılarak, araştırmacılar için daha etkin bir çalışma ortamı oluşturulur.

Grid yapılarda kullanıcı yapacağı işi sisteme gönderir. İşlemin yapılabilmesi için uygun kaynak seçilir. Daha sonra sonuç kullanıcıya gönderilir. Burada amaç tek bir sistem görünümü vermektir. Grid, verilerle dosya seviyesinde ilgilenir ve ayrıca veri yapılarıyla ilgilenmez. Depolama kaynaklarında saklanan grid dosyaları salt-okunur dosyalardır.

Grid sistemler için geliştirilmiş bazı araçlar ve uygulamalar vardır. En bilinen uygulamalardan biri SETI@home'dur [7] (Şekil 2.9). GridMiner [8] sistemi ise grid üzerinde veri madenciliği için geliştirilmiş bir yazılımdır.




Şekil 2. 9: Seti@home BOINC istemcisi (sürüm: 4.45)

Türkiye'de grid ile ilgili çalışmalar ULAKBİM tarafından 2003 yılında TR-Grid adı altında başlamıştır [6]. TR-Grid'in amaçları ulusal grid altyapısını kurmak ve kullanıcı kitlesini bilgilendirmek, çeşitli bölgesel uygulamalar geliştirmek, uluslararası grid projelerinde aktif görev almak olarak sayılabilir.

## 2.6 Paralel Programlama Modelleri

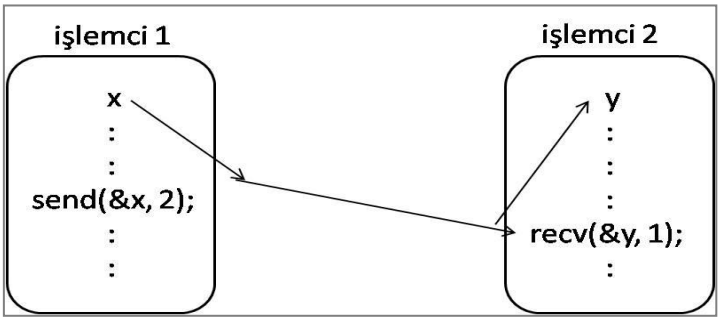
Genel olarak paralel programlama modelleri şu şekildedir [3]:

- Paylaşımlı Bellek Modeli (*Shared Memory Model*): Bu modelde görevler ortak bir hafıza alanını paylaşırlar. Burada eş zamanlı olarak verilere ulaşmada karşılaşılacak sorunları engellemek için bazı yapılar kullanılır (semafor, kilit vb.)
- İş Parçacığı Modeli (*Threads Model*): Bu modelde bir program önce seri olarak çalışmaya başlayıp daha sonra iş parçacıkları yaratılarak eş zamanlı olarak çalışmaya devam edebilir. Her iş parçacığı kendi yerel verisine sahiptir.
- İleti Geçme Modeli (*Message Passing Model*): Burada gerçekleştirilecek iş parçalara ayrılarak düğümlere dağıtılır. Her düğüm kendi işi için gerekli olan verileri ileti geçme *(send-receive)* sayesinde alır. Bu iletim esnasında bir gönderici ve bir alıcı olması gerekir. Burada düşünülmesi gereken şey, verinin doğru olarak alınıp alınmadığıdır.
- Veri Paralel Model (*Data Parallel Model*): Bu modelde her düğüm işlenecek verinin bir kısmı üzerinde kendi görevini gerçekleştirir. Dolayısıyla bir mesaj iletimi söz konusu değildir.
- Melez Model (*Hybrid Model*): Bu modelde herhangi iki ya da daha fazla paralel programlama modeli birlikte çalışır.

## 2.7 MPI

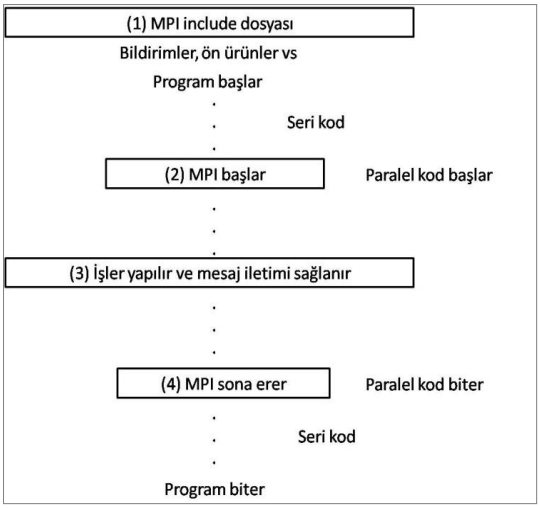
MPI (*Message Passing Interface*) 1993-94 yıllarında bir grup araştırmacı tarafından geliştirilmiş bir dizi fonksiyon ve makrodan ibaret bir yazılım kütüphanesidir. MPI kütüphanesine çeşitli diller aracılığı ile erişmek mümkündür (C, Fortran, C++ gibi). MPI'ı diğer eski mesaj geçirmeli kütüphanelerden ayıran en önemli özellik taşınabilirlik ve kolay kullanılabilir olmasıdır.

MPI "ileti geçme / gönder-al (message passing / *send-receive*)" mantığına dayalıdır (Şekil 2.10). Değişik işlemciler, birbirleriyle iletiler sayesinde haberleşirler [2].




Şekil 2. 10: MPI İleti Geçme Şeması

MPI kütüphanesi oldukça fazla sayıda fonksiyona sahip olmasına rağmen, paralel çalışabilecek bir program yazmak için az sayıda temel MPI fonksiyonu yeterlidir. Genel olarak MPI'ın program yapısı Şekil 2.11'de verildiği gibidir [9].




Şekil 2.11: MPI Genel Program Yapısı

MPI ile paralel bir kod yazabilmek için gerekli bazı temel fonksiyonlar aşağıda verilmiştir:

- MPI_Init: Herhangi bir MPI fonksiyonu kullanılmadan önce çağrılmalıdır. (Şekil 2.11: (2))
- MPI_Finalize: MPI fonksiyonları çağrıldıktan sonra bu metot çağrılarak MPI'ın sonlandığı belirtilmelidir. (Şekil 2.11: (4))
- MPI_Comm_rank: MPI içerisinde her düğüme bir numara verilir. Aynı zamanda her düğüm işçi ve yönetici düğümler olarak sınıflandırılır. Atanan bu numaralarla her düğümün hangi sınıfa dahil olduğu belirlenir. (Şekil 2.11: (2) ve (3) arası)
- MPI_Comm_size: Toplam düğüm sayısını verir. (Şekil 2.11: (2) ve (3) arası)
- MPI_Send: Mesaj gönderimini sağlayan metottur. İçinde, gönderdiği mesaj, mesajın uzunluğu, türü ve göndereceği yer bilgileri tutulur. Aynı zamanda, alıcıyla haberleşmeyi sağlayan bir mesaj etiketi bilgisi tutulur. Örneğin, gönderici tarafında mesaj etiketi "10" ise alıcı tarafında da aynı mesaj için mesaj etiketi "10" olmalıdır.

- MPI_Recv: Mesaj alımını sağlayan metottur. İçinde MPI_Send metodunun ilettiği bilgilerin aynısı mevcuttur. Bunlara ek olarak, MPI önceden tanımlı bir durum değişkeni tutar. Bu değişken, mesaj ile ilgili bilgileri tutar.

Paralel bir kod yazarken haberleşmeyi mümkün olduğu kadar az yapmak gerekir. Eğer çok haberleşme gerekiyorsa haberleşmelerin peşpeşe yapılması daha verimli olur. Çünkü verinin iletiminde soket açma zamanı iletim zamanına göre oldukça uzun süreye sahiptir. Bir paralel algoritmanın çalışma süresi, hesaplama ve haberleşme zamanının toplamı olarak ifade edilir. Bir algoritmada çok yoğun haberleşme olacağı gibi çok az ya da hiç haberleşme olmayabilir. Haberleşmenin çok yoğun olduğu algoritmalarda haberleşme zamanı hesaplama zamanına baskın (*overhead*) hale gelebilir. Bu, kaçınılması gereken bir durumdur. Özet olarak, haberleşme zamanının mümkün olduğu kadar kısa tutulması ve işlemcilerin devamlı hesap yapar halde tutulmaları verimli bir algoritmayı ortaya çıkaracaktır.

Şekil 2.12, örnek bir MPI programı içermektedir. Bu kod, bir dizi toplamı yapmaktadır. Yönetici işlemci diziyi oluşturup diğer işlemcilere gönderir. İşçi işlemciler ise öncelikle gönderilen diziyi alır. Daha sonra, her bir işlemci dizinin hangi parçası üzerinde işlem yapacağını bulur. Her işçi işlemci kendi parçası üzerinde toplama işlemi yaparak sonucu yönetici işlemciye gönderir. Son olarak, yönetici işlemci işçi işlemcilerden gelen ara toplamları alarak toplar ve tam sonucu bulur.

```c
/*DIZI TOPLAMI YAPAN BASIT BIR MPI KODU
*Dizinin icindeki sayilar 1'den baslayip 1000 ile biter
*0. islemci diziyi olusturur ve diziyi diger islemcilere gonderir
*Diger islemciler diziyi alirlar
*0 haricinde kalan islemciler dizinin kendilerine ayrilmis kisminin parca toplamini yapar
*Parca toplamlari 0. islemciye gonderilir.
*0. islemci tum parca toplamlarini kendi icinde toplayarak sonuc degiskenine atar.
*/
#include<stdio.h>
#include<stdlib.h>
#include "mpi.h"
int main(int argc, char *argv[]){
int *dizi;
int islemciNo,islemciSayisi,i,j,k,toplam=0,parcaTop=0,toplanacakSayi,diziLength;
    MPI_Init(&argc, &argv);      //MPI baslar
    MPI_Comm_rank(MPI_COMM_WORLD, &islemciNo);  //Islemciye Atanan Numaranın Tespiti
    MPI_Comm_size(MPI_COMM_WORLD, &islemciSayisi);  //Toplam Islemci Sayisini Bulma
    MPI_Status status1, status2, status3;
    if (islemciNo == 0){
        diziLength = 1000;
        dizi = malloc(sizeof(int) * diziLength);     //Dizi icin Hafizada Yer Acma
        for (i = 0; i < diziLength; i++)     //Diziyi Olusturma
            dizi[i] = i+1;
        for (j = 1; j < islemciSayisi; j++){
            MPI_Send(&diziLength, 1, MPI_INT, j, 10, MPI_COMM_WORLD); //Dizi Uzunlugunu Gonderme
            MPI_Send(dizi, diziLength, MPI_INT, j, 20, MPI_COMM_WORLD); //Diziyi Gonderme
        }
    }
    else{
        MPI_Recv(&diziLength, 1, MPI_INT, 0, 10, MPI_COMM_WORLD, &status1); //Dizi Uzunlugunu Alma
        dizi = malloc(sizeof(int) * diziLength); //Dizi icin Hafizada Yer Acma
        MPI_Recv(dizi, diziLength, MPI_INT, 0, 20, MPI_COMM_WORLD, &status2); //Dizi alma
        toplanacakSayi = diziLength/(islemciSayisi-1); //Toplancak Sayi Bulma Islemi
        for (j = (islemciNo-1)*toplanacakSayi; j < toplanacakSayi*(islemciNo); j++)
            parcaTop += dizi[j];     //Parca Toplama Islemi
        //Dizinin Uzunluğu Toplama Yapacak Islemci Sayisinin Tam Kati Olmadigi Durumda
        if (((diziLength % (islemciSayisi-1)) != 0) && (islemciNo == (islemciSayisi - 1)))
            for (k = (diziLength - (diziLength % (islemciSayisi - 1))); k < diziLength; k++)
                parcaTop += dizi[k];
        printf("islemciNo%d uzerindeki parca toplam: %d\n", islemciNo, parcaTop);
        fflush(stdout); //Parca Toplamlarin Ekranda Gosterilmesi
        MPI_Send(&parcaTop, 1, MPI_INT, 0, 30, MPI_COMM_WORLD); //Parca Toplami Gonderme Islemi
    }
    if (islemciNo == 0){
        for (k = 1; k < islemciSayisi; k++){ //Parca Toplamlari Birlestirme Islemi
            MPI_Recv(&parcaTop, 1, MPI_INT, k, 30, MPI_COMM_WORLD, &status3);
            toplam += parcaTop;
        }
        printf("Dizi toplami: %d\n", toplam);
        fflush(stdout);  //Genel Toplami Ekranda Gosterme
    }
    free(dizi);
    MPI_Finalize();
    return 0;
}
```

Şekil 2.12: MPI Kod Örneği (C Dilinde)

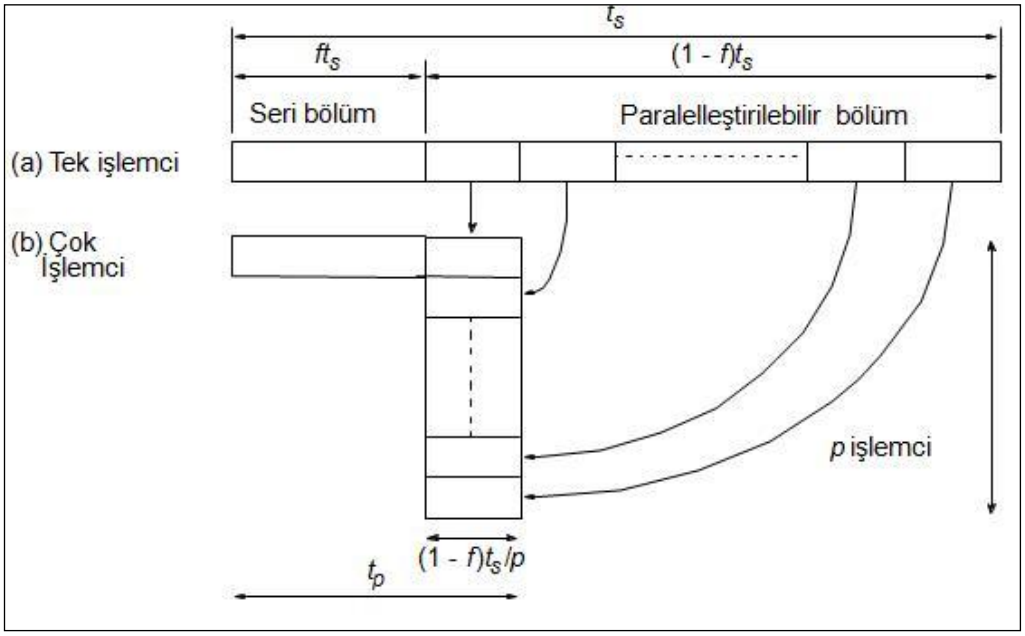
## 2.8 Paralel Hesaplama Ölçütleri

Paralel hesaplamada elde edilen kazanımı gösteren bazı ölçütler vardır. Bunlar hızlanma faktörü *(speedup)* ve verimliliktir *(efficiency)*. Hızlanma faktörü çok işlemci kullanıldığında hızdaki artışı verir. Bunun için seri algoritmanın zamanı bulunur. Bu zaman, seri kod için önerilen en iyi algoritmayla elde edilmiş zaman olmalıdır. Aynı şekilde paralel hesaplama için gerekli olan zaman da bulunduktan sonra hızlanma faktörü, *S(p)*, şöyle ifade edilir [2]:

$$S(p) = \frac{t_s}{t_p} \tag{2.1}$$

Burada, $p$ işlemci sayısını, $t_s$ seri çalışma zamanını, $t_p$ ise paralel çalışma zamanını göstermektedir.

Hızlanma faktörü bazı etkenlerden dolayı belli bir üst sınıra ulaşmaktadır. Bunlar, işlemciler arasındaki haberleşme zamanı, seri algoritmada olmayıp sadece paralel algoritmada olan ek hesaplamalar ve bazen işlemcilerin iş yapamadığı ve boş olduğu zamanlardan kaynaklanabilir. Problemlerin çözümünde genelde tüm hesaplamalar paralelleştirilemez. Bir kısım sadece seri olarak çalışabilir. Amdahl seri bir kodun içerisindeki paralelleştirilemeyecek, seri kalması gereken kod kısımlarının hızlanma faktörü üzerindeki etkisini açıklamıştır. Bu açıklama Amdahl Yasası olarak bilinmektedir. Şekil 2.13 bu yasayı daha açık bir biçimde göstermektedir [2].



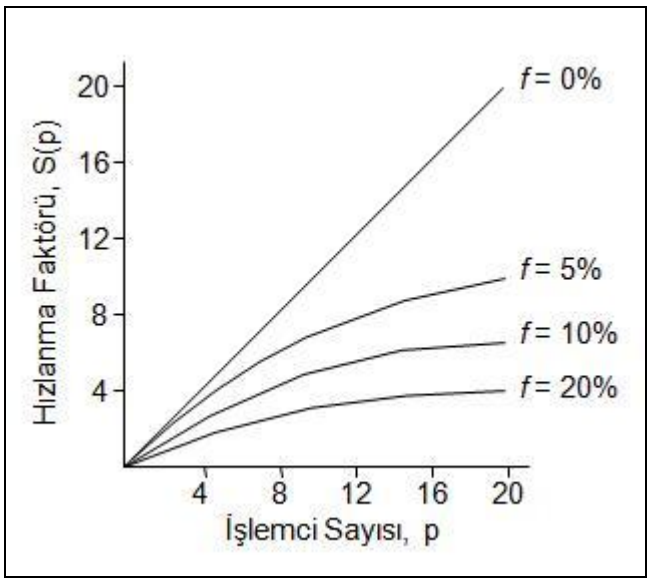

Şekil 2. 13: Amdahl Yasasının Şematik Gösterimi

Matematiksel olarak ise Amdahl Yasası şu şekilde ifade edilmektedir:

$$S(p) = \frac{t_s}{ft_s + (1-f)t_s / p} = \frac{p}{1+(p-1)f} \tag{2.2}$$

Buradaki $t_s$ seri çalışma zamanını, $f$ seri kod dilimi (paralelleştirilemeyen bölüm), $p$ işlemci sayısını göstermektedir.

Amdahl tarafından yukarıda açıklanan yasanın hızlanma faktörüne etkisi Şekil 2.14' de gösterilmektedir [2].




Şekil 2. 14: Amdahl Yasasıyla Hızlanma Faktörü

Paralel hesaplamada elde edilen kazanımı gösteren diğer bir ölçüt ise verimliliktir. Verimlilik, hesaplamaya katılan işlemcilerin performansını (kullanım miktarını) anlamaya yarayan bir ölçüttür. Verimlilik, *E*, aşağıdaki gibi hesaplanır. Burada *S(p)* hızlanma faktörünü, *p* ise işlemci sayısını göstermektedir.

$$E = \frac{S(p)}{p} \tag{2.3}$$

Bu kavramla üzerinde çalışılan problemin çözümü için gereken en iyi işlemci sayısı bulunabilir. Sisteme katılan her yeni işlemci çalışma süresini göreceli olarak düşürse bile, bu yeni işlemciyi sisteme katma kararını azalan süre ile değil verimlilik kavramı ile vermek gerekir. Örneğin, paralel olarak kodlanan bir programın tek işlemcide, 4 işlemcide ve 6 işlemcideki çalışma zamanları sırayla şöyle olsun: 120, 100 ve 80 sn. 4 işlemci üzerindeki hızlanma faktörü 1.2 , 6 işlemcideki hızlanma faktörü ise 1.5 olur. Buna göre 4 işlemcideki verimlilik 0.6 olurken 6 işlemcideki verimlilik ise 0.25 olur. Bu durumda 4 işlemci tercih edilmelidir. Çünkü 6 işlemci kullanıldığında sürede kazanç olmasına rağmen verim düşer. Bu da yeni işlemciler katılmasına rağmen daha önce bahsedilen etkenlerden dolayı, hızlanma faktörünün en iyi şekilde artmadığını gösterir.

Paralel hesaplamada bazen S(p)>p olabilir. Bu durumda süper lineer hızlanma faktörü (*superlinear speedup*) ortaya çıkar. Süper lineer hızlanma faktörünün sebepleri, sistemdeki fazla hafıza, verinin önbellekte bulunması olabilir. Süper lineer hızlanma (aşırı hızlanma) durumu verimlilik grafiklerinde de aşırı verimlilik olarak görülmektedir.

Hızlanma faktörü ve verimlilik dışında paralel hesaplamada ölçeklenebilirlik (*scalability*) kavramı mevcuttur. Ölçeklenebilirlik, bir problemin çözümü için kullanılan işlemci sayısının artmasına bağlı olarak performans artışını göstermek için kullanılır. Bunun dışında çalışma zamanı ve hafıza kullanımı da paralel hesaplama içinde incelenen parametrelerdir.

# BÖLÜM 3

# VERİ MADENCİLİĞİ GÖREVLERİ

Veri madenciliği, büyük veri ambarlarından yararlı bilgi keşfedilmesi sürecidir. Veri madenciliğinin temel olarak üç ana görevi vardır. Bunlar sınıflama *(classification)*, öbekleme *(clustering)* ve birliktelik *(association)*'tir [10].

Sınıflamada amaç yeni bir nesnenin özellikleri kapsamında önceden belirlenmiş bir sınıfa atanmasıdır. Sınıflama modelinin belirlenmesinde bazı gereksinimler vardır. Bunlardan ilki, modelin elde edilmesinde kullanılan veri kümesidir. Bu küme eğitim veri kümesi *(training set)* olarak nitelendirilir. Buna ek olarak, modeli doğrulamak için kullanılan veri kümesine doğrulama veri kümesi *(validation set)* ve modelin kontrolünü gerçekleştiren veri kümesine ise test veri kümesi *(test set)* denir. Sınıflama probleminde verilen eğitim kümesi kullanılarak bir sınıflama modeli bulunması (*model induction*) istenir. Belirlenen model sınıfından, eğitim veri kümesi kullanılarak öğrenilen modele sınıflayıcı (*classifier*) denir [10].

Veri madenciliğinin başka bir görevi ise öbeklemedir. Öbekleme algoritmaları veri kümesini alt kümelere ayırır. Her bir altkümede yer alan nesneler dahil oldukları grubu diğer gruplardan ayıran ortak özelliklere sahiptir [11, 12]. Öbeklemede, öncelikle olası sınıflar veriler kullanılarak oluşturulur ve daha sonra nesneler bu sınıflara atanır. Olası sınıflar belirlenirken bir grup nesnenin yoğun olarak birbirine yakın olduğu bölgeler seçilerek, bu bölgeler öbek olarak değerlendirilir. Bu yüzden öbekleme sınıflamadan daha zor bir süreçtir. Öbekleme genellikle biyoloji, meteoroloji, psikoloji, tıp bilimleri ve iş dünyasında kullanılır [1].

Veri madenciliğinin diğer bir görevi olan birliktelik ise, bir ilişkide özniteliklerin aldığı değerler arasındaki bağımlılıkları bulur. En bilinen birliktelik uygulaması market - sepeti verisi (*market-basket*) [13] uygulamasıdır.

Birliktelik problemi temel olarak, ögelerin sık olarak beraber satın alındığı ile ilgilidir. Örneğin, bir marketteki satışların %10'unda gazete ve sütün beraber alındığı görülürse bu iki ürün arasında satış kapsamında bir ilişki olduğu düşünülür. Bu şekilde bir bilgi, pazarlama ve rekabet açısından son derece yararlı bir bilgidir. Mesela, bu bilgi kapsamında bu marketin rafları yenilenerek, beraberce çok satılan bu iki ürün yakın raflara konarak bu ürünlerin satışı daha da arttırılabilir. Pazarlama ve rekabet dışında birliktelik, biyoenformatik, tıp, web madenciliği ve bilimsel veri analizi gibi alanlarda da sıklıkla kullanılır [1].

Birliktelikte temel alınan nokta, hangi malların hangi sıklıkta alındığıdır. Bu sıklık değerine destek (*support*) adı verilir. Belli bir destek eşik değerinin (*support threshold*) üzerinde desteğe sahip olan öge kümelerine (*itemset*) sık ögekümesi *(frequent itemset)* denir. Sık öge kümesi madenciliği (*frequent itemset mining, FIM*) problemi ise bu koşulu sağlayan tüm öge kümelerinin bulunmasıdır [10].

Destek değeri yanında diğer bir ölçü ise eminlik (*confidence*)'tir. Bir kuralın eminlik değeri belli bir eşik değerinden büyükse, kural önemli olarak adlandırılır. Bir veritabanındaki tüm sık ve önemli birliktelik kurallarının bulunması problemi ise birliktelik kuralı madenciliği (*association rule mining, ARM*) problemidir [10].

### 3.1 Birlikteliğin Matematiksel Modeli

Birliktelik içindeki kavramların tanımları matematiksel bir model içerisinde açıklanmıştır. Bu model [13, 1]'de incelenerek şu şekilde özetlenebilir:

Birliktelik içinde geçen öge kümesi $I=\{i_1, i_2, \ldots, i_d\}$ ve hareket kümesi ise $T=\{t_1, t_2, \ldots, t_N\}$ olacak şekilde ifade edilir [1]. Her bir hareket (*transaction*), öge kümesi içinden seçilen ögelerden oluşur. Tüm hareketler birleşerek bir öge kümesi veritabanını (D) oluştururlar. Birliktelik modeli içerisinde sıfır ya da daha fazla ögeden oluşan küme k-öge kümesi olarak adlandırılır. Örneğin, {Ekmek, Süt, Peynir, Gazete} 4-öge kümesi olarak isimlendirilir.

Her bir hareket bir öge kümesini (X) ancak ve ancak bu küme hareket $t_i$'nin alt kümesi olduğu sürece içerir. Bu durumda $t_i$ hareketi X öge kümesini destekler denir. Buna göre destek değeri şu şekilde ifade edilir [1]:

$$\sup_D(X) = |\{t_i \mid X \subseteq t_i, t_i \in T\}| \tag{3.1}$$

Her veritabanı için bir destek eşik değeri tanımlanır ve σ olarak gösterilir. D veritabanı için σ destek değerindeki sık öge kümesi şu şekilde ifade edilir [13].

$$F(D,\ \sigma) = \{X : X \subseteq I \text{ ve } \sup_D(X) \geq \sigma\} \tag{3.2}$$

Burada sık öge kümeleri (F), I kümesinin tüm alt kümelerinin hesaplanmasıyla bulunabilir. Ancak, göz önünde bulundurulması gereken şey, herhangi bir $X \subseteq I$ için $X \notin F(D, \sigma)$ sağlanıyorsa, $X \subseteq Y \subseteq I$ özelliğini sağlayan bir Y öge kümesinin sık olmasının mümkün olmadığıdır (Apriori özelliği) [10].

Birliktelik içinde amaç birliktelik kurallarını bulmaktır. Bir birliktelik kuralı şu şekilde ifade edilir [13]:

$$X \subseteq I, Y \subseteq I \text{ ve } X \cap Y = \varnothing, \quad X \Rightarrow Y \tag{3.3}$$

Bu kuralın güçlülüğünü ifade eden ölçütler destek değeri ve eminlik derecesidir. Eminlik derecesi şu şekilde ifade edilir [1]:

$$conf_D(X \Rightarrow Y) = \sup(X \cup Y) / \sup(X) \tag{3.4}$$

Birliktelikte bir kuralın hem destek değerinin hem de eminlik değerinin belirtilen eşik değerlerinden yüksek olması istenir. Destek değeri ve eminlik değerinin belirlenen destek eşik ($\sigma$) ve eminlik eşik ($\gamma$) değerlerinden yüksek olan tüm

kuralların bulunması problemi birliktelik kuralları madenleme problemidir. Problem şu şekilde formüle edilir [13]:

$$B(D,\sigma,\gamma)=\{X \Rightarrow Y : X \subseteq I, Y \subseteq I, X \cap Y = \varnothing, \sup_D(X \cup Y) \geq \sigma \text{ ve } conf_D(X \Rightarrow Y) \geq \gamma\} \quad (3.5)$$

Literatürde birliktelik için önerilen seri ve paralel algoritmalar mevcuttur. Bunlardan seri birliktelik algoritmaları Bölüm 3.2'de, paralel birliktelik algoritmaları ise Bölüm 3.3'de anlatılacaktır.

## 3.2 Seri Algoritmalar

Birliktelik algoritmaları genel olarak iki aşamada gerçekleşir. İlk aşamada, belirlenen destek değerine göre sık öge kümeleri bulunur, daha sonra sık öge kümeleri içinden önemli kurallar çıkarılır [10].

Literatürde birliktelik için birçok algoritma geliştirilmiştir. Bunlar arasında en bilinenleri AIS [13], Apriori [14], DHP [15], Partition [16] dır. Birliktelik algoritmaları içinde en sık kullanılan Apriori ve Apriori tabanlı algoritmalardır. Apriori algoritması özünde veritabanının sürekli taranmasını gerektirir. Örneğin, 1 elemanlı sık öge kümeleri bulunurken veritabanı bir kez taranır. Daha sonra 1 elemanlılar kullanılarak 2 elemanlılar oluşturulur ve onların sıklığını bulmak için veritabanı bir kez daha taranır. Dolayısıyla çok verimli değildir.

Birliktelikte amaç sık ve önemli kuralları bulmaktır. Bazen sadece sık kurallar bulunmak istenir. Veri madenciliğinde sık kuralların bulunması problemi sık öge küme madenciliğidir. Sık öge kümesi madenciliğiyle ilgili olarak Apriori benzeri yaklaşımlardan farklı olarak FP-Growth adında bir algoritma geliştirilmiştir [17]. FP-Growth temelde sık öge kümelerini hızlı bir şekilde bulmayı hedefler. Bu algoritma, bu amaçla geliştirilen algoritmalar içinde en verimli olarak düşünülebilir. Algoritmanın kendisi çalışmamıza da temel oluşturduğu için ayrıntılı olarak Bölüm 4.1'de anlatılacaktır.

## 3.3 Paralel Algoritmalar

Günümüzde, verinin büyümesi sonucu problemlerin çözümüne paralel yaklaşımlar sunulmuştur. Birçok paralel algoritma geliştirilmiştir. Bu algoritmaların amacı, seri kodun çözemediği veriler üzerinde çalışıp çözümler bulmaktır. Bunun yanında, bu algoritmalar serinin çözebildiği problemler için de daha kısa zamanda çalışan daha etkili çözümler sunmayı hedeflerler. Paralel çalışmalar içerisinde, temel olarak çalışma zamanı incelenir. Bunların dışında hafıza kullanımı, ölçeklenebilirlik *(scalability)*, hızlanma faktörü *(speedup)* ve verimlilik *(efficiency)* ölçütleri de incelenir.

Literatürde birliktelik için önerilen dağıtık algoritmaların büyük bir kısmı Apriori algoritması tabanlıdır. Burada paralellik olarak esas alınan şey verinin bölünmesidir. Veriler bölünerek işlemcilere dağıtılır. Daha sonra tüm işlemciler yerel verisinde 1-öge kümesini ve destek değerlerini hesaplar ve bunu diğer işlemcilere gönderir. Gelen tüm 1-öge kümesi ve destek değerleri birleştirilerek global 1-öge kümesini tüm işlemciler eş zamanlı oluşturur. Destek eşik değeri kullanılarak, sık olmayanlar elenir. Süreç tüm olası ögeler için tekrarlanır, 2-öge kümesi, 3-öge kümesi vb. [18]. Verinin bölünmesi dışında bu çalışma içerisinde iki farklı paralel yaklaşım daha önerilmiştir. Bunlardan ilki, bağımsız arama *(independent search)* olarak adlandırılan verinin tüm işlemcilerde mevcut olduğu ancak her işlemcinin verinin belli bir kısmı üzerinde işlem yaptığı yaklaşımdır. İkinci yaklaşım ise seri bir algoritmanın paralelleştirilmesidir.

Özel mimariler (örn, ortak hafıza) için özel algoritmalar da geliştirilmiştir [19, 20]. [20] çalışmasında "JavaSpaces" teknolojisi (Linda hesaplama yaklaşımı benzeri bir teknoloji) kullanılarak paralel ve dağıtık veri madenciliği üzerinde çalışılmıştır.

Burada toplam destek ağacı *(total support tree, T-tree)* denilen yeni bir veri yapısı kullanan Apriori-T algoritmasının nasıl paralelleştirileceği konusunda farklı yaklaşımlarda bulunulmuştur. Temel yaklaşım, verinin işlemcilere nasıl

bölüneceğidir. JavaSpaces ortamı kullanılarak bu yaklaşımların performans analizi gerçekleştirilmiştir. Sonuçta, aday dağıtım yaklaşımı diğerlerine göre daha az mesajlaşma içerdiğinden iyi bir performans göstermiştir [10]. Başka bir paralel çalışma ise [21]'de verilmiştir. Burada, 100 adet ATM anahtar ile birbirine bağlı PC içeren bir öbek bilgisayarda yapılan Apriori algoritması çalışılmıştır.

Paralel birliktelik algoritmaları, çok sayıda aday k-öge kümesinin desteğini paralel olarak sayarak sayma süresini kısaltırlar. Yani, k-öge kümesi kafesi (*lattice*) üzerindeki adaylar işlemci sayısına göre ayrık kümelere ayrılır ve herbirinin destek değerleri veritabanında paralel ve birbirinden bağımsız olarak sayılır. Bu yaklaşım görev-paralel hesaplamaya örnektir [19, 18].

Apriori tabanlı algoritmalardan farklı olan ve bu algoritmalara göre daha iyi bir performans gösteren FP-Growth algoritması için önerilen temel paralel yaklaşım [22] çalışmasında geliştirilmiştir. Bu çalışmanın ve diğer paralel yaklaşımların detayları Bölüm 4.2'de verilecektir.

# BÖLÜM 4

# FP-GROWTH

Bu bölümde sık öge kümesi madenciliği (*frequent itemset mining*) algoritmalarından biri olan FP-Growth üzerinde durulacaktır. FP-Growth, sık örüntüleri bulmak için kullanılan bir birliktelik (*association*) algoritmasıdır. Bu algoritmanın önceki çoğu algoritmadan daha etkili bir şekilde çalışarak maliyeti azalttığı görülmüştür. Bunun en büyük nedeni, tüm veritabanını daha küçük ve daha yoğun bir veri yapısı, sık örüntü ağacı (*FP-Tree*), içinde tutmasıdır. Apriori tabanlı algoritmalardan farklı olarak FP-Growth içinde tüm veritabanı sadece iki kez taranır. İlki tüm ögelerin destek değerinin hesaplanması için, ikincisi ise ağaç yapısının oluşturulması içindir [10].

## 4.1 Seri FP-Growth

FP-Growth, yeni aday üretimine ve her defasında veritabanının taranmasına gerek kalmadığı için büyük veritabanları için bir kazanımdır. Algoritmanın çalışması [10]'da da anlatıldığı gibi şu şekilde çalışır: Veritabanı bir kez taranarak her ögenin (*item*) destek değeri (*support*) hesaplanır. Destek değerleri, algoritma içerisinde atanan destek eşik değerine büyük ve eşit olan ögeler büyükten küçüğe sıralanarak bir liste içine konur. Aynı şekilde veritabanında bulunan her hareket (*transaction*) içerisindeki ögelerde destek değerine göre büyükten küçüğe sıralanır. Sık örüntü ağacını oluşturmak için öncelikle root adında yeni bir düğüm (*node*) oluşturulur. Daha sonra her bir hareket (ögeler bahsedilen sırada olmak üzere) ağaç içerisine yerleştirilir. Bu süreç şu şekilde gerçekleşir: işlem içerisinde yer alan bir öge eğer ağaçta yoksa o öge için yeni bir düğüm oluşturulur ve destek değeri 1 yapılır. Destek değerleri de ögelerle beraber tutulur. Eğer o öge daha önce oluşturulmuşsa sadece o düğümün destek değeri 1 arttırılır. Düğümler arasındaki ilişkiyi tutmak için de bir başlık tablosu (*header table*) tutulur. Bu tabloda her düğümün başlangıç noktası işaretlenir. Aynı zamanda ağaç içerisindeki aynı düğümler birbirine işaretçilerle bağlanır. Ağaç oluşturulduktan sonra üzerinde Fp-Growth algoritması çalıştırılır.

Bunun için sıklığı en az olan ögeden başlanır. İçinde o ögenin geçtiği yollar belirlenir. Her yol için de, o ögenin destek değeri o yolun destek değeri olarak atanır. Bu yollar o ögenin şartlı örüntü temelini (*conditional pattern base*) oluşturur. Her bir şartlı örüntü temelinden şartlı örüntü ağacı (*conditional pattern tree*) oluşturulur. Daha sonra bu şartlı örüntü ağacı üzerinde algoritma özyineli (*recursive*) olarak yeniden çalışır. Tablo içindeki her bir öge için bu süreç tekrarlanır ve böylece sık ögeler kümesi belirlenir. Algoritma böl ve fethet yaklaşımına uygun olarak ana görevin kendi içinde daha küçük görevlere ayrılmasına olanak verir.

Yukarıda anlatılan süreci bir örnek üzerinde incelemek gerekirse, elimizde Çizelge 4.1'deki gibi bir market veritabanı olsun [10]. Bu veritabanında ilk sütun hareket numaralarını, ikinci sütun her bir hareket içerisinde yer alan ögeleri, yani her bir alışveriş sırasında alınan ürünleri gösterir. Burada harfle temsil edilen her bir öge gerçekte herhangi bir market alışverişinde alınabilecek herhangi bir ürünü gösterir.

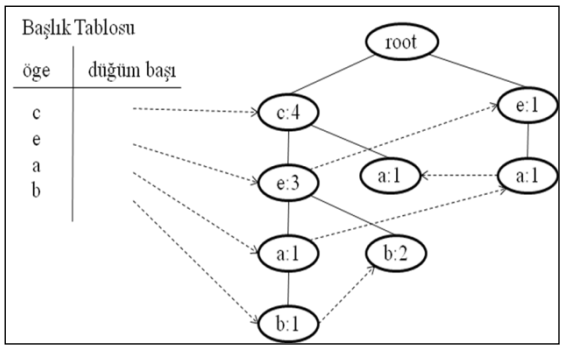
Çizelge 4. 1: FP-Growth Algoritması İçin Örnek Veritabanı

| Hareket No | Alınan Ögeler | Sıralı Sık Ögeler |
|---|---|---|
| 100 | a,b,c,d,e | c,e,a,b |
| 200 | b,c,e,f | c,e,b |
| 300 | a,e | e,a |
| 400 | b,c,e | c,e,b |
| 500 | a,c,d | c,a |

Bu örnek için destek eşik değerini ($\sigma$=3) olarak alalım [10]. İlk olarak veritabanı her bir ögenin destek değerini hesaplamak için bir kez taranır. Buradan, (a:3), (b:3), (c:4), (d:2), (e:4), (f:1) olarak bulunur. Destek eşik değeri 3 olduğundan dolayı d ve f bir sonraki adıma geçemez. Bulunan sık ögeler büyükten küçüğe sıralanarak bir liste içine konur L={<(c:4), (e:4),

(a:3),(b:3)>}. Bu liste içerisinde aynı destek değerine sahip ögelerin öncelik sırası önemli olmamakla beraber listenin sırası önemlidir. Çünkü bundan sonraki tüm işlemlerde burada belirlediğimiz liste *L* kullanılacaktır. Daha sonra veritabanındaki her bir hareket içerisinden sık olmayan ögeler elenir ve kalan sık ögeler ise bu listeye göre kendi içinde sıralanır. Örneğin, 100 nolu harekette d sık olmayan bir öge olduğu için elenir ve kalanlar ise sıralanarak <c,e,a,b> haline gelir. 200 nolu harekette f sık olmayan ögedir ve elenir, kalanlar <c,e,b> olacak şekilde sıralanır. Veritabanının sıralanmış hali Çizelge 4.1'in son sütununda gösterilmiştir.

Bundan sonra sık örüntü ağacı oluşturulmaya başlanır. Bunun için öncelikle *root* adında bir düğüm oluşturulur. Sonra veritabanı ağacın dallarını oluşturmak için ikinci kez taranır. 100 no'lu hareketten başlanır. Bu hareketin ilk elemanı olan c'nin ağaç içerisinde olup olmadığına bakılır. Olmadığı için c adında yeni bir düğüm oluşturulur ve destek değerine 1 atanır. Daha sonra e, a ve b elemanları için de yeni bir düğüm oluşturularak destek değerlerine 1 atanır. 200 nolu hareketin ilk elemanı c'dir. root'tan başlamak üzere ağaç dolaşılır, root'tan c'ye giden bir dal olduğu için bu düğümün daha önceden ağaçta varolduğu anlaşılır. Böylece yeni bir düğüm oluşturulmaz. Sadece olan düğümün destek değeri 1 arttırılır. Sonra e'ye bakılır, o da önceden oluştuğundan dolayı onun da sadece destek değeri 1 arttırılır. Sonraki eleman b içinse c ve e'den sonra gelen bir b olmadığından dolayı e'nin çocuğu olacak şekilde yeni bir düğüm oluşturulur ve destek değerine 1 atanır. Veritabanındaki her bir hareketin ağaç içine bu şekilde yerleştirilmesiyle Şekil 4.1'deki sık örüntü ağacı oluşturulmuş olur [10].




Şekil 4. 1: Örnek veritabanı için FP-tree

Ağaç üzerinde dolaşmayı kolaylaştırmak adına bir başlık tablosu oluşturulur. Bu tabloda her bir ögenin ilk yeri işaretlenir ve ağaç içerisinde de aynı düğümler arasındaki bağlarla her bir düğümün takibi kolaylaştırılır (Şekil 4.1).

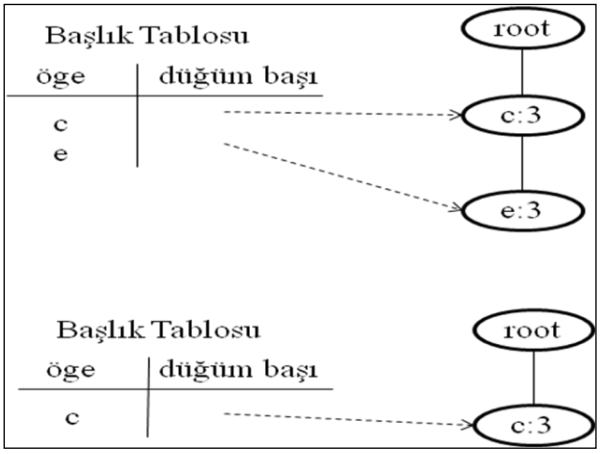
**Girdi:** FPTree
**Çıktı:** Tüm sık ögekümeleri
**Metod:** FP-Growth(FP-tree, null)

FP-Growth(*Tree*, α)
{
**if** *Tree* tek bir yol *P* içeriyorsa **then**
**for each** *β* (*P* yolu içerisindeki düğümlerin kombinasyonu) **do**
*sup* ← ***min***{b.*support* | b in *β* }
Sık ögekümesi αU{b.*item* | b in *β* } yı *sup* destek değeri ile üret
**else**
**for each** item *a* (*a* in *Tree.header*)
*sup*←a.*support*
*a*U*α* için sık ögekümelerini *sup* destek değeri ile üret
*β* = *a*U*α* için şartlı örüntü temellerini ve sonra *β* için şartlı örüntü ağacını $Tree_{\beta}$ oluştur
**if** $Tree_{\beta} \neq \emptyset$ **then**
**FP-Growth**($Tree_{\beta}$, *β*)
}

Şekil 4. 2: FP-Growth Algoritması

FP-Growth algoritmasının ilk bölümü olan sık örüntü ağacının oluşturulmasının ardından bu ağaç üzerinde algoritma (Şekil 4.2) çalıştırılır [17]. Algoritma *L*

içerisindeki en az sıklığı olan ögeden başlayarak tüm ögeler için tekrarlanır. Bu örnekte en az sıklığı olan öge b olduğu için algoritma öncelikle b için başlar. İlk adım, ağaç içerisinde b'nin beraber geçtiği tüm yolların bulunması işlemidir. Bunun için başlık tablosu kullanılarak b'nin geçtiği ilk yol tespit edilir. Oradan düğümler arasındaki bağlar kullanılarak, içinde b olan tüm yollar tespit edilir. Burada bu yollar <c:4, e:3, a:1, b:1>, <c:4, e:3, b:2> dır. Bu durumda algoritma b için çalıştırıldığı için bunlardan b çıkartılabilir. Bunların destek sayısı ise b ile beraber geçen hareket sayılarına eşit olacağı için b'nin destek değerine eşit olur. Yani, <c:1, e:1, a:1>, <c:2, e:2> olur. Bu iki yol b'nin şartlı örüntü temelini oluşturur. Bunlar için algoritma yeniden çalıştırılır. Bunların küçük bir veritabanı oluşturduğunu düşünecek olursak, buradaki her bir ögenin destek değeri hesaplanır. Buradan (c:3), (e:3), (a:1) olur. Minimum eşik değeri 3 olduğundan dolayı, a bir sonraki adıma geçemez. Eşik değerini geçen c ve e ögeleri için yeni bir sık örüntü ağacı oluşturulur (Şekil 4.3). Şekil 4.3'deki ağaç üzerinde tekrar algoritma çalıştırılır. Öncelikle e'ye göre çalıştırılır. Buradan çıkan sonuç, (ceb:3) ve (eb:3)'dür. Daha sonra ağaçta kalan tek öge c olur. c'ye göre algoritma çalıştırıldığında ise (cb:3) elde edilir. b'nin zaten en başta sık olduğu bilindiğinden, içinde b geçen sık öge kümeleri {<(b:3), (cb:3), (eb:3), (ceb:3)>} olarak bulunur.




Şekil 4. 3: b için Oluşturulan FP-tree

Sık örüntü ağacı üzerinde tek bir yol kaldıktan sonra sık öge kümelerini bulmanın başka bir yolu da Şekil 4.3'deki ağaç üzerindeki tüm kombinasyonların bulunması işlemidir. O da yukarıda elde edilen sonuçla aynı sonucu verir.

b için sık örüntü kümelerinin bulunmasının ardından sıra a'ya gelir. İçinde a'nın geçtiği tüm yollar bulunur. Bunlar <c:4, e:3, a:1>, <e:1, a:1>, <c:4, a:1> dir. Bunların a ile beraber geçtiği hareketler ise şöyle olur: <c:1, e:1>, <e:1>, <c:1>. Bu ögelerin yeni destek değerleri (c:2), (e:2) olur. Bu değerler eşik değerinin altında olduğundan dolayı işlem burada son bulur. Bu öge için tek sık öge kümesi kendisi yani <a:3>'dür. Sırada e vardır. e'nin beraber geçtiği yollar <c:4, e:3>, <e:1> dır. e'nin şartlı örüntü temelini <c:3> oluşturur. Buradan elde edilen sık öge kümesi ise <ce:3> olur. e için sık öge kümeleri {<e:4>, <ce:3>} olur. Son olarak içinde c'nin geçtiği yollar <c:4>'dür. Bunun için herhangi bir şartlı örüntü ağacı oluşturulamaz. Dolayısıyla kendisi dışında herhangi bir sık öge kümesi yoktur.

[17]'deki çalışma içerisinde FP-Growth algoritması gerçekleştirilerek diğer sık öge kümesi madenciliği algoritmalarıyla karşılaştırılmıştır. Sonuçta FP-Growth'un daha etkili ve ölçeklenebilir olduğu görülmüştür.

## 4.2 Paralel FP-Growth

Paralel FP-Growth için önerilen en temel algoritma [22]' dir. Bu çalışmada birbirine Ethernet anahtarıyla bağlı 32 düğümden (*nodes*) oluşan bir öbek kullanılmıştır. FP-Growth algoritması paralel bir şekilde düğümler tarafından gerçekleştirilmiştir. Bunun için veritabanı yatay bölünerek öbek üzerindeki her bir düğüme dağıtılır. Bunun ardından her düğüm kendi yerel destek değerlerini hesaplayarak bunları bir liste içine koyar. Sonra tüm listeler birleştirilerek global liste elde edilir. Daha sonra her düğüm kendi yerel veritabanını ve global listeyi kullanarak yerel sık örüntü ağacını oluşturur. Bu ağaçtan sonra şartlı örüntü ağaçları oluşturularak sık öge kümeleri bulunur. Buradaki problem düğümler arasındaki yük dengesini

sağlayabilmektir. Çalışma içerisinde bu sorun "yol derinliği" adında bir kavramın tanımlanmasıyla aşılmaktadır. Yük dengesi bir bakıma statik olarak çözülmüştür. Böylece paralel FP-Growth kendi içerisinde iyi bir sonuç vermektedir [10].

[23] çalışmasında da PFPTC *(Parallel FP-Tree Constructing)* adında bir algoritma önerilmiştir. Bu algoritmada, eşzamanlı olarak sık örüntü ağaçları üretilir. Daha sonra bu ağaçlar *FP-merge* adında bir algoritmayla ikili bir şekilde birleştirilmektedir. Bu çalışmada aynı zamanda *QFP-Growth* adında bir algoritma geliştirilmiştir. Bu, FP-Growth'un kendiliğinden gelen büyük miktarlardaki sonuç üretmesinin zorluğunu engellemek için üretilmiştir.

[24] çalışmasında MLFPT *(Multiple Local Frequent Pattern Tree)* adında bir algoritma önerilmiştir. Bu çalışma da özü itibarıyla yerel sık örüntü ağaçları oluşturur. Ancak burada global değişkenlerin varlığından dolayı gerçekleşen karşılıklı dışlama *(mutual exclusion)* yok edilmiştir. Bunun için birbirine içten bağlı yerel değişkenler kullanılmıştır.

[25] çalışmasında ise dağıtık sistemler üzerinde çalışacak paralel FP-Growth algoritması önerilmiştir. Daha önce yapılan [24] ve [22] çalışmalarında yüksek oranda iletişim zamanının olduğu görülmüştür. Bu çalışma bunu engellemek için *Map-Reduce* yaklaşımını kullanan daha iyi ölçeklenebilir ve web veri madenciliği için de kullanışlı olan bir metot bulmuştur.

# BÖLÜM 5

# ÖBEK BİLGİSAYARLARDA FP-GROWTH

Literatürdeki paralel FP-Growth algoritmaları incelendiğinde veri paralel ve mesaj iletimine dayalı çalışmaların olduğu gözlenmektedir. Örneğin, paralel FP-Growth için en temel çalışma olan [22]'de veritabanı bölünerek veri paralel bir çalışma yapılmıştır. Bunun dışındaki diğer çalışmalar da genelde bu yaklaşımı baz alarak kurulmuştur. Bu çalışma, onlardan farklı olarak görev paralel yaklaşımı benimser. Burada veritabanı bölümlenmesi yoktur, aksine görev bölümlenmesi mevcuttur. Önerilen metotların ikisinde veritabanının tüm işlemcilerde bulunduğu varsayılır. Buradan kasıt, her işlemci veritabanını okuyarak ağacı oluşturur. Dolayısıyla işlenecek veri tüm işlemcilerde mevcut olduğundan bir sonraki aşamada görev dağıtımı yapılır. Burada hedeflenen, veritabanı bölünmeden paralellik uygulansa ne derece bir kazanım olacağıdır. Diğerinde ise veritabanı, dolayısıyla işlenecek verinin tutulduğu ağaç tek işlemcide bulunur. Bu işlemci algoritmayı çalıştırmaya başladıktan sonra özyinelemeli (*recursive*) olarak alt ağaçlar oluşturur ve bu alt ağaçları diğer işlemcilere gönderir. Burada, hedeflenen ağaç yapısının diğer bir düğüme (*nodes*) gönderilmesinin algoritmaya bir kazanımda bulunup bulunmayacağıdır. Her üç algoritmanın genel özellikleri Çizelge 5.1'de verilmiştir. Geliştirilen metotlar iki ayrı öbek (*cluster*) bilgisayar üzerinde test edilmiştir.

Çizelge 5. 1: Öbek FP-Growth Genel Özellikleri

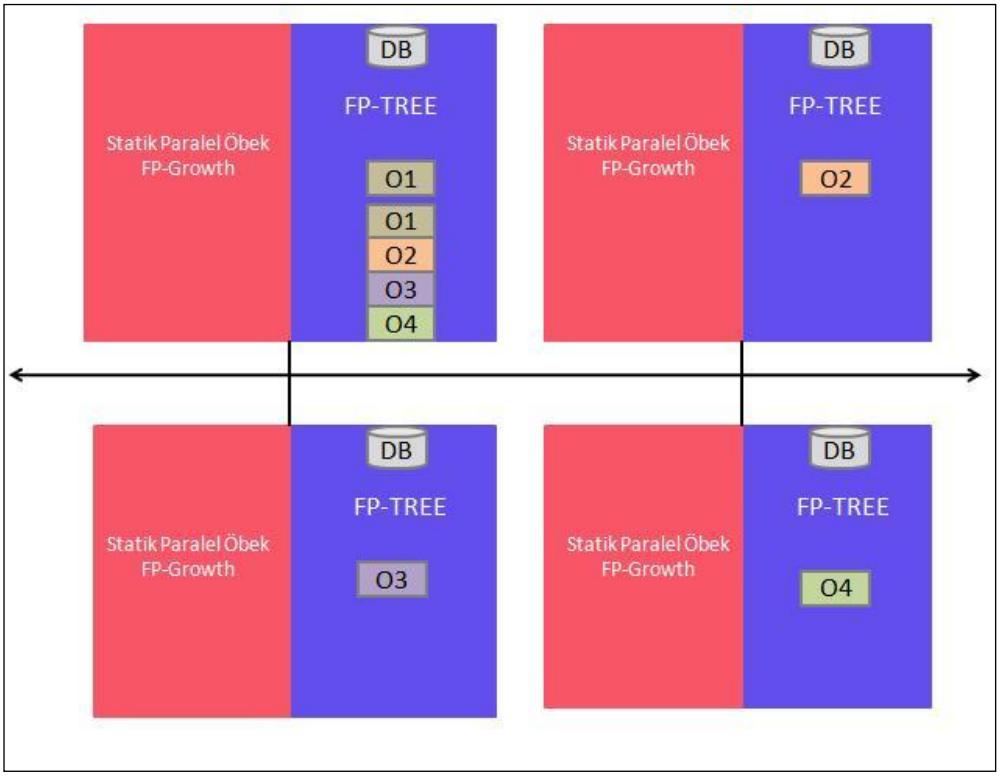
| **Statik Paralel Öbek FP-Growth** | **Dinamik Paralel Öbek FP-Growth** | **Ağaç Gönderimli Dinamik Paralel Öbek FP-Growth** |
|---|---|---|
| İşlemcilerin tümünde veritabanı mevcut | İşlemcilerin tümünde veritabanı mevcut | Sadece yönetici işlemcide veritabanı mevcut |
| İşlemcilerin tümünde sık örüntü ağacı mevcut | İşlemcilerin tümünde sık örüntü ağacı mevcut | Sadece yönetici işlemcide sık örüntü ağacı mevcut |
| Görev dağılımı statik | Görev dağılımı dinamik | Görev dağılımı dinamik |

Mesaj iletimli paralel hesaplamada kabul gören anlayışlardan biri de işlemcilerin sınıflandırılmasıdır. İşlemciler yönetici ve işçi olmak üzere ikiye ayrılır. İşlemci numarası 0 olan işlemci yönetici işlemci olarak adlandırılır ve bu işlemciye genelde bazı temel işler yaptırılır. Mesela, görev dağılımlarını bu işlemci gerçekleştirir. 0 dışında kalan diğer işlemciler işçi işlemciler olarak adlandırılır ve genelde kendilerine verilen işleri gerçekleştirirler. Bu sınıflamayı yapabilmek için MPI kodu içerisine *if/else* yerleştirilerek yönetici işlemci ve işçi işlemciler için farklı çalışma blokları oluşturulur. Bunu yaparken MPI tarafından işlemciye atanan işlemci numarasına da ihtiyaç vardır.

## 5.1 Statik Paralel Öbek FP-Growth

Bu yöntemde, her işlemci veritabanını okuyarak sık öge kümelerini bulur ve sık örüntü ağacını oluşturur. Bunu yaparken bir mesaj iletimi olmaz. Burada paralelliği sağlayan işin bölünmesidir. Bu yöntemde veritabanının her işlemcide bulunduğu varsayılmaktadır. Bu yönüyle algoritma seri algoritmaya benzer çalışır. Paralellik ise bu noktadan sonra başlar. Bundan sonra FP-Growth algoritması sık öge sayısı kadar tekrarlar. Seri algoritmada bu işlem tek bir işlemci üzerinde yapılırken, bu algoritmada farklı işlemciler üzerinde yapılmaktadır. Bu şekilde, her işlemci belli miktar sık ögeyi hesaplayarak, kendi bilgisayarının adıyla oluşturduğu bir dosya içine yazar. Böyle bir yazım, veri kaybını engeller ve kodun doğruluğunun testinin yapılmasını kolaylaştırır. Son olarak, yönetici işlemci (0. işlemci) bu dosyaların hepsini basit bir sistem komutuyla birleştirerek tek bir dosyaya dönüştürür. Böylece seri kodda üretilen gibi tek bir sonuç dosyası elde edilir.

Algoritmanın şematik gösterimi Şekil 5.1'de verilmektedir. 4 işlemciden oluşan bir sistem üzerinde çalışıldığı düşünülürse, tüm işlemciler veritabanını okuyarak sık örüntü ağacını (FP-TREE) oluşturur. Daha sonra hepsi ayrı bir kısım veri üzerinde çalışarak bir sonuç üretir. Bu sonuçlar bir dosyaya yazılır (O1, O2, O3, O4). Dolayısıyla her bir işlemci üzerinde bir sonuç dosyası mevcuttur. En sonunda, yönetici işlemci bu sonuç dosyalarını birleştirir.




Şekil 5. 1: Statik Paralel Öbek FP-Growth Şematik Gösterim

Statik Paralel FP-Growth için önerilen algoritma Şekil 5.2'de verilmiştir.

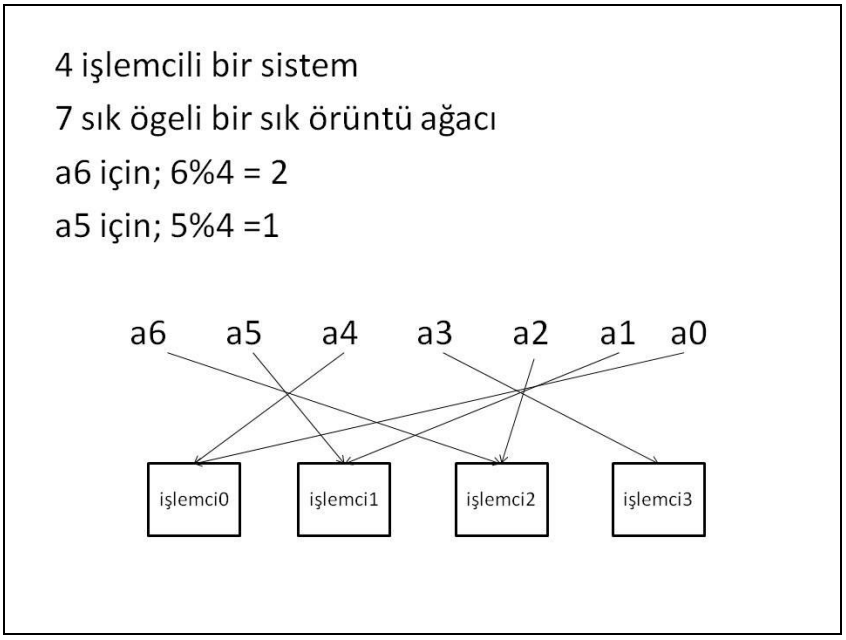
```
1. main(int argc, char *argv[])
2. {
3.      Mpi_Init(&argc, &argv);
4.      MPI_Comm_rank(MPI_COMM_WORLD, &islemciNo)
5.      MPI_Comm_size(MPI_COMM_WORLD, &islemciSayisi)
6.
7.     Tree = Scan1() & Scan2()
8.
9.     if  islemciSayisi == 1 then
10.            FP-Growth(Tree, Ø)
11.     else
12.          if Tree tek bir yol P içeriyorsa then
13.              if  islemciNo == 0 then
14.                   for  each  β  (P  yolu  içerisindeki  düğümlerin
                      kombinasyonu) do
15.                        sup ← min{b.support | b in β }
16.                        Sık  öge  kümesi  {b.item  |  b  in  β  }  yı
                           sup destek değeri ile üret
17.          else
18.              for each öge a (a in Tree.header)
19.                   calisacakNo ← a % islemciSayisi
20.                   if calisacakNo == islemciNo then
21.                        sup←a.support
22.                        a  için  sık  öge  kümesini  sup  destek  değeri
                           ile üret
23.                        β = {a} için şartlı örüntü temellerini ve sonra
                           β için şartlı örüntü ağacını Treeβ oluştur
24.                        if Treeβ ≠ Ø then
25.                             FP-Growth(Treeβ, β)
26.     MPI_Finalize()
27. }
```

Şekil 5. 2: Statik Paralel Öbek FP-Growth Algoritması

Şekil 5.2'deki 3. satır MPI'in ve aynı zamanda paralelliğin başladığı yerdir. 4. ve 5. satırlar sırasıyla işlemcilerin numarasını ve toplam işlemci sayısını bulmaya yarar. 7. satırda veritabanı içindeki tüm sık öge kümeleri bulunur ve büyükten küçüğe sıralanarak sık örüntü ağacı oluşturulur. 9. ve 10. satırlar, tek işlemci olduğu durumda çalışan satırlardır, yani seri kodu çalıştırır. İşlemci sayısının iki ve ikiden büyük olduğu durumda eğer ağaç üzerinde tek bir dal varsa, bu yönetici işlemci (0. işlemci) tarafından yapılır (13-16. satırlar). 17. satırla beraber paralel kod başlar. Burada yapılan işlem, eğer ağaç üzerinde birden fazla dal varsa başlık tablosundaki her bir öge için algoritmanın özyinelemeli olarak tekrar çağrılmasıdır. Bunu

yapabilmek için, hangi işlemcinin hangi öge için algoritmayı çalıştıracağı belirlenmelidir. Yani, bir görev dağılımı gerçekleştirilmelidir. Burada görev dağılımı statik olarak yapılır (19. satır). 20. satır her işlemcinin kendi bloğunu gerçekleştirmesini sağlar. 21. satırda ögenin destek değeri yeni destek değeri olarak atanır. 22. satırda yeni destek değerine göre sık ögeler üretilir. 23. satırda yeni örüntü ağacı oluşturulur. 24. ve 25. satırda ise algoritma kendini yeniden çağırır. 26. satırdaki komut MPI'i sonlandırır.

Bu yöntemi diğer iki yöntemden ayıran temel özellik görev dağılımının statik olmasıdır. Bunu yaparken göz önünde bulundurulması gereken şey işlemciler üzerindeki yük dağılımıdır. Çünkü algoritma her öge için aynı uzunlukta çalışmaz. Bazıları için uzun çalışırken bazıları için de çok kısa çalışır. Bunu önlemek için işleri peş peşe aynı işlemciye göndermek yerine aşağıdaki gibi bir gönderimin daha uygun olacağı düşünülerek uygulama bu şekilde gerçekleştirilmiştir (Şekil 5.3).




Şekil 5. 3: Statik FP-Growth İçindeki Görev Dağılımı

## 5.2 Dinamik Paralel Öbek FP-Growth

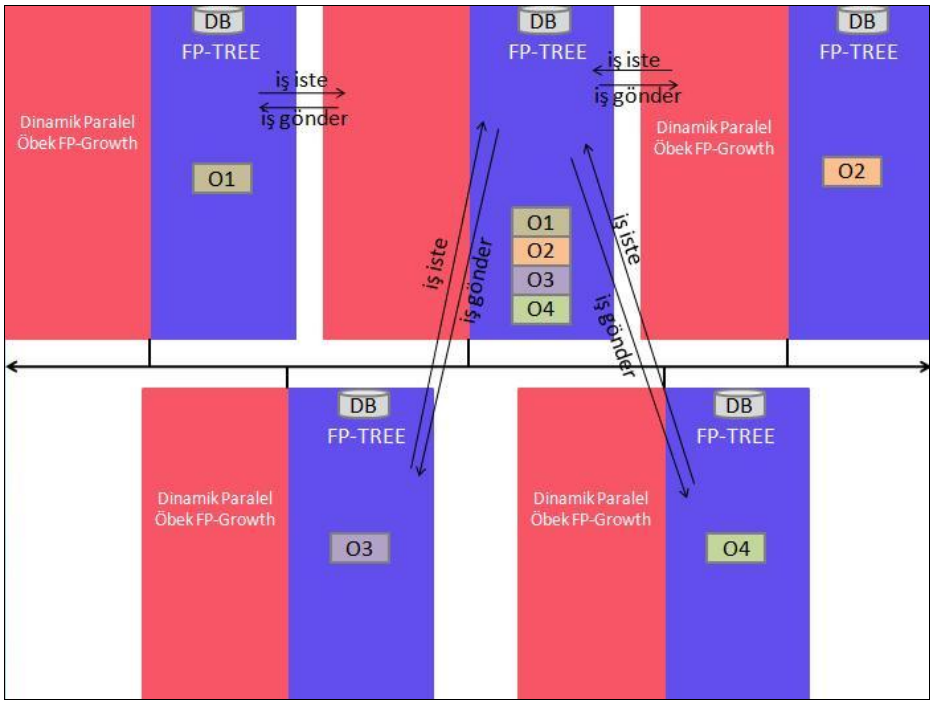
Bu yöntemde de, bir öncekinde olduğu gibi her işlemci veritabanını okur ve sık örüntü ağacını oluşturur. Dolayısıyla, bu algoritma için de her işlemcide veritabanının tutulduğu varsayılmaktadır. Statik Paralel Öbek FP-Growth algoritmasından farkı, burada görev dağılımının dinamik olarak gerçekleştirilmesidir. Dinamik ile gerçekleştirilmek istenen işlemcilerin boş kalmadan işleri bittikçe yeni iş istemeleridir. Bununla amaçlanan statik yöntemde olabilecek bekleme süresini en aza indirmektir. Çünkü statik görev dağılımında işlemci kendisine verilen işi kısa sürede tamamlayıp kendisine yeni iş gelene kadar bekleyebilir. Ancak, dinamik görev dağılımında işlemci kendi işini bitirdikçe bunu iş dağıtıcıya bildirir ve yeni iş ister. Burada yönetici işlemci ve işçi işlemcilerin farklı görevleri vardır. Yönetici işlemci, işlemciler arasındaki görev dağılımını gerçekleştirir. Görev dağılımından kasıt, her bir sık ögeyi boş olan işçi işlemciye göndermektir. İşçi işlemciler ise işleri gerçekleştirirler ve her iş bittikçe yönetici işlemciden yeni iş isterler. İşçi işlemciler açısından iş, kendilerine gönderilen sık ögeyi alıp onun için şartlı sık örüntü ağacını oluşturmak ve bu ağaç üzerinde FP-Growth algoritmasını çalıştırmaktır.

Kod içerisinde dinamik yapı mesaj iletimi ile sağlanır. Burada yönetici işlemciler ve işçi işlemciler arasında gönderilip/alınan bir değişken vardır. İşçi işlemciler, boş olduğu sürece yönetici işlemciye bu değişkeni gönderirler. Yönetici işlemci ise her iş dağıtımından önce herhangi bir işlemciden bu değişkeni alır. Aldığı kaynağın numarası kendisine işi göndereceği işlemci bilgisini verir. Daha sonra, yönetici işlemci işçi işlemci için üzerinde çalışacağı ögeyi gönderir. İşçi işlemciler, gönderilen ögeyi alarak bu öge için yeni sık örüntü ağacını oluşturarak algoritmayı yeniden çağırır.

İşçi işlemcilere ne kadar iş verileceği bilgisi başta belirsiz olduğu için işçi işlemciler sonsuz bir döngü içinde iş isterler. İş bittiğinde, yönetici işlemci işçi işlemcilere işlemin bittiğini gösteren bir mesaj gönderir. Bu mesajı alan işçi işlemciler sonsuz döngüden çıkarlar.

Bu yaklaşımın bu haliyle bir dezavantajı vardır. O da, burada bir işlemcinin tamamen görev dağılımına ayrılmasıdır. Dolayısıyla üç işlemci üzerinde çalışmak istendiğinde aslında iki işlemci üzerinde çalışılır. Bunu önlemek için iş parçacığı (*thread*) kullanılır. Yönetici işlemci bir iş parçacığı yaratır. Böylece yönetici işlemcinin kendisi görev dağılımını gerçekleştirir, kendisi tarafından yaratılan iş parçacığı ise işçi işlemciler gibi iş gerçekleştirir.

Algoritma için önerilen şematik gösterim Şekil 5.4'de verilmektedir. Gösterilen sistem yine 4 işlemciden oluşur. Ancak burada bir tane iş parçacığı oluşturulmuştur. Yönetici işlemci görev dağılımını gerçekleştirirken iş parçacığı da iş gerçekleştirir. Öncelikle her işlemci veritabanını okuyarak sık örüntü ağacını oluşturur. Burada işlemciler arasında mesajlaşma olmaktadır. Bunun amacı, görev dağılımındaki dinamikliği sağlamak içindir. Her bir işlemci bulduğu sık öge kümelerini bir sonuç dosyasına yazar. Yönetici işlemci programın sonunda bu dosyaları birleştirir.




Şekil 5. 4: Dinamik Paralel Öbek FP-Growth Şematik Gösterim

Dinamik Paralel Öbek FP-Growth'da da Statik Paralel Öbek FP-Growth'da olduğu gibi programın sonunda yönetici işlemci işçi işlemcilerin ve kendisi tarafından yaratılan iş parçacığının oluşturduğu sonuç dosyalarını birleştirerek tek bir sonuç dosyası elde edilmesini sağlar.

Dinamik Paralel Öbek FP-Growth için önerilen algoritma Şekil 5.5'deki gibidir.

```
1. main(int argc, char *argv[])
2. {
3.    Mpi_Init(&argc, &argv);
4.    MPI_Comm_rank(MPI_COMM_WORLD, &islemciNo)
5.    MPI_Comm_size(MPI_COMM_WORLD, &islemciSayisi)
6.
7.    Tree = Scan1() & Scan2()
8.
9.    if islemciSayisi == 1 then
10.       FP-Growth(Tree, Ø)
11.   else
12.       if Tree tek bir yol P içeriyorsa then
13.           if islemciNo == 0 then
14.               for each β (P yolu içerisindeki düğümlerin
                   kombinasyonu) do
15.               sup ← min{b.support | b in β }
16.               Sık ögekümesi {b.item | b in β } yı
                   sup destek değeri ile üret
17.       else
18.           if islemciNo == 0 then
19.               for each öge a (a in Tree.header)
20.                   MPI_Recv( bayrak )
21.                   MPI_Send( a )
22.
23.                 if öge bittiğinde then
24.                   MPI_Send( -1)
25.           else
26.               while ( 1 )
27.                   MPI_Send( bayrak )
28.                   MPI_Recv( a )
29.                   if ( -1) then
30.                       break
31.
32.                   sup←a.support
33.                   a için sık öge kümesini sup destek değeri
                       ile üret
34.                   β = {a} için şartlı örüntü temellerini ve sonra
                       β için şartlı örüntü ağacını Treeβ oluştur
35.                   if Treeβ ≠ Ø then
36.                       FP-Growth(Treeβ, β)
37.   MPI_Finalize()
38. }
```

Şekil 5. 5: Dinamik Paralel Öbek FP-Growth Algoritması

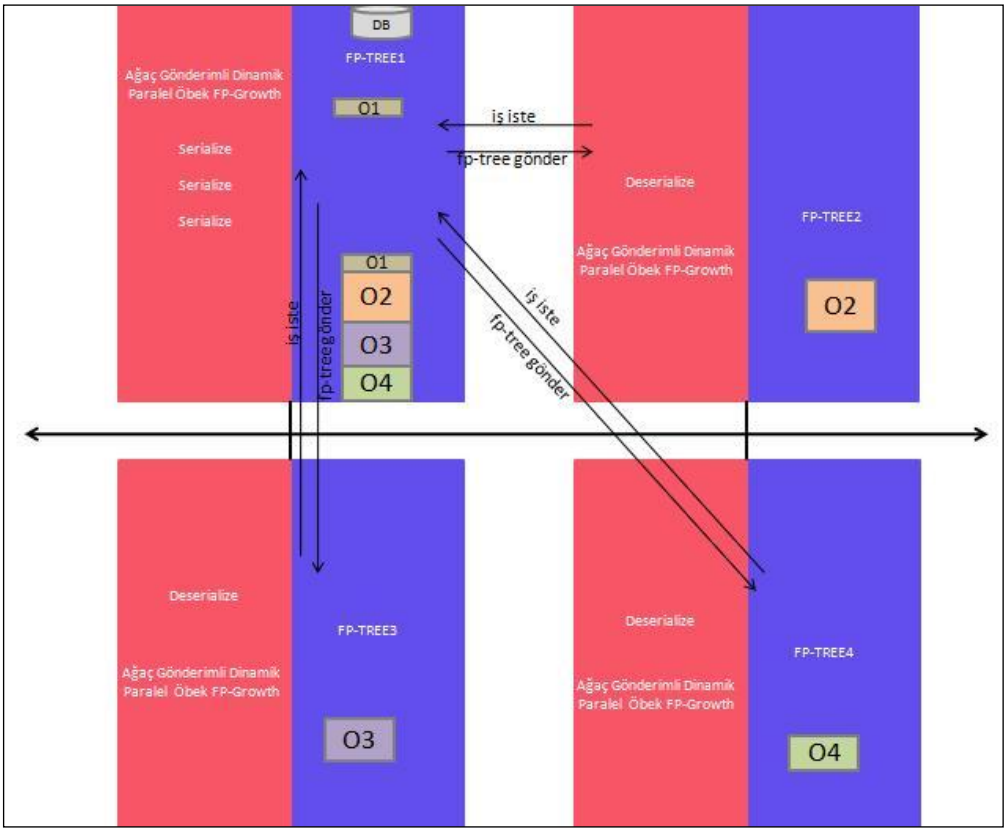
Şekil 5.5'de Statik Paralel Öbek FP-Growth yaklaşımından farklı olarak gerçekleştirilen adımlar şu şekildedir: Burada yönetici işlemcinin çalıştıracağı blok 18.-24. satırlardır. İşçi işlemcinin çalıştıracağı blok ise 25.-36. satırlardır. Yönetici işlemci 20. satırla dağıtacağı iş için boş bir işlemci bekler. Bu işlemciyi bulduğu durumda 21. satırda görüldüğü gibi başlık tablosunda yer alan sık ögelerden en alt sıradakini boş olan işlemciye gönderir. İşçi işlemci tarafında ise, daha önceden ifade edildiği gibi işçi işlemcilerin ne kadar iş gerçekleştireceği belli olmadığı için sonsuz döngü kullanılır. 26. satır bu döngüyü gösterir. İşçi işlemcilerin iş almadan önce yönetici işlemciye bir anlamda "ben boşum" demesi gerekir ve bunu 27. satırda olduğu gibi bir değişken göndererek yapar. Bundan sonra kendisine iş gelene kadar alma (*recv*) satırında bekler (28). Yönetici işlemci başlık tablosunda sık ögeler olduğu sürece bu şekildeki bir mesajlaşmayla görev dağıtımına devam eder. Başlık tablosunda hiçbir öge kalmadığı durumda ise 23. ve 24. satırlarda olduğu gibi diğer işlemcilere ve iş parçacığına negatif bir değer (-1) göndererek tüm işlerin bittiği bilgisini iletir. İşçi işlemciler tarafında ise bu bilgi 28. satırdaki alma komutuyla eşleşir. Çünkü yönetici işlemci görev dağıtımı sırasında gönderdiği değişkenle bu bilgiyi gönderir. İşçi işlemci her alma sonrasında bu bilgi, işlemin sonlandığını gösteren bilgi mi diye bakarak devam edip etmemesi gerektiğine karar verir (29. ve 30.).

## 5.3 Ağaç Gönderimli Dinamik Paralel Öbek FP-Growth

Bu yöntemde, diğer iki yöntemden farklı olarak veritabanının okunması ve ağacın oluşturulması yönetici işlemci tarafından gerçekleştirilir. Bazen, güvenlik gibi sebeplerle verinin tamamının tüm bilgisayarlarda bulunması istenmez. Diğer bilgisayarların bu veriye direk ulaşmadan problemi çözmesi beklenir. Ağaç Gönderimli Dinamik Paralel Öbek FP-Growth algoritması bu ihtiyaca cevap vermek için geliştirilmiştir. Dolayısıyla burada düşünülmesi gereken nokta, işçi işlemcilerin problemi çözerken ihtiyaç duyacakları verinin onlara nasıl iletileceğidir. Burada ilk olarak, yönetici işlemci veritabanını okuyarak sık ögeleri bulup sık örüntü ağacını oluşturarak FP-Growth algoritmasını çalıştırır. Seri koda göre, ağaç tek bir dal

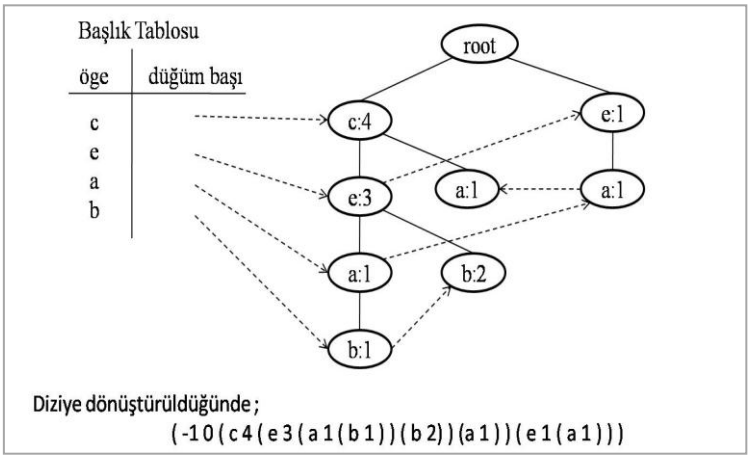
içermediği müddetçe her sık öge için yeni destek değeriyle yeni şartlı sık örüntü ağaçları ve örüntüler oluşturulur ve bunların üzerinde algoritma kendini yeniden çağırır. Ağaç gönderimli dinamik paralel öbek FP-Growth algoritmasında bundan sonra, yönetici işlemci her bir sık öge için şartlı sık örüntü ağaçlarını oluşturur. İşçi işlemciler bu aşamadan sonra devreye girer. Oluşturulan her yeni ağaç üzerinde algoritmayı yeniden çağırma işi işçi işlemcilere verilir. Buradaki sorun görevin işçi işlemcilere nasıl dağıtılacağıdır. Çünkü verilen görevde kullanılan veri yapısı ağaçtır. Ağaç MPI için karmaşık bir veri yapısıdır. Burada düşünülmesi gereken şey ağacın karşı tarafa MPI ile nasıl iletileceğidir. Bunun için önerilen çözüm, ağacın bir diziye dönüştürülerek *(serialization)* karşı tarafa gönderilmesidir. Çünkü MPI ile dizi gönderimi/alımı mevcuttur. Buraya kadar olan kısım gönderici tarafındaki sorundur. Alıcı, MPI ile diziye dönüştürülmüş ağacı alır. Fakat, bu dizi alıcı tarafında tam olarak anlamlı değildir. Çünkü alıcı tarafının işlemesi için gerekli olan veri yapısı dizi değil ağaçtır. O halde, alınan dizinin ağaca dönüştürülmesi gerekir *(deserialization).*

Algoritmanın şematik gösterimi Şekil 5.6'da verilmektedir. Buna göre, 4 işlemcili bir sistem üzerinde algoritma öncelikle yönetici işlemciden başlar. Burada, diğer iki metottan farklı olarak veritabanı yalnızca yönetici işlemcide mevcuttur. Yönetici işlemci veritabanından sık örüntü ağacını oluşturur ve algoritmayı çalıştırmaya başlar. Bundan sonraki süreçte, gerektiğinde şartlı sık örüntü ağaçlarını kendisinden görev isteyen işlemcilere gönderir. Göndermeden önce ağacı diziye dönüştürür. Bu diziyi alan işlemci de önce diziyi ağaca dönüştürür. Daha sonra algoritmayı çalıştırarak bir sonuç üretir. Bulunan sonuçlar işlemciler tarafından ayrı dosyalara yazılır. Bu sonuç dosyaları sonunda yönetici işlemci tarafından birleştirilir.




Şekil 5. 6: Ağaç Gönderimli Dinamik Paralel FP-Growth Şematik Gösterim

Ağacın diziye dönüştürülmesi ve dizinin yeniden ağaca dönüştürülmesi sırasında aynı temele dayanan bir strateji belirlenmesi gerekir. Bunun için öncelikle ağaç üzerinde dolaşmayı sağlayacak metotlardan birinin seçilmesi gerekir. Burada, kök-önce derinliğine geçme *(preorder depth-first traversal)* yöntemi seçilmiştir. Ek olarak, bazı özel karakterler kullanılmıştır. Bunlar '(' ve ')' 'dır. Bu karakterler, ağacın düğümleri arasındaki geçişleri belirler. Burada temel alınan şey, ağaç üzerine eklenen her bir yeni düğüm için öncelikle diziye bir '(' eklenir. Daha sonra yeni düğüm olarak eklenen her öge için diziye öge ve destek değerleri eklenir. Düğüme eklenecek yeni bir düğüm olmadığı durumda, bu bir anlamda çocuğunun olmadığını gösterir ve bu durumda ')' karakteri eklenir. Bu süreç Şekil 5.7'de, Bölüm 4.1'de anlatılan örnekte kullanılan sık örüntü ağacı (Şekil 4.1) üzerinde gösterilmektedir.




Şekil 5. 7: Ağacın Diziye Dönüştürülmesi (Lineerleştirme) İşlemi

Yönetici işlemci, işçi işlemcilere iş gönderirken Dinamik Paralel Öbek FP-Growth algoritmasında kullanıldığı gibi dinamik dağıtımı kullanır. Yani yönetici işlemciler ve işçi işlemciler arasında sabit bir değişken gönderilip/alınır. Bu değişken iş mevcutken ve işler tamamlanmışken farklı değerler alır.

İşçi işlemciler, kendilerine gönderilen ağaç üzerinde algoritmayı çalıştırarak sık ögeleri bulurlar. Bulunan sonuçlar, Statik ve Dinamik Paralel Öbek FP-Growth algoritmasında olduğu gibi kendi makinelerinin isimleri ve numaralarıyla oluşan dosyalara yazılır. Yönetici işlemci, programın sonunda bu dosyaları sistem komutuyla birleştirerek tek bir dosya haline getirir.

Ağaç gönderimli dinamik paralel öbek FP-Growth için önerilen algoritma Şekil 5.8 ve Şekil 5.9'da verilmiştir.

```
Girdi: FPTree
Çıktı: Tüm sık ögekümeleri
Metod: FP-Growth(FP-tree, null)

1. FP-Growth(Tree, α)
2. {
3.     MPI_Comm_rank(MPI_COMM_WORLD, &islemciNo)
4.     MPI_Comm_size(MPI_COMM_WORLD, &islemciSayisi)
5.
6.     if Tree tek bir yol P içeriyorsa then
7.       for each β (P yolu içerisindeki düğümlerin
8.         kombinasyonu) do
9.             sup ← min{b.support | b in β }
10.            Sık ögekümesi αU{b.item | b in β } yı
                sup destek değeri ile üret
11.    else
12.            for each item a (a in Tree.header)
13.                sup←a.support
14.                aUα için sık ögekümelerini sup destek değeri
                    ile üret
15.                β = aUα için şartlı örüntü temellerini ve sonra
                    β için şartlı örüntü ağacını Treeβ oluştur
16.
17.                if Treeβ ≠ Ø then
18.                    if islemciSayisi == 1 then
19.                        FP-Growth(Treeβ, β)
20.                    else
21.                        if islemciNo == 0 then
22.                            MPI_Recv( bayrak )
23.                            MPI_Isend ( bitisBayragi)
24.                            serialize()
25.                            MPI_Send(Treeβ )
26.                            MPI_Send( β )
27.                        else
28.                            FP-Growth(Treeβ, β)
29. }
```

Şekil 5. 8: Ağaç Gönderimli Dinamik Paralel Öbek FP-Growth Algoritması-1

```
1. main(int argc, char *argv[])
2. {
3.      Mpi_Init(&argc, &argv);
4.      MPI_Comm_rank(MPI_COMM_WORLD, &islemciNo)
5.      MPI_Comm_size(MPI_COMM_WORLD, &islemciSayisi)
6.
7.      if islemciNo == 0
8.         Tree =  Scan1() & Scan2()
9.
10.     if  islemciSayisi == 1
11.          FP-Growth(Tree, Ø)
12.     else
13.          if islemciNo == 0
14.               FP-Growth(Tree, Ø)
15.          else
16.               bitisBayragi ← 1
17.               while ( bitisBayragi ==1 )
18.                    MPI_Isend( bayrak )
19.                    MPI_Recv ( bitisBayragi)
20.                    if  bitisBayragi == -1
21.                         break;
22.                    MPI_Recv( Tree)
23.                    MPI_Recv(β)
24.                    deserialize()
25.                    FP-Growth(Treeβ, β)
26.
27.     MPI_Finalize()
28. }
```

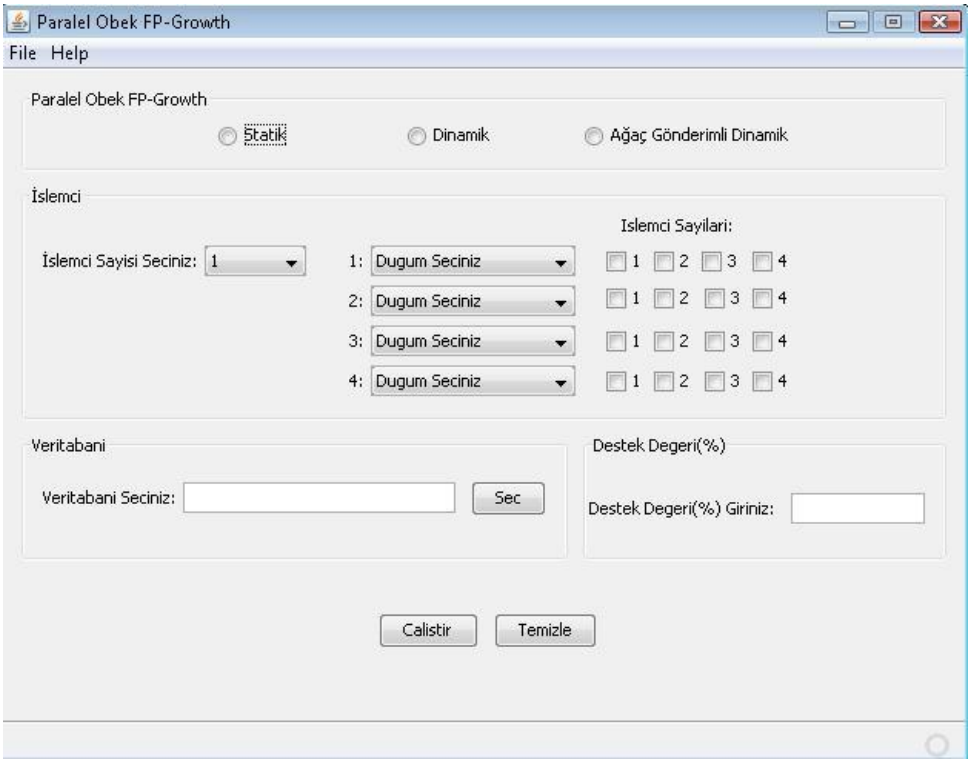
Şekil 5. 9: Ağaç Gönderimli Dinamik Paralel Öbek FP-Growth Algoritması-2

Algoritma her zamanki gibi main metoduyla başlar (Şekil 5.9). Öncelikle 3. satırla MPI başlar. 4. ve 5. satırlar MPI'e özgü metotlardır ve sırasıyla işlemci numarasını ve toplam işlemci sayısını bulmayı sağlar. Ağaç Gönderimli Dinamik Paralel Öbek FP-Growth algoritmasında sadece yönetici işlemci veritabanını okuyarak sık ögeleri bulup sık örüntü ağacını oluşturur (7.-8. satırlar). 10. ve 11. satırlar seri koda ait satırlardır. Algoritma öncelikle yönetici işlemciden başlar (13. ve 14. satırlar). Bu esnada, işçi işlemciler boş olduğu mesajını yönetici işlemciye gönderir (18.). Devamı için Şekil 5.8 'in incelenmesi gerekir. Burada yönetici işlemci öncelikle kendisinden görev isteyen bir işlemci bulur (22.). Daha sonra ağacı diziye dönüştürür (24.) ve ağacı ve örüntüyü görev isteyen işlemciye gönderir (25. ve 26.) İşçi işlemciler üzerindeki süreç için tekrar Şekil 5.9'un incelenmesi gerekir. Burada, işçi işlemciler yönetici işlemci tarafından gönderilen diziye dönüştürülmüş ağaç ve örüntüyü alır

(22. ve 23.). Diziyi yeniden ağaca dönüştürerek (24) algoritmayı tekrar çalıştırır (25.). Hangi işlemciye ne kadar görev verileceği yine belirsiz olduğundan bu sefer şu şekilde bir yapılanma gerçekleştirilir: Bitiş işlemini gösteren ayrı bir değişken tutularak, her görev öncesi işçi işlemcilere gönderilir (Şekil 5.8, 23. satır). İşçi işlemcilerde bu değişken üzerine kurulu bir döngü oluşturulur (Şekil 5.9, 16.-17. ve 19.-21.). Şekil 5.9'da 27. satırda görüldüğü gibi program sonlanmadan MPI sonlandırılır.

## 5.4 Performans Analizi

Geliştirilen uygulama C dilinde kodlanmıştır. Paralellik için MPI kütüphanesi kullanılmıştır. Uygulama için bir masaüstü arayüzü geçekleştirilmiş ve bu arayüz Java diliyle kodlanmıştır (Şekil 5.10). Bu arayüz, uygulamalar için gerekli olan betiklerin (*scripts*) otomatik olarak oluşturulmasını sağlamaktadır.




Şekil 5. 10: Betik Oluşturma Arayüzü

Testler için gerekli veritabanları IBM Dataset Generator tarafından oluşturulan sentetik veri kümeleridir. Bu veri kümelerinin özellikleri Çizelge 5.2'de verilmektedir. Buna göre `D` ile ifade edilen veritabanı içindeki toplam hareket sayısı ve `I` ile ifade edilen ise toplam öge sayısıdır. `T` ise her bir hareketteki ortalama öge sayısını gösterir [26].

Çizelge 5. 2: Veri Kümelerinin Genel Özellikleri

| İsim | \|T\| | \|I\| | \|D\| | Büyüklük(MB) |
|---|---|---|---|---|
| `T20.I5.500K` | 20 | 5 | 500K | 38 |
| `T20.I5.1000K` | 20 | 5 | 1000K | 76 |
| `T20.I5.1500K` | 20 | 5 | 1500K | 113 |
| `T20.I5.2000K` | 20 | 5 | 2000K | 151 |
| `T20.I5.2500K` | 20 | 5 | 2500K | 188 |

Ayrıca testler sırasında sentetik veri kümelerinin yanında gerçek bir veritabanı olan `retail` de kullanılmıştır. `Retail`, Belçika'daki bir markete ait alışveriş veritabanıdır [27]. Bu veritabanındaki öge sayısı 16470, toplam hareket sayısı ise 88163'tür.

Paralel kodun testleri TOBB ETÜ ve Çankaya Üniversitesi öbeği üzerinde gerçekleştirilmiştir. Her iki öbeğin donanım ve yazılım özellikleri Çizelge 5.3'de verilmektedir.

Çizelge 5. 3: Test Öbeklerinin Özellikleri

| **TOBB ETÜ ÖBEĞİ** | **ÇANKAYA ÜNİVERSİTESİ ÖBEĞİ** |
|---|---|
| 4 adet Sun Fire X2200 M2 sunucu | 4 adet PC |
| 8 adet AMD Opteron çift çekirdek | 4 adet Intel Core Quad işlemci |
| 1.8 GHz X64 işlemci | 2.83 GHz işlemci |
| 2 GB DDRR2 677MHz bellek | 16 GB bellek |
| Çift girişli gigabit Ethernet kartı | Fast Ethernet kartı |
| Solaris 10 x86 işletim sistemi | Linux işletim sistemi 64 bit |
| OpenMPI | OpenMPI |
| Sun N1 Grid Engine 6.1 | |
| C, C++, Fortran derleyicileri | C, C++, Fortran derleyicileri |

Bu öbeklerden TOBB ETÜ öbeği raflı (*rack*) bir yapıya, Çankaya Üniversitesi öbeği ise PC'lerden oluşan bir yapıya sahiptir (Şekil 5.11).

a) TOBB ETÜ Öbeği b) Çankaya Üniversitesi Öbeği

Şekil 5. 11: Test Öbekleri

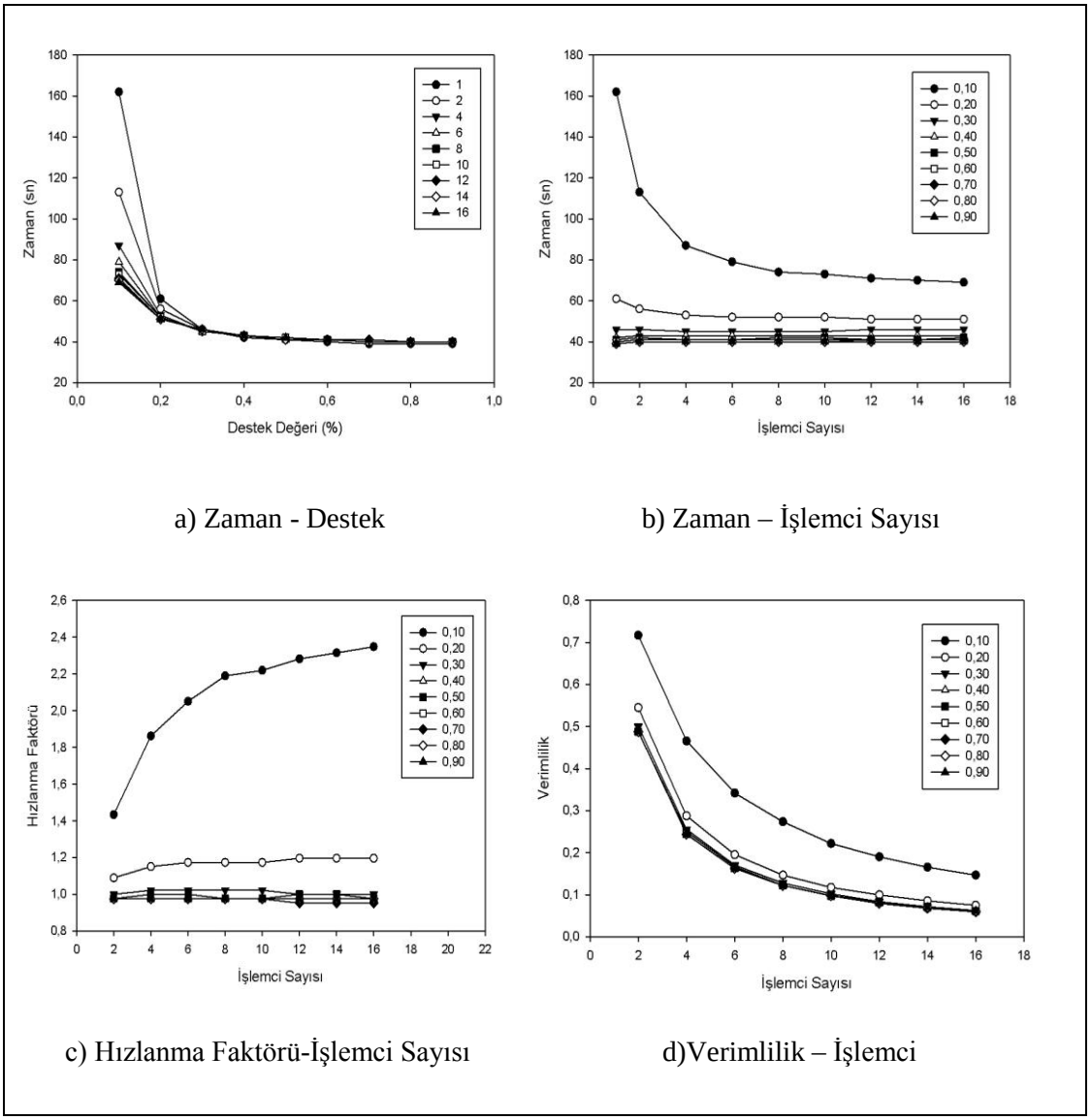
### 5.4.1 Statik Paralel Öbek FP-Growth Testleri

Statik Paralel Öbek FP-Growth algoritmasının çeşitli veritabanları üzerinde test edilen sonuçları Şekil 5.12-5.18 arasında verilmektedir. İşlemci sayılarına göre zaman-destek değeri grafikleri incelendiğinde, seriden (1 işlemci) paralele geçilen ilk durumda (2 işlemci) çalışma zamanında yaklaşık %50 oranında bir azalma olmuştur. İşlemci sayısı arttıkça zaman azalmış ve genel olarak 6 işlemciden sonra zamanda gözle görülür bir değişiklik olmamıştır. Destek değerlerine göre zaman-işlemci sayısı grafikleri incelendiğinde, algoritmanın en düşük destek değerinde en uzun çalıştığı gözlemlenirken, destek değeri arttıkça çalışma zamanının azaldığı gözlemlenmiştir. Çünkü destek değeri büyüdükçe sık öge sayısı azalır.

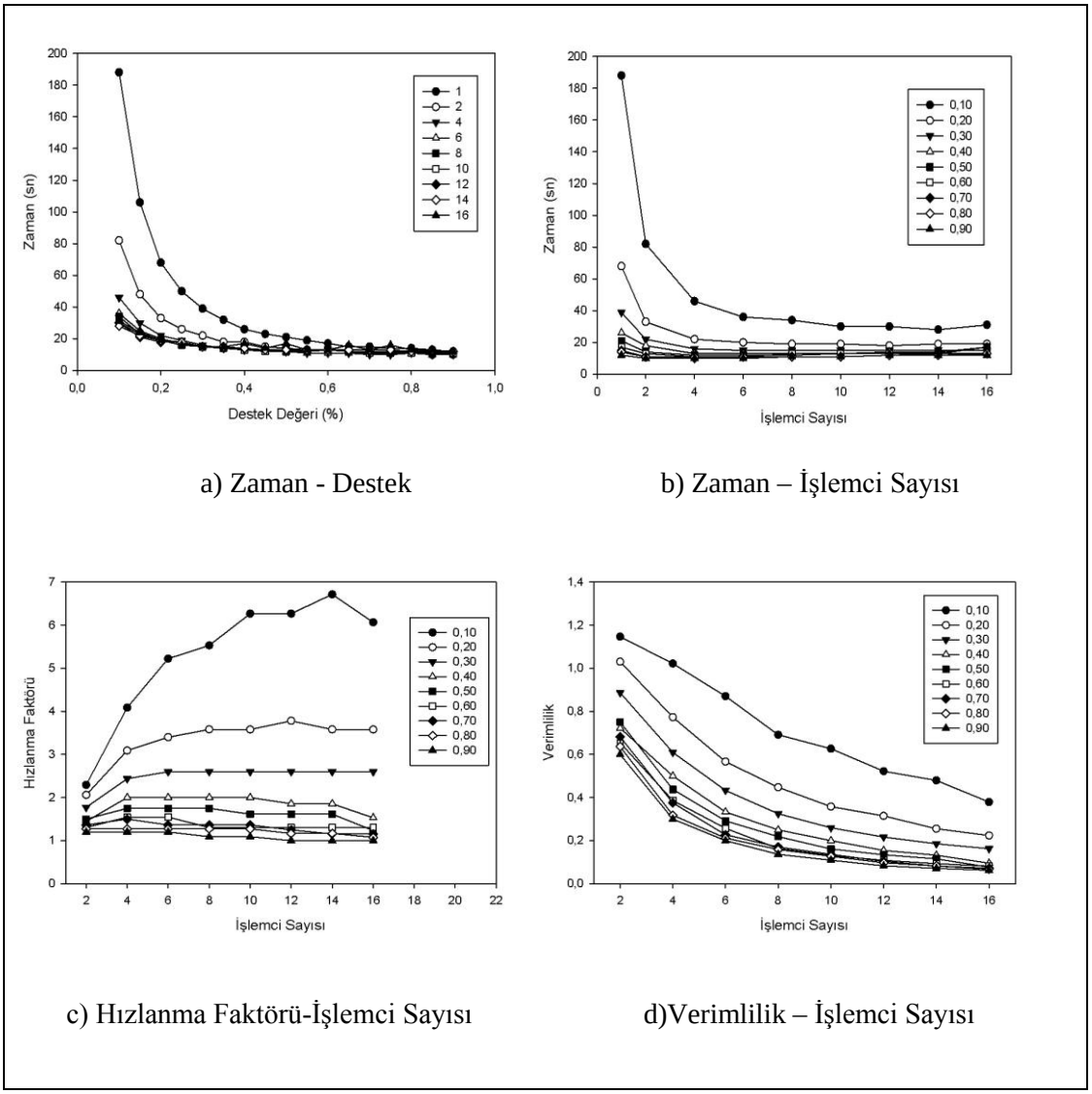
Hızlanma zamanı-işlemci sayısı grafiklerinin normalde artan bir eğilim göstermesi gerekir. Böyle olması gerekirken, bazı durumlarda (Şekil 5.16 c, %0,10 ve %0,20 destek değerleri için) hızlanma grafiğinin düşmeye başladığı görülür. Bunun anlamı,

bu destek değerleri için işlemcilere düşen veri miktarının optimum seviyeden uzaklaşmasından dolayı işlemcilerin tam performansla çalışmaması olabilir.

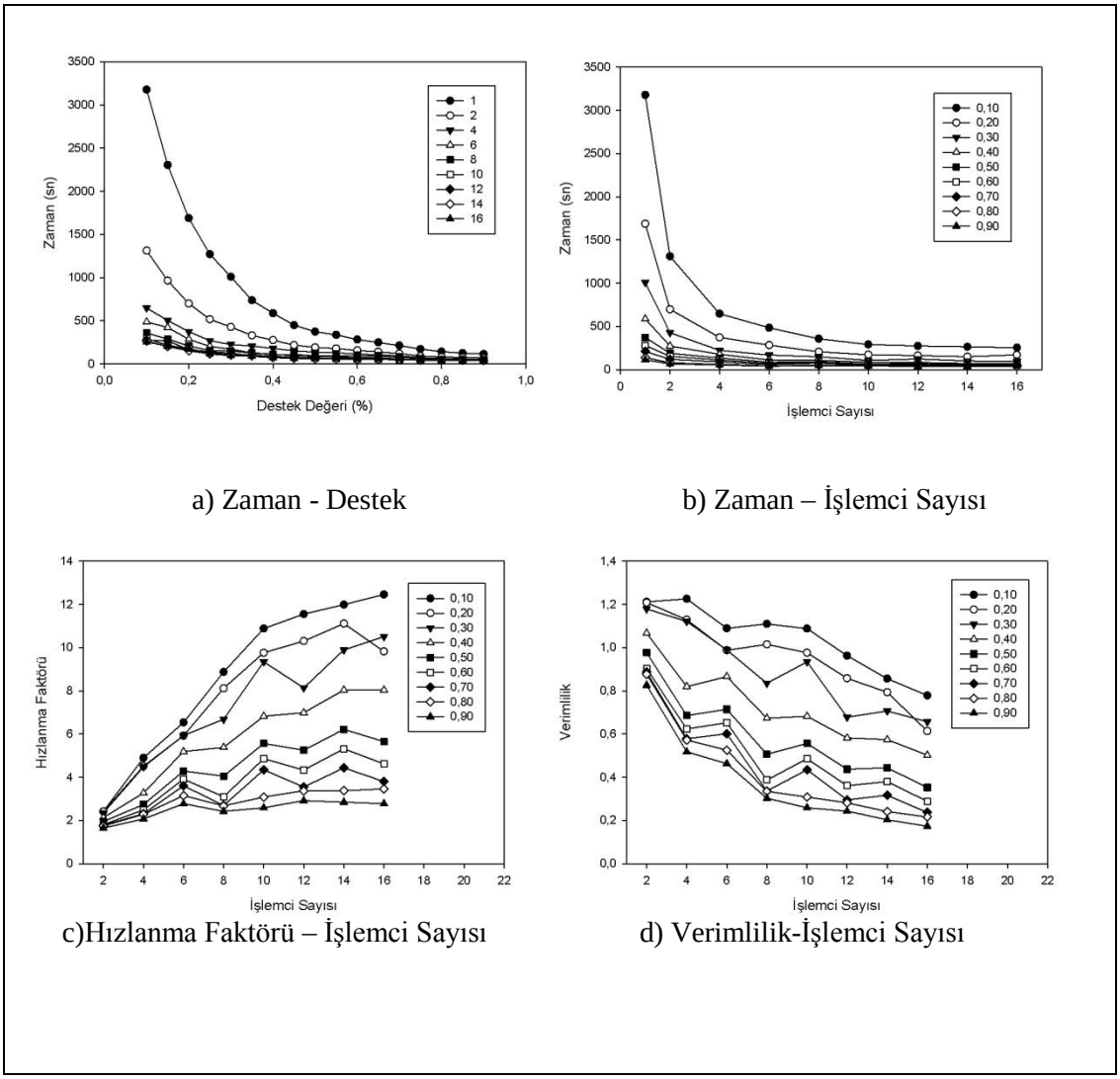
Verimlilik-işlemci sayısı grafiklerine göre, verimlilik işlemci sayısı arttıkça azalmıştır. Bunun nedeni, belli bir noktadan sonra sisteme eklenen işlemcilerin yük dağılımını azaltmaması, yeteri kadar yükün olmamasıdır. Aynı zamanda verimlilik grafiklerinden küçük ve büyük destek değerleri için işlemcilerin verimliliği ile ilgili bilgilere de ulaşılabilir. Küçük destek değerlerinde tüm işlemciler iyi bir performansla çalışırken, büyük destek değerlerinde işlemci sayısının 4'ün üstünde olduğu durumlarda verimlilikte bir düşüş gözlemlenmektedir.




Şekil 5. 12: `retail` (TOBB ETÜ) Test Sonuçları




Şekil 5. 13: `retail` Test Sonuçları



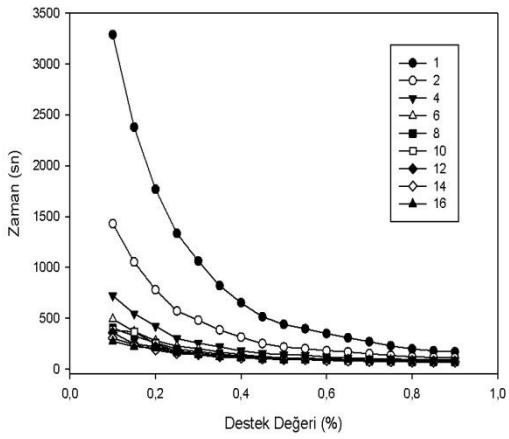

a) Zaman - Destek

b) Zaman – İşlemci Sayısı

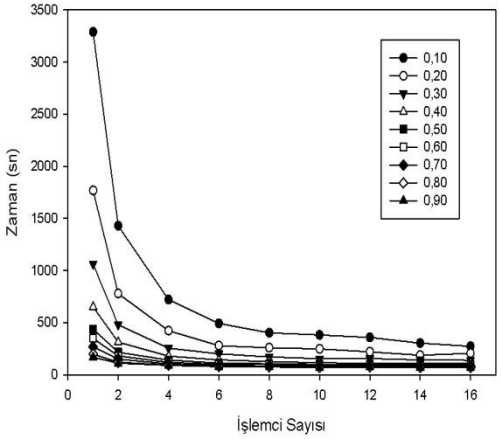
c)Hızlanma Faktörü – İşlemci Sayısı

d) Verimlilik-İşlemci Sayısı

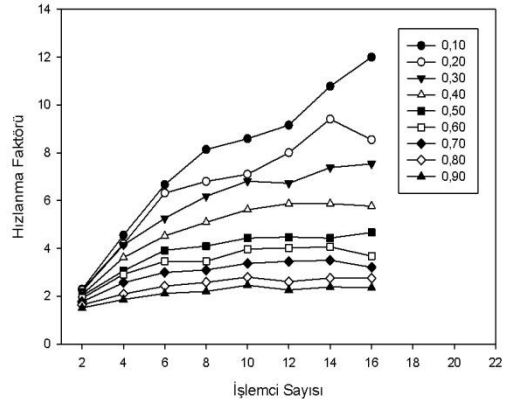
Şekil 5. 14: `T20.I5.500K` Test Sonuçları



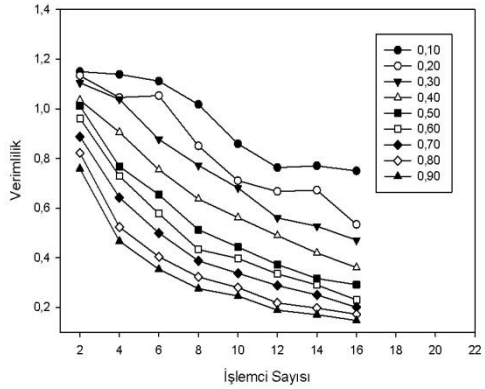

a) Zaman - Destek




b) Zaman – İşlemci Sayısı

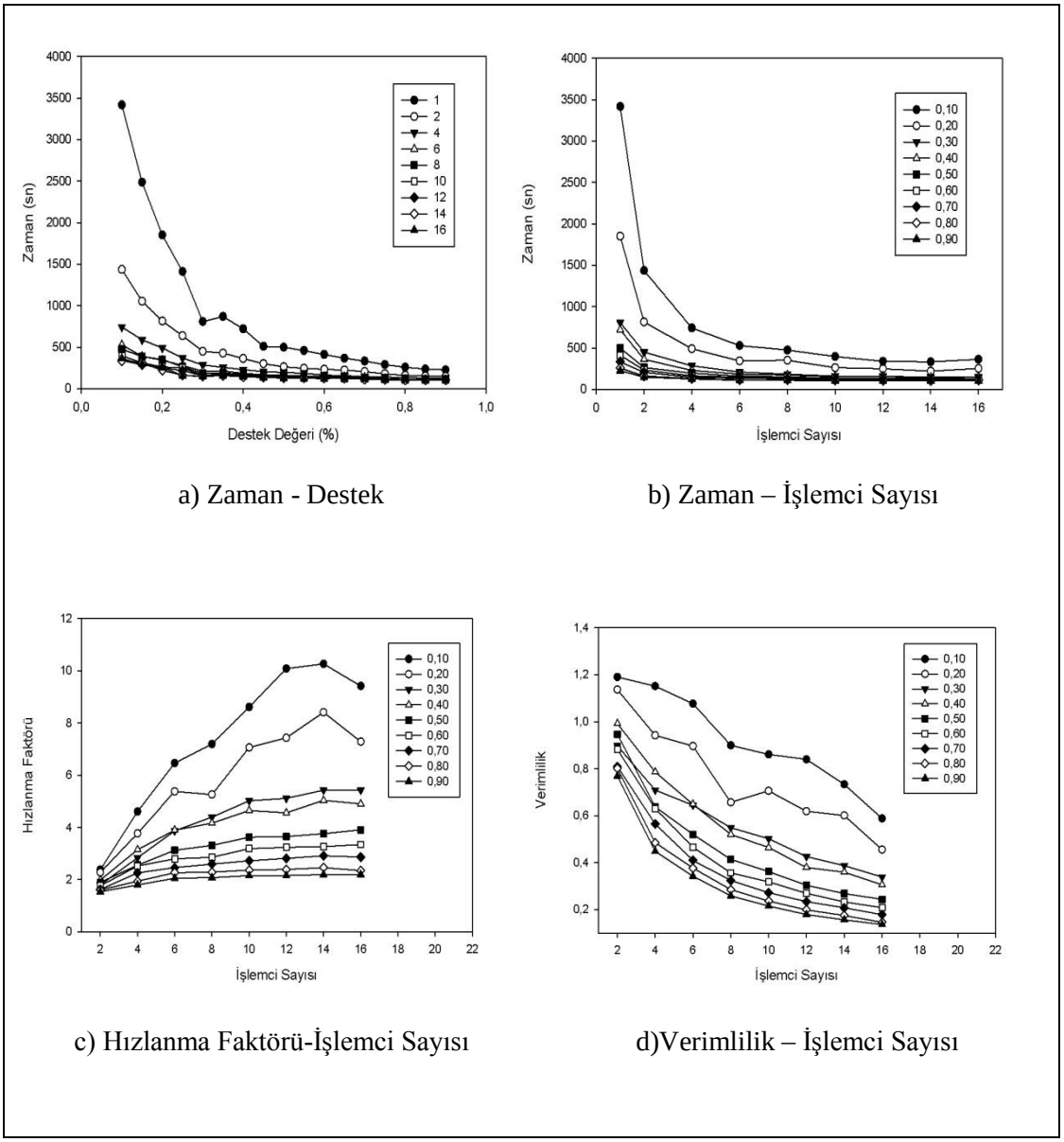



c) Hızlanma Faktörü-İşlemci Sayısı

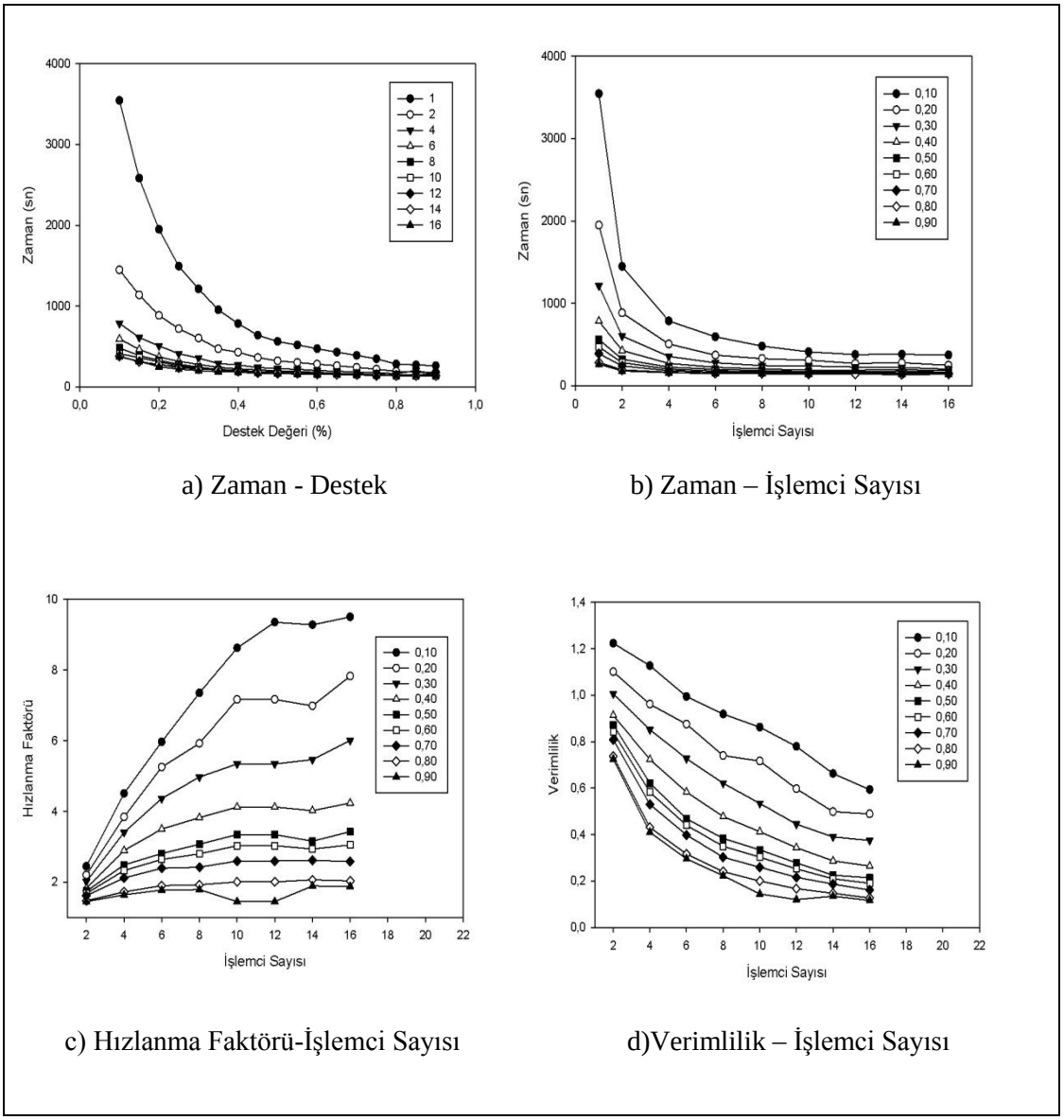



d)Verimlilik – İşlemci Sayısı

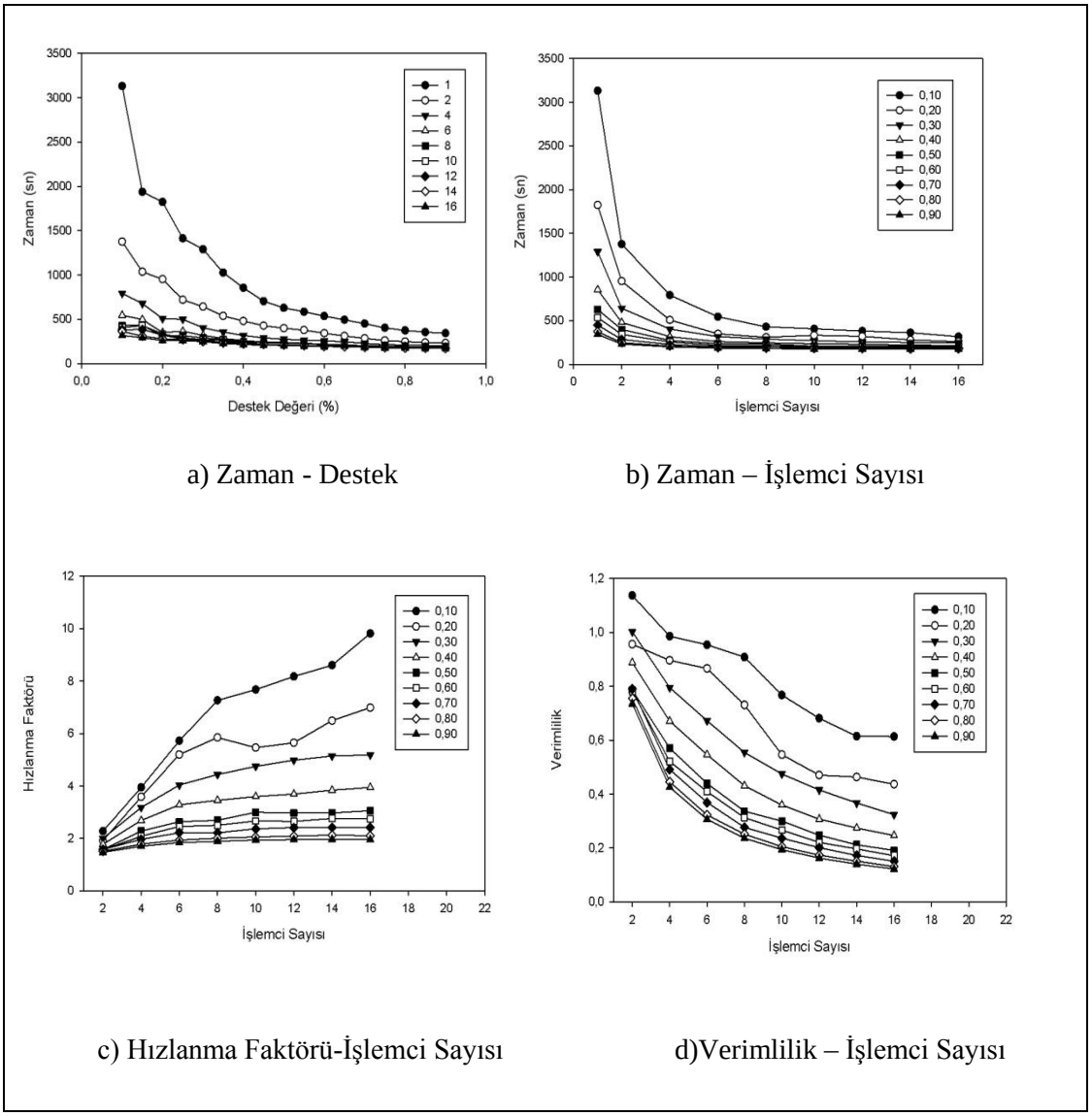
Şekil 5. 15: `T20.I5.1000K` Test Sonuçları




Şekil 5. 16: `T20.I5.1500K` Test Sonuçları




Şekil 5. 17: `T20.I5.2000K` Test Sonuçları




a) Zaman - Destek

b) Zaman – İşlemci Sayısı

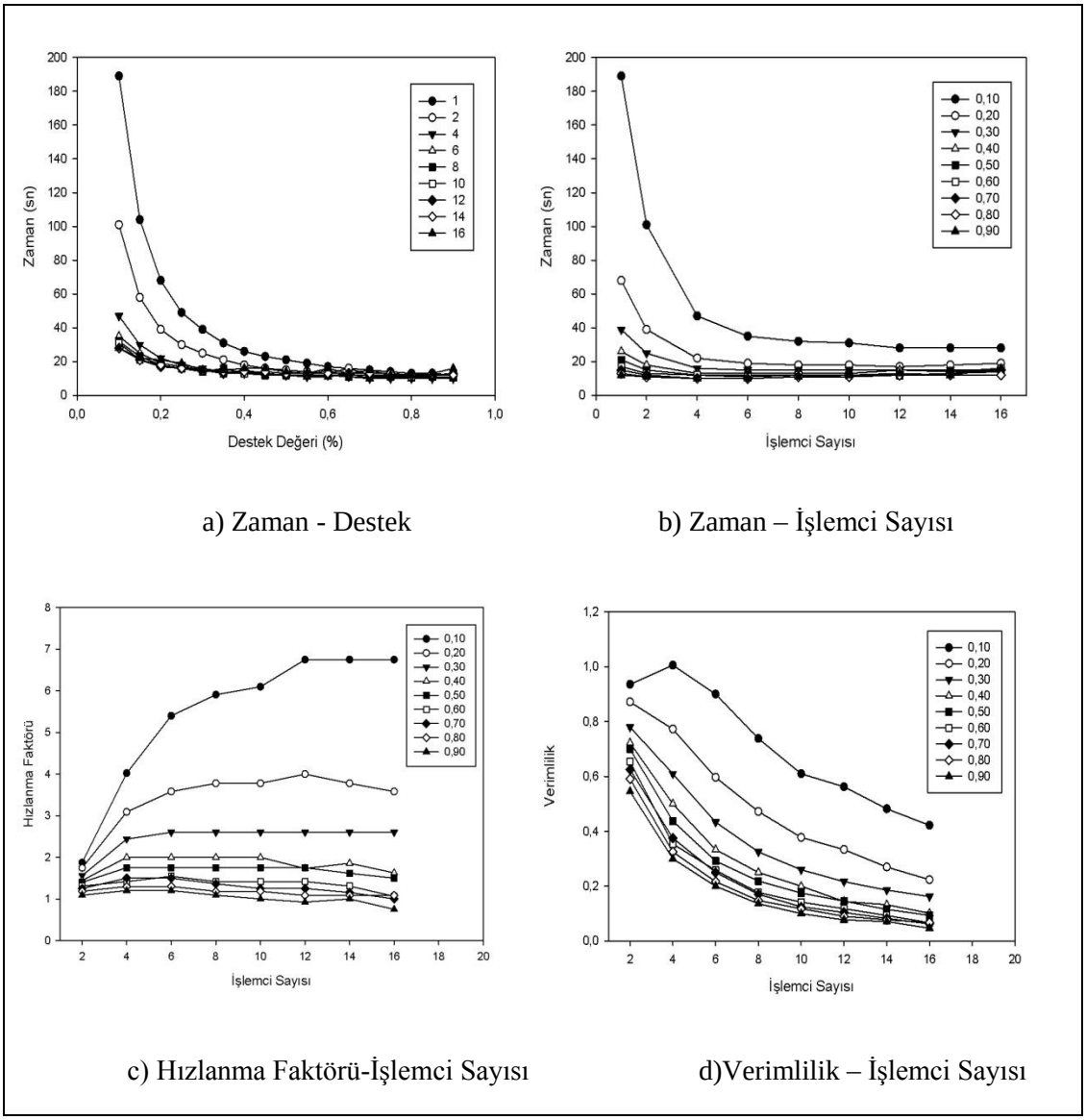
c) Hızlanma Faktörü-İşlemci Sayısı

d)Verimlilik – İşlemci Sayısı

Şekil 5. 18: `T20.I5.2500K` Test Sonuçları

### 5.4.2 Dinamik Paralel Öbek FP-Growth Testleri

Dinamik Paralel Öbek FP-Growth için önerilen algoritmanın çeşitli veritabanları üzerinde gerçekleştirilen sonuçları Şekil 5.19-5.24 grafikleri arasında verilmektedir. Bu yaklaşım da statik yaklaşıma benzer sonuçlar göstermektedir. Yani, işlemci sayısı arttıkça çalışma zamanında bir düşme görülür. Aynı zamanda destek değerleri düştükçe çalışma zamanı azalır. Ancak burada dinamik yaklaşımın statikten daha iyi sonuç vererek çalışma zamanını statiğe göre azaltması beklenmektedir. Bu durum yapılan testlerde gözlemlenmiştir. Bunun detayları karşılaştırmalı grafiklerde verilecektir (Bölüm 5.4.4). Ayrıca dinamik algoritma bazı durumlarda statik algoritmadan farklı bir davranış göstermektedir. Buradaki farklı davranış, Bölüm 2.8'de ifade edilen aşırı verimliliktir *(superlinearity)*. Bu durum en fazla olarak `T20.I5.500K` (Şekil 5.20 d), `T20.I5.1000K` (Şekil 5.21 d) veritabanlarında görülmüştür. Veritabanı büyüdükçe beklenilen verimlilik sınır değerlerine (E<=1) ulaşılmıştır. Bu veritabanları, diğerlerine göre daha küçüktür. Görülen aşırı verimliliğin sebebi olarak, verinin önbellekte daha kolay (hızlı) bir şekilde bulunmasından dolayı algoritmanın hızlı çalışması düşünülebilir. Yani algoritmik değil, aslında pratik olarak paralel hesaplamanın bir kazancıdır. Fakat algoritmaya zıt düşmemiş, aksine algoritmayı destekleyici bir unsur olarak gözlenmiştir. Bu durumun diğerlerine göre daha küçük olan `retail` veritabanında gözlenmemesinin sebebi ise sık öge kümelerinin az sayıda olmasıdır. Ayrıca verimlilik grafikleri incelendiğinde, büyük destek değerlerinde en uygun işlemci sayısının 4 olduğu söylenebilir. Destek değeri %0,4'den yukarı olanlar için işlemci sayısının arttırılmasının verimliliği çok da arttırmadığı gözlenir (E<0,5). Çünkü 4 işlemciden sonra birden hızlı bir düşüş, ardından işlemci sayısının artmasıyla daha yavaş bir düşüş gözlemlenmektedir. Küçük destek değerlerinde neredeyse tüm durumlarda 16 işlemcinin de verimli bir şekilde kullanıldığı gözlenmiştir.



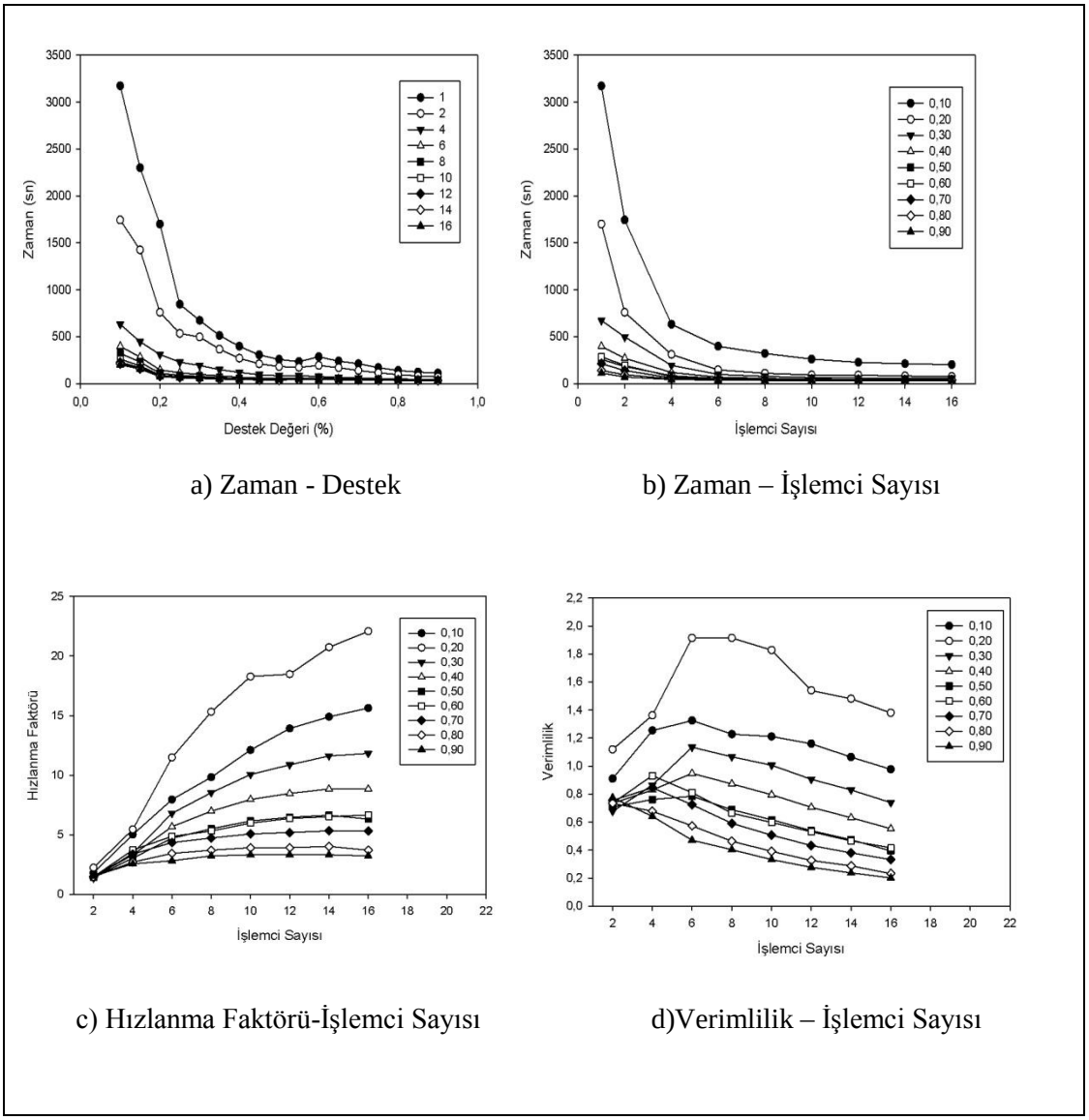

a) Zaman - Destek

b) Zaman – İşlemci Sayısı

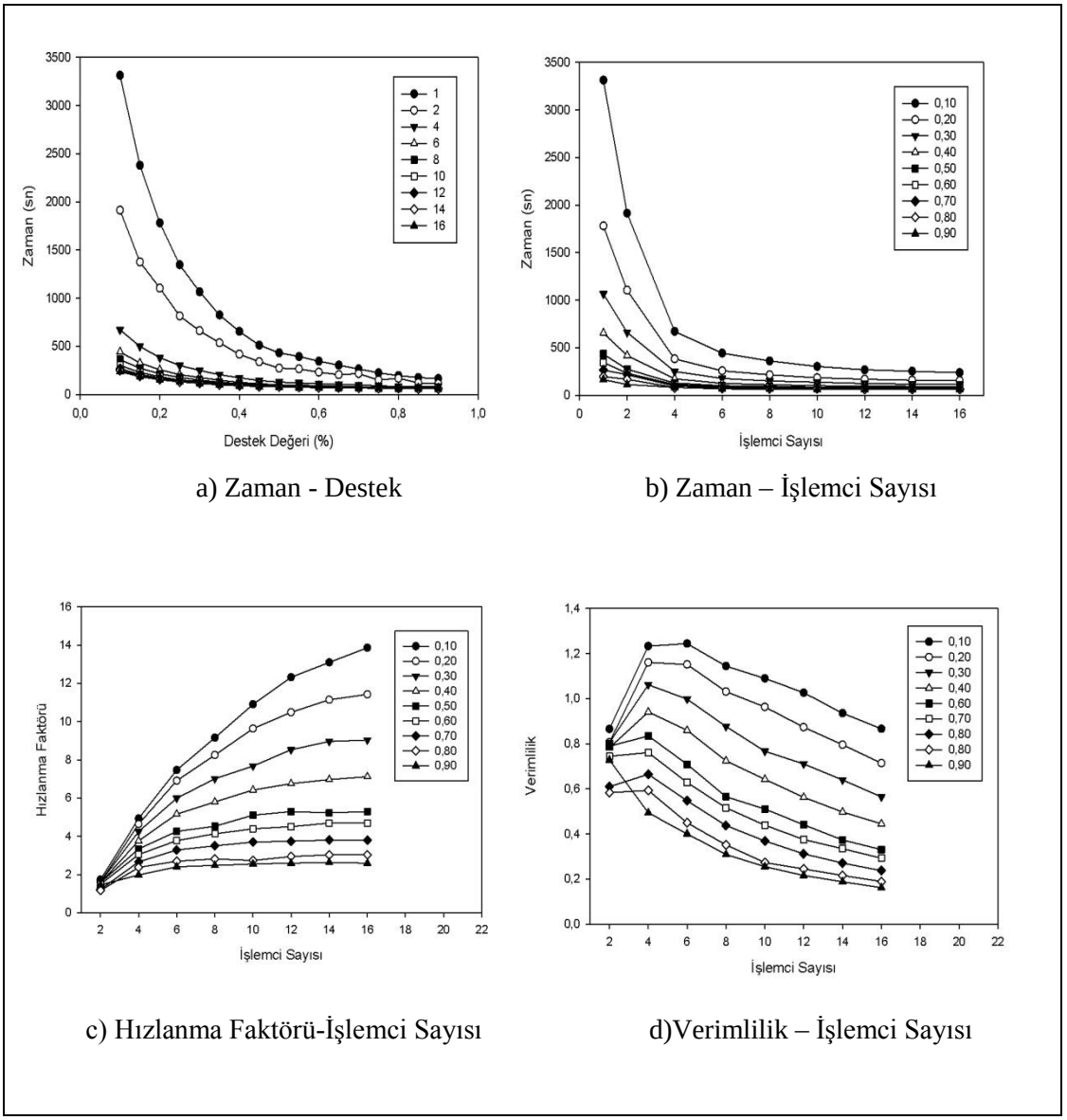
c) Hızlanma Faktörü-İşlemci Sayısı

d)Verimlilik – İşlemci Sayısı

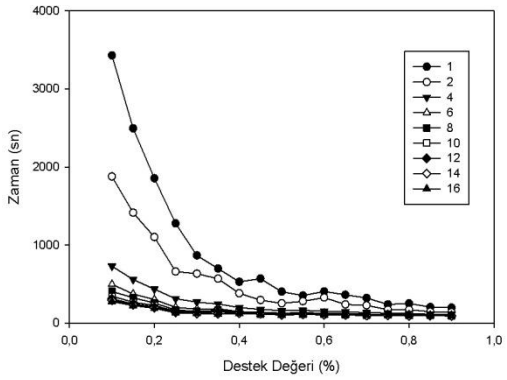
Şekil 5. 19: `retail` Test Sonuçları



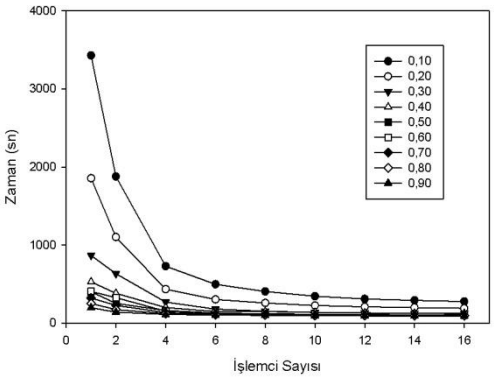

Şekil 5. 20: `T20.I5.500K` Test Sonuçları



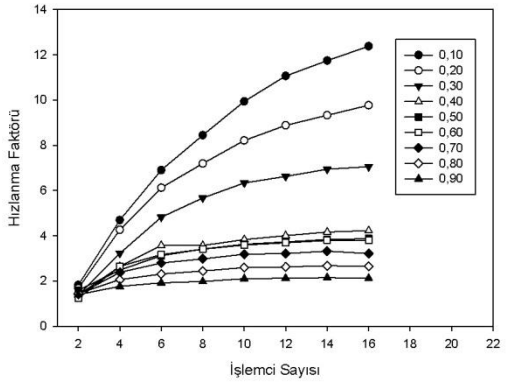

Şekil 5. 21: `T20.I5.1000K` Test Sonuçları



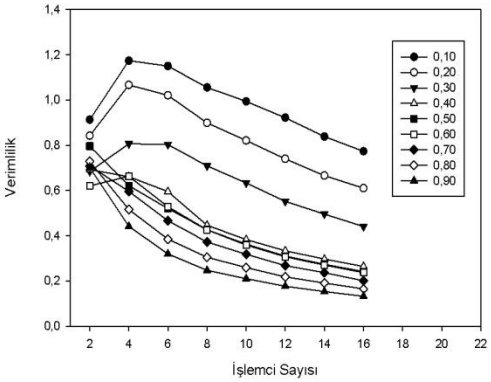

a) Zaman - Destek




b) Zaman – İşlemci Sayısı

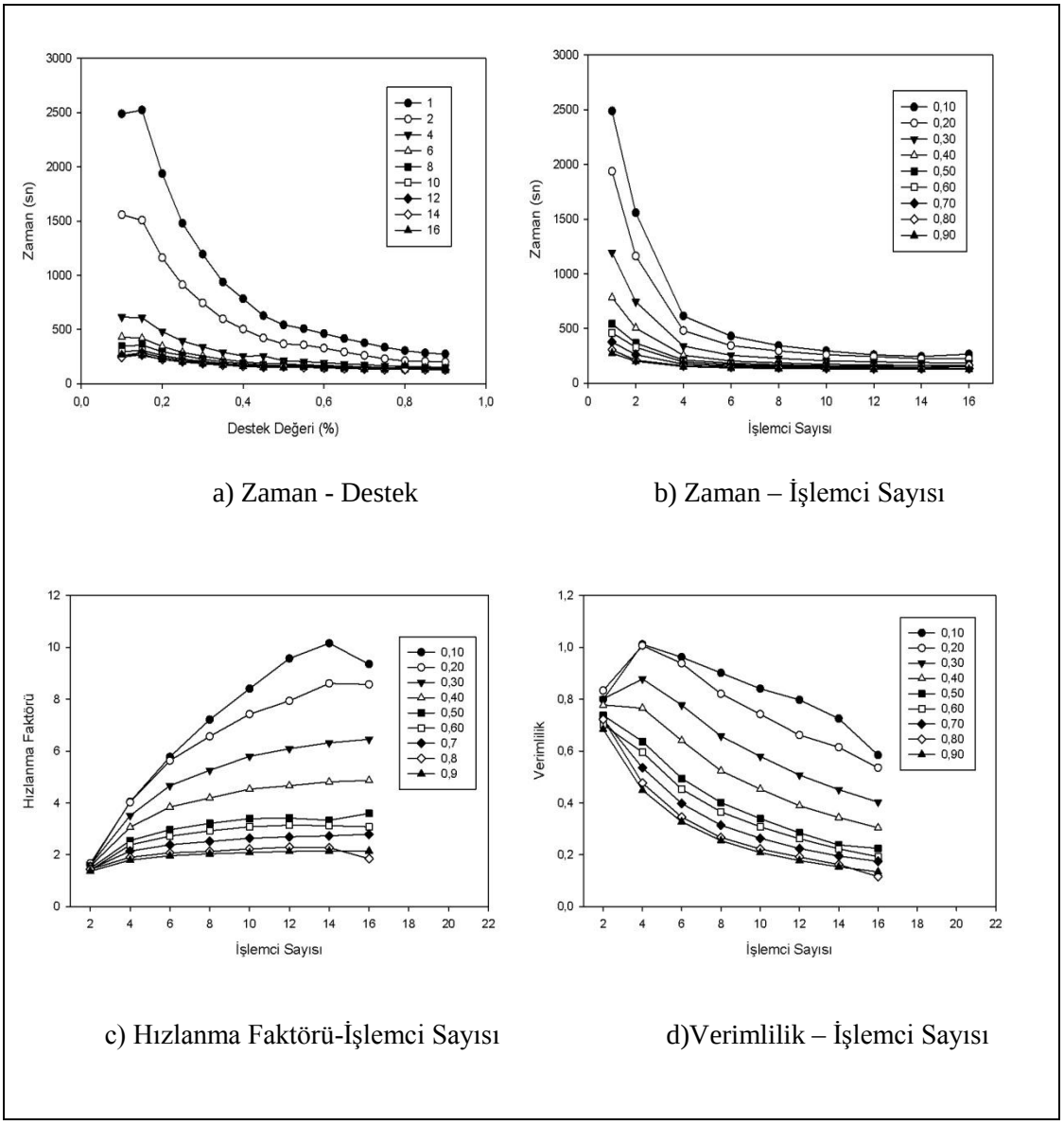



c) Hızlanma Faktörü-İşlemci Sayısı




d)Verimlilik – İşlemci Sayısı

Şekil 5. 22: `T20.I5.1500K` Test Sonuçları



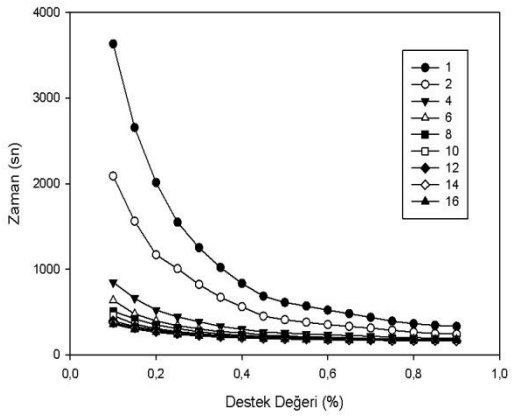

a) Zaman - Destek

b) Zaman – İşlemci Sayısı

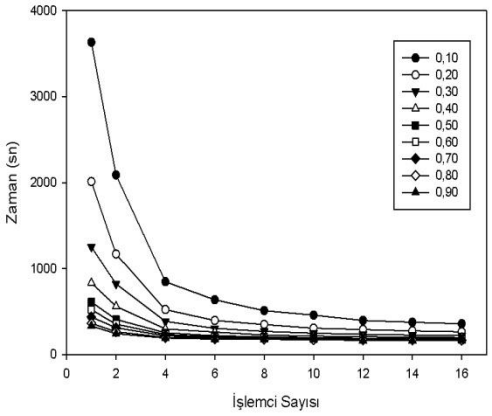
c) Hızlanma Faktörü-İşlemci Sayısı

d)Verimlilik – İşlemci Sayısı

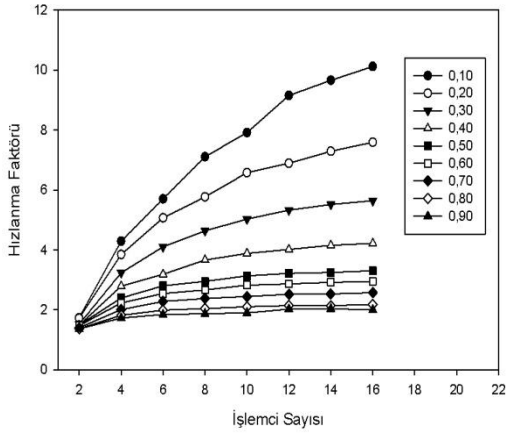
Şekil 5. 23: `T20.I5.2000K` Test Sonuçları



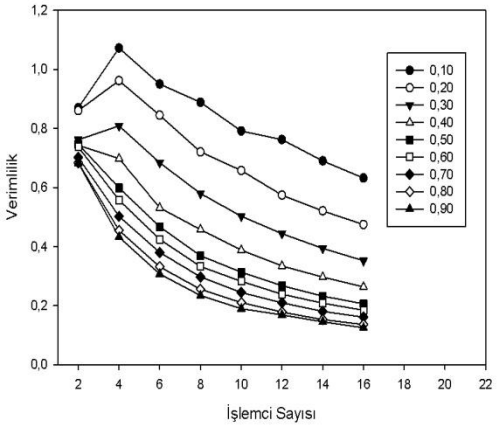

a) Zaman - Destek




b) Zaman – İşlemci Sayısı




c) Hızlanma Faktörü-İşlemci Sayısı




d)Verimlilik – İşlemci Sayısı

Şekil 5. 24: `T20.I5.2500K` Test Sonuçları

Dinamik Paralel Öbek FP-Growth algoritmasında statikten farklı olarak bir mesajlaşma söz konusudur. Bu mesajlaşmanın çok fazla bir iletişim zamanına sahip olduğu düşünülmezken, çalışma zamanı haberleşme ve hesaplama zamanı olarak incelendiğinde `T20.I5.2500K` için Çizelge 5.4'deki sonuçlar elde edilmiştir. Burada, *p* işlemci sayısını, $t_{haber}$ haberleşme zamanını, $t_{hesap}$ ise hesaplama zamanını göstermektedir. Elde edilen sonuçlar, haberleşme zamanının hesaplama zamanına göre oldukça küçük olduğunu göstermektedir. Bundan dolayı haberleşme zamanı ihmal edilebilir. Bu sonuç Bölüm 2.7'de anlatılan verimli paralel algoritmaların özelliğiyle de örtüşmektedir. Yani, işlemciler arasındaki haberleşme zamanı mümkün olduğunca azaltılarak işlemcilerin sürekli çalışır durumda tutulması sağlanmıştır.

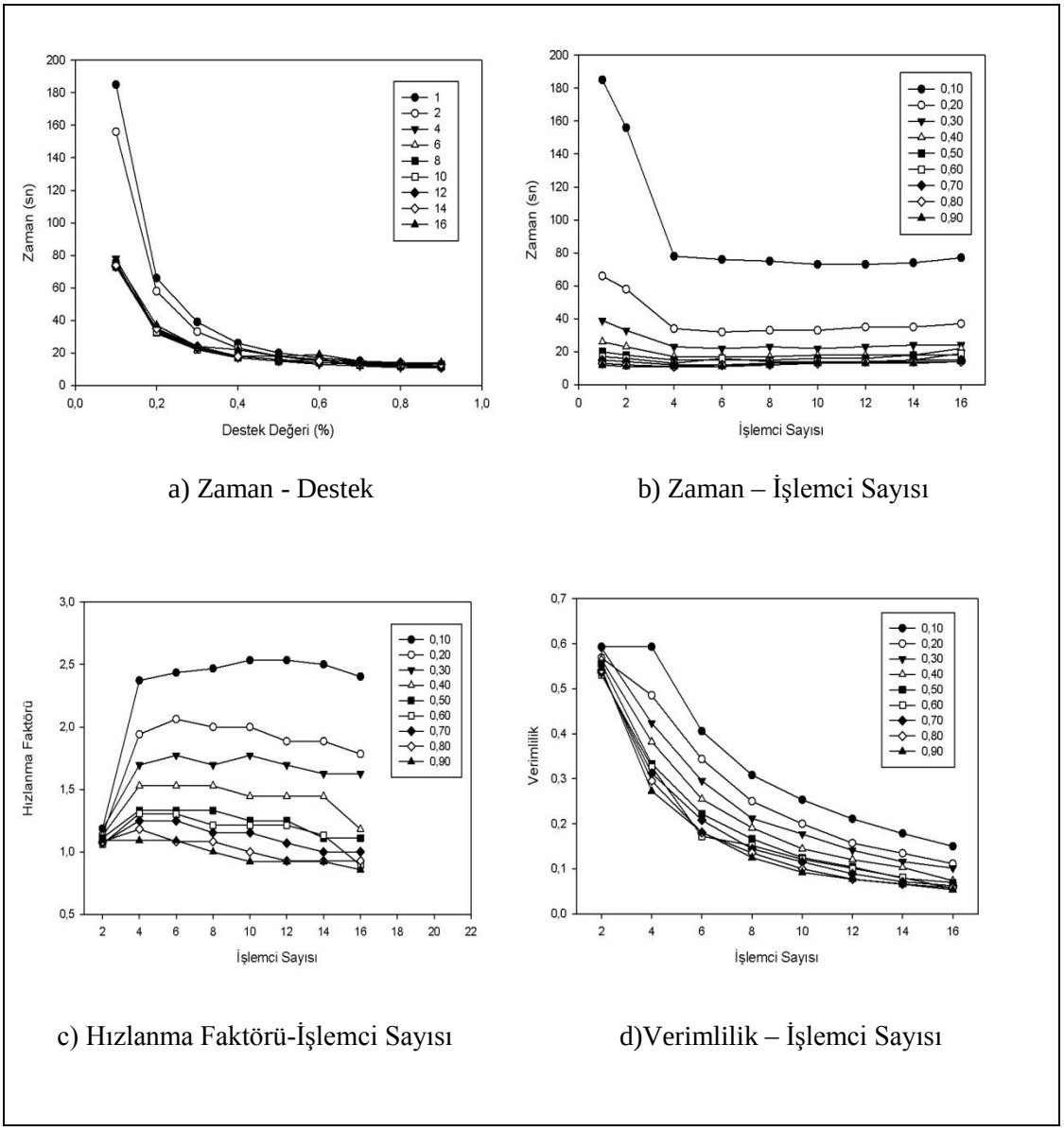
Çizelge 5. 4: `T20.I5.2500K` Çalışma Zamanı Analizi

| | destek = % 0,1 | | destek = % 0,2 | |
|---|---|---|---|---|
| *p* | $t_{haber}$ (sn) | $t_{hesap}$(sn) | $t_{haber}$ (sn) | $t_{hesap}$ (sn) |
| 2 | 0,016 | 1992,984 | 0,016 | 982,984 |
| 4 | 0,027 | 802,973 | 0,027 | 487,973 |
| 6 | 0,039 | 586,961 | 0,007 | 337,993 |
| 8 | 0,039 | 509,961 | 0,035 | 281,997 |
| 10 | 0,035 | 419,965 | 0,019 | 254,981 |
| 12 | 0,043 | 373,957 | 0,027 | 238,973 |
| 14 | 0,027 | 348,973 | 0,035 | 260,965 |
| 16 | 0,023 | 334,977 | 0,035 | 254,965 |

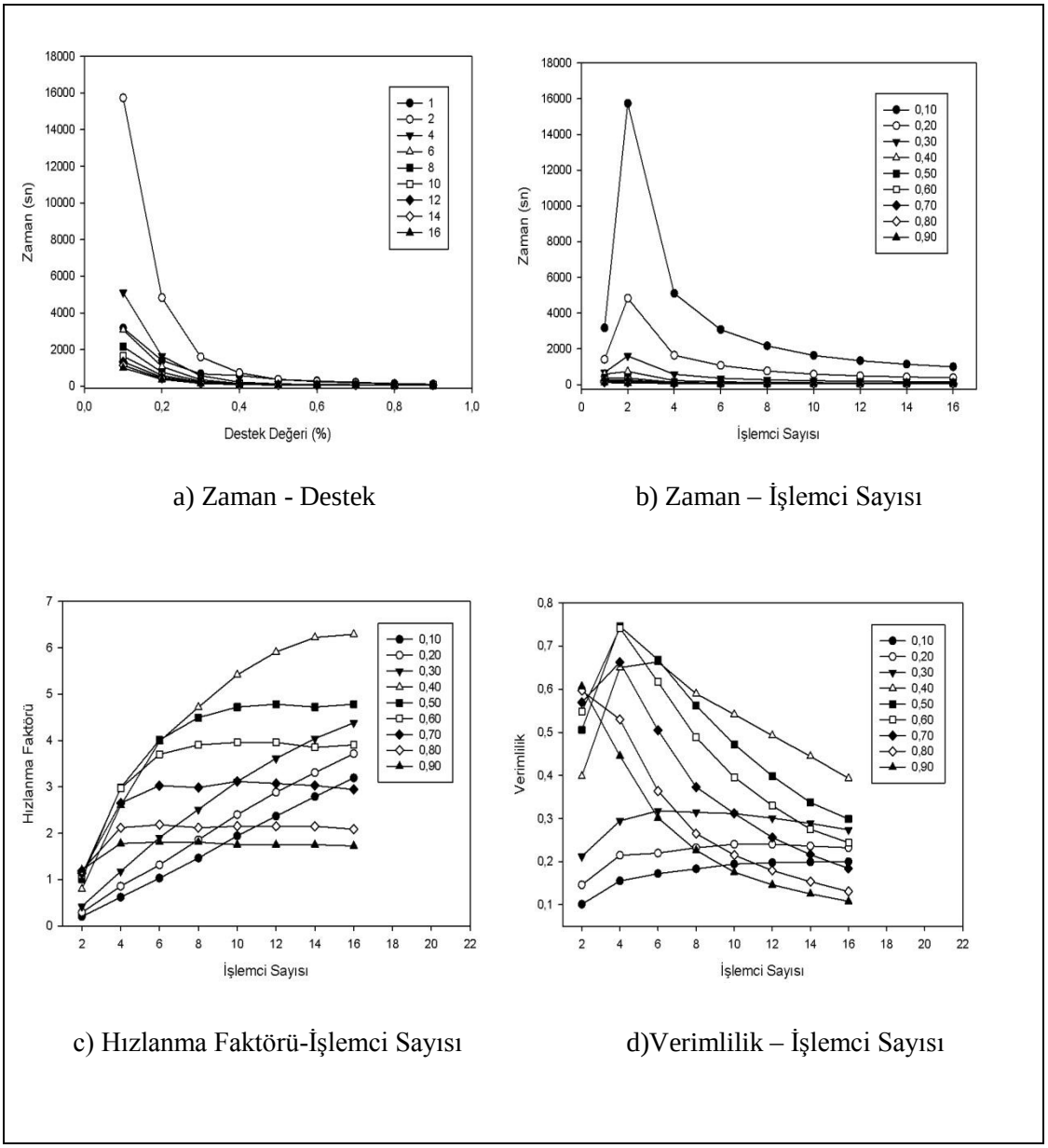
### 5.4.3 Ağaç Gönderimli Dinamik Paralel Öbek FP-Growth Testleri

Ağaç Gönderimli Dinamik Paralel Öbek FP-Growth algoritmasının `retail` ve `T20.I5.500K` veritabanları için test edilen sonuçları Şekil 5.25-5.26'dadır.

Ağaç Gönderimli Dinamik Paralel Öbek FP-Growth algoritmasının diğerlerinden farklı bir davranış gösterdiği gözlemlenmektedir. Çalışma zamanının diğer algoritmalara göre başlangıçta arttığı, ancak işlemci sayısının artmasıyla kendi içinde orantılı olarak azaldığı görülür. Bu algoritmanın çalışma karakteristiği, (`T20.I5.500K`)'da göründüğü gibi küçük destek değerlerinde (%0,1, %0,2, %0,3 için) tüm işlemci sayıları için büyük oranda zaman artışı olması ve ardından destek değerinin büyümesiyle zamanın göreceli olarak düşme eğiliminde olmasıdır. Bu durum Bölüm 5.3'de belirtildiği gibi algoritmanın doğasından kaynaklanır. Ancak `T20.I5.500K`'ya göre daha küçük olan `retail` veritabanında ise algoritmanın diğer algoritmalara benzer bir yaklaşım sergilediği görülür. Yani, önce yavaş, işlemci sayısının artmasıyla hızlı çalışmaya başlar. Veritabanı büyüklüğünden dolayı, algoritmanın doğası gereği çalışma sürelerinin az sayıdaki işlemcilerde çok arttığı ve işlemci sayısı arttıkça düştüğü gözlenmiştir. Ama veritabanının büyümesi bu çalışma sürelerini çok arttırdığı için bu testlerin grafikleri konmamıştır. Genel beklenti, algoritmanın bu şekliyle yüksek veritabanlarında daha da kullanışsız olacağı yönündedir.




Şekil 5. 25: `retail` Test Sonuçları




Şekil 5. 26: `T20.I5.500K` Test Sonuçları

Ağaç Gönderimli Dinamik Paralel Öbek FP-Growth algoritması da dinamik paralel öbekte olduğu gibi mesajlaşma içerir. `T20.I5.500K` için çalışma zamanının hesaplama ve haberleşme olarak nasıl olduğu incelenirse Çizelge 5.5'deki gibi bir tablo ortaya çıkar. Burada, *p* işlemci sayısını, $t_{haber}$ haberleşme zamanını, $t_{hesap}$ ise hesaplama zamanını gösterir. Burada, haberleşme zamanının işlemci sayısının çoğalmasına bağlı olarak arttığı görülmektedir. Ancak yine de haberleşme zamanının hesaplama zamanına göre baskın olmadığı gözlenmektedir.

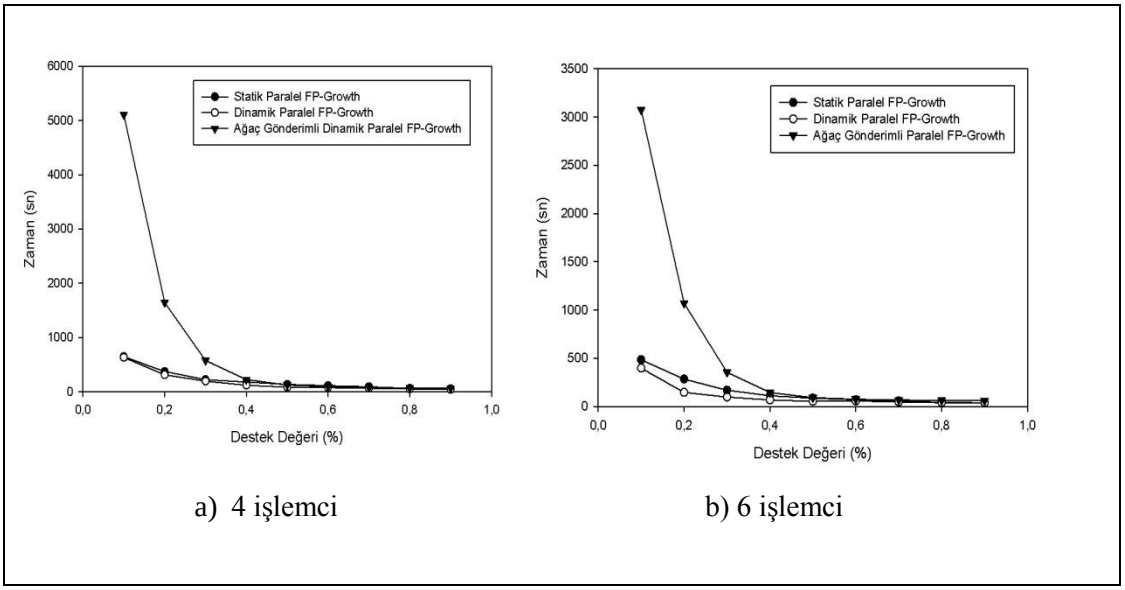
Çizelge 5. 5: `T20.I5.500K` Çalışma Zamanı Analizi

| | destek = % 0,1 | | destek = % 0,2 | |
|---|---|---|---|---|
| *p* | $t_{haber}$ (sn) | $t_{hesap}$ (sn) | $t_{haber}$ (sn) | $t_{hesap}$ (sn) |
| 2 | 0,59 | 15570,41 | 0,34 | 5076,66 |
| 4 | 0,65 | 5049,35 | 0,45 | 1691,55 |
| 6 | 10,31 | 3004,69 | 4,52 | 957,48 |
| 8 | 14,57 | 2152,43 | 6,32 | 723,68 |
| 10 | 16,81 | 1668,19 | 7,25 | 565,75 |
| 12 | 18,44 | 1368,56 | 7,93 | 470,07 |
| 14 | 19,43 | 1160,57 | 8,34 | 400,66 |
| 16 | 20,21 | 1011,79 | 8,66 | 334,34 |

### 5.4.4 Metotların Karşılaştırılması

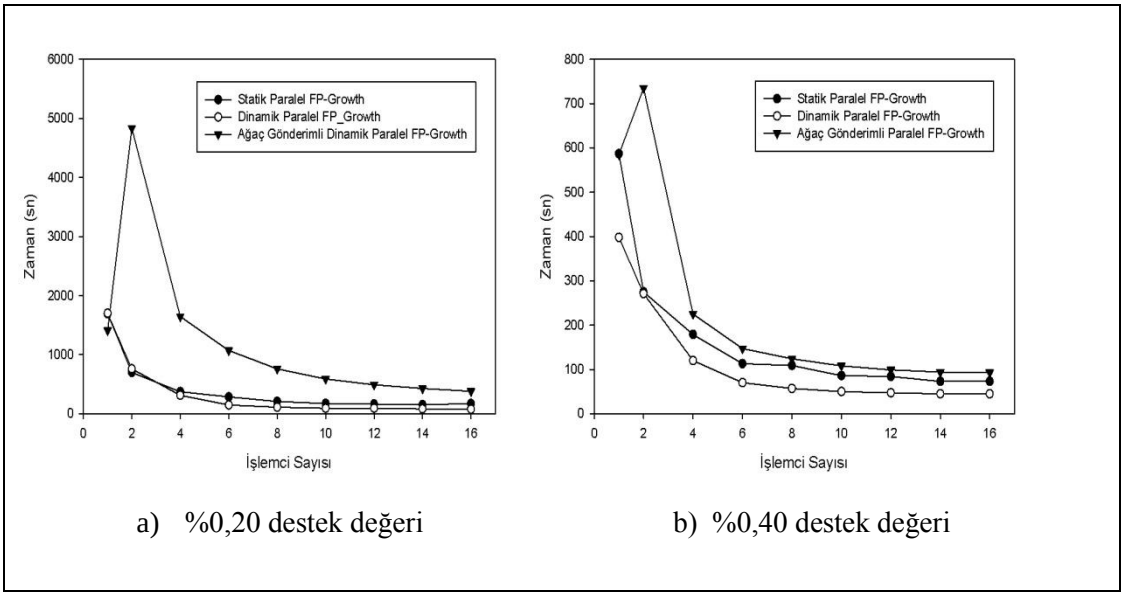
Şekil 5.27-5.31'de geliştirilen üç metodun karşılaştırıldığı grafikler mevcuttur. Her üç metot için zaman-destek değeri ve zaman-işlemci sayısı grafikleri incelendiğinde, en etkili metodun Dinamik Paralel Öbek FP-Growth algoritması olduğu görülmektedir. Bu sonuç veritabanı büyüdükçe daha iyi gözlemlenir. Nitekim T20.I5.500K veritabanında `retail` veritabanına göre daha iyi gözlenir. Ağaç Gönderimli Dinamik Öbek FP-Growth algoritmasının büyük veritabanlarında aşırı bir zaman artışına sebep olmasından dolayı ağaç gönderiminin beklenildiği kadar performanslı olmadığı gözlemlenmektedir. Ancak gözlemlenen diğer bir nokta ise,

bu algoritmanın küçük veritabanlarında aşırı bir hızlanmaya sebep olduğudur (Şekil 5.29). Buradan çıkan sonuç, ağaç gönderiminin küçük ağaçlar üzerinde etkili olabileceğidir. Zaman-veritabanı büyüklüğü grafikleri incelendiğinde ise, paralel algoritmaların seri algoritmaya göre büyük oranda kazançlı olduğu görülmektedir. İşlemci sayısı 4 iken çalışma zamanında dörtte bir oranında bir azalma görülmüştür. Bu sonuç, çalışılan veritabanı büyüklükleri için en uygun işlemci sayısının dört olduğunu gösterir. Yine zaman-destek değeri grafikleri incelendiğinde, 4 işlemciden 6 işlemciye geçildiğinde çalışma zamanında yaklaşık %50 azalma görülmektedir. Bu durum, 6 işlemcinin de başka bir seçenek olduğunu gösterir. Ama işlemci sayısını daha fazla artırmak önemli bir katkı yapmamaktadır. Veritabanı büyüdükçe daha yüksek işlemci sayılarının en verimli olabileceği duruma ulaşılabileceği, yani algoritmanın ölçeklenebilir olduğu sonucu çıkarılabilir. Diğer görülen sonuç ise dinamik yaklaşımın statiğe göre daha iyi sonuç verdiği ve çalışma zamanında azalmaya sebep olduğudur. Bu, veritabanı büyüklüğünün artmasıyla daha iyi gözlemlenebilir.



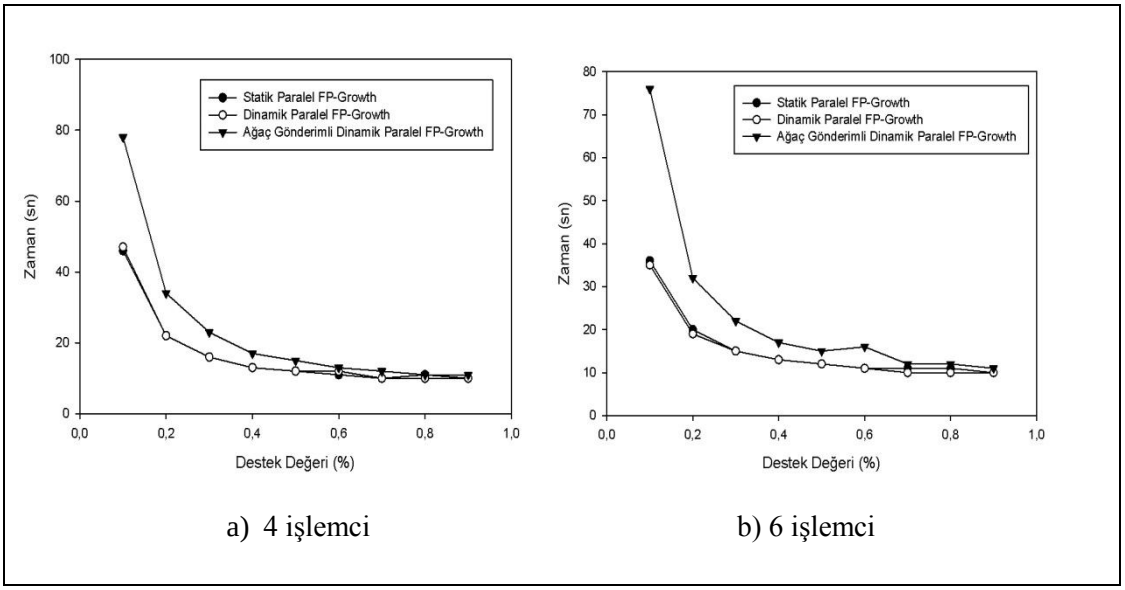

a) 4 işlemci b) 6 işlemci

Şekil 5. 27: T20.I5.500K için Zaman-Destek Değeri Grafiği



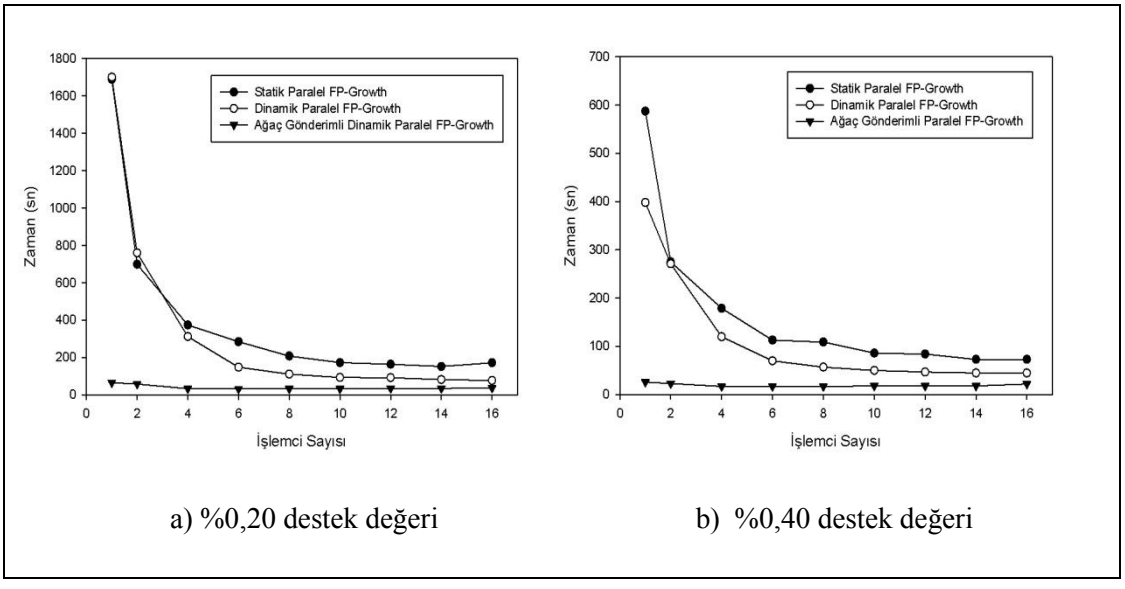

a) %0,20 destek değeri b) %0,40 destek değeri

Şekil 5. 28: T20.I5.500K için Zaman-İşlemci Sayısı Grafiği



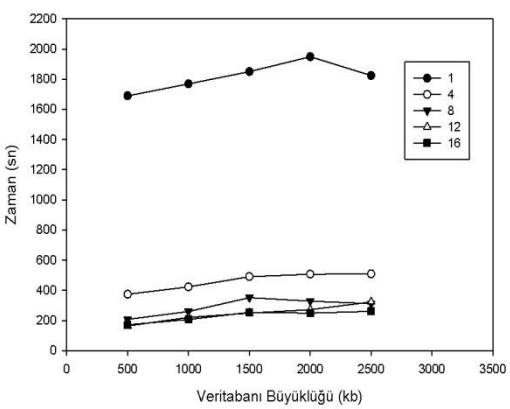

a) 4 işlemci b) 6 işlemci

Şekil 5. 29: `retail` için Zaman-Destek Değeri Grafiği

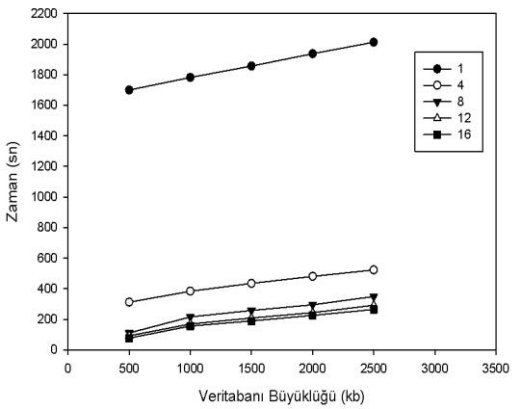



a) %0,20 destek değeri b) %0,40 destek değeri

Şekil 5. 30: `retail` için Zaman-İşlemci Sayısı Grafiği




a) Statik Paralel Öbek FP-Growth




b) Dinamik Paralel Öbek FP-Growth

Şekil 5. 31: Zaman-Veritabanı Büyüklüğü Grafiği

# BÖLÜM 6

# SONUÇ

Bu çalışmada FP-Growth algoritması için öbek bilgisayarlar (*clusters*) üzerinde çalışan 3 farklı yaklaşım gerçekleştirilmiştir. Çalışmada hedeflenen, daha önce yapılan ve veritabanının bölünmesi esasına dayanan yaklaşımlardan farklı olarak, veritabanının öbek bilgisayarın her düğümünde mevcut olduğu varsayılarak, veriyi bölmek yerine görevi bölmenin ne gibi bir kazanım sağlayacağıdır. Ayrıca, veritabanını bölmek yerine gerektiğinde sık örüntü ağacını diğer işlemcilere göndererek algoritma paralelleştirildiğinde bir kazanım sağlayıp sağlamayacağını gözlemlemektir. Bunlardan çalışma çerçevesinde geliştirilen Statik Paralel Öbek FP-Growth ve Dinamik Paralel Öbek FP-Growth algoritmasında veritabanının dolayısıyla sık örüntü ağacının her işlemcide mevcut bulunduğu varsayılarak görev dağılımı şeklinde gerçekleştirilen paralel bir algoritma tasarlanmıştır. Ağaç Gönderimli Dinamik Paralel Öbek FP-Growth'da ise veritabanı dolayısıyla sık örüntü ağacı sadece bir işlemcide tutularak, gerektiğinde alt ağaçlar diğer işlemcilere gönderilerek yine görev dağılım esasına dayanan paralel bir algoritma gerçekleştirilmiştir. Bu üç algoritma iki farklı öbek üzerinde IBM Dataset Generator tarafından oluşturulan veritabanları ve `retail` gerçek veritabanı üzerinde test edilerek sonuçları analiz edilmiştir.

Çıkan sonuçlara göre, statik ve dinamik paralel öbek FP-Growth algoritmaları seri koda göre oldukça iyi bir performans göstermiştir. Öyle ki, iki işlemci kullanıldığı durumda bile dörtte bir oranında çalışma zamanını azalttığı görülmüştür. Beklenen bir durum olarak, algoritmaların küçük destek değerlerinde daha yavaş, büyük destek değerlerinde ise daha hızlı çalıştığı görülmüştür. Bu iki algoritma ayrı olarak incelendiğinde ise dinamik yaklaşımın çalışma zamanını statik yaklaşıma oranla daha çok azalttığı görülmüştür. Aynı zamanda öbek içerisinde en verimli çalışacak işlemci sayısı bulunarak, veritabanı büyüklüğünün artmasıyla bu kazancın artabileceği sonucuna varılmıştır.

Ağaç gönderimi esasına dayanan yaklaşımda ise büyük ağaçlar üzerinde çalışma zamanında çok yavaşlama görülürken, küçük ağaçlar üzerinde de hızlanma görülmüştür. Dolayısıyla, büyük ağaçlar için bu şekilde bir gönderimin bir kazanım sağlamayacağı, küçük ağaçların gönderilmesi esasına dayanan başka bir paralel algoritma içerisinde verimli bir çalışmaya dönüşebileceği sonucuna varılmıştır.

Gelecekte, Ağaç Gönderimli Dinamik Paralel Öbek FP-Growth'un daha verimli bir algoritması üzerinde çalışmak hedeflenmektedir. Bu çalışmada, literatürde geçen paralel FP-Growth algoritmasının gerçekleştirdiği gibi hem veritabanı bölünmesi hem de yapılan bu çalışmada olduğu gibi ağaç gönderiminin olması düşünülmektedir.

## KAYNAKLAR


[1] Tan, P., Steinbach, M., Kumar, V., Introduction to Data Mining, *Pearson Education*, 2006.

[2] Wilkinson, B., Allen, M., Parallel Programming: Techniques and Application Using Networked Workstations and Parallel Computers, *Prentice Hall*, Upper Saddle River, New Jersey, 2005.

[3] "Introduction to Parallel Computing" erişim adresi: https://computing.llnl.gov/tutorials/parallel_comp/, erişim tarihi: 19 Aralık 2009.

[4] "Wikipedia, the free encyclopedia" erişim adresi: http://en.wikipedia.org/wiki/Distributed_computing, erişim tarihi: 19 Aralık 2009.

[5] "Wikipedia, the free encyclopedia" erişim adresi: http://en.wikipedia.org/wiki/Cluster_(computing), erişim tarihi: 20 Aralık 2009.

[6] "Tr-Grid Sayfası" erişim adresi: http://wiki.grid.org.tr/index.php/Ana_sayfa, erişim tarihi: 20 Aralık 2009.

[7] "Wikipedia, the free encyclopedia" erişim adresi: http://tr.wikipedia.org/wiki/SETI@home, erişim tarihi: 21 Aralık 2009.

[8] "GridMiner" erişim adresi: http://www.gridminer.org/, erişim tarihi: 20 Aralık 2009.

[9] "Message Passing Interface" erişim adresi: https://computing.llnl.gov/tutorials/mpi/, erişim tarihi: 26 Aralık 2009.

[10] Özdoğan, G.Ö., Abul, O., Yazıcı, A., Paralel Veri Madenciliği Algoritmaları, 1. Ulusal Yüksek Başarım ve Grid Konferansı, Proc. of BAŞARIM'09, 131-137, Ankara, Nisan 2009.

[11] Michalski, R.S., Stepp, R.E., Automated Construction of Classifications: Conceptual Clustering Versus Numerical Taxonamy, IEEE Transactions on Pattern Analysis and Machine Intelligence, 5, 219-243, 1983.

[12] Zhong, N., Ohsuga, S., Discovering Concept Clusters by Decomposing Databases, Data and Knowledge Engineering, 12(2), 223-244, 1994.

[13] Agrawal, R., Imielinski, T. Swami, A. Mining Association Rules Between Sets of Items in Large Databases, Proc. of the 1993 ACM SIGMOD International Conference on Management of Data, 207-216, Washington, D.C., United States, 1993.

[14] Agrawal, R., Srikant, R., Fast Algorithms for Mining Association Rules, Proc. of the 20th International Conference on Very Large Data Bases, 487-499, Santiago de Chile, Chile, 1994.

[15] Zaki, M.J., Ogihara, M., Theoretical Foundation of Association Rules, Proc. of 3rd. SIGMOD'98 Workshop on Research Issues in Data-Mining and Knowledge Discovery, 1-8, Seattle, Washington, USA, 1998.

[16] Chen, M.S., Han, J., Yu, P.S., Data Mining: An Overview From a Database Perspective, IEEE Transactions on Knowledge and Data Engineering, 8(6), 866-883, 1996.

[17] Han, J., Pei, J., Yin, Y., Mining Frequent Patterns Without Candidate Generation, Proc. of the 2000 ACM SIGMOD International Conference on Management of Data, 1-12, Dallas, Texas, United States, 2000.
[18] Skillicorn D., Strategies for Parallel Data Mining, IEEE Concurency, 7(4), 26-35, 1999.
[19] Zaki, M.J., Parallel and Distributed Association Mining: A Survey, IEEE Concurrency, 7(4), 14-25, 1999.
[20] Coenen, F., Leng, P., Partitioning Strategies for Distributed Association Rule Mining, The Knowledge Engineering Review, 21(1), 25-47, 2006.
[21] Oguchi, M., Kitsuregawa, M., Parallel Data Mining on ATM-Connected PC Cluster and Optimization of Its Execution Environments, Lecture Notes in Computer Science, 1800, 366-373, 2000.
[22] Pramudiono, I., Kitsuregawa, M., Parallel FP-Growth on PC cluster, Proc. of the 7th Pacific-Asia Conference of Knowledge Discovery and Data Mining (PAKDD03), 467-473, Seoul, Korea, April-May 2003.
[23] Lan, Y-J., Qiu, Y., Parallel Frequent Itemsets Mining Algorithms Without Intermediate Results, Proc. of 2005 International Conference on Machine Learning and Cybernetics, 2102-2107, Guangzhou, China, August 2005.
[24] Zaiane, O.R., El-Hajj, M., Lu, P., Fast Parallel Association Rule Mining Without Candidacy Generation, Proc of the 2001 IEEE International Conference on Data Mining, 665-668, San Jose, CA, USA, 2001.
[25] Li, H., Wang, Y., Zhang, D., Zhang, M., Chang, E.Y., PFP: Parallel FP-Growth for Query Recommendation, Proc. of the 2008 ACM Conference on Recommender Systems, 107-114, Lausanne, Switzerland, 2008.
[26] Savasere, A., Omiecinski, E., Navathe, S.B., An Efficient Algorithm for Mining Association Rules in Large Databases, Proc. of the 21th International Conference on Very Large Data Bases, 432 - 444, 1995.
[27] "Frequent Itemset Mining Dataset Repository" erişim adresi: http://fimi.cs.helsinki.fi/data/, erişim tarihi: 1 Kasım 2009.

# ÖZGEÇMİŞ

## Kişisel Bilgiler


| | |
|---|---|
| Soyadı, adı | : Özdemir Özdoğan, Gülistan |
| Uyruğu | : T.C. |
| Doğum tarihi ve yeri | : 05.03.1983 Eskişehir |
| Medeni hali | : Evli |
| Telefon | : 0 (312) 292 42 68 |
| Faks | : 0 (312) 292 41 80 |
| e-mail | : gozdemir@etu.edu.tr |


## Eğitim

| Derece | Eğitim Birimi | Mezuniyet tarihi |
|---|---|---|
| Lisans | Çankaya Üniversitesi/Bilgisayar Mühendisliği | 2006 |

## İş Deneyimi

| Yıl | Yer | Görev |
|---|---|---|
| 2007-2010 | TOBB Ekonomi ve Teknoloji Üniversitesi | Burslu Yüksek Lisans Öğrencisi |
| 2007 | Altay Grup | Yazılım Mühendisi |

## Yabancı Dil

İngilizce
Almanca

## Yayınlar


Özdoğan, G.Ö., Abul, O., Yazıcı, A., Paralel Veri Madenciliği Algoritmaları, 1. Ulusal Yüksek Başarım ve Grid Konferansı, Proc. of BAŞARIM'09, 131-137, Ankara, Nisan 2009.