KOCAELİ ÜNİVERSİTESİ * MÜHENDİSLİK FAKÜLTESİ

Fonksiyonel Programlama Dilleri ile Paralel Programlama

BİTİRME ÇALIŞMASI

Uğurcan Ergün

Bilgisayar Mühendisliği

Danışman: Yard. Doç. Dr. Ahmet SAYAR

KOCAELİ, 2013

KOCAELİ ÜNİVERSİTESİ * MÜHENDİSLİK FAKÜLTESİ

Fonksiyonel Programlama Dilleri ile Paralel Programlama

BİTİRME ÇALIŞMASI

Uğurcan Ergün

Bilgisayar Mühendisliği

Danışman: Yard. Doç. Dr. Ahmet SAYAR

KOCAELİ, 2013

Fonksiyonel Programlama Dilleri ile Paralel Programlama

Uğurcan Ergün

Anahtar Kelimeler: Paralel ve Dağıtık Sistemler, Fonksiyonel Programlama, Aktör Modeli, İşlemsel Bellek, Erlang, Scala

Özet: Günümüzde paralel sistemler büyük bir önem arz etmektedirler. Bu sistemler kullanılarak hesaplamalar daha hızlı ve verimli şekilde yapılabilmektedir. Ancak yazılımlarında bu yapıdan faydalanabilecek şekilde tasarlanmaları gerekmektedir ve hakim programlama paradigması olan imperatif dillerle bu iş görece zor ve maliyetli olabilmektedir. Bu sorunla daha basit çözümler üretebilen fonksiyonel paradigma günümüzde gittikçe yaygınlaşmaktadır. Bu çalışmada bazı parelel programlama modelleri ve teknikleri incelenmiş ve bazı fonksiyonel programlama dillerinde bu model ve tekniklerin nasıl gerçeklendiği ele alınmıştır.

Paralel Programming with Functional Programming Languages

Uğurcan Ergün

Keywords: Paralel and Distributed Systems, Functional Programming, Actor Model, Ttransactional Memory, Erlang, Scala

Abstract: In present day paralel systems has a significant importance. These systems could make computations faster and more efficent. However software has to be designed accordingly and using the common imperative paradigm it could be relatively harder and costs more. Because of coming with easier solutions to this problem, functional paradigm is getting more popular nowadays. In this paper some paralel programming models and techniques are reviewed. It also examines how these models and techniques are implemented on functional programming languages.

ÖNSÖZ ve TEŞEKKÜRLER

Mikroişlemci teknolojisi geçtiğimiz 8-10 yıl içerisinde büyük bir değişim yaşanmıştır. İşlemci çekirdek hızlarını daha fazla arttıramayacağını anlayan mühendisler çareyi işlemcileri birlikte kullanmakta görmüşlerdir. Ancak programların daha fazla işlemci kullanılınca beklenildiği kadar hızlanmamasıyla yazılımların da bu mimariye uygun olarak tasarlanmaları gerektiği anlaşılmıştır.

Bugüne kadar çok sınırlı bir alan olarak kalmış paralel programlama önem kazanmıştır. Ancak geçen zamanla beraber alışılageldik programlama yöntemlerimizin paralellikle uyuşmasında sorunlar olduğu gözükmüştür. Bu problemlere yeni çözümler ararken aslında oldukça eski olan ancak akademi dışında pek kullanılmayan fonksiyonel programlama paradigmasının paralel sistemlerle daha iyi çalışabileceği düşünülmeye başlanmış ve günümüzde bu paradigma oldukça ilgi görmektedir.

Beni bu konuda araştırma yapmaya yönelten, araştırma sürecinde her türlü kolaylığı sağlayan ve yardımlarını esirgemeyen proje danışmanım Yard. Doç. Dr. Ahmet SAYAR'a ve süreç içerisinde beni destekleyen aileme teşekkürlerimi sunmak isterim.

İÇİNDEKİLER

ŞEKİLLER DİZİNİ

TABLOLAR DİZİNİ

SEMBOLLER

pk : Maksimum performans kazancı (speedup)

f : Bir problemin seri kısmının probleme oranı

n : İşlemci sayısı

Kısaltmalar

ALU : Aritmetik ve Mantık Ünitesi / Arithmetic and Logic Unit

SISD :Tekil Komut, Tekil Veri Akışı / Single Instruction Single Data Stream

SIMD : Tekil Komut, Çoğul Veri Akışı / Single Instruction Multiple Data Stream

MISD : Çoğul Komut, Tekil Veri Akışı / Multiple Instruction Single Data Stream

MIMD : Çoğul Komut, Çoğul Veri Akışı / Multiple Instruction Multiple Data Stream

GPGPU: Grafik İşlemcisinde Genel Amaçlı Hesaplama / General Purpose Computing on Graphical Processing Unit

I/O : Giriş-Çıkış / Input-Output

OTP : AçıkTelekom Platformu / Open Telecom Platform

MPL : Mozilla Kamu Lisansı / Mozilla Public Licence

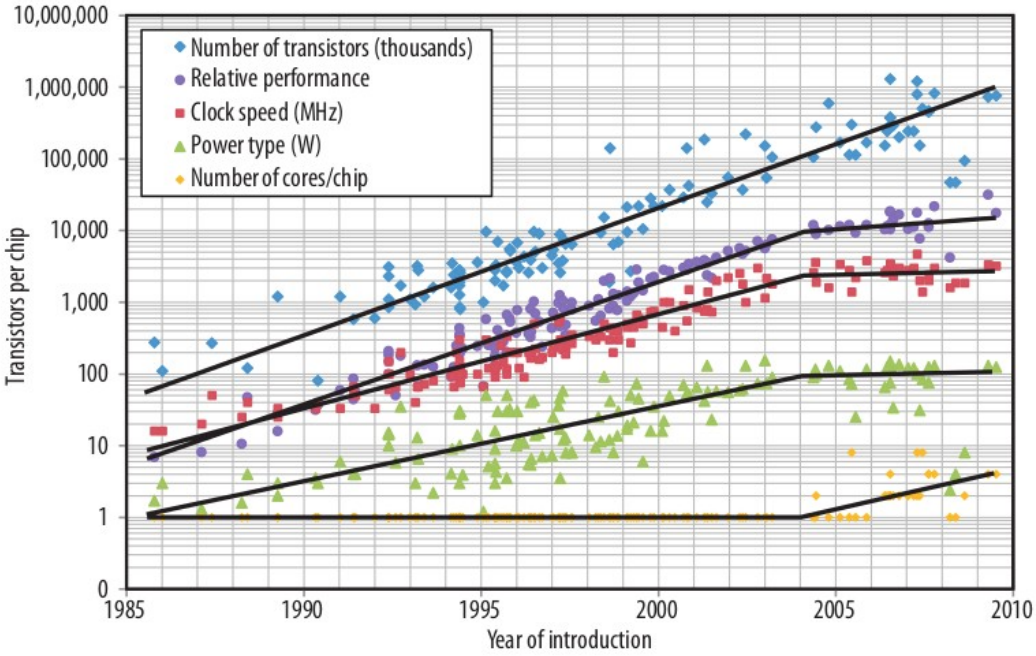
VM : Sanal Makina / Virtual Machine

JVM : Java Sanal Makinası / Java Virtual Machine

EPFL : Lozan Teknik Üniversitesi / École polytechnique fédérale de Lausanne

1. Giriş

Mikroişlemciler 1970'lerin başında ortaya çıktıklarından beri sürekli gelişmektedirler. Moore Yasası (Moore, 1965) entegre devrelerdeki transistör sayısının her iki yılda bir ikiye katlanacağını öngörmektedir. Artan transistör miktarı çoğunlukla performans artışı anlamına gelmekteydi. Donanımların sürekli gelişmesiyle beraber yazılımlarda hiç bir değişiklik yapılmadan performans artışı sağlamak mümkündü. Herb Sutter bu durumu "bedava yemek" olarak adlandırmıştır (Sutter, 2005). Ancak Şekil 1.1'de (Fuller,Millett, 2011) görüldüğü gibi transistör sayısı Moore'un öngürdüğü üzere artmaya devam etse de işlemci saat hızları 2004'ten sonra pek bir artış gösterememiştir. Bu duruma sebep olan etkenlerden biri de gerekli olan güç artışıdır. Bir noktadan sonra saat hızında yapılan ufak artışlar bile işlemciye gereken gücü büyük miktarda arttırabilmektedir. Donanım endüstrisi bu problemle başa çıkabilmek ve performansı makul koşullarla artırabilmek için bir entegre üzerine birden fazla işlem birimi koymaya başladılar.



Şekil 1.1: Mikroişlemcilerin tarihsel gelişimi

Bu geçişi destekleyen gözlemlerden biri işlemcinin mimarisinde yapılacak değişikliklerin veya eklenecek özelliklerin, kullanılan transistör sayısı ve gereken güce oranla en az doğrusal performans kazancı sağlaması gerektiğini yoksa uygulanmamaları gerektiğini söyleyen "KILL" (İng. Kill if Less Than Linear) kuralıdır (Agarwal, Levy 2007). Bu prensip günümüzde bazı çok çekirdekli işlemcilerin tasarımına öncülük etmektedir. (Vajda, 2011).

Donanımlara getirilen bu çoklu-çekirdek yaklaşımı beraberinde yazılımların tekrar değerlendirilmesi ihtiyacını getirdi. Çünkü bu yaklaşımda performans artışı sağlayabilmek için yazılımların bu minariye uygun tasarlanmaları gerekiyordu.
Ortaya çıkan birden fazla işlem birimini ortaklaşa ve verimli bir şekilde kullanma ihtiyacı önceleri oldukça dar bir alan olarak görülen paralel hesaplamayı bir anda önemli kıldı.

Ancak hakim nesneye yönelik programlama paradigması ile paralel programlama yapmak günümüzde hala oldukça zor olabilmektedir. Aynı problemlere yapısı gereği daha basit çözümler üretebilen fonksiyonel paradigma gittikçe daha çok önem kazanmaktadır.

Bu çalışmada paralel programlama modelleri fonksiyonel paradigma çerçevesinde incelenmeye çalışılmıştır. Öncelikle paralel hesaplama temellerine değinilmiş, ardından paralel programlama modelleri tanıtıldıktan sonra bu modellerin bazı fonksiyonel programlama dillerinde nasıl uygulandıkları gösterilmiştir. En son olarak sonuçlar aktarılmaya çalışılmıştır.

2. Paralel Bilgisayar Mimarileri

Bir yazılımın nasıl çalıştığının anlaşılabilmesi için üzerinde çalıştığı donanım mimarisinin bilinmesi oldukça faydalıdır. Bu bölümde önce klasik seri bilgisayar mimarilerinden başlanarak, paralel bilgisayar mimarileri tanıtılmaya çalışılacaktır.

2.1. Von Neumann Mimarisi

Klasik Von Neumann mimarisi bir işlemci, ana bellek ve bu unsurları birleştiren bir veriyolundan oluşur. Ana bellek içinde hem veri hem komut bulundurabilen bir konumlar bütünüdür. Her konum bir adres ile temsil edilir.

İşlemci kontrol ünitesi ve aritmetik ve mantık ünitesinden (ALU) oluşur. Kontrol ünitesi hangi komutların işletileceğine karar verirken ALU komutların işletilmesinden sorumludur. İşletilecek veriler veya komutlar önce bellekten veriyolu aracılığıyla işlemciye aktarılır. Bu durumda veriyolunun hızı okunabilecek en fazla veri veya komut sayısını belirler. Buna Von Neumann darboğazı adı da verilir (Backus, 1977). Modern bilgisayarlarda bu darboğazın etkisini azaltmak ve işlemcilerin daha hızlı çalışmalarını sağlamak için önbellekleme, sanal bellek, pipelining gibi belli yöntemler kullanılır. (Pacheco, 2011)

2.2. Flynn Taksonomisi

Paralel bir bilgisayar, iletişim kuran ve bir problemi hızlı bir şekilde çözmek için birlikte çalışan işlem birimlerinin bütünü olarak algılanabilir. Ancak bu tanım oldukça geniştir ve paralel platformların çoğunu içine alır. Paralel hesaplama tarihinde önerilen ve uygulanan oldukça fazla mimari vardır. Ayrıca bu tanım bir paralel bilgisayarın yapısına dair önemli detaylara da yer vermemektedir. Paralel mimarilerin önemli özelliklerine göre daha detaylı incelenebilmeleri için bir sınıflandırma yapmak faydalı olacaktır. Bu sınıflandırmaya dair basit bir model için Flynn taksonomisi (Flynn, 1972) incelenebilir. Bu sınıflandırma metodunda paralel bilgisayarlar bir anda işlenebilen komut sayısı ve veri akışı türlerine göre 4 farklı

mimari olarak sınıflandırılmıştır.

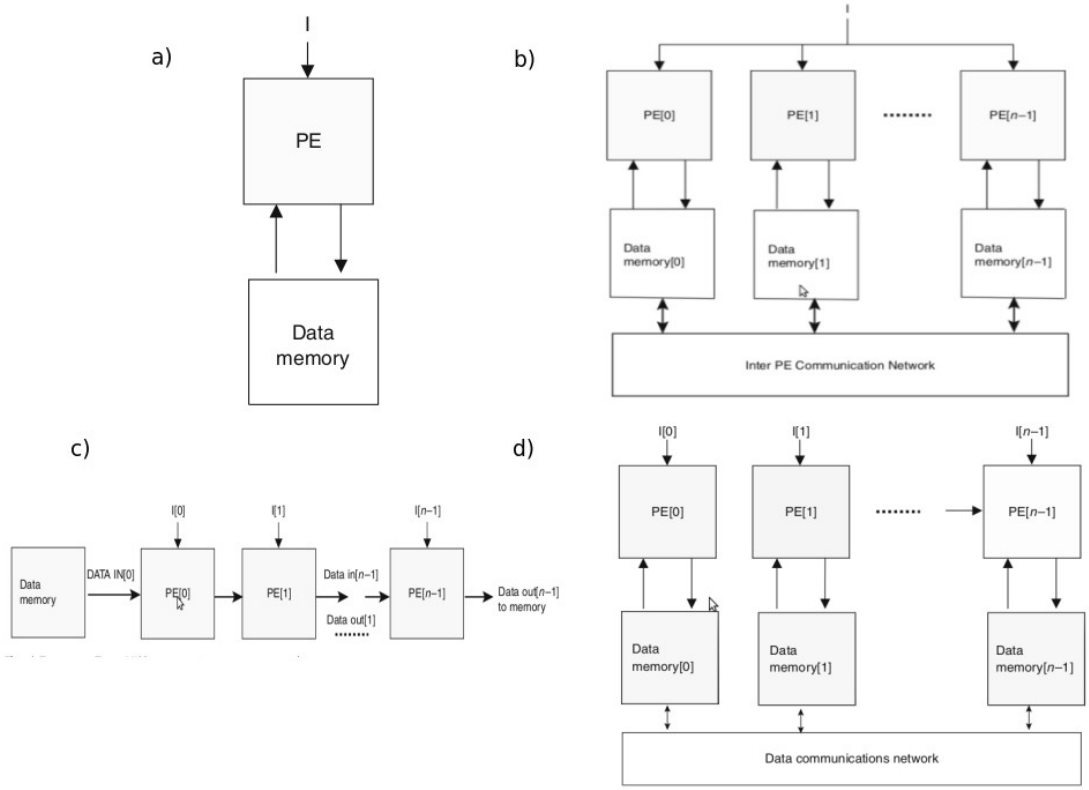
Tekil komut, tekil veri akışı (SISD) : Bu mimaride tekil bir programa ve veriye erişmeye çalışan bir tane işlem birimi bulunur. Her adımda veri ve komut okunur işletilir ve tekrar belleğe yazılır. SISD sistemlere örnek Von Neumann mimarisini kullanan klasik seri bilgisayarlardır.

Tekil komut, çoğul veri akışı (SIMD) : Bu mimaride her biri kendi özel veri belleğine (bu bellek dağıtık veya paylaşımlı olabilir) sahip birden fazla işlem birimi bulunur. Ancak ortada tek bir program belleği vardır. Özel bir işlemci bu komutu okur ve diğer birimlere yönlendirir. Her adımda işlem birimleri bu işlemciden işletilecek komutu alır ve bu komutu kendine ayrılan veri üzerinde işletir. Böylece bir komut farklı veri akışlarında eş zamanlı olarak işletir. SIMD sistemlere örnek dizi işlemcilerdir. Bu işlemci mimarisi 1990'larda modern mikroişlemciler çıktıktan sonra büyük oranda terk edilmiştir. Ancak bu mimari multimedya ve bilgisayar grafikleri uygulamalarında çok verimli olabildiği için Sony Playstation 3 için geliştirilen Cell işlemcisinde bir adet PowerPC işlemcisiyle beraber sekiz adet SIMD işlemci kullanılmıştır (IBM Research, *tarih bilinmiyor*) .

Çoğul komut, tekil veri akışı (MISD) : Bu mimaride her biri kendi program belleğine sahip birden fazla işlem birimi vardır. Ancak tek bir veri hafızası vardır. Program aşamaları programcı tarafından belirlenir. Her işlem birimi kendine gelen veri üzerinde kendi komutunu işletir ve sonucu işlem sırasında bir sonra gelen işlemciye iletir. Bu mimariye uygun bir paralel bilgisayar yapılmamış olsa dahi modern grafik işlemcileri bu sınıfa girmektedir. Grafik işlemciler çok sayıda MISD işlemci içermelerinin yanında çoğu zaman MIMD mimariyi de kullanırlar. Örneğin Nvidia Geforce GTX 660 grafik işlemcisi 960 tane işlem birimi içermektedir. Günümüzde Nvidia CUDA veya OpenCL gibi GPGPU teknolojileriyle grafik işlemciler üzerinde paralel programlama yapmak mümkündür ve oldukça revaştadır.

Çoğul komut, çoğul veri akışı (MIMD) : Bu mimari birden fazla herhangi bir türden işlemci ve bu işlemciler arası bağlantıdan oluşur. İşlemciler birbirlerinden

bağımsız olsalarda bir problemi çözmek yardımlaşabilirler. Bu yardımlaşmanın sağlanabilmesi ve işlemciler arası bilgi ve veri paylaşımı için bir tür senkronizasyon mekanizmasına ihtiyaç duyulur. MIMD mimarisinde kullanılan işlemcilerin birbirinin aynı olması gibi bir gereksinim olmasa da bu sistemler genelde homojen bir şekilde tasarlanırlar. Modern çok çekirdekli işlemciler ve küme sistemleri bu mimari sınıfına girer.



Şekil 2.1: Flynn taksonomisinde paralel mimariler

Şekil 2.1'de (Encyclopedia, 2011, syf 690-692) yukarıda bahsedilen mimarilerin görsel temsilleri yer almaktadır. Bu grafikte a) SISD b) SIMD c) MISD d) MIMD mimarilerini temsil etmektedir. (Rauber,Rünger, 2010)

2.3. Paralel Sistemlerde Bellek Organizasyonu

Paralel sistemlerin hemen hemen tamamı MIMD modelini temel almaktadırlar. MIMD'ler üzerinden yapılacak daha detaylı bir sınıflandırma bu sistemlerin bellek

organizasyonu incelenerek yapılabilir. MIMD bellek mimarileri üçe ayrılabilir. Dağıtık bellek, paylaşımlı bellek ve sanal paylaşımlı bellek. Dağıtık bellek organizasyonunda tekil makinalara düğüm adı verilir. Her düğümün kendi yerel kaynakları bulunur ve bunlara sadece o düğümün kendi işlemcisi erişibilir. Düğümler veri veya bilgi paylaşma işini birbirlerine mesaj göndererek yaparlar. Dağıtık bellekli bir paralel sistem kurmak herhangi bir bilgisayar düğüm olarak kullanılabileceği için kolaydır. Paylaşımlı bellek organizasyonunda modern çok çekirdekli işlemcilerde olduğu gibi işlemciler ortak bir bellek alanını paylaşırlar. Buna global bellek denir. İşlemciler arası veri transferi paylaşılan değişkenler aracılığıyla olur ancak bir değişkene aynı anda birden fazla işlemcinin erişmesi engellenmelidir. Çünkü yarış durumları ve beklenmeyen sonuçlar ortaya çıkabilir.

3. Paralelliğin Temel Kanunları

Paralel hesaplama teorisinin 40 yılı aşkın bir geçmişi vardır. O günden beri paralelliğin temel kavramları, kanunları ve temel algoritmaları bu çabanın bir sonucu olarak belirlenmiştir. Bu bölüm paralelliğin gümünze kadar araştırmaları ve pratik uygulamaları etkileyen kanunları tanıtılmaya çalışılacaktır. Bu kanunlar modern bilgisayar mimarilerine yol göstermekte olup bilimsel araştırma için temel bir çatı sunmaktadır. (Vajda, 2011)

3.1. Amdahl Yasası

Amdahl Yasası (Amdahl, 1967) verilen bir uygulama için teorik olarak en fazla ne kadar paralel performas kazancı elde edilebileceğini tanımlayan ve büyük ihtimalle en çok bilinen ve başvurulan kanunlardan bir tanesidir. Bu yasa 1967 yılında Gene Amdahl tarafından paralel hesaplamanın tartışıldığı bir konferansta paralel hesaplamaya karşı argüman olarak sunulmuştur (Encyclopedia, 2011, syf 53-54). Bu gün paralel hesaplamayı açıklamak için kullanılsa da argüman ortaya koyulduktan 40

sene sonrasına kadar seri performans artışı devam ettirebildiği için zamanında haklı olduğu söylenebilir.

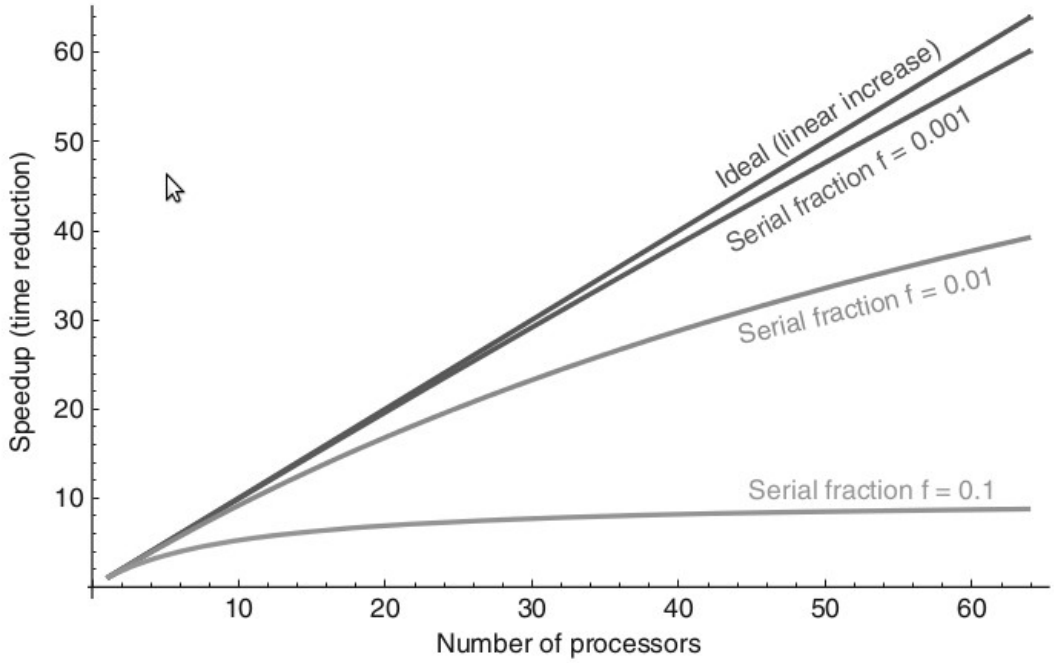
Eğer f kadarı seri olan bir problem n tane işlemcide çalışıyorsa maksimum performans kazancı pk

$$pk = 1 / (f + (1-f) / n)$$

olacaktır. Hatta bu sistemde sonsuz adet işlemci olduğunu düşünürsek

$$pk = 1 / f$$

olacaktır. Yasaya göre performans artışının nasıl değiştiği Şekil 3.1 'de görülebilir.



Şekil 3.1: Amdahl Yasasına göre performans artışı

Ancak asimetrik paralel mimariler kullanıldığında Amdahl yasasının öngördüğünden daha iyi bir performans artışı sağlanabilmektedir. Örneğin 64 tane aynı işlemciden kullanmak yerine 60 tane aynı işlemci kullanmak ve bir tane bu işlemcilerden iki katı hızlı bir işlemci kullanıp, hızlı işlemciye programın seri kısmını işletmek daha performanslı olacaktır (Vajda, 2011).

Amdahl yasası genel olarak yanlış olmamakla beraber Amdahl'ın paralel

programlamanın anlamsız olması sonucunu çıkarırken yaptığı iki hatalı varsayım vardır.

Birincisi o zamanda paralel bilgisayarlar mevcut olmadığı için programcılar programları paralelleştirmek için çaba harcamamışlardır. Bu tip bilgisayarlar ortaya çıkınca programların paralel kısımlarını arttıracak yöntemler geliştirmeye başlamışlardır.

İkincisi ise problemin büyüklüğü değiştikçe paralel ve seri kısımların toplam işletme hızları eşit hızlı büyümemektedir. Bu nedenle problemin boyutu büyüdükçe f azalacak bu sayede performans yükselecektir. (Dowd,Severance, 1998)

3.2. Gustafson Yasası

Amdahl yasası problemlerin boyutunu sabit kabul ederek yanlış bir varsayımda bulunmuştur. Amdahl asıl olarak f'i 0.25 ila 0.4 olarak tahmin etmiştir (Encyclopedia, 2011, syf. 819-824) günümüz f bu değerin çok çok altındadır. Maksimum performans kazancı genelde problemin büyüklüğü ile orantılıdır. Gustafson yasasının (Gustafson, 1988) temel gözlemlerinden biri güçlü bilgisayarların aynı sorunları daha hızlı çözmedikleri daha büyük sorunları çözmeye çalıştıkları için daha büyük problemlerin daha fazla işlemci sayısı ile benzer zamanlarda çözülebileceğidir. Maksimum performans kazancı aşağıdaki formül ile hesaplanabilir.

$$pk = n – (n-1)\, f$$

Formüle göre pk'nın üst sınırı problemin seri kısmı değil, büyüklüğüdür. Bu durum ölçeklenmiş hızlanma olarak da adlandırılır.

Amdahl ve Gustafson yasaları birbirleriyle çelişiyor gibi gözükseler de aslında farklı durumları anlatmaktadırlar ve f tanımlamaları arasında farklılıklar vardır. Ortak tanımlamalar yaparak bu iki yasayı birleştirmeye çalışan çalışmalar mevcutsa da geniş kabul görmemişlerdir ve bu iki yasanın birleştirilip birleştiremeyeceği hala tartışma konusudur.

3.3. Gunther Yasası

Gunther yasası (Gunther, 2002) evrensel ölçeklenebilirlik yasası olarak da bilinir. Bu formül ölçeklenebilirliği tutarlılık kayıpları gibi faktörleri de hesaba katarak incelemeye çalışmaktadır. Formül şu şekildedir.

$$C(N , s, k) = N /(1 + s * (N - 1) + k * N * (N - 1))$$

Burada n işlemci sayısını, s programın paralel kısmını, k ise tutarlılık gecikmesini (senkronizasyon işlemleriyle geçen süre vs.) ifade etmektedir. Eğer s ve k = 0 ise ideal doğrusal artış gözlenmektedir eğer k = 0 ise bu yasa Amdahl yasasına eşit olmaktadır.

Bu formülün en büyük eksilerinden biri k ile neyin ifade edildiğinin tam olarak açık olmamasıdır.

3.4. Karp-Flatt Ölçütü

Karp-Flatt ölçütü (Karp,Flatt, 1990) bir uygulamanın kullanılan işlemci sayısına göre performansını ölçmenin pratik yollarından biridir. Bir uygulamanın performans artışına bakarak seri kısımının kestirilmesinde kullanılır. Formül şu şekildedir.

$$f = (1/pk - 1/N)/(1 - 1/N)$$

Bu ölçüt kısıtlı ölçeklenebilirlik üzerine mantık yürütmek için elimizdeki güçlü araçlardan biridir. Her ne kadar yasa olmasada Karp-Flatt ölçütü paralelliğe dair temel ölçümlerden biridir ve bu alanda uygulanabilirliği de oldukça geniştir (Vajda, 2011).

4. Paralel Programlama Modelleri

Paralel programlamanın genel prensipleri üzerine çalışabilmek için paralel mimariler genelde bazı özellikleri değerlendirilerek daha soyut şekillerde düşünülürler. Bunu yapmanın sistematik yollarında biriyse tekil sistemlerden biraz uzaklaşıp daha soyut bir bakış getiren modelleri göz önüne almaktır.

Paralel işleme için belirtilen içerdikleri soyutlama seviyelerine göre ayrılmış 4 çeşit model vardır (Haywood,Ranka, 1992). Bunlar makine modeli, mimari model, hesaplama modeli ve programlama modelidir.

Programlama modeli bir paralel hesaplama sistemini programlama dilleri ve kavramlarıyla açıklamaya çalışır. Bir paralel programlama modeli bir programcının bir paralel bilgisayara bakışını belirler. Bu bakış mimari tasarımdan, programlama dilinden, derleyiciden, dilin kütüphanelerinden vs. etkilenebilir.

Paralel programlama modellerinin birbirinden farklılaşacağı bir kaç farklı kriter vardır.

- Paralleliğin hangi seviyede uygulandığı
- Paralellik tanımlamasının açık mı yoksa ortülü mü olduğu
- Paralel program parçalarının nasıl tanımlandığı
- İşlem birimlerinin nasıl haberleştiği
- Paralel birimler arasında senkronizasyonun nasıl sağlandığı

Paralelliği destekleyen her programlama dili yukarıdaki kriterleri bir şekilde gerçekler ve düşünüldüğünde ortaya çok farklı kombinasyonlar çıkabilir. (Rauber,Rünger, 2010)

4.1. Paralellik seviyeleri

4.1.1. Komut seviyesinde paralellik

Bir programa ait birden fazla komut normalde aynı anda paralel bir şekilde işletilebilir. Ancak komutlar arasında bir veri bağımlılığının varlığı paralelliğe engel olacaktır. Veri bağımlılığı aşağıdaki şekillerde gerçekleşebilir.

- **Akış bağımlılığı:** Eğer i1 komutu daha sonra i2 komutunda ihtiyaç duyulacak bir değer hesaplıyorsa i1 ve i2 arasında akış bağımlılığı var denir. (solda)
- **Karşı bağımlılık**: Eğer i1 daha sonra i2'nin hesaplama sonucunu yazacağı bir değişkeni kullanıyorsa i1 ve i2 arasında karşı bağımlılık var denir. (ortada)
- **Çıkış bağımlılığı:** Eğer i1 ve i2 aynı değişkeni hesaplama sonuçlarını tutmak için kullanıyorlarsa i1 ve i2 arasında çıkış bağımlılığı var denir. (sağda)

i1 : r1 = r2+r3 i1 : r1 = r2+r3 i1 : r1 = r2+r3

i2 : r5 = r1+r4 i2 : r2 = r4+r5 i2 : r1 = r4+r5

4.1.2. Veri Paralelliği

Programlarda aynı bir işlem büyük bir veri yapısının farklı elemanlarına uygulanması oldukça yaygın bir işlemdir. Eğer bu işlemler birbirlerinden bağımsızsa veri yapısı farklı işlemcilere dağıtılarak paralel bir şekilde işletilebilir. En temel örneklerinden biri dizi işlemleridir. Tek bir akış kontrolünün olup verinin dağıtıldığı model SIMD olarak da bilinmektedir. Veri paralelliği dağıtık ve paylaşımlı bellek organizasyonu kullanan sistemlerde gerçeklenebilir. Veri paralelliğinin pek göze çarpmayan kullanımlarından biri olay güdümlü sistemlerdir. Örnek olarak haberleşme sistemlerini verebiliriz.

4.1.3. Görev (Fonksiyon) Paralelliği

Seri programların pek çoğunda birbirinden bağımsız işletilebilecek kısımlar bulunur. Basit ifadeler,döngüler veya fonksiyonlardan oluşabilecek olan bu kısımlar paralel işletilebilir. Veya matematiksel bir ifadeyle eğer işletilecek bir h(x) fonksiyonu varsa ve bu fonksiyonu f(x), g(x) gibi farklı fonksiyonların toplamı şeklinde yazabilirsek bu fonksiyonlar paralel olarak işletilebilir. Bu tür paralelliğe görev veya fonksiyon paralelliği adı verilir. Görev paralelliğini kullanabilmek için görevlerin ve aralarındaki bağımlılıkların bilinmesi gerekir. Bunlar bir graf (çizge) ile gösterilebilir. Daha sonra bu graftan belirli bir işlemci grubu için programın nasıl işletileceğinin planı belirlenmeye çalışılır (Rauber,Rünger, 2010). Bu statik veya dinamik olarak belirlenebilir. Statik zamanlama kullanıldığında görevler derleme veya çalıştırma zamanında işlemcilere atanırken, dinamik zamanlama kullanıldığında görevler işlemcilere çalıştırma süresince atanır. Statik zamanlamanın uygulama alanı geniş değildir ve pek yaygınlaşamamıştır. Dinamik zamanlama ise daha yüksek potensiyele sahip olduğu düşünüldüğünden oldukça ilgi görmektedir. Clik, Intel Thread Building Blocks, OpenMP, Pthreads gibi yaygın çözümler bu tekniği kullanmaktadır (Vajda, 2011).

4.2. Paralelliğin Temsili

Paralel programlama modelleri, görevlerin nasıl belirlendiği,iletişim ve senkronizasyonun temsil edilme şekilleriyle de birbirinden ayrılabilir. Paralellik örtülü veya açık bir şekilde temsil edilebilir. Örtülü paralellik için basit programlar yazılması yeterlidir ancak derleyicilerin oldukça karmaşık olması gerekir. Açık paralellik için daha basit derleyiciler yeterliyken, program yazmak daha büyük çaba gerektirir.

- **Örtülü Parallellik:** Bu modelde rogramcının paralel işletime dair herhangi bir şeyi düşünmesi gerekmediği için programcı açısından en basit modeldir. Program sadece yapılacak hesaplamaları içerir. Böylece programcı sadece seri algoritmanın nasıl gerçekleneceğine yoğunlaşabilir, paralel organizasyonun nasıl yapılacağı ile ilgilenmesi gerekmez. Bu konudaki yaklaşımlardan biri otomatik paralelleştirmedir. Otomatik paralelleştirme derleyicinin seri koddan paralel kod üretmesi anlamına gelir. Paralelliğin kutsal kasesi olarak da bilinir. Bunun yapılabilmesi için program kısımlarındaki bağımlılıkların analiz edilebilmesi gerekir ki sonradan paralel çalıştırma için işlemcilere atanabilsin. Ancak hem pointer tabanlı işlemlere ve dolaylı adreslemeye bağımlılık analizi yapmak hem defonksiyonların ve belirsiz sınırlı döngülerin işletim zamanını tahmin etmek zor olduğu için bu yöntem pek başarıya ulaşamamıştır. Yaygın kullanılan programlama dilleri ya kısmi örtülü veya açık paralellik kullanmaktadır. Ancak pek az da olsa tamamen örtülü kısmi diller mevcuttur (SISAL, Paralel Haskell, Verilog, VHDL)
- **Kısmi Örtülü Paralellik:** Bu modelde programın paralel kısmının açık bir şekilde belirtilmesine karşın programın proses ve threadlere dağıtımı ve atanması derleyici tarafından yapılır. Bu nedenle paralel kısmın iletişim ve senkronizasyon tarafı da gene derleyici tarafından yapılır. Bu yaklaşımın derleyici için avantajı paralel kısımlar belirlendiği için oldukça karmaşık olan bağımlılık analizlerinin yapılmasına gerek olmamasıdır. Bu modeli kullanan paralel diller genelde bazı seri dillerin belirli yapılarla (Ör. Paralel döngüler)

genişletilmiş halleridir. OpenMP kütüphanesi bu modeli kullanır.

- **Açık Paralellik:** Bu modelde programcı iletişim ve senkronizasyon işlemleri dahil olmak üzere paralel işletimin bütün detaylarını açık bir şekilde belirtmelidir. Bu yaklaşımın avantajı ise standart bir derleyici kullanmayı mümkün kılması ve programcıya paralel işletim üzerinde tam kontrol vermesidir. Bu tip programların çok verimli olabilmelerine karşın yazılmaları oldukça fazla çaba gerektirmektedir. Bu model MPI, Pthreads gibi araçlarca kullanılmaktadır.

4.3. Paralel Birimlerin Tanımlanması

Paralel programlama modellerinde kullanılan temel prensiplerden biri uygulamanın belirli görevlere ayrıştırılmasıdır. Bu belli görevler farklı kontrol akışlarına atabilir ve farklı çekirdek veya işlemcilerde işletilebilirler. Ancak bu ayrıştırmanın eşlenmesinden bahsederken işletim sisteminin uygulamaları nasıl gördüğü ve uygulamaların kendi alanlarında davrandığını anlamak önemlidir

Uygulama seviyesinde paralellik uygulanmak istediğinde işletim sisteminde iki temel kavramdan bahsedilebilir, process ve threadler. Bu kavramlar kontol akışlarına dair soyutlamalardır. Temelde benzeseler de birbirlerinden ayrıldıkları hiyerarşi ve izolasyon gibi iki karekteristik bulunur.

4.3.1. Processler

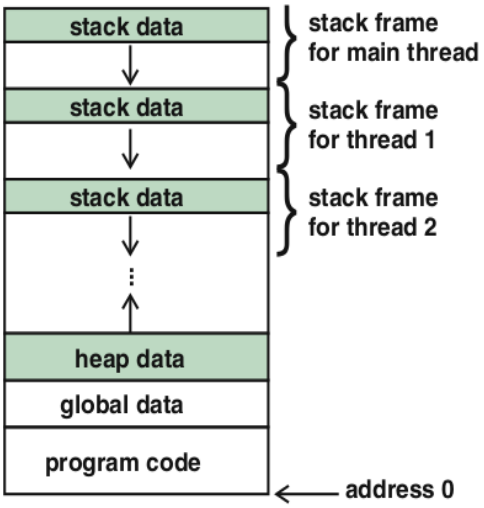
Processler en basit haliyle bir programın çalışması ile tanımlanır. Bir process programın çalıştırılabilir hali ve programın çalışması için gereken bütün veriyi içerir buna çalışma anında stackte bulunan program verisi, registerların mevcut değeri ve program sayacının içeriği de dahildir. Bütün bu veriler programın çalıştırılması esnasında dinamik olarak değişir. Her process sadece kendi adres alanına erişebilir. Bu açıdan her process diğerlerinden kesin çizgilerle ayrılır, aynı önceliğe sahip her process işletim sisteminden eşit varlıklar olarak muamele görür. Eğer iki process kendi aralarında veri paylaşımı yapmak isterse bunu birbirleriyle ayrıca haberleşerek yapmak zorundadırlar. Processler işletim sistemlerinde güvenli ve öngörülebilir bir şekilde yöneyilebilen en küçük izole edilmiş birimlerdir. Processler özellikle bellek

ve hata yönetimi izolasyon birimleri olarak görülebilirler.

Processler iş yapabilmek için işletim kaynaklarına (işlemci,çekirdek vs.) atanmalıdırlar. Eğer işlem birimi sayısından fazla process varsa bütün processleri işleyebilmek adına zaman içerisinde farklı processler işletilir. Bir processten bir başka processe geçmek için mevcut process durdurulmadan önce daha sonra işletime devam edebilmesi amacıyla kaydedilir. Ardından diğer process işletilmeye başlanır. Bu işleme context switch adı verilir ve bazı performans kayıplarına sebep olur. Eğer ortada bir işlem birimi varsa processler zaman paylaşımlı olarak eşzamanlı bir şekilde yürütülürler. Bu durum da paralellikten bahsedilemez. Farklı processler farklı işletim kaynaklarına atanabiliyorsa paralel işletim mümkün olabilir.

4.3.2. Threadler

Processler içinde bulunan birbirinden bağımsız kontrol akışlarına thread denir. Thread sözcüğü genellikle işletilen uzun ve devamlı komut dizisini belirtmek için kullanılır. Modern işletim sistemlerinde her process en az bir thread içermektedir. Bir processin işletimi sırasında bu processe ait farklı threadler işletim kaynaklarına atanabilir. Threadler bağlı bulundukları processlerin adres alanlarını kullanırlar. Process adres alanının içinde bir threadin sadece kendisinin ulaşabildiği bir özel adres alanı bulunur. Bunun haricinde threadler processlerin ortak adres alanlarına da erişebilir.(Rauber,Rünger,2010). Bkz. Şekil 4.1



Şekil 4.1: Çoklu thread kullanan bir process adres alanı

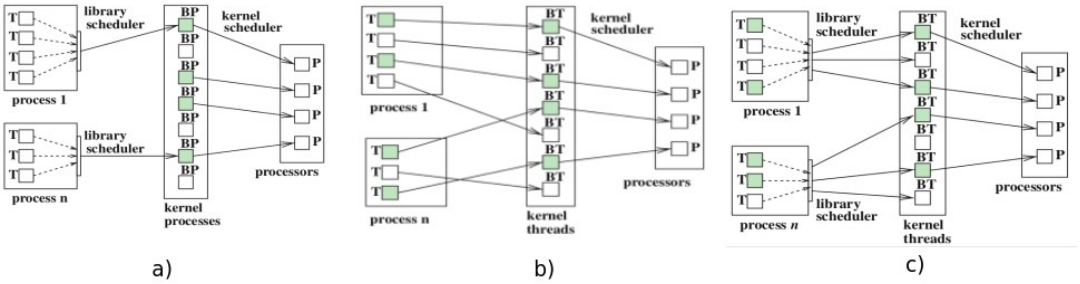
Thread kullanmak ortamda fiziksel bellek paylaşımlı sistem (ör. Çok çekirdekli işlemci) mevcutsa anlamlı olacaktır. Process yaratırken olduğu gibi tüm adres alanının kopyalanması gerekmediği için thread oluşturmak çok daha kolaydır. Threadler processlere göre çok daha esnek bir paralel işletim imkanı sağlarlar. Threadler programlama ortamı veya işletim sistemi tarafından sağlanabilir. Bunlardan birine kullanıcı threadleri denirken diğerine çekirdek threadleri adı verilir.

Kullanıcı threadleri programlama diline ait thread kütüphanesi tarafından sağlanır. Kullanıcı threadleri için işletim sistemi desteğine ihtiyaç yoktur. İşletim sistemi ile iletişime geçilmediği için threadler arasında çok hızlı bir şekilde geçiş yapılabilir. Ancak işletim sistemi threadlerin varlığından haberdar olmadığı için bir processe ait farklı threadler farklı işletim kaynaklarına atanamazlar. Dahası bir thread bloklanırsa bu bütün processin bloklanmasına sebep olacaktır. Bu dezavantajları ortadan kaldırmak için çekirdek threadleri kullanılabilir. İşletim sistemi threadlerin varlığından haberdar olduğu için buna uygun davranışlar sergileyebilir. Çok çekirdekli bir sistemin verimli kullanımı için çekirdek threadlerinin kullanımı önemlidir. Modern işletim sistemlerinin pek çoğu bu desteği sağlamaktadır. Threadlerin kullanımı için belli işletim modelleri bulunmaktadır. Bunlar çoktan bire (N:1), bire bir (1:1) ve çoktan çoğa (N:M) olarak üçe ayrılır

- **N:1 eşleme:** Thread zamanlamasından sadece thread kütüphanesinin sorumlu olduğu modeldir. Kütüphane hangi anda hangi threadin işletileceğine karar verir. Processlerin kaynaklara atanması işletim sistemi tarafından yapılır. Aynı processe ait threadler farklı işletim kaynaklarına atanamazlar. Bu model Şekil 4.2a (Rauber,Rünger,2010) 'da gösterilmiştir

- **1:1 eşleme:** Her kullanıcı threadine bir çekirdek threadi yaratılır. Çekirdek threadlerinin nasıl işletileceğine işletim sistemi karar verir. Bu modelde zamanlayıcı bir kütüphaneye ihtiyaç duyulmaz. Aynı processe ait threadler farklı işletim kaynaklarına atanabilirler. Bu model Şekil 4.2b 'de gösterilmiştir

- **N:M işleme:** Bu modelde iki seviyeli bir zamanlama mevcuttur. Thread kütüphanesi kullanıcı threadleri verilen çekirdek threadleri grubuna atar. Çekirdek threadlerinin nasıl işletileceğine işletim sistemi karar verir. Herhangi bir anda bir çekirdek threadi farklı kullanıcı threadleri işletiyor olabilir. Thread kütüphanesine bağlı olarak programcı kütüphane zamanlayıcısını etkileyebilir. Bu model 1:1 modelinde çok daha fazla esneklik sağlamaktadır. Bu model Şekil 4.2c 'de gösterilmiştir



Şekil 4.2: Thread eşleme modelleri

4.4. Veri Alışverişinin Sağlanması

Paralel bir programın değişiklik parçalarının eşgüdümlü bir şekil işletilmesini kontrol etmek için işlem birimlerinin veri alış verişi yapması gerekmektedir. Bu veri alış verişinin nasıl olacağının belirlenmesi ve gerçeklenmesi büyük oranda hangi paralel platfomun kullanıldığı ile belirlenir.

4.4.1. Bellek Paylaşımlı Model

Bellek paylaşımlı paradigma, threadler arası iletişim için en çok kullanılan ve anlaması en kolay olan yöntemdir. En çekici yanlarından biri ise programlama üzerine tekrar düşünülmesinden ziyade alışılageldik programlama tarzının bir uzantısı olarak görünmesidir. Konseptin kendisi oldukça basittir, iletişime geçmek isteyen threadler bellek içinde her yerden erişilebilecek bir konuma iletmek istediği verileri yazar.

Paylaşılan değişkenler kullanılırken aynı değişkene aynı anda birden fazla threadin okuması veya yazması önlenmelidir. Aksi takdirde yarış durumları ortaya çıkabilir. Yarış durumlarında hangi threadin değişkene yazacağını zamanlama algoritması belirlediği için paylaşılan değişkenin hangi değeri alacağı önceden bilinemez. Buna determenistik olmayan davranış denir. İşletim sırasına göre farklı sonuçlar elde edilebilir ve kesin sonuç tahmin edilemez.

Program içerisinde paylaşılan bir değişkenle işlem yapılan yani tutarsız değer içerme tehlikesi olan parçalara kritik kısım adı verilir. Hataların önlenmesi için bir anda sadece bir tane thread kritik kısmını işletmelidir. Buna karşılıklı dışarlama adı verilir.

Ancak bellek paylaşımlı modelin iki temel problemi vardır. Bellek içeriğinin tutarlılığının sağlanması için belleğe erişim kontrol edilmeli ve bütün threadlerin senkronize olmaları sağlanmalıdır. Birinci sorunun çözümü tamamen uygulama geliştirinin elindedir. Geleneksel olarak kilit ve semaforlar (senkronizasyon nesneleri kullanılır. Sadece senkronizasyon nesnesini elde edebilmiş olan thread paylaşılan belleğe erişebilir) kullanılsa da yakın zaman da bu problemin çözümü için işlemsel bellek (veritabanı işlemlerine benzemektedir. Ara değerlerden korunmak için atomik işlemleri temel alır. Yapılacak işlem başarısız olursa o ana kadar yaptığı değişiklikleri siler ve daha sonra tekrarlanmayı bekler) adında bir yöntem önerilmiştir

4.4.2. Mesaj İletimli Model

Mesaj iletimli paradigma bellek paylaşımlı modelin tam tersi olarak görülür. Temel yaklaşımı "hiçbir şey paylaşma"dır. Mesaj iletimi kullanan bir sistemde threadler herhangi bir veri paylaşmazlar, bütün iletişim threadlerin birbirleri arasında iletilen mesajlarla sağlanır. Mesaj iletimi en temel thread izolasyonu çözümüdür. Bellek paylaşımlı sistemlerde eğer bir thread çökerse paylaşılan belleğin bozulması olasılığı olduğu için diğer threadler de çökebilir. Mesaj iletimli sistemlerde threadler mesajları analiz ederek ve hatalı mesajları çöpe atarak kendilerini koruyabilirler. Mesaj iletimli model gerçek zamanlı sistemlerde (Ör. telekominasyon) yaygın olarak kullanılmaktadırlar.

Mesaj iletimleri sistemler farklı şekillerde gerçeklenebilirler örneğin işletim sistemi, ara yazılım ve programlama dilleri düzeyinde. Bu metodlar birden fazla şekilde sınıflandırılabilirler, iletimin güvenilirliği, sıralı iletim garantisi, iletişim ortamının mimarisi, iletimin senkron mu asenkron mu yapılacağı gibi. Pek çok sistem güvenilirlik, sıralı iletim gibi koşulları yerine getirirken en az bire bir iletime imkan sağlar. Asenkron haberleşme ise daha yaygın olarak kullanılmaktadır. (Vajda, 2011)

4.5. Senkronizasyon Mekanizmaları

4.5.1. Kilitler

Kilitler, pek çok senkronizasyon primitiflerinin temeliyle beraber, büyük ihtimalle en

çok kullanılan ve en çok tartışılan senkronizasyon mekanizmasıdır. Bir kilit temelde aynı anda pek az (genelde sadece bir) threadin sahip olabileceği bir kaynaktır. Bir kilidi belli metodları olan bir nesneymiş gibi görmek de yanlış olmayacaktır. Kilitler de üç temel metod bulunur.

- **Acquire:** Bu çağrıyı yapan thread kilit boşsa kilidi kendi üzerine alır, değilse bloklanır.
- **Release:** Bu çağrıyı yapan thread kilidin sahipliğinden vazgeçer ki başka threadler de işlem yapılabilsin.
- **Test:** Çağrıyı yapan threadin bloklamadan kilidin üzerine alıp alamayacağını öğrenmesini sağlar.

Kilitleri kullanmanın olağan yolu onları birbirleriyle ilişkili kaynakları korumak için kullanmaktır. Böylece bu kaynaklara ulaşmak isteyen her threadin önce kilidin sahibi olması gerekir. Bazen kilitlere kapsamının eldeki duruma uyarlanması gerekebilir. Örneğin bir kaynağı her threadin okuması sorun olmayacakken bu kaynağa yapılan yazma işlemlerinin korunması gerekebilir.

Bazı özel kilit şekilleri de mevcuttur. Spinlocklar kullanıldığında acquire çağrısında eğer kilit elde edilemiyorsa thread bloklanmaz sadece bir hata belirtisi döndürülür. Daha sonra thread kilidi tekrar elde etmeye çalışabilir. Bu avantajının yanı sıra eğer bir thread sürekli kilidi elde etmeye çalışırsa performans sorunları yaşanacaktır. Okuma/Yazma kilitlerinde okuma kilidi pek çok thread tarafında elde edilebilirken yazma kilidi sadece tek bir thread tarafından elde edilebilir. Okuma kilidi elde edilmişken yazma kilidi elde edilemez veya tam tersi de geçerlidir.

Kilitlerin kullanımı yarış durumlarını ve determenistik olmayan davranışları önleyebilseler de dikkatli bir şekilde kullanılmadığında deadlock,livelock gibi sorunlara yol açabilmektedirler. Birden fazla threadin birbirleri tarafından gerçekleştirilecek eylemleri beklemesi gibi durumlar deadlock oluşturur.

Örneğin aşağıdaki gibi bir durumda

thread1	thread2
acquire(s1)	acquire(s2)
acquire(s2)	acquire(s1)
birsey_yap()	baska_birsey()
...	...

thread1 s1 ve thread2 s2 kilidine sahiptir. Ancak thread1 s2 kilidine sahip olamayacağı için ve thread2 s1 kilidine sahip olamayacağı için bloklanır. Threadlerin ikisi de bloklanmışken ellerindeki kilidi bırakamayacakları için bu iki thread sonsuza kadar beklemek zorunda kalır. Bu deadlock durumuna örnektir. Bu threadler bloklanmasalardı da birbirlerinin kilitlerine ihtiyaç duydukları için işlem yapamayacaklardı. Bu duruma ise livelock denir.

4.5.2. Semaforlar

Semaforlar, kilitlerin genelleştirmiş hali olarak görülebilir ilk defa Djisktra (Djikstra,1974) tarafından önerilmişlerdir. En basit ikili semaforlar kilitlerle aynı işlevi görürler. Threade izin verebilir veya bloklayabilirler. Sayaç semaforları ise N tane threade izin verebilirler. Sayaç semaforları basit bir tamsayı sayaçtan oluşur. Ve atomik operasyona sahiptirler, P(s) ve V(s) (wait(s) ve signal(s) olarak da gösterilirler.) P(s) sadece sıfırdan büyük değerler için threade izin verir ve işlem başarıyla yapıldığında s'i bir azaltır. V(s) ise s'i bir arttırır. Semaforlar şu sekilde kullanılabilirler.

wait(s)
kritik kısım
signal(s)

Semaforlar, kilitlere göre daha esnek ve genel olsalar da kilitlerle aynı dezavantajlara sahiptirler. Buna rağmen modern işletim sistemleri senkronizasyon için semafor kullanılır. Semaforlar ayrıca daha yüksek düzeyde yapılara temel oluştururlar.

4.5.3. Koşul Değişkenleri ve Monitörler

Koşul değişkenleri işletim sistemi mekanizmaları olarak Hoare ve Hansen tarafından ortaya atılmıştır (Hoare,1974)(Hansen,1975). Temelde threadlerin koşullar üzerinde bekletilebilmelerini sağlar. Bir koşul değişkeni bir koşulla ilişkilendirilir ve üzerinde wait ve signal işlemleri gerçekleştirebilir. Eğer verilen koşul sağlanmıyorsa işlemi yapmaya çalışan thread bloklanır. Bu thread daha sonra başka bir thread tarafından

mevcut koşulun sağlanması ile uyandırılabilir.

Monitörler (Hoare, 1974) ise birden fazla thread tarafından güvenle kullanabilecek nesneler yaratmaya yarayan programlama dili seviyesinde bir mekanizmadır. Monitör nesnelerinin metodları karşılıklı dışarlama ilkesine uyar, ayrıca bu yapı kullanırken koşullu erişim desteği de sağlanabilir. Bu özellikler ise semaforlar ve durum değişkenleriyle gerçeklenir.

Koşul değişkenleri ve monitörler pek çok programlama dilinden kullanılmaktadır. Ör . Ada, Java, .NET ailesi, Python, vs. (Vajda, 2011)

4.5.4. İşlemsel Bellek

Thread senkronizasyonunda kullanılan temel yaklaşımlar kritik kısımların belirlenmesi ve karşılıklı dışarlamadır. Senkronizasyon mekanizması olarak kilitlerin kullanılması kritik kısımların serileştirilmesini getirmektedir. Bu performans kayıplarını beraberinde getirmekte ve hatta büyük sistemlerde darboğazlara sebep olmaktadır. Kilit mekanizmasını kullanan programcı çok dikkatli olmalı ve programdaki paylaşılan bütün alanları korumalıdır ki programın doğru çalışabildiğinden emin olunabilsin. Eğer korunması gereken bir kısım atlanırsa programda hatalar oluşabilir. Program sonucu işletim sırasına bağlı olduğu için oluşabilecek hatalar her zaman tekrarlanmayabilir ve bu hata ayıklamayı oldukça zorlaştırmaktadır. Kilitlerle programlama yapmak oldukça hataya açık ve düşük seviyeli olmaktadır. Hatta bu teknikler yer yer assembly diliyle programlamaya benzetilmektedir. (Sutter,Larus,2005)

Kilit mekanizmalarına bir alternatif olarak işlemsel bellek (Herlihy,Moss,1993) önerilmiştir. İşlemsel bellek, belleğe eş zamanlı erişim sırasında kilitlerin yol açtığı bazı sorunları çözmeyi amaçlamaktadır. Donanım veya yazılım seviyesinde gerçeklenebilmektedir. Günümüzde yazılımsal işlemsel bellek kullanan çok sayıda kütüphane mevcutken donanımsal işlemsel belleği destekleyen ticari bir platform bulunmamaktadır. İşlemsel belleğin arkasındaki temel fikir şöyledir. Bir grup bellek erişim operasyonu bir işlem olarak muamele görür. Bu işlem ya tamamen başarılı olur, ya da işlem sırasında yapılan bütün değişiklikler geriye alınır. Bunu gerçekleştirmek için yapılan bütün bellek erişimleri önbelleklenir eğer işlem sürerken

bellekte bir değişiklik yapılırsa işlem geri alınır ve daha sonra tekrar işletilmeye çalışılır.

Uygulama seviyesinde kritik kısımlardaki kodun bir işlem olarak işaretlenmesi yeterlidir. İşlemsel belleğin güçlü yanlarından biri işlemlerin şeffaf bir şekilde gerçeklenmesidir. Başarısız olan işlemler kullanıcıya belli etmeden başarılı olana kadar denenecektir.

Kilit yapılarında kullanılan kötümser ve defansif yaklaşımın (çakışmanın mutlaka yaşanacağı varsayılır ve programın sadece doğruluğu garantilenmiş koşullarda ilerlemesi sağlanır) aksine işlemsel bellek daha iyimser bir yaklaşım (İşlemin gerçekleşeceğini varsayar aksi gerçekleştiğinde işlemi geri alır.) getirir.

Ancak işlemsel bellek kullanılırken geriye alınması zor, I/O gibi işlemlerde uygulanması zordur. Ayrıca işlemler çok fazla tekrarlanmaya başlarsa performans kayıplarına sebep olabilir.

İşlemsel bellek, kilitlere göre daha soyut ve daha esnek programlama imkan sağlasa bellek paylaşımlı modele içkin sorunları bünyesinde barındırmaktadır. İşlemsel bellek ümit vaadeden bir yaklaşım olmasına karşın hala bir aktif araştırma konusu olduğu ve mevcut uygulamaların henüz tam olgunlaşmadığı unutulmamalıdır.

5. Fonksiyonel Programlama

İmperatif diller doğrudan Von Neumann mimarisini temel alarak tasarlanmışlardır. Hatta bu diller kolektif olarak eldeki temel modelin devamlı olarak geliştirilmiş hali olarak da düşünülebilir. En temel amaç Von Neumann mimarisini verimli bir şekilde kullanmaktır. Hatta imperatif dillerdeki temel konseptlerin donanım işlemleriyle oldukça paralel olduğu görülebilir. Örneğin

Sabit olmayan değişkenler – hafıza hücreleri,

Dereference (C'deki "*" operatörünün gerçekleştirdiği) işlemi – Bellekten veri yükleme

Atama – bellekte depolama

Kontrol yapıları – atlamalar

Backus'a göre (Backus,1978) saf imperatif programlama Von Neumann darboğazı tarafından kısıtlanmaktaydı. Temel problem ise veri yapılarının kelime kelime kavramlaştırılmasıydı.

Programlama dilleri büyüyen yazılım ihtiyaçları için ölçeklenmek istiyorlarsa yüksek seviyeli soyutlamalar tasarlayabilmek için teknikler geliştirmek gerekiyordu. Aynı zamanda bu yapılar üzerinde mantık yürütebilmek için teorilere (Matematikte bir teori belli veri tiplerinden,bu tipler üzerinde tanımlı operasyonlardan ve operasyon ve değer arasındaki ilişkileri tanımlayan kanunlardan oluşur.) ihtiyaç olacaktır. Ancak imperatif dillerde değişkenler bellek hücreleri olarak düşünüldüğü için çokça kullanılan değişim (mutation) operasyonunun matematikte bir karşılığı bulunmamaktadır. Eğer dayandıkları matematiksel teorileri kullanarak yüksek seviyeli soyutlamalar oluşturmak istiyorsak sorun yaratmaktadır. Matematikte değişim operatörü olmadığı eklediğimiz takdirde teoride mevcut olan kanunları da bozmaktadır. Burada yeni bir yaklaşıma gidilmesi gerektiği düşünülebilir.

Öne süreceğimiz fonksiyonel programlama ise temel programlama paradigmalarından bir tanesidir. Komutların fonksiyonlarla ifadesine dayanır. Değişim işleminden uzak durulmasını öğütler, hatta bazı çoğu fonksiyonel dilde bu işlem mümkün değildir. Fonksiyonların oluşturulması ve soyutlanmasıyla ilgili kuvvetli yöntemler içerir.. Bu özellikleriyle yukarıda tarif ettiğimiz soruna bazı çözümler getirebilir. Fonksiyonel dillerde bilgisayara yapılmak istenen bir işin nasıl yapılacağı tarif edilmez, işlemin kendi beliltilir programcı sistemin işletim detaylarıyla ilgilenmez böylece programcıya daha yakın bir kodlama düzeyi sağlanmış olur. Ayrıca matematiksel bir programlama yaklaşımı getirdiği de söylenebilir Temellerini ise hesaplama teorisinde yer alan modellerden biri olan lambda calculusten (Church,1941) alır. Lambda calculus çok genel bir ifadeyle değişkenlerin yer değiştirmesi yoluyla fonksiyon indirgemesi kullanılarak hesaplama yapar. İfade indirgenemeyecek hale geldiğinde işletim sonlandırılır ve indirgenemeyen son hale normal form adı verilir. Fonksiyonel programlama üzerinde uzlaşılmış net bir tanımlama bulunmasa da genel olarak saf ve saf olmayan diller olarak bir ayrım yapmak mümkündür. Değişim, atama gibi işlemlere izin vermeyen, imperatif kontrol yapılarına sahip olmayan ve fonksiyonları yan etki barındırmayan

fonksiyonel dillere saf fonksiyonel diller denilir. Pratik olarak kullanılan pek çok işlem bu tanımlanın bir kısmını ihlal ettiği için pek az saf fonksiyonel dil mevcuttur. Örneğin, Haskell(bazı monadları çıkartıldığında), XSLT,XPath gibi belirli bir alana özel diller bu sınıfa girer. Öte yandan temel yazılım yaklaşımını fonksiyonlar üzerinde yoğunlaştığı ve fonksiyonlarla çalışmak için belirli mekanizmaları sağlayan ancak imperatif dillerde kullanılan özellikleri de barındıran dillere saf olmayan fonksiyonel diller denir. Pek çok fonksiyonel dil bu sınıfa girer. Örneğin Lisp, Racket, Clojure, Erlang, Scala, SML, OCaml, F# vs. ayrıca Javascript,Ruby,Smalltalk,Python,C#, C++11, Java8 gibi bazı imperatif dillerde bazı fonksiyonel programlamaya ait mekanizmalar bulunmaktadır. Fonksiyonel diller, imperatif dillere kıyasla daha basit syntax,semantic ve daha fazla esneklik sağlamasına rağmen işletilmeleri bu dillere kıyasla daha verimsizdir.
Fonksiyonel programlama geçmişte çoğunlukla akademik araştırma alanı olarak görüldüyse de günümüzde bu paradigmayı kullanan dillerin pek çoğu olgunlaşmıştır. Ayrıca paralel sistemlerin yaygınlaşmasıyla beraber fonksiyonel dillere karşı olan ilgi de artmaktadır. Değerlerin değişmesinin engellenmesiyle bellek paylaşımlı sistemlerdeki senkronizasyon problemi ortadan kalktığı için paralel programlar yazmak çok daha kolay olmaktadır.

5.1. Temel Özellikleri

Fonksiyonel paradigmanın getirdiği temel kavram ve özellikler aşağıda maddeler halinde açıklanmaya çalışılmıştır.

- **Birinci sınıf fonksiyonlar:** Bir programlama dilinde değişkenler ve veri yapılarında saklanabilen, bir fonksiyona veya alt rutine parametre olarak gönderilebilen, bir fonksiyondan veya alt rutinden sonuç olarak döndürülebilem ve çalışma zamanında oluşturulabilen yapılara birinci sınıf vatandaşlar adı verilir. Fonksiyonel dillerde fonksiyonlar birinci sınıf vatandaşlar olarak tanımlanmışlardır. Söylenenlere ek olarak program içerisinde herhangi bir noktada fonksiyon yaratılabilir. İçe içe yuvalanmış fonksiyonlar gibi yapılar da mümkündür.

- **Yan etkiler:** Matematikte fonksiyonlar sadece bir girdi setini bir çıktı setine belirli işlemler kullanarak eşlerler. Fonksiyonel programlamada gerçek fonksiyonlara benzer şekilde sadece verilen argümanlardan bir çıktı üretmesi beklenir. Bundan gayrı fonksiyon alanından (scope) dışarı ile girilen her etki yan etki olarak adlandırılır. Yan etki bulundurmayan fonksiyonlar saf fonksiyonlar olarak adlandırılır. Yan etkilerden mümkün olduğu kadar kaçınılması gerektiği önerilmektedir. Fonksiyonların bir şekilde kendi alanlarından dışarı çıkması pek çok durumda zorunluluk olduğu için programların tamamen saf olması oldukça zordur. İlginç bir durum olarak saf bir dil olan Haskell'de fonksiyonlar yan etki barındıramayacakları için gerektiğinde monad adı verilen özel yapılar kullanılır.

- **Yüksek mertebeden fonksiyonlar:** Bir başka fonksiyonu argüman olarak alabilien fonksiyonlara veya bir başka fonksiyonu sonuç olarak döndürebilen fonksiyonlara yüksek mertebeden fonksiyonlar denir. Matematiksel türev fonksiyonu yüksek mertebeden fonksiyonlara örnek gösterilebilir. Çünkü argüman olarak bir fonksiyon alıp, sonuç olarak başka bir fonksiyon döndürmektedir. Programlamada en çok bilinen yüksek mertebeden fonksiyonlar map ve reduce fonksiyonlarıdır. Map fonksiyonu verilen bir listenin her elemanına kendisine argüman olarak gönderilmiş fonksiyonu uygular, ve sonuç olarak her eleman için dönen sonuçları içeren yeni liste döndürür. Reduce fonksiyonu ise, birden fazla parametresi bulunan bir fonksiyonu argüman olarak alır ve listeden elemanlar seçerek fonksiyonu işletir, sonuç olarak bütün elemanlar işletildiğinde elde edilen bir sonuç değeri döndürür. Yüksek mertebeden fonksiyonlar imperatif diller de dahil olmak üzere pek çok dilde desteklenmektedir.

- **Anonim fonksiyonlar:** Bir tanıtıcıya bağlanmadan tanımlanabilen ve çağrılabilen fonksiyonlara anonim fonksiyonlar denir. Lambda fonksiyonları olarak da bilinirler. Çok kısa işleri yapmak için veya yüksek mertebeden fonksiyonlarla kullanılmak için uygundurlar. Günümüzde yaygın kullanılan

pek çok dilde anonim fonksiyon desteği vardır veya eklenmesi planlanmaktadır.

- **Rekürsif fonksiyonlar:** İteratif hesaplama, programalama esnasında çok fazla kullanılan bir hesaplama yöntemidir. Döngülerle veya rekürsif fonksiyonlarla ifade edilebilirler. İmperatif diller rekürsif fonksiyonlar destekleseler de temel iterasyon yapısı olarak döngüleri kullanmaktadırlar. Ancak değişim işlemi olmadan döngülerle iterasyon yapmak mümkün olmamaktadır. Bazı saf olmayan fonksiyonel diller döngüleri de destekleseler de temel iterasyon yapısı olarak rekürsif fonksiyonları kullanırlar. Rekürsif bir fonksiyon kendi içerisinde kendisini bir veya daha fazla kere çağıran fonksiyonlara denir. Hesaplama teorisine göre salt rekürsif fonksiyon kullanan diller hesapsal olarak imperatif diller kadar kuvvetlidir. Bu da demektir ki döngülerle yapılan her türlü hesaplama, rekürsif fonksiyonlarla da yapılabilir. Rekürsif çağrı her yapıldığında çağrıyı yapan fonksiyonun durumu işletim sistemi tarafından kullanılan akış yapısının stack alanına daha sonra fonksiyonun geri kalanı işletilmek üzere kaydedilir. Bu rekürsif algoritmaların, imperatif algoritmalardan daha fazla bellek kullanması anlamına da gelmektedir. Çok fazla rekürsif çağrı yapıldığında o programa ayrılmış olan stack alanı dolabilir (buna stack overflow'da denir.) Bu istenmeyen bir durumdur. Ancak buna karşı tail rekürsif fonksiyonlar adı verilebilen yöntem kullanılabilir. Eğer bir fonksiyon son yapacağı işlem olarak kendisini veya başka bir fonksiyonu çağırısa buna tail çağrısı adı verilir. Derleyiciler veya runtime'lar bu durumu anlayabilir ve bu çağrı fonksiyonun en sonunda yapıldığı için fonksiyonun artık stack alanına kaydedilmesine gerek olmadığı için kayıt işlemini atlayarak doğrudan çağrıyı gerçekleştirirler buna tail çağrısı optimizasyonu adı verilir. Kendi kendine yaptığı çağrı başka bir hesaplamaya bağlı bulunmadan fonksiyonun sonunda bulunan özel rekürsif fonksiyonlara, tail rekürsif fonksiyon adı verilir. Bu fonksiyonlar, performans olarak iteratif döngülerle eşdeğerdirler ve yukarıda tarif edilen problemin çözümünde kullanılabilirler

- **Referential transparency:** Referential transparency özelliğinin sağlanabilmesi için bir fonksiyonun sonucunun sadece ve sadece girdilerine bağlı olması gerekmektedir. Bu da o fonksiyonun aynı değerlerden her zaman aynı sonuçları üretmesi anlamına gelir. Bunun için fonksiyon dışından herhangi bir değerin kullanılmaması gerekmektedir. Bütün matematiksel fonksiyonların bu özelliği sağladığı söylenebilir. Eğer bu özelliği sağlayan bir fonksiyon bir argüman almıyorsa her zaman aynı sonucu döndürmelidir (örneğin pi sayısının değerini veren fonksiyon). Argüman almadan farklı sonuçlar döndürebilen random gibi fonksiyonların bu özelliği sağlamaması anlamına gelir. Fonksiyonların bu şekilde dış etkilere bağışık oluşu derleyicinin de programcının da işini kolaylaştırmaktadır. Programı anlamak, onun üzerinde çalışmak, programın doğruluğunu ispatlamak kolaylaşır. Ayrıca program memoization (Bir fonksiyonun döndürdüğü değer kaydedilir, eğer tekrar aynı argümanla çağrı yapılırsa işlem tekrar yapılmadan hafızadaki değer döndürülür), yaygın ifadelerin elenmesi, paraleleştirme gibi optimizasyon teknikleri de kolayca uygulanabilir.

- **Tembel işletim:** Programların işletilmesinde kullanılan stratejilerden bir tanesidir. Programlama dillerinin pek çoğunun kullandığı hevesli işletim stratejisinde bir değer bir değişkene atandığı anda hesaplanır. Daha sonra kullanılıp kullanılmayacağıyla veya aynı ifadenin daha sonra tekrar işletilip işletilmeyeceğiyle ilgilenmez. Daha sonra kullanılmayacak değerlerin hesaplanması veya aynı değerin tekrar tekrar hesaplanması bir işlem yükünü beraberinde getirir ancak işletim sırasının kod organizasyonuyla belirlendiği imperatif dillere en uygun olan yaklaşımdır ve neredeyse hemen hemen hepsi bu stratejiyi kullanır. Tembel işletimdeyse bir değer belirlendiğinde o değere başka bir hesaplamada ihtiyaç duyulana kadar ifadenin işletimi ertelenir. Bu tip stratejiler katı olmayan işletim stratejileri grubuna dahildir. Bir ifadenin tekrarı bir fonksiyonun çalışma zamanını çok fazla arttırabilir. Tekrar lı ifadelerin tembel işletimle engellenmesi bu gibi durumlarda fazlaca performans kazancı sağlayabilir. Hatta hatalı bir ifade varsa ancak kullanılmamışsa programda hata ortaya çıkmaz. Tembel işletim sadece

gerektiğinde çağrı yapısıyla sonsuz ifadelerin tanımlanabilmesini de sağlar. Python, C# gibi hevesli işletim kullanan dillerde bazı tembel işletim mekanizması bulunur.

6. Örnek Programlama Dili: Erlang

Erlang, 1986 yılında Ericsson tarafından geliştirilmiş (Armstrong, 2007) bir fonksiyonel programlama dilidir. Adını hem İngilizce Ericsson dilinden (ERicsson LANGauge), hem de günümüz telekominikasyon ağlarını temellerinden biri haline gelmiş telefon ağı analizi alanını yaratmış Danimarkalı matematikçi Agner Krarup Erlang'dan almaktadır.

Tarihsel olarak 80'lerin ortasına doğru Ericsson Bilgisayar Bilimleri Laboratuarı'na yeni nesil telekom altyapı ve ürünlerinin geliştirebilmesi için uygun bir programlama dili araştıması görevi verilir. Bjarne Däcker'ın gözetiminde Joe Armstrong, Robert Virding, ve Mike Williams'dan oluşan bir ekip iki sene boyunca mevcut programlama dilleriyle telekom uygulaması prototipleri geliştirirler ancak pek çok dilde ilginç ve işlerine yarayan yapılar gördülerse istedikleri özelliklerin tamamını kapsayan bir dile rastlayamadılar. Böylece kendi dillerini tasarlamaya karar verdiler. Erlang fonksiyonel ML ve Miranda dillerinden, paralel ADA ve Simula dillerinden mantıksal prolog dilinden ve bazı özellikleri açısından Smalltalk dilinden esinlenmiştir (Cesarini,Thompson,2009) . Erlang dili bir süre sadece Ericsson'un kendi iç süreçlerinde kullanılmış ardından bir ürün olarak dışarıya da açılmıştır. Dil belirli bir olgunluğa ulaştıktan sonra 1996 yılında Erlang derleyici ve yorumlayıcısı, Ericsson'un bazı telekominikasyon çözümleri, çok sayıda kütüphane, yardımcı araç ve protokollerle beraber OTP (Open Telecom Platform) adıyla piyasaya sunulmuştur. Bu platform 1998 yılında MPL türevi bir lisansla açık kaynak hale getirilmiştir ve o günden beri telekom endüstrisinde yaygın kabul görmektedir. Ericsson'un yanı sıra Motorola ve T-Mobile altyapılarında da kullanılmaktadırlar. Erlang'ın telekom endüstrisi için geliştirildiğinden dolayı sahip olduğu özellikler, günümüzde paralel sistemlerin ihtiyaçlarını da karşılayabildiğinden dolayı Erlang telekominikasyon

dışındaki alanlarda da kullanılmaktadır. Örneğin Erlang ile geliştirilen CouchDB, RabbitMQ, Ejabberd gibi yazılımlar piyasada yaygın olarak kullanılmaktadır, Ayrıca Amazon,Facebook,Yahoo gibi şirketler de Erlang ile geliştirilmiş servisler kullanmaktadır. (Cesarini,Thompson,2009).

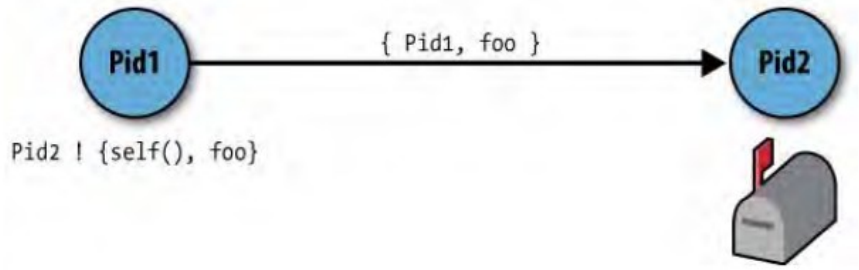
Bir programlama dili olarak Erlang'ın karekteristikleri şöyle açıklanabilir. Erlang saf olmayan bir fonksiyonel programlama dilidir. Dinamik bir dildir ve katı işletim stratejisini kullanır. Her değişkene tek bir defa değer atanabilir. Bundan gayrı atama operatörü örüntü bulma (pattern matching) işlemini gerçekleştirebilir. Sadece yüksek seviye yapılarda değil bit dizilerinde (Erlang dilinde bit dizileri "<<...>>" şeklinde tanımlanır) de örüntü bulma işlemi yapabilmesi onu bu konuda diğer dillerden ayrı kılar. Erlang paralel ve eş-zamanlı programlama için tasarlanmıştır, aynı zamanda gerçek zamanlı uygulamalarla da kullanılabilmektedir. Haberleşme asenkron mesaj iletimi ile gerçekleşir. Erlang temel paralel birim olarak thread kullanmaz kendi VM'i içerisinde hafif process adı verilen bir yapı kullanılır. Bunlar threadlere benzeseler de herhangi bir veri paylaşmazlar. Erlang VM oluşturulan her hafif process için bir işletim sistemi threadi başlatmaz, hafif processlerin oluşturulması, zamanlaması ve yönetilmesi VM tarafından yapılır. Böylece mevcut hafif processlerin sayısından bağımsız olarak yaratılma zamanları mikrosaniyelerle ölçülür. Erlang VM aynı zamanda otomatik bellek yönetimi (çöp toplama) da yapmaktadır. Erlang telekom sektörü için geliştirildiği için hataya oldukça dayanıklı olmalıdır. Çünkü telefon hatlarının olası hatalarda çalışmaması kabul edilemez. Erlang tasarlanırken kabul edebilen çalışma süresi %99.999 olarak belirlenmiştir. Yani senede sadece 5 dakika çalışmamasına tahammül gösterilebilir. (Williams,2003)
Erlang'da hataları ele alma stratejileri "Bırak çakılsın" şeklinde özetlenebilir. Processler birbirleriyle ilişkilendirilebilir ve ilişkili processlerden her biri olası bir hatada çöktüğünde diğerlerine haber verir. Diğer processler hatayı yakalayabilir veya kendileri de çökebilirler hatta işi bağlı bulunduğu processleri hatalara karşı izleyip gerektiğinde sonlandırmak olan gözetmen processler tanımlanabilir. Joe Armstrong bu duruma şöyle örnek vermektedir. "Eğer insanlarla dolu bir salonda biri ölürse diğerleri fark edecektir." (Armstrong,2007). Telekom sistemlerini yeni modüller ekleme veya güncelleme yapma gibi sebeplerle kapatmak kabul edilemez

olduğu için Erlang çalışma anında dinamik kod yükleme veya mevcut çalışan kodu değiştirme gibi özellikleri de bulunmaktadır. .Erlang içerisinden C,Java,Perl, Python,Lisp gibi farklı dillere ait kodları çalıştırmak da mümkündür.

6.1. Erlang ile Paralel Programlama

Herşeyden önce Erlang ile paralel programlama yapmak için kullanılan temel mekanizmaların açıklanması daha isabetli olacaktır.

Pid = spawn(Fun) komutu Fun fonksiyonunu işletecek bir paralel process yaratır. Ve yaratılan processin process tanımlayıcısını döndürür. (Erlang'da process tanımlayıcıları şuna benzer <0.30.0>). *self()* bir processin kendi Pidsini barındırır.



Şekil 6.1: Erlang'da mesaj iletimi

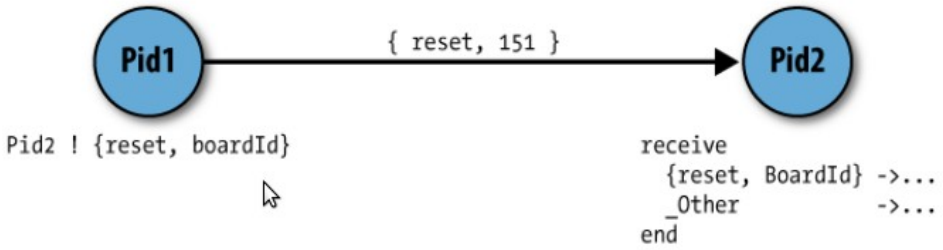
Pid ! Message komutu Pid processine asenkron bir mesaj yollar yani process cevabı beklemeden işletimine devam edecektir. Erlang'da "!" gönderme operatörü olarak tanımlanmıştır. Her Erlang processinin başka processlerden gelen mesajları depoladığı bir posta kutusu mevcuttur. Bir mesaj gönderildiğinde mesaj gönderen process'ten iletilen process'in posta kutusuna kopyalanır. Eğer bir process başka bir processe birden fazla mesaj gönderiyorsa mesajları sıralı olarak iletileceği garanti altına alınmıştır. Ancak bu garanti birden fazla processten gelen mesajların sıralanmasını kapsamaz bu durumda sıralama VM'e bağlıdır. Erlang'da mesaj gönderimi asla başarısız olmaz. Olmayan bir processe mesaj gönderseniz dahi herhangi bir hata mesajı alınmayacaktır. Mesaj iletimi Şekil 6.1'da (Cesarini,Thompson,2009) gösterilmiştir.

receive ... end : komutu processe gelen mesajları işlemek için kullanılır. Kullanımı şu

şekildedir. Gelen mesajlar verilen örüntülerle eşleştiği zaman eşleştiği örüntünün ifadeleri işletilir. After ifadesiyle timeout tanımlanabilir

```
receive                                    after Time ->
        Pattern1 [when Guard1] ->                  Expressions3
                Expressions1;              end.
        Pattern2 [when Guard2] ->
                Expressions2;
```

Şekil 6.2'de gönderilen mesajların nasıl işlendiği gösterilmiştir.



Şekil 6.2: Erlang'ta alınan mesajların işlenmesi

Burada anlatılan temel işlemlerin kod içerisinde nasıl kullanabildiği Şekil 6.3 (Armstrong, 2007) 'de görülebilir.

```
-module(area_server2).
-export([loop/0, rpc/2]).

rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        {Pid, Response} ->
            Response
    end.
```

```
loop() ->
    receive
        {From, {rectangle, Width, Ht}} ->
            From ! {self(), Width * Ht},
            loop();
        {From, {circle, R}} ->
            From !  {self(), 3.14159 * R * R},
            loop();
        {From, Other} ->
            From ! {self(), {error,Other}},
            loop()
    end.
```

Erlang Shell:

```
1> Pid = spawn(fun area_server2:loop/0).
<0.37.0>
3> area_server2:rpc(Pid, {circle, 5}).
78.5397
```

Şekil 6.3: Erlang mesaj iletimi örneği

Her zaman Pidlerle çalışmak uygun olmayabilir Erlang'da Pidleri isimlerle eşleştirmek mümkündür. Bunun için aşağıdaki ifadeleri kullanmak gerekir.

register(Atom,Pid): Bir atom verilen pid ile eşleştirilir. Atom farklı bir Pid ile

eşlenmişse hata dönecektir.

unregister(Atom): Atom, Pid eşleştirmesini kaldırır.

whereis(Atom) -> Pid | undefined : Verilen atomun Pid'sini döndürür, atom için kayıtlı Pid yoksa undefined döndürülür.

Yukarıda processlerin birbirleriyle ilişkilendirilebildğinden bahsetmiştim. Bu dil içerisinde şu ifadelerle yapılır.

link(Pid): Çağrıyı yapan processle Pidsi verilen process arasında çift yönlü bir bağ oluşturur.

unlink(Pid): Çağrıyı yapan processle Pidsi verilen process arasındaki bağı kaldırır.

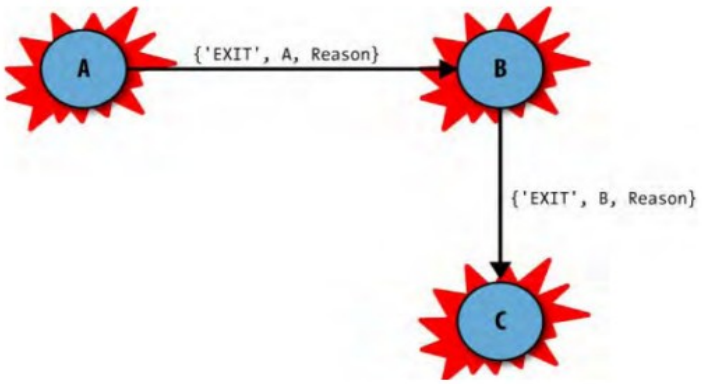
spawn_link(Fun): Çağrıyı yapan process yeni bir process yaratır ve onu kendine bağlar.

Bağlanmış processler olası hatalara karşı birbirlerini gözlemeye başlarlar. Eğer bu processlerden biri çakılırsa diğerine veya diğerlerine exit sinyali adı verilen bir sinyal gönderir. Exit sinyali aşağıdaki gibi elle de gönderilebilir.

exit(Reason) : Process kendini öldürür, göndereceği mesajda ölme nedeni Reason olarak belirtilecektir

exit(Pid,Reason): Process, başka bir Pid processine exit sinyali gönderir

Eğer başka bir şey yapılmamışsa sinyali alan processte kendini öldürür ve ilk processten aldığı sinyali kendi bağlı olduğu processlere gönderir. Bu durum Şekil 6.4'de görülebilir.

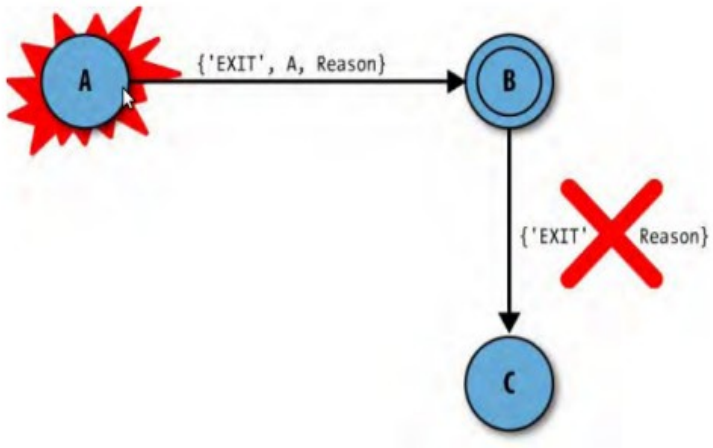


Şekil 6.4: Exit sinyalinin yayılması

Birbirine bağımlı processleri birbirine bağlamak önemlidir, böylece bir çökme yaşandığında processlerin hepsinin birden çöktüğünden emin olunur. Elbette bir

process exit sinyali aldığında tek yapabilceği kendisi de ölmek değildir. Hataları yakalamak da mümkündür. Bir hata yakalandğında hata mesajı yakalayan processin posta kutusuna düşürülür, ve mesaj başka processlere yayınlanmaz. Bunun için aşağıdaki komut kullanılır.

process_flag(trap_exit,true): Bu bayrakla işaretlenmiş bir processe sistem processi denir. Sistem processleri hata yakalama özelliğine sahiptirler.



Şekil 6.5: Erlang'da hata yakalama

Erlang'da hata yakalamak için 3 temel yaklaşım vardır.

- Eğer yaratılan processin ölüp ölmeyeceği ile ilgilenilmiyorsa *spawn* metodu
- Eğer yaratılan processin ölmesi durumunda mevcut processin de ölmesi gerekiyorsa *spawn_link*
- Eğer yaratılan processin ölmesi durumunda oluşan hata yakalanmak isteniyorsa *spawn_link* ve *trap_exit*

kullanılması gerekir.

7. Örnek Programlama Dili: Scala

Scala, 2001 yılında EPFL'de Martin Odersky tarafından tasarlanmaya başlanmış ilk sürümünü 2003 yılında çıkartmış bir programlama dilidir (Odersky, 2008). Scala paradigma olarak hibrit bir dildir. Hem nesneye yönelik programa, hem de fonksiyonel programlama paradigmalarının iyi taraflarından faydalanmaya çalışır.

Adını ölçeklenebilir dil anlamına gelen SCAble LAnguage'dan alır. Dil kullanıcıların isteklerine göre genişleyebilmesi amacıyla tasarlanmaya çalışılmıştır. Eric Raymond kitabında (Raymond, 1999) katedral ve pazar yapılarını açık kaynaklı yazılım geliştirme metodolojisini anlatmak için benzetme olarak kullanmıştır. Burada Katedral, yapımı uzun zaman alan ancak uxun süreler boyunca değişmeden kalan mükemmele yakın bir yapı iken pazar orada çalışan insanlar tarafından devamlı, uyarlanan ve geliştirilen yapıları ifade etmektedir. Guy Steele (Steele, 1999) bu benzetmenin programlama dilleri için de uygulanabileceğini öne sürmüştür. Bu benzetmeyi kullanarak Scala'yı pazara benzetmek doğru olacaktır çünkü Scala bir programcı ihtiyacı olabilecek her türlü yapıyı sağlayan mükkemmel bir dil olmaktan gayrı, kullanıcısı bu tarz yapıları oluşturabilecek araçları sağlamaya çalışır. Scala derleme platformu olarak JVM kullanır. Her ne kadar ana platformu olan Java eskiyip, diğer dillere göre geride kalmaya başladıysa da, JVM gelişmeye devam etmektedir. JVM bugün yeryüzündeki en başarılı derleyiciler arasında görüldüğü için JVM üzerinde çalışacak programlama dilleri geliştirmek günümüzde yaygın bir çabadır. JVM üzerinde çalışmasından dolayı Java byte koduna derlenmektedir. Bu sebeple performansı Java'ya oldukça yakındır.Mevcut bütün Java kütüphaneleri Scala ile uyumludur. Hatta Scala kendi yapılarının, kütüphanelerinin çoğunu mevcut Java yapıları üzerine kurmuştur bu sebeple. Scala içerisinden ek bir syntax veya arayüz kullanmadan Java methodları çağrılabilir, Java sınıfları ve arayüzleri kullanılabilir.

Scala pek çok programlama dilinden etkilenmiştir, aslında Scala'nın pek az özelliği özgündür. Scala'nın getirdiği önemli ilerlemelerin pek çok bu yapıların birlikte kullanılmasına yöneliktir. Scala syntaxını büyük oranda C/C++,Java,C# gibi dillerden almıştır. Java'nın işletim modelini, basit tiplerini ve sınıf kütüphanelerini kullanır. Standartlaşmış nesne yapısında Smalltalk'tan, evrensel yuvalama özelliği Simula,Algol'dan, fonksiyonel programlama anlayışı ML'den, Scala standart kütüphanesindeki yüksek mertebeden fonksiyonları ML ve Haskell'den, örtük parametreleri Haskell'den ve paralel yapıları Erlang'tan esinlenmiştir.

(Odersky et. al ,2010) Her ne kadar fonksiyonel yapılar içeren nesne yönelimli diller ve nesne yapıları içeren fonksiyonel diller bulunsa da Scala'nın bu iki paradigmayı birleştirme adına yapılmış en başarılı çalışmalardan biri olduğu söylenebilir. Scala görece yeni bir dil olsa da piyasada çabukca kabul görmüştür. Bugün pek çok büyük

firma ve kurum çeşitli operasyonlarında Scala kullanmaktadır. Örneğin. Twitter, Amazon ,IBM ,Intel, NASA, HSBC vb.

Bir programlama dili olarak Scala'nın temel özellikleri şöyle açıklanabilir. Scala, Java gibi statik bir dildir ancak tip sistemi Java'ya kıyasla daha güçlüdür. Çünkü Scala statik dillerin iki temel problemine de belli çözümler getirir. Scala derleyicisi tip çıkarımı yapabilir (val x = 3+3 ifade için Scala'ya x'in integer olduğunu söylemeniz gerekmez) bu özellik programların daha sade olmasını sağlar. Ayrıca yeni tipler üretmek için örüntü bulma ve başka yeni teknikler kullanarak dinamik dillerdeki esnekliğe yaklaşabilir. Scala nesneye yönelik yapısı sebebiyle değişim işlemine izin verir. Ancak saf olmayan bir fonksiyonel dilde mevcut olan özelliklerin hemen hemen tamamına izin verir. Scala'nın nesneye yönelik yapısı ise Java'dan farklıdır, Smalltalk'ın adımlarını takip eder ve daha saf bir nesneye yönelik yapı ortaya koyar. Scala'da primitiv tipler, statik yapılar gibi bir nesnenin parçası olmayan hiç bir yapı bulunmaz. Bütün değerler birer nesneyken bütün operasyonlarda birer metod çağrısıdır. Fonksiyonel programlama yapısına uygun olarak bütün metodlar veya fonksiyonlar her zaman bir değer döndürür, void yapılara rastlanmaz. Scala nesne yaratma açısından mevcut en güçlü dillerden bir tanesidir. Scala trait'leri metodları gerçeklenebilen Java arayüzlerine benzese de daha güçlüdür. Bir sınıf elemanlarına birden fazla traitin elemanları eklenebilir. Bu yolla bir sınıfın farklı kısımları, farklı traitlerle kapsülleniyor olabilir.

7.1. Scala ile Paralel Programlama

Scala birden pazla paralel programlama yapısı desteklemektedir. Java uyumluluğu sayesinde java.util.concurent kütüphanesiyle standart Java'da bulunan bellek paylaşım model kullanılabilir. Mesaj iletimli aktör modeli (Hewitt,1973) Scala'nın temel paralel programlama modelidir. Scala dilinde ayrıca işlemsel bellek ve future yapıları da kullanılabilir.

Erlang'ın aksine Scala'nın temel paralel birimi threadlerdir. Aktör modelinin gerçeklenmesi sırasında da threadler kullanılır. En başta 4 threadden oluşan bir

thread havuzu kullanılırken bu ihtiyaca göre arttırılabilir. Scala'da aktör tanımlamak için iki farklı yol bulunmaktadır. Birincisinde bir aktör sınıfı yaratıp içine act methodunu gerçekleyebiliriz. Örneği Şekil 7.1'de (Wampler, Payne, 2009) görülebilir.

```
import scala.actors.Actor

class Redford extends Actor {
  def act() {
    println("A lot of what acting is, is paying attention.")
  }
}

val robert = new Redford
robert.start
```

Şekil 7.1: Scala aktör tanımı (nesnel)

Bu yaklaşım biraz nesneye dayalıdır. Daha fonksiyonel bir yaklaşım şöyle getirilebilir.

```
import scala.actors.Actor
import scala.actors.Actor._

val paulNewman = actor {
  println("To be an actor, you have to be a child.")
}
```

Şekil 7.2: Scala aktör tanımı (fonksiyonel)

Şekil 7.2'deki (Wampler, Payne, 2009) yaklaşımın daha fonksiyonel olduğu görülebilir. Mesaj göndermek için Erlang'da olduğu gibi "!" operatörüyle yapılmaktadır. Scala'da buna ek olarak mesaj gönderimiyle ilgili iki tane daha operatör vardır. "!!" operatörü bir future (şu anda boş olan ancak gelecekte bir değer döndürebilecek bir nesne, asenkron hesaplamalar için kullanılır) döndürürken, "!?" operatörü senkron bir mesaj gönderimi başlatır.

```
import scala.actors.Actor
import scala.actors.Actor._

val fussyActor = actor {
  loop {
    receive {
      case s: String => println("I got a String: " + s)
      case i: Int => println("I got an Int: " + i.toString)
      case _ => println("I have no idea what I just got.")
    }
  }
}

fussyActor ! "hi there"
fussyActor ! 23
fussyActor ! 3.33
```

Şekil 7.3: Scala'da alınan mesajların işlenmesi

Şekil 7.3'te (Wampler, Payne, 2009) Scala'da alınan mesajların işletilmesi görülebilir. Erlang'da olduğu gibi aktöre gelen mesajlar posta kutusuna düşerler ve burada işletilmeyi beklerler. Scala'da posta kutusundaki mesaj sayısını öğrenmek için bir de *mailboxSize()* metodu bulunur. *Recieve* metoduna ek olarak bir de timeout kullanılabilen *recieveWithin(timeout)* metodu bulunur. *Recieve* çağrısıyla bekleyen bir aktör mesaj gelmese dahi sürekli işletilecek ve hiç bir şey yapmasa da havuzdaki bir threadi işgal edecektir. Aktörlerin olay tabanlı kullanabilmeleri için *react* adındaki metod kullanılabilir. Bu metod kullanıldığında aktör sadece mesaj geldiğinde işletilecektir. Ancak *recieve* işlemi yapan bir threadin değer döndürmesi beklenirken *react* işlemi gerçekleştiren bir threadden bu beklenmez. Scala'da işlemsel bellek kullanımı Şekil 7.4'de (ScalaSTM) gösterilmiştir. Paylaşılacak değişkenler Ref

```
import scala.concurrent.stm._

val x = Ref(0) // allocate a Ref[Int]
val y = Ref.make[String]() // type-specific default
val z = x.single // Ref.View[Int]

atomic { implicit txn =>
  val i = x() // read
  y() = "x was " + i // write
  val eq = atomic { implicit txn => // nested atomic
    x() == z() // both Ref and Ref.View can be used inside atomic
  }
  assert(eq)
  y.set(y.get + ", long-form access")
}

// only Ref.View can be used outside atomic
println("y was '" + y.single() + "'")
println("z was " + z())

atomic { implicit txn =>
  y() = y() + ", first alternative"
  if (x getWith { _ > 0 }) // read via a function
    retry // try alternatives or block
} orAtomic { implicit txn =>
  y() = y() + ", second alternative"
}
```

Şekil 7.4: Scala ile işlemsel bellek kullanımı

ifadesi ile işaretlenir. Ref ifadeleri sadece atomic yapıların içerisinde işletilebilirler. Bütün bunların haricinde Scala dilinde paralel koleksiyon desteği mevuttur. Scala derleyicisi .par ile işaretlenmiş veri yapılarını paralelleştirmeye çalışacaktır.

8. Sonuç

Günümüz bilgisayar mimarisinde paralelliğin önemi oldukça açıktır. Çalışma içerisinde paralel sistemler ve paralel programlama modelleri incelenmiş ve imperatif dillerle senkronizasyon mekanizmaları kullanarak paralel programlama yapmanın oldukça zor olduğu kanısına varılmıştır. Köklü bir geçmişi olan ancak kullanım alanı genelde akademi ile sınırlı kalmış fonksiyonel programlama dillerinin temel özelliklerinin günümüzde paralel programlamaya dair sorunlarımızın bazılarını çözebileceği fark edilmiştir. Bu sebepten dolayı fonksiyonel programlama dillerinin günümüzde ilgi görmeye başladığı anlaşılmıştır. Paralel programlama için yaygın olarak kullanılan iki fonksiyonel programlama dili incelenmiştir. Erlang dağıtık sistemlerde özellikle haberleşme sistemlerinde çok iyi performans vermektedir ancak bilişim sektöründe kullanım alanı sistem yazılımlarında olmuştur. Scala ise daha genel amaçlı olup web teknolojileri alanında yaygın bir şekilde kullanılmaktadır.

KAYNAKLAR

Agarwal, A., Levy, M. (2007), The KILL Rule for Multicore. Design Automation Conference 2007: 750-753

Amdahl, G. (1967) Validity of the single-processor approach to achieve large scale computing capabilities. AFIPS Joint Spring Conference Proceedings 30 (Atlantic City, NJ, Apr. 18– 20), AFIPS Press, Reston VA, pp 483-485, At http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf Erişim Tarihi 27 Mayıs 2013

"Amdahl's Law", (2011) Encyclopedia of Paralel Programming,Springer Science+Business Media,

Armstrong, J., (2007) "History of Erlang", in HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages,

Armstrong, J., (2007) Programming Erlang: Software in a Concurent World, Pragmatic Bookshelf, ABD

Cesarini, F., Thompson , S., (2009) Erlang Programming, Oreilly Media, ABD

Church, A. (1941) Annals of Mathematics Studies. Volume 6: Calculi of Lambda Conversion. Princeton Univ. Press, Princeton, ABD

Dijkstra, E. (1974) Over seinpalen (Flemenkçe). EWD 1974. http://www.cs.utexas.edu/users/ EWD/ewd00xx/EWD74.PDF. Erişim Tarihi 21 Haziran 2013

Flynn, M.J., (1972) Some computer organizations and their effectiveness. IEEE Transactions on Computers, 21(9): 948–960,.

"Flynn's Taxonomy",(2011). Encyclopedia of Paralel Programming, Springer Science+Business Media,

Fuller, S. H., Millett, L. I., (2011) Computing Performance: Game Over or Next Level?, Computer, IEEE, 44(1), syf 31-38

Gabrielli, M., Martini, S., (2010) Programming Languages: Principles and Paradigms , Springer, İngiltere

Gunther, N. J.,(2002) A New Interpretation of Amdahl’s Law and Geometric Scaling. http://arxiv.org/PS_cache/cs/pdf/0210/0210017v1.pdf. Erişim Tarihi 27 Mayıs 2013

Gustafson, J. L., (1998) Reevaluating Amdahl's Law, Communications of the ACM 31(5), syf. 532-533

"Gustafson's Law", (2011). Encyclopedia of Paralel Programming,Springer Science+Business Media

Hansen, B. P., (1975) The Programming Language Concurrent Pascal. IEEE Transactions on Software Engineering 2: syf. 199-206

Herlihy, M, Moss, J. E. B., (1993) Transactional Memory: Architectural Support for Lock-free Data Structures. Proceedings of the 20th International Symposium on Computer Architecture: syf. 289-300

Hewitt, C. et. al (1973). *A Universal Modular Actor Formalism for Artificial Intelligence*. IJCAI.

Heywood, T.,Ranka, S., (1992) A practical hierarchical model of parallel computation. Journal
of Parallel and Distributed Computing, 16: syf. 212–249

Hoare, C. A. R. Monitors: An operating systems structuring concept. Communications of the ACM, 17(10): 549–557, 1974.

IBM Research,The Cell architecture, http://domino.research.ibm.com/comm /research.nsf/pages/r.arch.innovation.html Erişim tarihi 27 Mayıs 2013,

Karp, A. H., Flatt, H. P. (1990) Measuring Parallel Processor Performance. Communications of the ACM 33(5): syf. 539 - 543

Moore G (1965) Cramming More Components onto Integrated Circuits. Electronics 38(8), Available from Intel's homepage: ftp://download.intel.com/muse

um/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf . Erişim Tarihi 5 Mayıs 2013

Odersky, M. (2008), Scala's prehistory, http://www.scala-lang.org/node/239, Erişim tarihi 24 Haziran 2013

Odersky, M., Spoon, L.,Venners, B, (2010), Programming in Scala, Artima Press, İkinci Baskı, ABD

Pacheco, P. S., (2011) An Introduction to Parallel Programming , Morgan Kauffman Publishers, ABD

Rauber, T., Rünger, R., (2010) Parallel Programming For Multicore and Cluster Systems , Springer, İkinci Baskı, Almanya

Raymond, E. (1999) The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. O'Reilly, ABD

ScalaSTM, http://nbronson.github.io/scala-stm/syntax_cheat_sheet.html, Erişim Tarihi 25 Haziran 2013

Scott, M. L., (2005), Programming Language Pragmatics, Morgan Kauffman Publishers,İkinci Baskı, ABD

Steele, Jr., G. L., (1999) "Growing a Language." Higher-Order and Symbolic Computation, 12:221–223 , 1998'te OOPSALA'da yaptığı konuşmanın transkriptinden alınmıştır.

Sutter, H., Larus, J. (2005) Software and the concurrency revolution. ACM Queue, 3(7): 54–62,

Sutter, H. (2005) The Free Lunch is Over: A Fundamental Turn toward Concurrency

in Software. Dr. Dobb’s Journal 30(3)

Vajda, A., (2011) Programming Many-Core Chips, Springer, İngiltere

Van Roy, P., Haidi, S. (2004) Concepts, Techniques, and Models of Computer Programming , The MIT Press, ABD

Wampler, D., Payne, A. (2009) Programming Scala, Oreilly Press, ABD

Williams, M. (2003) Erlang Rationale, 2. ACM SIGPLAN Erlang Workshop, Uppsala,İsveç

ÖZGEÇMİŞ

1991 yılında İstanbul'da doğdu. İlk ve orta öğretimini İstanbul'da tamamladı. 2009 yılında başladığı Kocaeli Üniversitesi Bilgisayar Mühendisliği bölümündeki lisans öğrenimini hala sürdürmektedir.