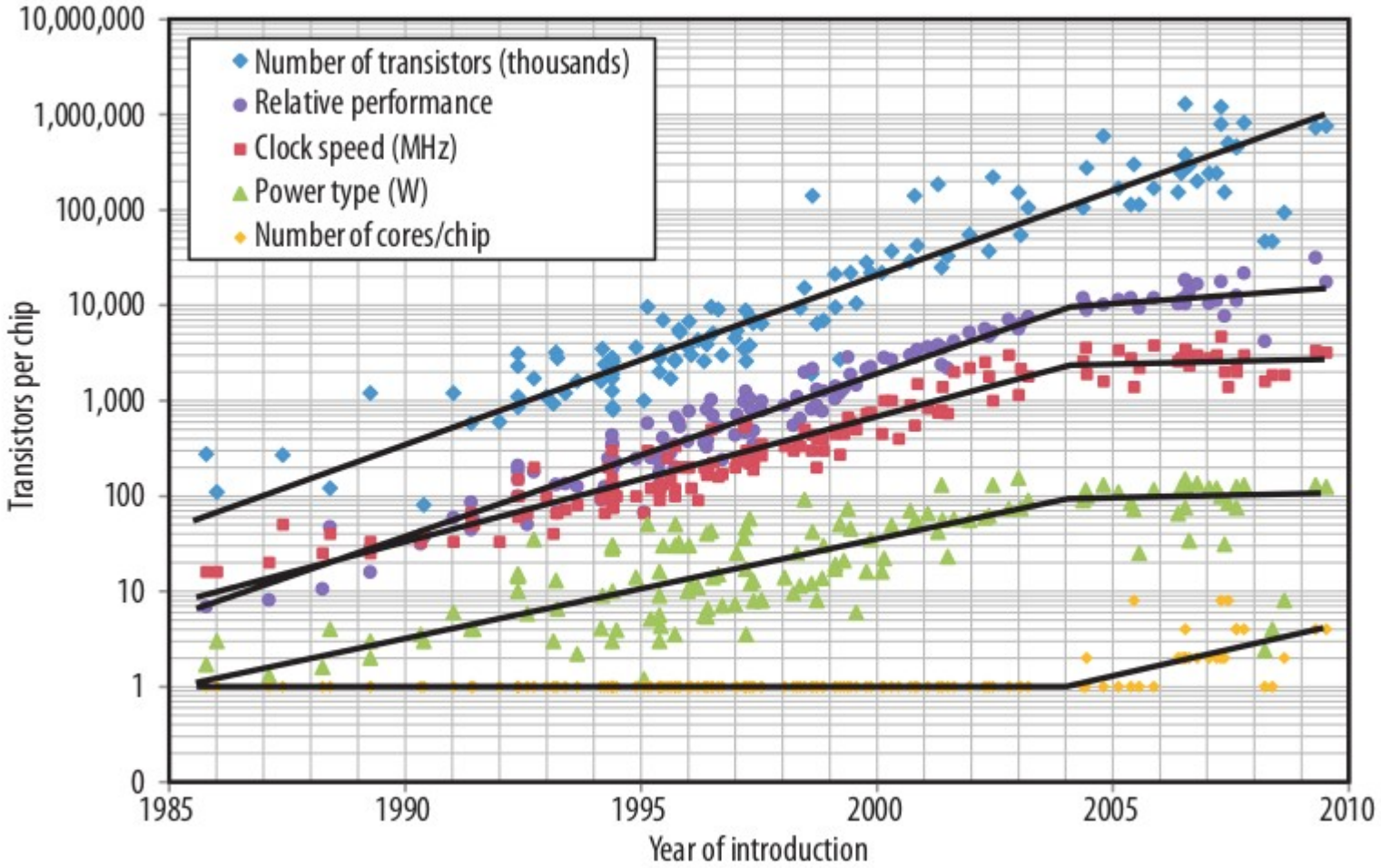
Fonksiyonel Dillerle Paralel Programlama

Uğurcan Ergün 090202003

Evren Bir Zamanlar Gaz ve Toz Bulutuydu

- Mikroişlemciler 70'lerde icat edildiklerinden beri sürekli gelişmektedirler.
- İşlemcilerdeki transistör sayısı her 18 ayda bir ikiye katlanmaktadır. (Moore yasası)
- İşlemci saat hızları da artmaktaydı.
- Yazılım değiştirilmeden yeni donanım kullanılarak performans artışı sağlamak mümkün
- Bedava öğle yemeği

İşlemci Parametrelerinin Zamanla Değişimi



Bedava Yemeğin Sonu

- İşlemci saat hızları 2004'ten sonra pek artamamıştır.
- Saat hızlarında yapılan artışlar güç tüketimini ciddi oranda arttırmaya başlamıştır.
- İşlemci tasarımında eğer bir değişiklik kullanılan transistör sayısı ve gereken güce oranla en az lineer performans kazancı sağlamıyorsa o değişiklik yapılmamalıdır. (KILL kuralı)

Paralel Sistemler İmdada Yetişiyor

- Saat hızı arttırmak maliyetli olmasına rağmen transistör sayısı artışı hala geçerli.
- Donanım üreticileri bu sorunu bir entegreye birden fazla işlem birimi koyarak çözmeye çalışmıştır.
- Ancak performans kazancı lineerden düşük.
- Mevcut yazılımlar bu sistemlerle uyumlu olmayabilir.

Dokuz Kadın ve Bir Çocuk

- Yeni çok çekirdekli işlemciler == paralel sistemler
- Eldeki problem parçalara ayırılamıyorsa fazladan işlemci kullanmak anlamlı değil
- Ortaya çıkan birden fazla işlem birimini ortaklaşa ve verimli bir şekilde kullanma ihtiyacı oluştu paralel hesaplama teknikleri kullanılmalı

Von Neumann Mimarisi

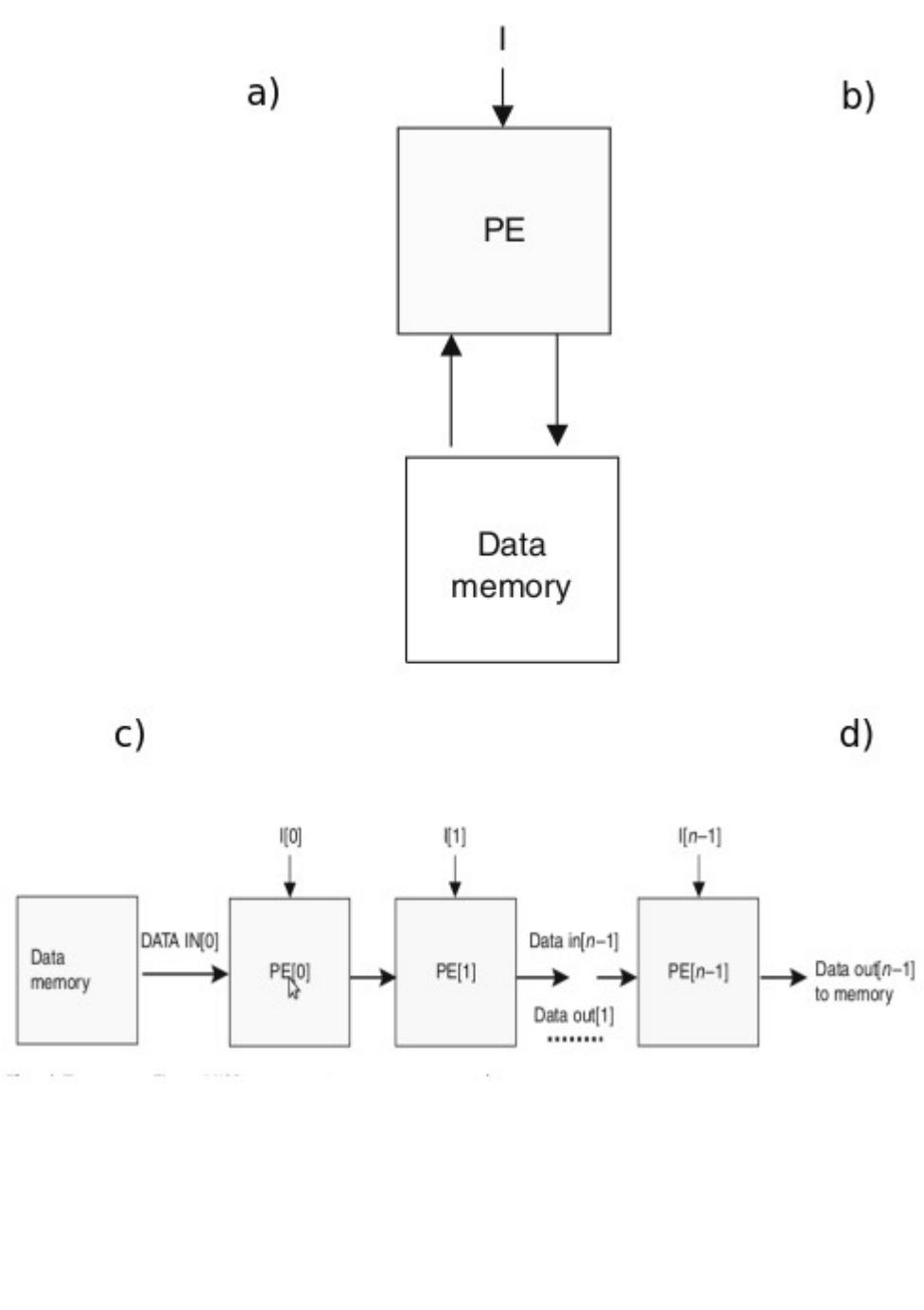
- Klasik Von Neumann mimarisi bir işlemci, ana bellek ve bu unsurları birleştiren bir veriyolundan oluşur.
- Bu durumda veriyolunun hızı okunabilecek en fazla veri veya komut sayısını belirler. Buna Von Neumann darboğazı adı da verilir.
- Olası çözümler: önbellekleme, sanal bellek, pipelining

Paralel Sistemlerin Sınıflandırılması (Flynn Taksonomisi)

- Paralel sistemler çok çeşitli olup 4 gruba ayrılabilir
- SISD: Tek bir program ve tek bir veri seti vardır. Ör. Klasik Von Neumann mimarisi
- SIMD: Her birimin kendi veri seti var ancak ortak bir program kullanılır. Multimedya ve grafik işlemede çok verimli olabilir. Örnek dizi işlemciler, Sony PS3

Paralel Sistemlerin Sınıflandırılması (Flynn Taksonomisi)

- MISD: Her birim tekil bir veri setinde kendilerine ait olan programı işletir. Ör. Modern grafik işlemcileri
- MIMD: Ortak kullanılan birden fazla bağımsız işlemci ve bunların bağlantılarını içerir. Ör. Modern çok çekirdekli işlemciler, cluster sistemler

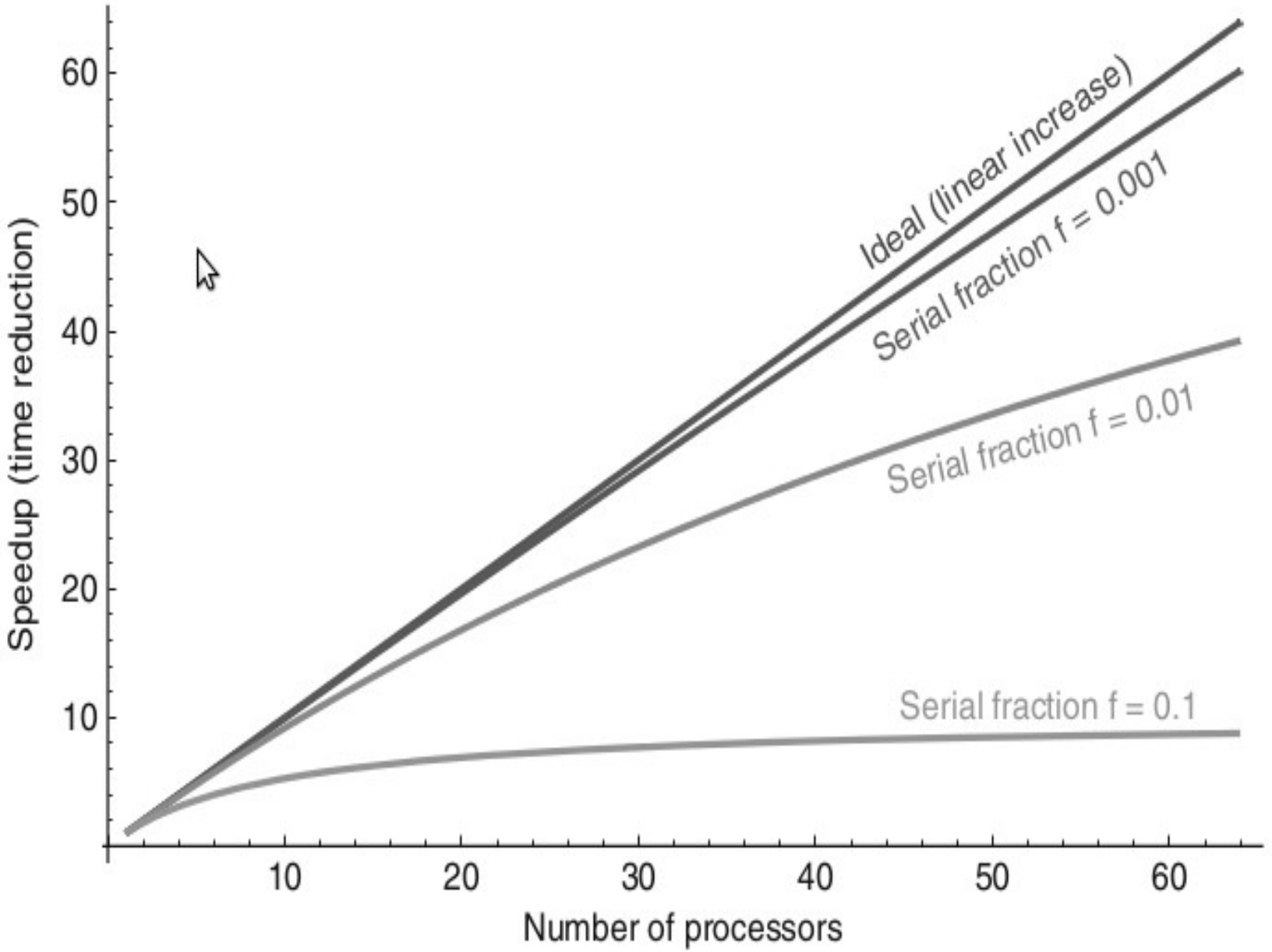

a)
I
PE
Data memory
b)
I
PE[0]
PE[1]
PE[n−1]
Data memory[0]
Data memory[1]
Data memory[n−1]
Inter PE Communication Network
c)
I[0]
I[1]
I[n−1]
Data memory
DATA IN[0]
PE[0]
PE[1]
Data in[n−1]
Data out[1]
PE[n−1]
Data out[n−1] to memory
d)
I[0]
I[1]
I[n−1]
PE[0]
PE[1]
PE[n−1]
Data memory[0]
Data memory[1]
Data memory[n−1]
Data communications network

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 kanunlar modern bilgisayar mimarilerine yol göstermekte olup bilimsel araştırma için temel bir çatı sunmaktadır.

Amdahl yasası

- Bir program paralel olarak işletildiğinde elde edilecek performans kazancı programdaki seri kısmın büyüklüğüne bağlıdır.
- Bu yasa 1967 yılında Gene Amdahl tarafından paralel hesaplamanın tartışıldığı bir konferansta paralel hesaplamaya karşı argüman olarak sunulmuştur


Speedup (time reduction)
60
50
40
30
20
10
10
20
30
40
50
60
Number of processors
Ideal (linear increase)
Serial fraction f = 0.001
Serial fraction f = 0.01
Serial fraction f = 0.1

- Kanunda iki temel hatalı varsayım var
- 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.

Gustafson Yasası

- Amdahl yasası problemlerin boyutunu sabit kabul ederek yanlış bir varsayımda bulunmuştur.
- Problemin boyutu büyüdüğünde paralel kısım daha hızlı büyüdüğünden f küçülür.
- Amdahl asıl olarak f'i 0.25 ila 0.4 olarak tahmin etmiştir günümüz f bu değerin çok çok altındadır.

Paralel Programlama Modelleri

- 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

  bir paralel programlama modelinin yapısını belirler

Paralellik seviyeleri

- İşletilecek komutlar arasında bağımlılık var mı varsa hangi türde ?
- Veriler birbirlerinden bağımsızlarsa veri seti dağıtılabilir. (Veri paralelliği)
- Eğer yapılacak işlem alt işlemlerin toplamı gibi tanımlanabiliyorsa veri setine dokunulmadan farklı işlemler aynı veri setine uygulanabilir. (Fonksiyonel paralellik)

Paralleliğin Temsili

- Paralleştirme kim tarafından sağlanacak ?
- Bilgisayar seri yazılmış bir kodu paralele çevirse buna örtülü paralellik denir. Ancak kodun içerisindeki değerlerin arasındaki ilişkileri inceleyip, paralel model üretmek çok karmaşık.
- Programın içerisinde paralel kısımlar belirtilerek derleyiciye yardım edilebilir (Kısmi örtülü paralellik)
- Programcı iletişim ve senkronizasyon işlemleri dahil olmak üzere paralel işletimin bütün detaylarını açık bir şekilde belirtebilir (Açık paralellik)

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.
- Uygulama seviyesinde paralellik uygulanmak istediğinde işletim sisteminde iki temel kavramdan bahsedilebilir, process ve threadler. Bu kavramlar kontol akışlarına dair soyutlamalardır.
- Temel paralel birim olarak bunlardan hangisi kullanılacak ?
- Thread kullanılacaksa hangi işletim modeli kullanılacak?
- N:1 ?, 1:1 ? , N:M ?

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.

Bellek Paylaşımlı Model

- Bellek paylaşımlı paradigma, threadler arası iletişim için en çok kullanılan ve anlaması en kolay olan yöntemdir.
- İletişime 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 edilmelidir
- Bütün threadlerin senkronize olmaları sağlanmalıdır.

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.

Bellek Paylaşımlı ve Mesaj İletimli Model Karşılaştırması

- Paylaşılacak verinin büyük olması durumunda kullanılır.
- Senkronizasyon gerektirir.
- Yarış durumları mutlaka engellenmelidir.

- Mesaj ile gönderilecek verinin büyük olmaması gerekir aksi takdirde overhead meydana gelir.
- Yarış durumları mümkün değil
- Daha güvenli

Senkronizasyon: Kilitler

- Bir kilit temelde aynı anda pek az (genelde sadece bir) threadin sahip olabileceği bir kaynaktır.
- Acquire,release ve test (opsiyonel) metodları bulunur.
- Kilitleri kullanmanın olağan yolu onları birbirleriyle ilişkili kaynakları korumak için kullanmaktır.

- Özel kilit şekilleri mevcuttur: Spinlock, Okuma/Yazma kilidi
- Kilitlerin kullanımı yarış durumlarını ve determenistik olmayan davranışları önleyebilseler de dikkatli bir şekilde kullanılmadığında deadlock,livelock gibi sorunlara yol açabilmektedirler.

Semaforlar

- Semaforlar, kilitlerin genelleştirmiş hali olarak görülebilir ilk defa Djisktra tarafından önerilmişlerdir.
- En basit ikili semaforlar kilitlerle eşdeğerdirler
- Sayaç semaforları ise N tane threade izin verebilirler.
- Sayaç semaforları basit bir tamsayı sayaçtan oluşur. Ve iki atomik operasyona sahiptirler, P(s) ve V(s) (wait(s) ve signal(s))

Koşul Değişkenleri ve Monitörler

- Koşul değişkenleri işletim sistemi mekanizmaları olarak Hoare ve Hansen tarafından ortaya atılmıştır.
- Temelde threadlerin koşullar üzerinde bekletilebilmelerini sağlar. Bir koşul değişkeni bir koşulla ilişkilendirilir.
- 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.

İşlemsel Bellek

- Senkronizasyon mekanizması olarak kilitlerin kullanılması kritik kısımların serileştirilmesini getirmektedir.
- Bu performans kayıplarını beraberinde getirmekte ve hatta büyük sistemlerde darboğazlara sebep olmaktadır.
- Kilitlerle programlama yapmak oldukça hataya açık ve düşük seviyeli olmaktadır.
- Kilit mekanizmalarına bir alternatif olarak işlemsel bellek (Herlihy,Moss,1993) önerilmiştir. İ
- 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.

Fonksiyonel Programlama

- İmperatif diller doğrudan Von Neumann mimarisini temel alarak tasarlanmışlardır.
- İmperatif dillerdeki temel konseptlerin donanım işlemleriyle oldukça paralel olduğu görülebilir.
- Sabit olmayan değişkenler – hafıza hücreleri,
- Dereference (C'deki "*" operatörünün gerçekleştirdiği) işlemi – Bellekten veri yükleme
- Atama – bellekte depolama
- Kontrol yapıları – atlamalar

- Backus'a göre 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 soyutlamaların tasarlanabilmesi için teknikler geliştirmek gerekiyordu.
- Aynı zamanda bu yapılar üzerinde mantık yürütebilmek için teorilere ihtiyaç olacaktır.
- Matematikte bir teori belli veri tiplerinden,bu tipler üzerinde tanımlı operasyonlardan ve operasyon ve değer arasındaki ilişkileri tanımlayan kanunlardan oluşur.

- Önemli bir problem var.
- Matematiksel teorileri kullanarak yüksek seviyeli yapılar tanımlamak istiyoruz ancak elimizde matematikte bulunmayan bir işlem var. (Atama,değişim,mutation)
- Bu işlem teorilerdeki kanunları geçersiz kılabiliyor.
- Yeni bir yönteme gidilmesi gerekebilir. Fonksiyonel programlama bu duruma karşın önerilen temel paradigmalardan biridir

- Hesaplamaların matematiksel fonksiyonlarla ifadesine dayanır.
- Kökenini lambda-calculus'ten alır.
- Değişim, atama gibi işlemlere izin vermeyen, imperatif kontrol yapılarına sahip olmayan ve fonksiyonları yan etki barındırmayan fonksiyonel dillere saf fonksiyonel diller denilir.
- Fonksiyonel diller, imperatif dillere kıyasla daha basit syntax,semantic ve daha fazla esneklik sağlamasına rağmen işletilmeleri bu dillere kıyasla daha verimsizdir.

.

Saf ve Saf Olmayan Fonksiyonel Diller

Saf Fonk. Diller

- Haskell (Bazı monadları çıkartıldığında)
- Xpath
- XSLT

Saf Olmayan Fonk. Diller

- Lisp (Racket,Clojure)
- Erlang
- Scala
- SML
- Ocaml
- F#

Temel Özellikleri

- Birinci Sınıf Fonksiyonlar
- Yan etkiler
- Yüksek Mertebeden Fonksiyonlar
- Rekürsif Fonksiyonlar
- Referencial Transparency
- Tembel İşletim

Erlang

- 80'lerde Ericsson tarafından telekominikasyon altyapılarından kullanmak üzere geliştirilmiştir
- Paralel ve eş-zamanlı programlama için tasarlanmıştır.
- ML,Miranda,Simula,ADA,Prolog ve Smalltalk dillerinden etkilenmiştir
- Uptime yüzdesi çok yüksek %99.999

Temel Karekteristikleri

- Dinamik Yazımlı
- Hevesli İşletim
- VM tarafından yönetilen hafif processler kullanılır
- Mesaj iletimli aktör modeli.
- Processler arası bağlarla hata kontrolü
- Bırak çakılsın

Erlang ile Paralel Programlama

- Pid = spawn(Fun)
- Pid ! Message
- receive ... end,after
- register(Atom,Pid)
- unregister(Atom)
- whereis(Atom) -> Pid | undefined
- link(Pid)
- unlink(Pid)
- spawn_link(Fun)

Örnek

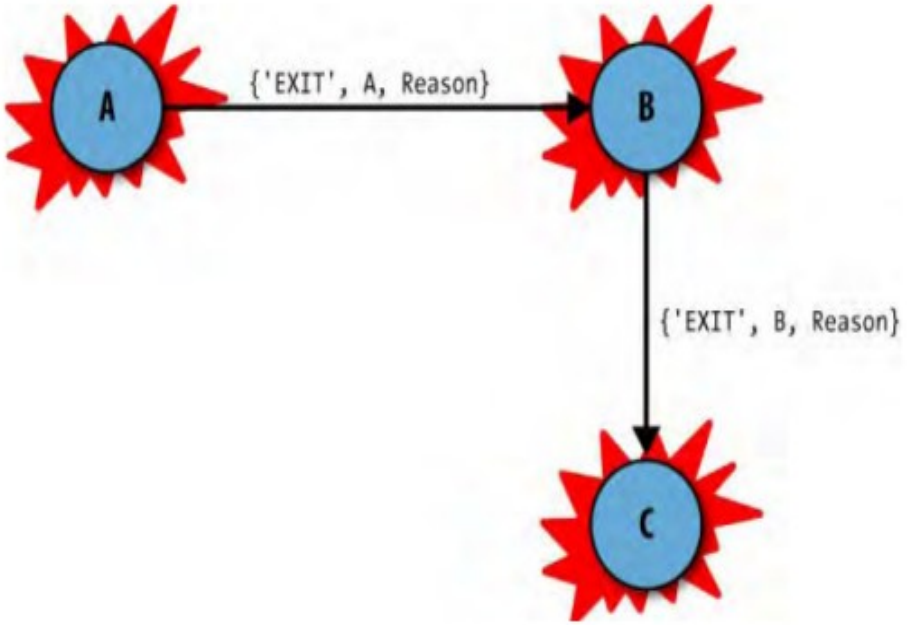
```
-module(area_server2).
-export([loop/0, rpc/2]).

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

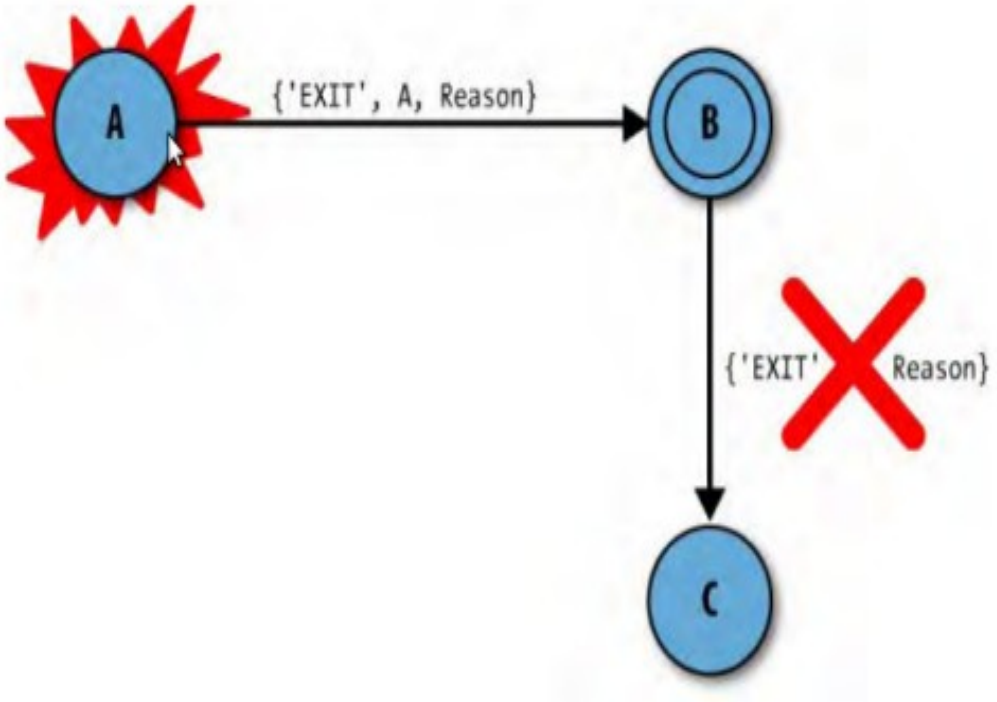
```
loop() ->
    receive
        {From, {rectangle, Width, Ht}} ->
            From ! {self(), Width * Ht},
            loop();
        {From, {circle, R}} ->
            From !  {self(), 3.14159 * R * R},
            loop();
        {From, Other} ->
            From ! {self(), {error,Other}},
    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

Erlang Shell:

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

Hata Yakalama





Scala

- 2001'de EPFL'de Prof. Dr. Martin Odersky tarafından geliştirilmeye başlanmıştır
- Ölçeklenebilirik sağlamak üzere tasarlanmıştır.
- Hem fonksiyonel hem nesne yönelimli bir dildir.
- JVM üzerinde çalışır, Java uyumludur.
- Java,ML,Haskell,Simula,Smalltalk,Erlang dillerinden etkilenmiştir.

Temel Karekteristikleri

- Statik yazımlıdır. Scala derleyicisi tip çıkarımı yapabilir.
- Hevesli işletim kullanır, tembel işletim desteği var.
- Her fonksiyon,metod bir değer döndürmelidir.
- Primitif tipler ve Statik yapılar bulunmaz.
- Immutable veri yapıları var.
- Traitler

Scala ile Paralel Programlama

- Bellek paylaşımlı ve mesaj iletimli model kullanılabilir.
- İşlemsel bellek desteği var
- Paralel koleksiyon desteği var.
- Receive,react çağrıları
- Thread pool kullanır

```scala
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
```

```scala
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"
}
```

Sonuçlar ve Sorular

Dinlediğiniz için Teşekkür Ederim