Niğde Üniversitesi Mühendislik Bilimleri Dergisi, Cilt 3, Sayı 2, (2014), 1-17

FONKSİYONEL PROGRAMLAMA DİLLERİ İLE PARALEL PROGRAMLAMA

Uğurcan ERGÜN, Ahmet SAYAR*

Bilgisayar Mühendisliği Bölümü,Mühendislik Fakültesi, Umuttepe Kampüsü,Kocaeli Üniversitesi, 41380, Kocaeli,Türkiye

ÖZET

Günümüzde paralel sistemler büyük bir önem arz etmektedirler. Bu sistemler kullanılarak hesaplamalar daha hızlı ve verimli şekilde yapılabilmektedir. Ancak yazılımların da 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 paralel programlama modelleri ve teknikleri incelenmiş ve bazı fonksiyonel programlama dillerinde bu model ve tekniklerin nasıl gerçeklendiği ele alınmıştır.

Anahtar Kelimeler: Dağıtık Sistemler, Paralel hesaplama, Fonksiyonel Programlama, Erlang, Scala

PARALLEL PROGRAMMING WITH FUNCTIONAL PROGRAMMING LANGUAGES

ABSTRACT

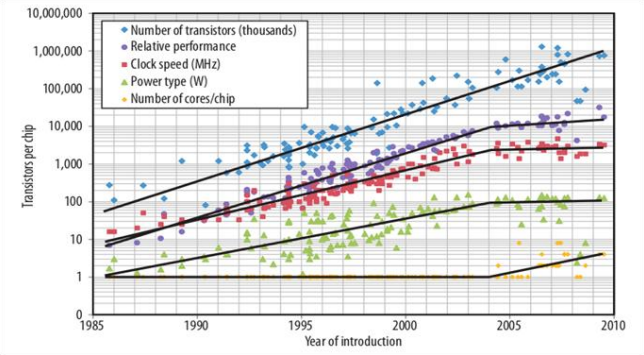
In the present day parallel systems has a significant importance. These systems could make computations faster and more efficent. However software has to be designed accordingly and using the common imperative paradigm it could be relatively harder and costs more. Because of coming with easier solutions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

Keywords: Distributed Systems, Parallel Computation, Functional Programming, Erlang, Scala

* Corresponding author. Tel.: +90 262 303 3580; Fax: +90 262 303 3003 e-mail: ahmet.sayar@kocaeli.edu.tr

1. GİRİŞ

Mikroişlemciler 1970'lerin başında ortaya çıktıklarından beri sürekli gelişmektedirler. Moore Yasası [1] entegre devrelerdeki transistor sayısının her iki yılda bir ikiye katlanacağını öngörmektedir. Artan transistor 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 [2]. Ancak Şekil 1'de [3] görüldüğü gibi transistor sayısı Moore'un öngördüğü 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. Mikroişlemcilerin tarihsel gelişimi

Bu geçişi destekleyen gözlemlerden biri işlemcinin mimarisinde yapılacak değişikliklerin veya eklenecek özelliklerin, kullanılan transistor sayısı ve gereken güce oranla en az doğrusal performans kazancı sağlaması gerektiğini yoksa uygulanmamaları gerektiğini söyleyen "KILL" (İng. Kill if Less Than Linear) kuralıdır [4]. Bu prensip günümüzde bazı çok çekirdekli işlemcilerin tasarımına öncülük etmektedir [5]. Donanımlara getirilen bu çoklu-çekirdek yaklaşımı beraberinde yazılımların tekrar değerlendirilmesi ihtiyacını getirdi. Çünkü bu yaklaşımda performans artışı sağlayabilmek için yazılımların bu mimariye 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 BİLGİSAYAR MİMARİLERİ

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.

i. 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 [6]. Modern bilgisayarlarda bu darboğazın etkisini azaltmak ve işlemcilerin daha hızlı çalışmalarını sağlamak için önbellekleme, sanal bellek, pipelining gibi belli yöntemler kullanılır [7].

ii. 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 [8] 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.

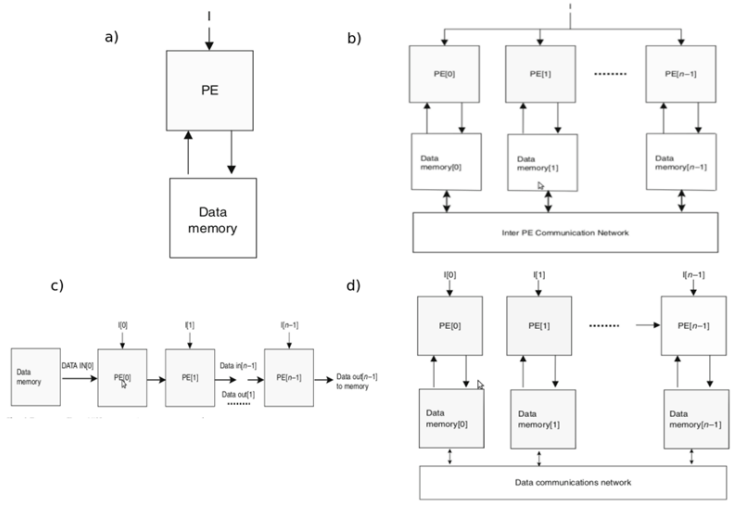
Tekil komut, çoğul veri akışı (SIMD) : Bu mimaride her biri kendi özel veri belleğine (bu bellek dağıtık veya paylaşımlı olabilir) sahip birden fazla işlem birimi bulunur. Ancak ortada tek bir program belleği vardır. Özel bir işlemci bu komutu okur ve diğer birimlere yönlendirir. Her adımda işlem birimleri 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) .

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 OpenGL gibi genel amaçlı grafik işlemci (GPGPU) teknolojileriyle grafik işlemciler üzerinde paralel programlama yapmak mümkündür ve oldukça revaçtadır.

Çoğul komut, çoğul veri akışı (MIMD) : Bu mimari birden fazla herhangi bir türden işlemci ve bu işlemciler arası bağlantıdan oluşur. İşlemciler birbirlerinden bağımsız olsalar da bir problemi çözmek için 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'de yukarıda bahsedilen mimarilerin görsel temsilleri yer almaktadır. Bu grafikte a) SISD b) SIMD c) MISD d) MIMD mimarilerini temsil etmektedir [9].

iii. *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 erişebilir. 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.



Şekil 2. Flynn taksonomisinde paralel mimariler

3. PARALELLİĞİN TEMEL KANUNLARI

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 günümüze kadar araştırmaları ve pratik uygulamaları etkileyen kanunları tanıtılmaya çalışılacaktır. Bu kanunlar modern bilgisayar mimarilerine yol göstermekte olup bilimsel araştırmalar için temel bir çatı sunmaktadır [5].

i. ***Amdahl Yasası:*** Amdahl Yasası [10] verilen bir uygulama için teorik olarak en fazla ne kadar paralel performans 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. 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.

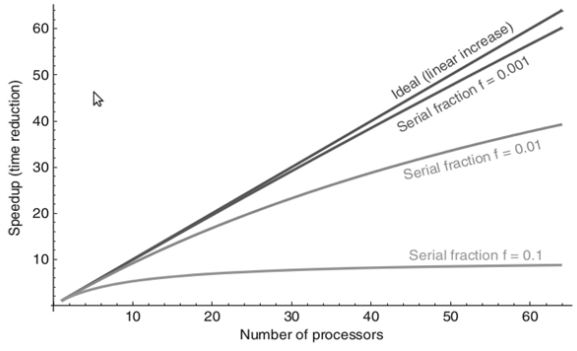
Eğer f kadarı seri olan bir problem n tane işlemcide çalışıyorsa maksimum performans kazancı *pk*

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

olacaktır. Hatta bu sistemde sonsuz adet işlemci olduğunu düşünürsek

$$pk = 1 / f$$

olacaktır. Yasaya göre performans artışının nasıl değiştiği Şekil 3 'de görülebilir.



Şekil 3. 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 [5].

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 [11].

ii. 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 günümüz f bu değerin çok çok altındadır. Maksimum performans kazancı genelde problemin büyüklüğü ile orantılıdır. Gustafson yasasının [12] 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.

iii. Gunther Yasası: Gunther yasası [13] 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.

iv. Karp-Flatt Ölçütü: Karp-Flatt ölçütü [14] bir uygulamanın kullanılan işlemci sayısına göre performansını ölçmenin pratik yollarından biridir. Bir uygulamanın performans artışına bakarak seri kısmının 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 olmasa da Karp-Flatt ölçütü paralelliğe dair temel ölçümlerden biridir ve bu alanda uygulanabilirliği de oldukça geniştir [5].

4. PARALEL PROGRAMLAMA MODELLERİ

Paralel programlamanın genel prensipleri üzerine çalışabilmek için paralel mimariler genelde bazı özellikleri değerlendirilerek daha soyut şekillerde düşünülürler. Bunu yapmanın sistematik yollarından 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 [15]. Bunlar makine modeli, mimari model, hesaplama modeli ve programlama modelidir.

Programlama modeli bir paralel hesaplama sistemini programlama dilleri ve kavramlarıyla açıklamaya çalışır

[16]. 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.

- Paralelliğin hangi seviyede uygulandığı
- Paralellik temsili: açık mı yoksa örtülü 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 [9].

i. Paralelliğin hangi seviyede uygulandığı: Çeşitli seviyelerde paralellik uygulamaları görülmüştür. Bunların başlıcaları: Komut Seviyesinde paralellik, veri paralelliği ve görev (fonksiyon) paralelliği.

Komut Seviyesinde paralellikte bir programa ait birden fazla komut aynı anda paralel bir şekilde işletilebilir. Ancak komutlar arasında bir veri bağımlılığının varlığı paralelliğe engel olacaktır. Veri bağımlılığı ise 3 tipte gerçekleşebilir, akış bağımlılığı, karşı bağımlılık ve çıkış bağımlılığı. Eğer i_1 komutu daha sonra i_2 komutunda ihtiyaç duyulacak bir değer hesaplıyorsa i_1 ve i_2 arasında akış bağımlılığı var denir. Eğer i_1 daha sonra i_2'nin hesaplama sonucunu yazacağı bir değişkeni kullanıyorsa i_1 ve i_2 arasında karşı bağımlılık var denir. Eğer i_1 ve i_2 aynı değişkeni hesaplama sonuçlarını tutmak için kullanıyorlarsa i_1 ve i_2 arasında çıkış bağımlılığı var denir.

Veri seviyesinde paralellik ise büyük bir veri yapısının farklı elemanlarına farklı işlemler uygulanması durumda ortaya çıkabilir. 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.

Görev paralelliğinde ise, eğer h(x) fonksiyonu f(x) ve g(x) gibi farklı fonksiyonların toplamı şeklinde yazabiliyorsa, h(x) bulmak için f(x) ve g(x) fonksiyonları paralel olarak işletilebilir. Görev paralelliğini kullanabilmek için görevlerin ve aralarındaki bağımlılıkların bilinmesi gerekir. Bunlar bir grafik (çizge) ile gösterilebilir [9]. Statik ve dinamik olarak iki çeşidi vardır. Clik, Intel Thread Building Blocks, OpenMP, Pthreads gibi yaygın çözümler bu tekniği kullanmaktadır [5].

ii. Paralellik temsili: açık mı yoksa örtülü mü olduğu: 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. Kısmi örtülü paralellikte ise programın paralel kısmının açık bir şekilde belirtilmesine karşın programın proses ve threadlere dağıtımı ve atanması derleyici tarafından yapılır. Örtülü paralelliğe örnek olarak SISAL, Paralel Haskell, Verilog ve VHDL verilebilir. OpenMP kütüphanesi bu modeli kullanır. Açık paralellik ise MPI ve Pthreads gibi araçlarca kullanılmaktadır.

iii. Paralel program parçalarının tanımlanması: 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 karakteristik bulunur. Thread lerin user yada kernel level olması, sistem threadleri ile user threadlerin eşleşme algoritmaları paralel programlamada kullanılacak yaklaşımları etkilemektedir [9].

iv. İşlem birimlerinin haberleşmeleri: 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 platformun kullanıldığı ile belirlenir. İki tür paralel platform vardır. Bunlar, bellek paylaşımlı model ve mesaj iletimli model.

Bellek paylaşımlı modelde, 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. Ancak bellek paylaşımlı modelin iki temel problemi vardır. Bellek içeriğinin tutarlılığının sağlanması için belleğe erişim kontrol edilmeli ve bütün threadlerin senkronize olmaları sağlanmalıdır. Bunu çözmek için, geleneksel olarak kilit ve semaforlar kullanılır.

Mesaj iletimli paradigma 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.

v. Paralel birimler arasında senkronizasyon: Senkronizasyonda kullanılan mekanizmalar genellikle şu şekilde özetlenebilir: Kilitler, semaforlar, koşul değişkenkeri, monitörler ve son zamanlarda ortaya atılan işlemsel bellek [17].

Kilitleri kullanmanın olağan yolu onları birbirleriyle ilişkili kaynakları korumak için kullanmaktır. Böylece bu kaynaklara ulaşmak isteyen her threadin önce kilidin sahibi olması gerekir. Kilitlerin kullanımı yarış durumlarını ve determenistik olmayan davranışları önleyebilseler de dikkatli bir şekilde kullanılmadığında deadlock, livelock gibi sorunlara yol açabilmektedirler. 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.

Semaforlar, kilitlerin genelleştirmiş hali olarak görülebilir ilk defa Dijkstra tarafından önerilmişlerdir. En basit ikili semaforlar kilitlerle aynı işlevi görürler. Threade izin verebilir veya bloklayabilirler. Sayaç semaforları ise N tane threade izin verebilirler. 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.

Koşul değişkenleri işletim sistemi mekanizmaları olarak Hoare ve Hansen tarafından ortaya atılmıştır [18, 19]. 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 [18] ise birden fazla thread tarafından güvenle kullanabilecek nesneler yaratmaya yarayan programlama dili seviyesinde bir mekanizmadır. Monitör nesnelerinin metotları 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.

İşlemsel Bellek: Senkronizasyon mekanizması olarak kilitlerin kullanılması kritik kısımların serileştirilmesini getirmektedir. Bu performans kayıplarını beraberinde getirmekte ve hatta büyük sistemlerde darboğazlara sebep olmaktadır. 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 [20].

Kilit mekanizmalarına bir alternatif olarak işlemsel bellek [21] önerilmiştir. İşlemsel bellek, belleğe eş zamanlı erişim sırasında kilitlerin yol açtığı bazı sorunları çözmeyi amaçlamaktadır. Donanım veya yazılım seviyesinde gerçeklenebilmektedir. 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.

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 vadeden bir yaklaşım olmasına karşın hala bir aktif araştırma konusu olduğu ve mevcut uygulamaların henüz tam olgunlaşmadığı unutulmamalıdır.

5. FONKSİYONEL 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 ve kontrol yapıları – atlamalar eşleştirilebilir.

Backus'a göre [6] 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 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 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 belirtilir 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 [22] 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.

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 döndürülebilen 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. İç 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 alabilen 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, programlama 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 çağırsa 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, paralelleştirme 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. Tekrarlı 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.

5.1. Paralel Hesaplamaya Olası Katkıları

Performans artırmaya yönelik olarak geliştirilen donanımsal yöntemlerde işlemcilerin saat hızını artırmak aşırı güç kullanımına sebep olduğundan son yıllarda araştırmacıları yazılımsal çözüm bulma arayışlarına itmiştir. Bu anlamda akla gelen ilk yaklaşım cluster hesaplama yada paralel hesaplama yaklaşımlarıdır. Bu tür yaklaşımlarda, bir iş fiziksel olarak alt parçalarına ayrılır. Bu alt parçalar farklı makinelera atanabileceği gibi, bir entegre üzerineki birden fazla işlem birimine de atanabilir (çok çekirdekli makinalar). 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. Ayrıca paralel sistemlerin yaygınlaşmasıyla beraber fonksiyonel dillerin bu alanda kullanılasına karşı ilgi de artmaya başladı. Bu ilginin nedenının başlıcaları aşağıdaki şekilde özetlenmeye çalışılmıştır.

Imperatif dillerde (C, Java vb.) veriler değişkenler üzerine atanır. Paralel hesaplamada bu verilerin birden fazla proses tarafından güncellenmesi yada okunması istenebilir. İstenen sonucun çıkması ise, paralel proseslerin belirli bir sırada veriye erişimine bağlıdır. Prosesler asenkron çalıştıkları için, istenen sırada çalışmaları çeşitli senkronizasyon tekniklerinin kullanılması ile sağlanır. Senkronizasyon işlemi ise kilit ve semaphore gibi yazılımsal olarak çözülebilmekte ancak bunlar da performans kayıplarına yol açmaktadır.

Bunların yanında, imperatif dillerde çalışan kodun debug edilmesi ve hata ayıklanması da çok zordur. Birçok proses ve threadin aynı değişkeni okuma ve yazması yarış durumu (race condition) olarak adlandırılır ve programcılar için hata ayıklamada zorluk çıkarmaktadır.

Diğer yandan fonksiyonel programlarda ise, proseslerin çalışma sırası sonuç üzerinde etkili değildir. Yani aynı ifadenin farklı kısımları her zaman paralel olarak çalıştırılabilir. Bunun nedeni değişken mantığının kullanılmaması ve değerlerin değişmesinin engellenmesidir. Bu şekilde, bellek paylaşımlı sistemlerdeki senkronizasyon problemi ortadan kalktığı için paralel programları yazmak çok daha kolay olmaktadır.

Bu genel olası katkıların yanında, her fonksiyonel programlama dilinin kendi özelliklerine bağlı olarak da ek bazı katkıları olabilir. Bunlar ise takip eden bölümlerde uygulama durumları olarak Erlang ve Scala dilleri için analiz edilmiştir.

6. ÖRNEK PROGRAMLAMA DİLİ: ERLANG

Erlang, 1986 yılında Ericsson tarafından geliştirilmiş [23] bir fonksiyonel programlama dilidir. Adını hem İngilizce Ericsson dilinden (ERicsson LANGauge), hem de günümüz telekomünikasyon 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 araştırması 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 gördüy selerde 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 [24]. 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 telekomünikasyon 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 telekomünikasyon 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 [24].

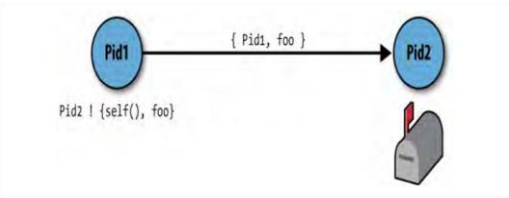
Bir programlama dili olarak Erlang'ın karakteristikleri şöyle açıklanabilir. Erlang saf olmayan bir fonksiyonel programlama dilidir. Dinamik bir dildir ve katı işletim stratejisini kullanır. Her değişkene tek bir defa değer atanabilir. Bundan gayrı atama operatörü örüntü bulma (pattern matching) işlemini 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 (sanal makina) 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 [25].

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" [26]. 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.

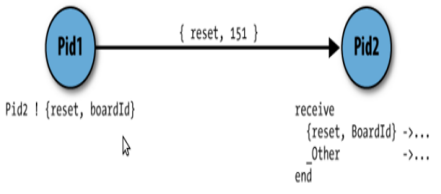
6.1. Erlang ile Paralel Programlama

Her şeyden 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 prosesin process tanımlayıcısını döndürür. Erlang'da prosess tanımlayıcıları şuna benzer <0.30.0>. *self()* bir prosesin kendi Pid'sini (proses no) barındırır.



Şekil 4. Erlang'da mesaj iletimi



Şekil 5. Erlang'ta alınan mesajların işlenmesi

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 4'de [24] gösterilmiştir.

receive ... end : komutu prosese gelen mesajları işlemek için kullanılır. Kullanımı şu şekildedir. Gelen mesajlar verilen örüntülerle eşleştiği zaman eşleştiği örüntünün ifadeleri işletilir. After ifadesiyle timeout tanımlanabilir. Şekil 5'de gönderilen mesajların nasıl işlendiği gösterilmiştir.

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

Burada anlatılan temel işlemlerin kod içerisinde nasıl kullanabildiği Şekil 6 [23] 'da görülebilir.

Her zaman Pid'lerle çalışmak uygun olmayabilir Erlang'da Pid'leri 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.

```
-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. Erlang mesaj iletimi örneği

Yukarıda proseslerin birbirleriyle ilişkilendirilebildiğinden bahsedilmişti. Bu, dil içerisinde şu ifadelerle yapılır.

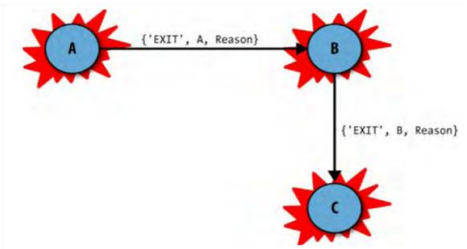
- *link(Pid):* Çağrıyı yapan processle Pidsi verilen process arasında çift yönlü bir bağ oluşturur.
- *unlink(Pid):* Çağrıyı yapan processle Pidsi verilen process arasındaki bağı kaldırır.
- *spawn_link(Fun):* Çağrıyı yapan process yeni bir process yaratır ve onu kendine bağlar.

Bağlanmış processler olası hatalara karşı birbirlerini gözlemeye başlarlar. Eğer bu proseslerden biri çakılırsa

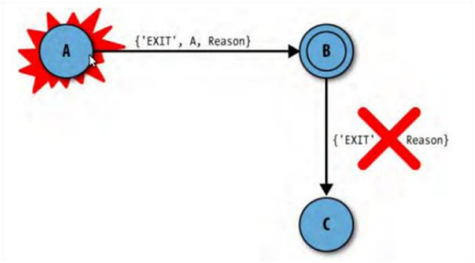
diğerine veya diğerlerine exit sinyali adı verilen bir sinyal gönderir. Exit sinyali aşağıdaki gibi elle de gönderilebilir.

- *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 7' de görülebilir.



Şekil 7. Exit sinyalinin yayılması



Şekil 8. Erlang'da hata yakalama

Birbirine bağımlı processleri birbirine bağlamak önemlidir, böylece bir çökme yaşandığında processlerin hepsinin birden çöktüğünden emin olunur. Elbette bir process exit sinyali aldığında tek yapabileceği kendisi de ölmek değildir. Hataları yakalamak da mümkündür. Bir hata yakalandığında 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.

Erlang'da hata yakalamak için 3 temel yaklaşım vardır: (1) Eğer yaratılan processin ölüp ölmeyeceği ile ilgilenilmiyorsa *spawn* metodu, (2) Eğer yaratılan processin ölmesi durumunda mevcut processin de ölmesi gerekiyorsa *spawn_link, (3)* Eğer yaratılan processin ölmesi durumunda oluşan hata yakalanmak isteniyorsa *spawn_link* ve *trap_exit* kullanılması gerekir.

7. ÖRNEK PROGRAMLAMA DİLİ: SCALA

Scala, 2001 yılında EPFL'de Martin Odersky tarafından tasarlanmaya başlanmış ilk sürümünü 2003 yılında çıkartmış bir programlama dilidir [27]. 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 [28] katedral ve pazar yapılarını açık kaynaklı yazılım geliştirme metodolojisini anlatmak için benzetme olarak kullanmıştır. Burada Katedral, yapımı uzun zaman alan ancak uzun süreler boyunca değişmeden kalan mükemmele yakın bir yapı iken pazar orada çalışan insanlar tarafından devamlı, uyarlanan ve geliştirilen yapıları ifade etmektedir. Guy Steele [29] 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 mükemmel 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 sözdizimi (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 [30]. 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 çabukça 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. Scala statik dillerin iki temel problemine 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 ilkel (primitive) 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 metotları 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 fazla paralel programlama yapısı desteklemektedir. Java uyumluluğu sayesinde java.util.concurent kütüphanesiyle standart Java'da bulunan bellek paylaşım model kullanılabilir. Mesaj iletimli aktör modeli [31] 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 9'da [32] 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 9. Scala aktör tanımı (nesnel)

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

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

Şekil 10. Scala aktör tanımı (fonksiyonel)

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

Şekil 10'daki [32] 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 11. Scala'da alınan mesajların işlenmesi

Şekil 11'de [32] 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 12'de gösterilmiştir. Paylaşılacak değişkenler Ref 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.

```
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 12. Scala ile işlemsel bellek kullanımı

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.

9. KAYNAKLAR

[1] MOORE, G. E., "Cramming more components onto integrated circuits", Proceedings of the IEEE, 86, 82-85, 1998.

[2] SUTTER, H., "The free lunch is over: A fundamental turn toward concurrency in software", Dr. Dobb's Journal, 30, 202-210, 2005.

[3] FULLER, S. H. and MILLET, L. I., "Computing performance: Game over or next level", Computer, 44, 31-38, 2011.

[4] AGARWAL, A. and LEVY, M., "The kill rule for multicore", In Proc. IEEE/ACM Design Automation Conf. (DAC), 750–753, San Diego, CA, USA, 2007.

[5] VAJDA, A. s. Programming many-core chips. Springer, Jorvas, Finland, 2011.

[6] BACKUS, J., "Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs", Communications of the ACM, 21, 613-641, 1978.

[7] PACHECO, P., An introduction to parallel programming. Access Online via Elsevier, Burlington, MA, USA, 2011.

[8] FLYNN, M. J., "Some computer organizations and their effectiveness", IEEE Transactions on Computers, 100, 948-960, 1972.

[9] RAUBER, T. and RUNGER, G., Parallel programming: For multicore and cluster systems. Springer, Berlin, Hiedelberg Germany, 2010.

[10] AMDAHL, G. M., Validity of the single processor approach to achieving large scale computing capabilities. ACM, City, 1967.

[11] DOWD, K., SEVERANCE, C. R. and LOUKIDES, M. K., High performance computing. O'Reilly, 1998.

[12] GUSTAFSON, J. L., "Reevaluating Amdahl's law", Communications of the ACM, 31, 532-533, 1988.

[13] GUNTHER, N. J., A New Interpretation of Amdahl's Law and Geometric Scalability. arXiv preprint cs/02100172002.

[14] KARP, A. H. and FLATT, H. P., "Measuring parallel processor performance", Communications of the ACM, 33, 539-543, 1990.

[15] HEYWOOD, T. and RANKA, S., "A practical hierarchical model of parallel computation I", The model. Journal of Parallel and Distributed Computing, 16, 212-232, 1992.

[16] PANDEYy, A. K., Programming languages: principles and paradigms. Alpha Science International, Springer, İngiltere, 2010.

[17] VAN R. P. and HARIDI, S., Concepts, techniques, and models of computer programming. The MIT Press, USA, 2004.

[18] HOARE, C. A. R., Monitors: An Operating System Structuring Concept. Springer, City, 2002.

[19] HANSEN, P. B., "The programming language concurrent Pascal", IEEE Transactions on Software Engineering, 2, 199-207, 1975.

[20] SUTTER, H. and LARUS, J., "Software and the concurrency revolution", Queue, 3, 54-62, 2005.

[21] HERLIHY, M. and MOSS, J. E. B., "Transactional memory: Architectural support for lock-free data structures", ACM, 1993.

[22] CHURCH, A., The calculi of lambda-conversion, Volume 6 of Annals of Mathematics Studies. Princeton University Press, Princeton, USA, 1941.

[23] ARMSTRONG, J., A history of Erlang. ACM, City, 2007.

[24] CESARINI, F. and THOMPSON, S., Erlang programming. O'Reilly Media, Inc., Sebastopol, CA, USA, 2009.

[25] WILLIAMS, R., OCHSENBEIN, F., DAVENHALL, C., DURAND, D., FERNIQUE, P., GIARETTA, D., HANISCH, R., MCGLYNN, T., SZALAY, A. and WICENEC, A., VOTable: "A Proposed XML Format for Astronomical Tables", Standard Specification, US National Virtual Observatory, 2002.

[26] ARMSTRONG, J., "Erlang", Communications of the ACM, 53, 68-75, 2007.

[27] ODERSKY, M., SPOON, L. and VENNERS, B., Programming in Scala: a comprehensive step-by-step guide. Artima Inc, 2008.

[28] RAYMOND, E. S., The Cathedral & the Bazaar: Musings on linux and open source by an accidental revolutionary. O'Reilly, Sebastopol, CA, USA, 2008.

[29] STEELE, G. L., "Growing a language", Higher-Order and Symbolic Computation, 12, 221-236, 1999.

[30] ODERSKY, M., The Scala experiment: can we provide better language support for component systems? ACM, City, 2006.

[31] HEWITT, C., BISHOP, P. and STEIGER, R., A universal modular actor formalism for artificial intelligence. Morgan Kaufmann Publishers Inc., City, 1973.

[32] WAMPLER, D. and PAYNE, A., Programming Scala: Scalability= Functional Programming+ Objects. O'Reilly Media, Sebastopol, CA, USA, 2009.