

Paralel Hesaplama

Öğr. Gör. Zafer SERİN

PARALEL HESAPLAMA

- Paralel programlama, kod talimatlarının eşzamanlı olarak çalıştırılması için hazırlanmış ortamlarla uyumlu programlar oluşturmayı amaçlayan bir model olarak tanımlanabilir. Önceden işlemciler, belirli bir zaman aralığında yalnızca bir talimatın çalıştırılabilıldığı, diğer bileşenler arasında tek bir Aritmetik Mantık Birimi (ALU) içeriyordu. Yıllarca, yalnızca bir işlemcinin belirli bir zaman aralığı içinde işleyebileceği talimat sayısını belirlemek için hertz cinsinden ölçülen bir saat dikkate alındı.

PARALEL HESAPLAMA

- 80'li yılların ortalarında, görevlerin önceden belirlenmiş bir şekilde yürütülmesine olanak tanıyan, yani bir programın yürütülmesini periyodik olarak kesip işlemci zamanını başka bir programa vermek mümkün olan devrim niteliğinde bir işlemci ortaya çıktı: Intel 80386. Bu, zaman dilimine dayalı sahte paralellik anlamına geliyordu.

PARALEL HESAPLAMA

- Intel'in kurucu ortağı Gordon Moore 1965'te kendi adıyla anılan Moore Yasasını ortaya attı. Buna göre işlemcilerde bulunan transistör sayısı her yıl 2 katına çıkmalıydı; ancak fiziksel sınırlamalar, ısı ve güç tüketimi gibi konular bu yasanın sürdürülmesini güç bir hale getirdi. 1975'te Moore Yasası buna göre güncellenerek işlemcilerde bulunan transistör sayısının her 2 yıl da bir 2 katına çıkması haline gelmiştir. Bu süre günümüzde halen devam ettirilmeye çalışılsa da süreler uzamaktadır.

NEDEN PARALEL HESAPLAMA? FİZİKSEL SINIRLAR

- Moore Yasası'nın öngördüğü hız artışı, fiziksel engellere çarptı.
- Frekans Duvarı (Frequency Wall): İşlemci saat hızlarını (GHz) artırmak, sinyal gecikmeleri ve senkronizasyon sorunları nedeniyle çok zorlaştı.
- Güç Duvarı (Power Wall): Hız (frekans) arttıkça, güç tüketimi ve ortaya çıkan ısı katlanarak arttı.
- Isı Duvarı (Heat Wall): Bu aşırı ısını işlemciden uzaklaştırmak (soğutmak) pratik olarak imkansız hale geldi.
- Çözüm: Daha hızlı tek bir çekirdek yerine, daha yavaş ama çok sayıda çekirdek (multi-core) kullanmak.

PARALEL HESAPLAMA

- Paralel Hesaplama, ilgili problemi daha hızlı bir şekilde çözmek için birden çok işlem biriminin kullanılmasını ifade etmektedir.
- Gerçek hayattan bir örnek ile açıklamak gerekirse; bir işin tamamlanma süresi yaklaştıkça arkadaşlarınızdan ve çevrenizden yardım isteyerek işin daha kısa sürede bitirilmesini sağlamanız gibi.
- Bilgisayar bilimleri ve programlama açısından bu arkadaşlar Process(İşlem) ve Thread(İş Parçacığı) olarak düşünülebilir. Bu 2 yapı programın hızını artırmak için kullanılır.

TEMEL MİMARİ: TALİMAT (I) VE VERİYE (D) ERİŞİM

1. Von Neumann Mimarisi (Birleşik Depolama)
 - Hem program kodları (Talimatlar) hem de o kodların işleyeceği veriler (Veri) aynı Hafıza Birimi'nde (RAM) tutulur.
 - İşlemci, bu tek hafıza alanına tek bir Veri Yolu (Bus) üzerinden erişir.
 - İşlemci, aynı anda hem talimatı hem de veriyi hafızadan çekemez. Biri için beklerken diğerini durur. Bu, bir "darboğaz" yaratır.

TEMEL MİMARİ: TALİMAT (I) VE VERİYE (D) ERİŞİM

2. Harvard Mimarisi (Ayrık Depolama)

- Talimat Hafızası ile Veri Hafızası fiziksel olarak ayrılmıştır.
- İşlemcinin bu iki hafızaya giden iki ayrı Veri Yolu (Bus) vardır.
- İşlemci, aynı anda bir yoldan "talimatı" çekerken, diğer yoldan "veriyi" çekebilir. Darboğaz yoktur.

Not: Günümüz modern CPU'ları, RAM'e erişirken (Von Neumann) ve kendi iç L1 Önbellegine (Cache) erişirken (Harvard) bu iki modelin Hibrit (Melez) bir yapısını kullanır.

TEMEL MİMARI: "TALİMAT (I)" NEDİR? (CISC vs. RISC)

- İşlemcinin anladığı "Talimat Seti Alfabesi", iki zıt felsefeye sahiptir:
 1. CISC (Complex Instruction Set Computer) - Karmaşık Komut Seti
- İşlemci akıllı olmalı, tek bir komutla çok karmaşık işler (örn: hafızadan al, matematik yap, sonucu geri yaz) yapabilmelidir.
- İşlemci devresi karmaşıklasır, daha çok güç tüketir ve daha çok ısınır.
- Intel / AMD (x86)

TEMEL MİMARI: "TALİMAT (I)" NEDİR? (CISC vs. RISC)

2. RISC (Reduced Instruction Set Computer) - Azaltılmış Komut Seti

- İşlemci çok yetenekli değil ama süper hızlı olmalı. Sadece çok temel (Al, Bırak, Topla) komutları anlamalı.
- Karmaşık bir iş için derleyicinin (programın) daha fazla sayıda basit komut vermesi gereklidir.
- İşlemci devresi basit, verimli, az ısınır ve az güç tüketir (Mobil cihazlar, Apple M-Serisi).
- ARM

PARALEL MİMARİLER: FLYNN TAKSONOMİSİ

- Paralel sistemler, talimat (Instruction) ve veri (Data) akışlarına göre sınıflandırılabilir:
- SISD (Single Instruction, Single Data): Tek talimat, tek veri. (Geleneksel tek çekirdekli işlemciler).
- SIMD (Single Instruction, Multiple Data): Tek talimat, çoklu veri.
Örn: GPU'lar. Bir veriye (pixel) yapılan işlem, binlerce veriye (tüm piksellere) aynı anda uygulanır.
- MISD (Multiple Instruction, Single Data): Flynn Taksonomisi'ndeki en nadir mimaridir. Birden fazla (ve birbirinden farklı) talimat, aynı anda tek bir veri akışı üzerinde çalışır.
Örn: Genellikle hataya dayanıklı (fault-tolerant) sistemlerde kullanılır (Örn: Uzay mekiği uçuş kontrolcüleri, nükleer santral güvenlik sistemleri).

PARALEL MİMARİLER: FLYNN TAKSONOMİSİ

- MIMD (Multiple Instruction, Multiple Data): Çoklu talimat, çoklu veri.

Örn: Günümüz çok çekirdekli CPU'ları. Her çekirdek, kendi talimatını kendi verisi üzerinde eş zamanlı çalıştırır.

PARALEL MİMARİLER: BELLEK YAPILARI

- İşlemcilerin (işçilerin) veriye nasıl eriştiğine göre iki temel bellek mimarisi vardır:
 1. Paylaşımılı Bellek (Shared Memory):
 - Tüm işlemciler (çekirdekler) ortak bir bellek alanını (RAM) paylaşır.
 - İletişim, veriyi doğrudan belleğe yazıp okuyarak yapılır.
 - Programlaması görece kolaydır.
 - Dezavantaj: Bellek darboğazı (Tüm çekirdekler aynı veriye ulaşmaya çalışırsa tikanıklık olur).

PARALEL MİMARİLER: BELLEK YAPILARI

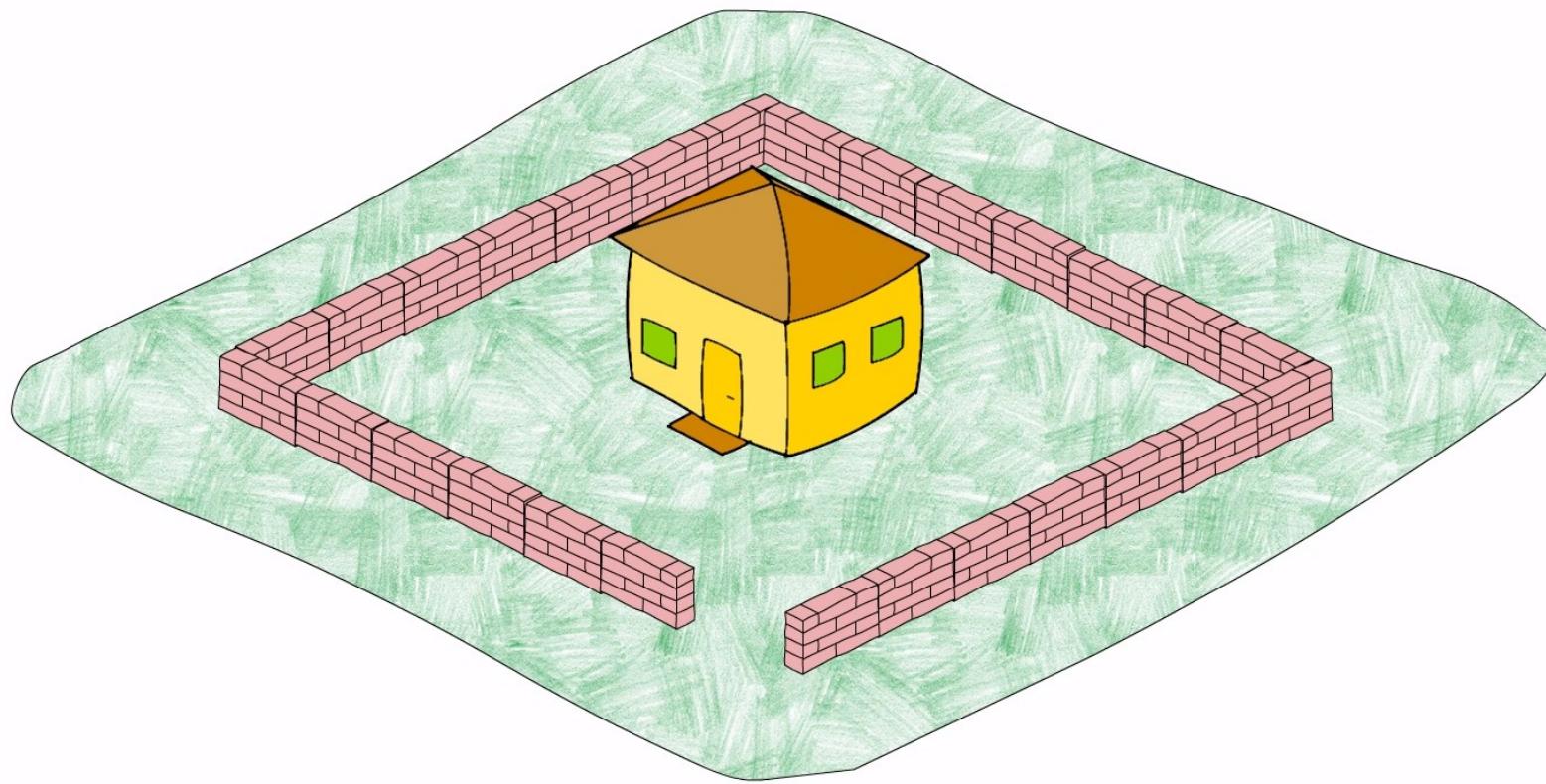
2. Dağıtık Bellek (Distributed Memory):

- Her işlemcinin kendine ait, özel bir belleği vardır.
- İletişim, ağ üzerinden "Mesaj Gönderme" (Message Passing) veya chunk(çukur) gibi çeşitli yöntemlerle yapılır.
- Dezavantaj: Programlaması daha karmaşıktır.

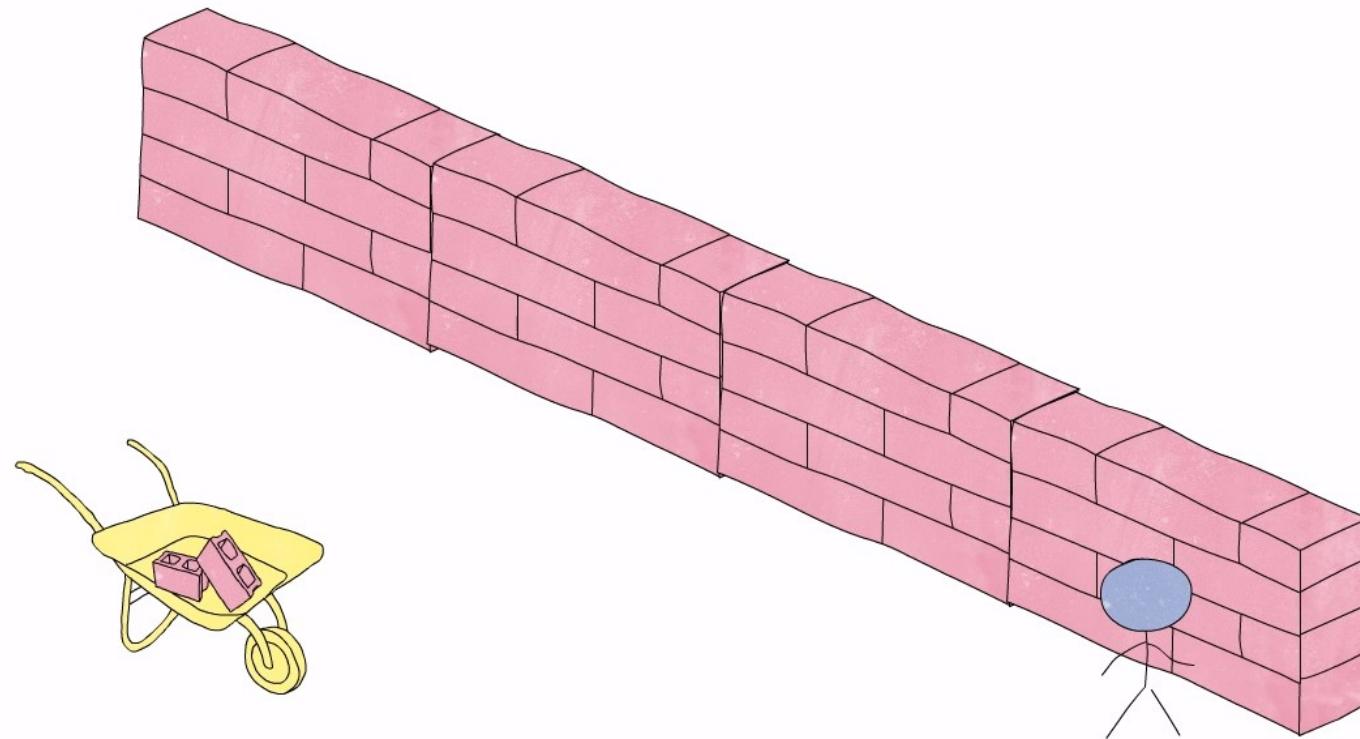
PARALEL HESAPLAMA

- Bir evimiz olduğunu ve bu evin etrafına bir duvar yapmak istediğimizi düşünelim.
- Duvarın yapılması için 1 kişi ile anlaştığımıza ve 1 kişinin bu işi 8 günde bitirdiğini varsayıyalım.
- Aynı iş için 1 yerine 2 kişi ile anlaşılmış olunsaydı iş 8 yerine 4 günde biterdi.
- Bu şekilde yeni kişiler ile anlaşarak işin bitme süresi azaltılabilir, daha hızlı bir şekilde sonuca ulaşılabilir.
- Bununla birlikte anlaşılan kişi sayısı arttıkça iş sonsuza kadar hızlanmaz bir yerde bir sınır ile karşılaşılırıldı.

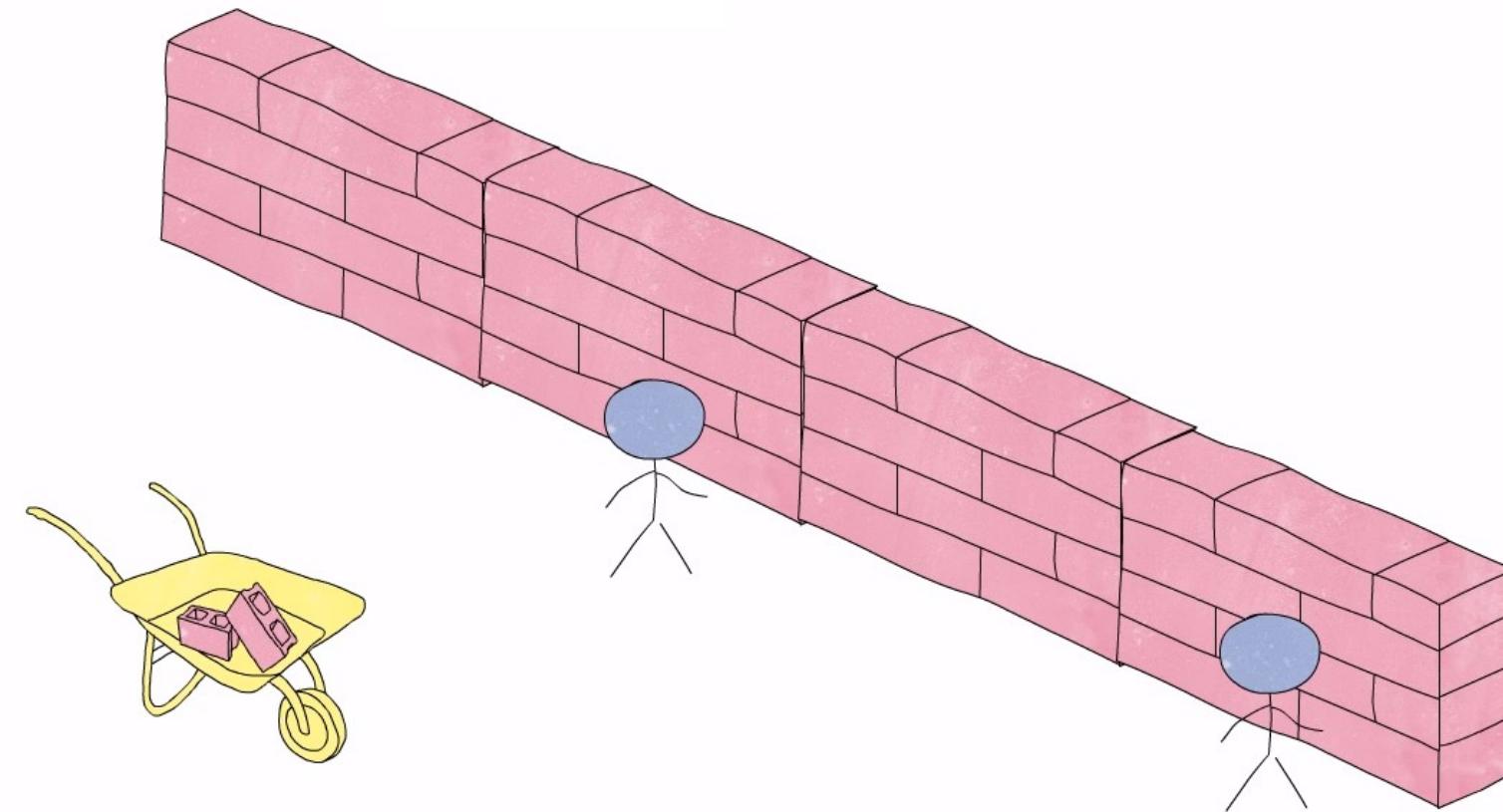
PARALEL HESAPLAMA



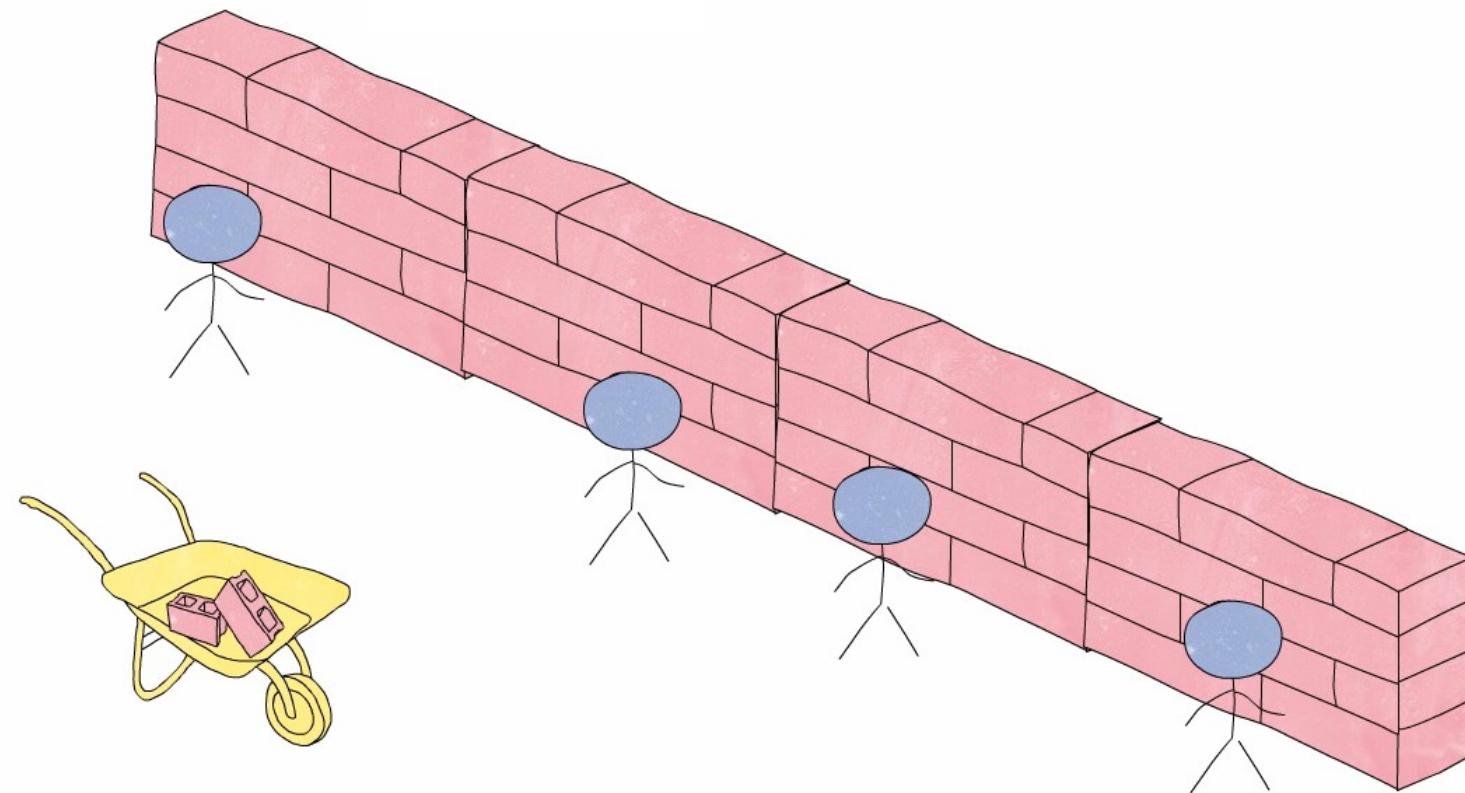
PARALEL HESAPLAMA



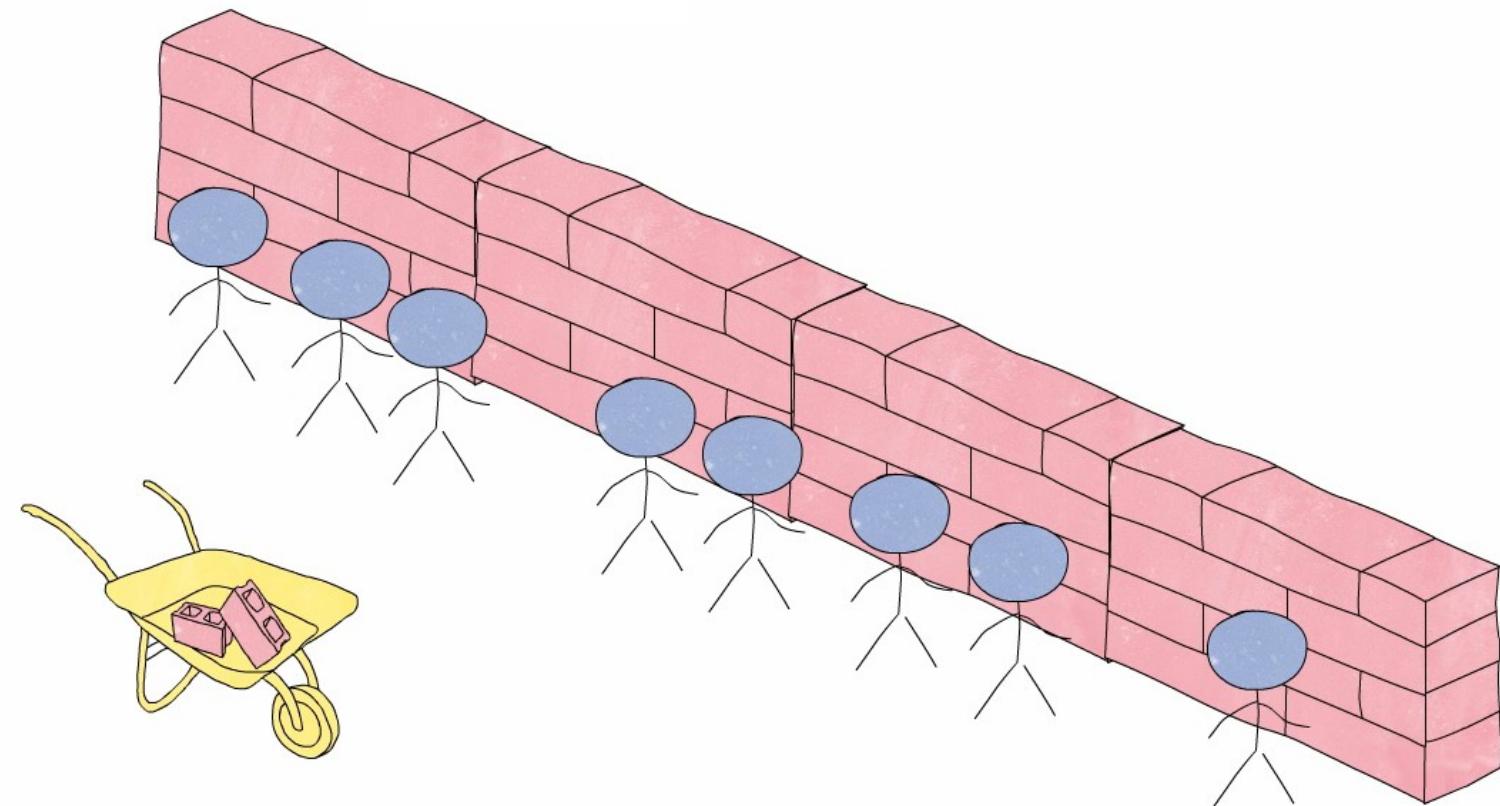
PARALEL HESAPLAMA



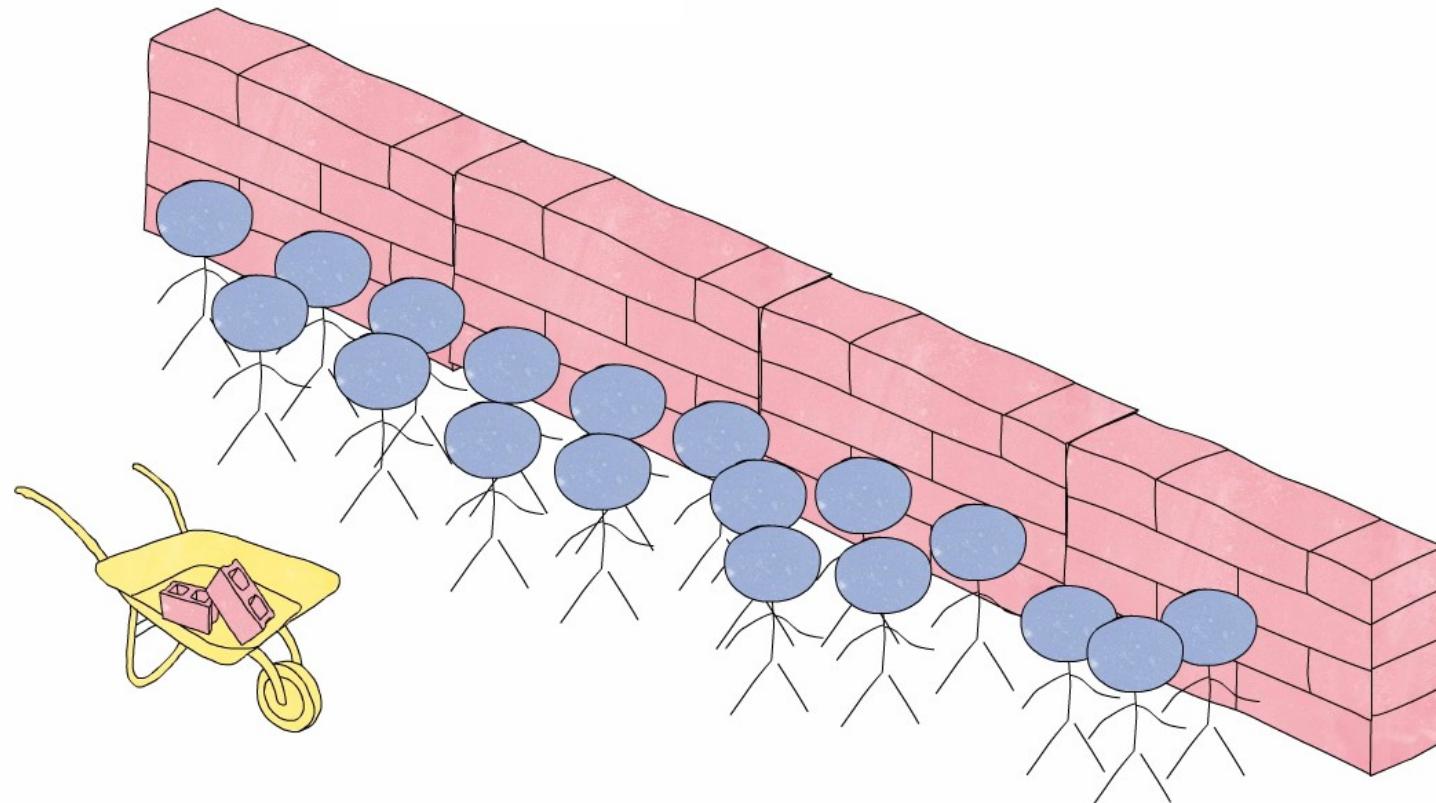
PARALEL HESAPLAMA



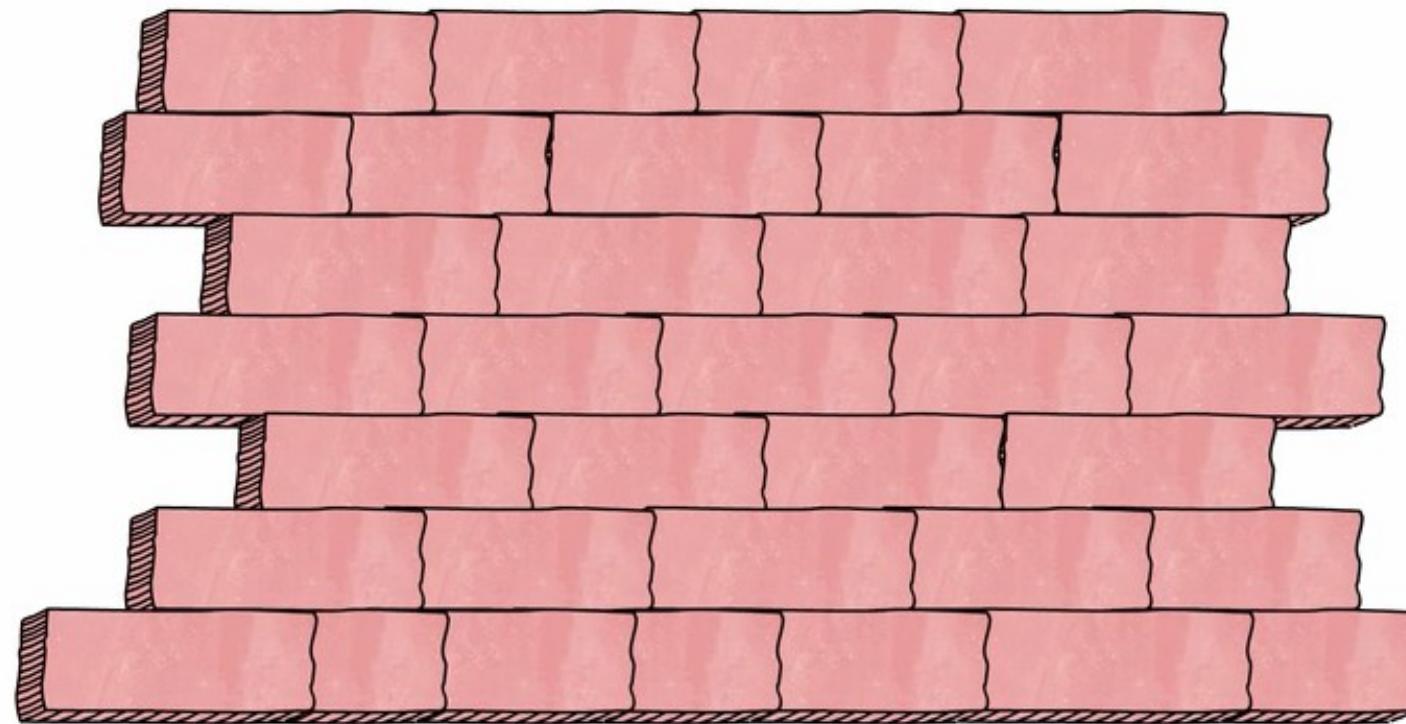
PARALEL HESAPLAMA



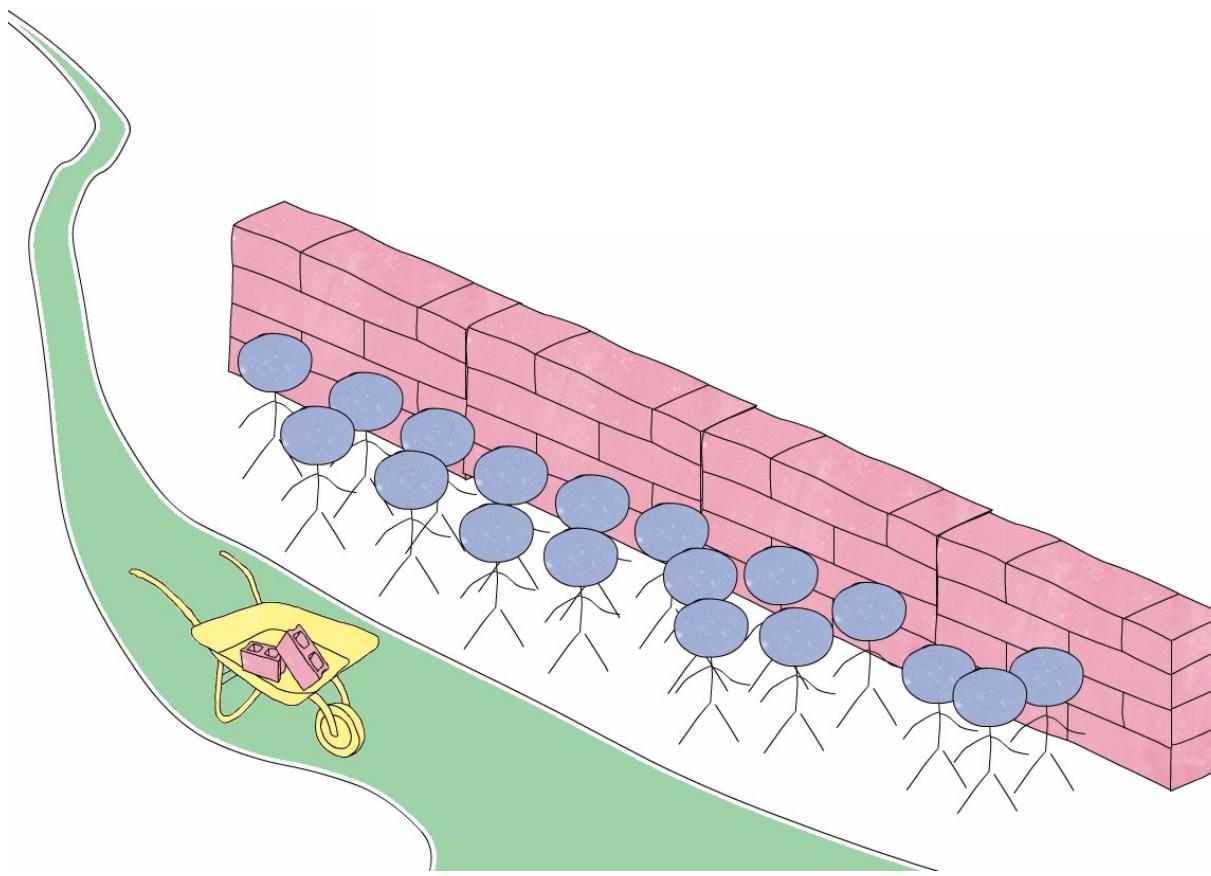
PARALEL HESAPLAMA



PARALEL HESAPLAMA



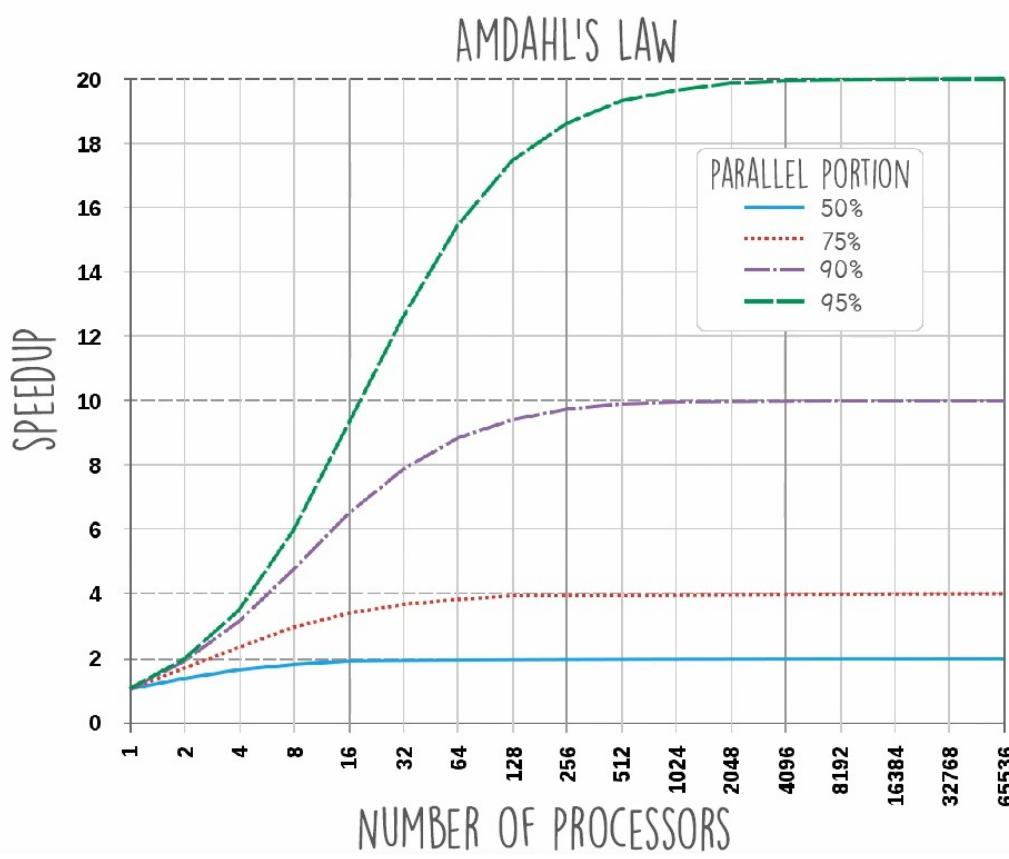
PARALEL HESAPLAMA



AMDAHL KANUNU

- AMDAHL Kanunu işlem birimi sayısı arttıkça hızımızın ne kadar artacağını gösteren bir kanundur.
- Eğer problemimizde paralelleştirilebilecek kısım fazla ise dolayısıyla hız artışı da daha yüksek olacaktır. Bununla birlikte problemimizdeki seri (sequential - sıralı) kısım fazla ise işlem birimi sayısı artsa bile hızda ki artış bir müddet sürecek ve daha sonra duracaktır.

AMDAHL KANUNU



$$S(P) = \frac{1}{\alpha + \frac{1 - \alpha}{P}}$$

Burada;

$S(P)$: P sayıda işlemci ile elde edilen hızlanma

P: İşlemci sayısı

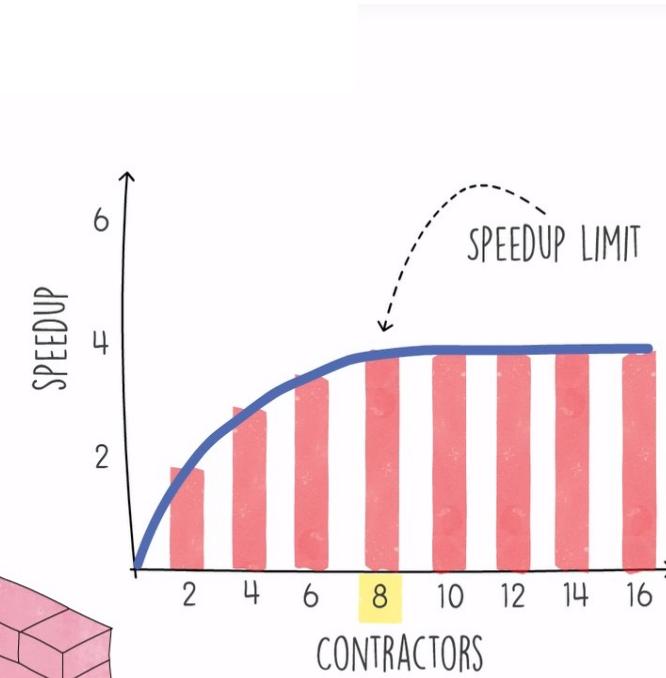
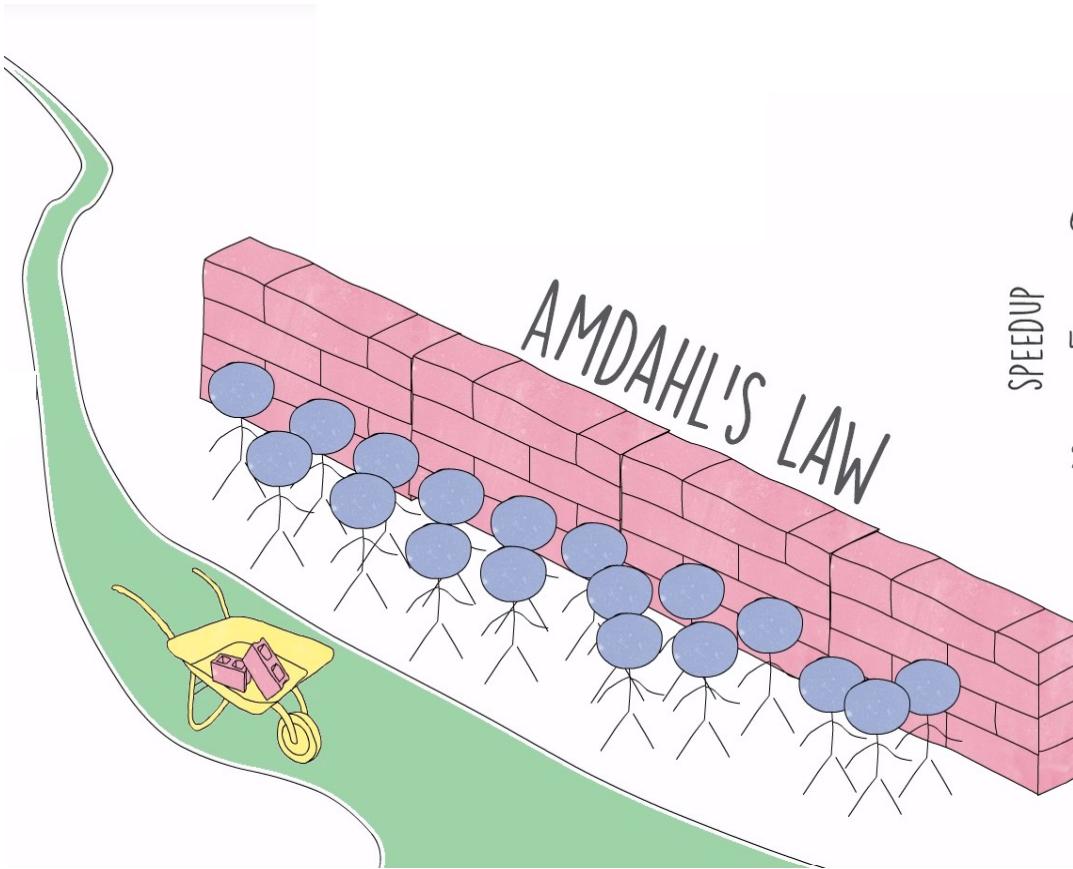
α : Seri kısmın oranı(0 ile 1 arasında bir değer)

Örneğin bir programın %10'u seri olarak çalışıyor ve %90 i paralel olarak çalışıyor. Bu durumda $\alpha=0.1$ olur. Eğer 10 işlemci kullanılırsa

$$S(10) = \frac{1}{0.1 + \frac{0.9}{10}} \approx 5.26$$

Yani 10 işlemci ile yaklaşık 5.26 kat hızlanma elde edilebilir.

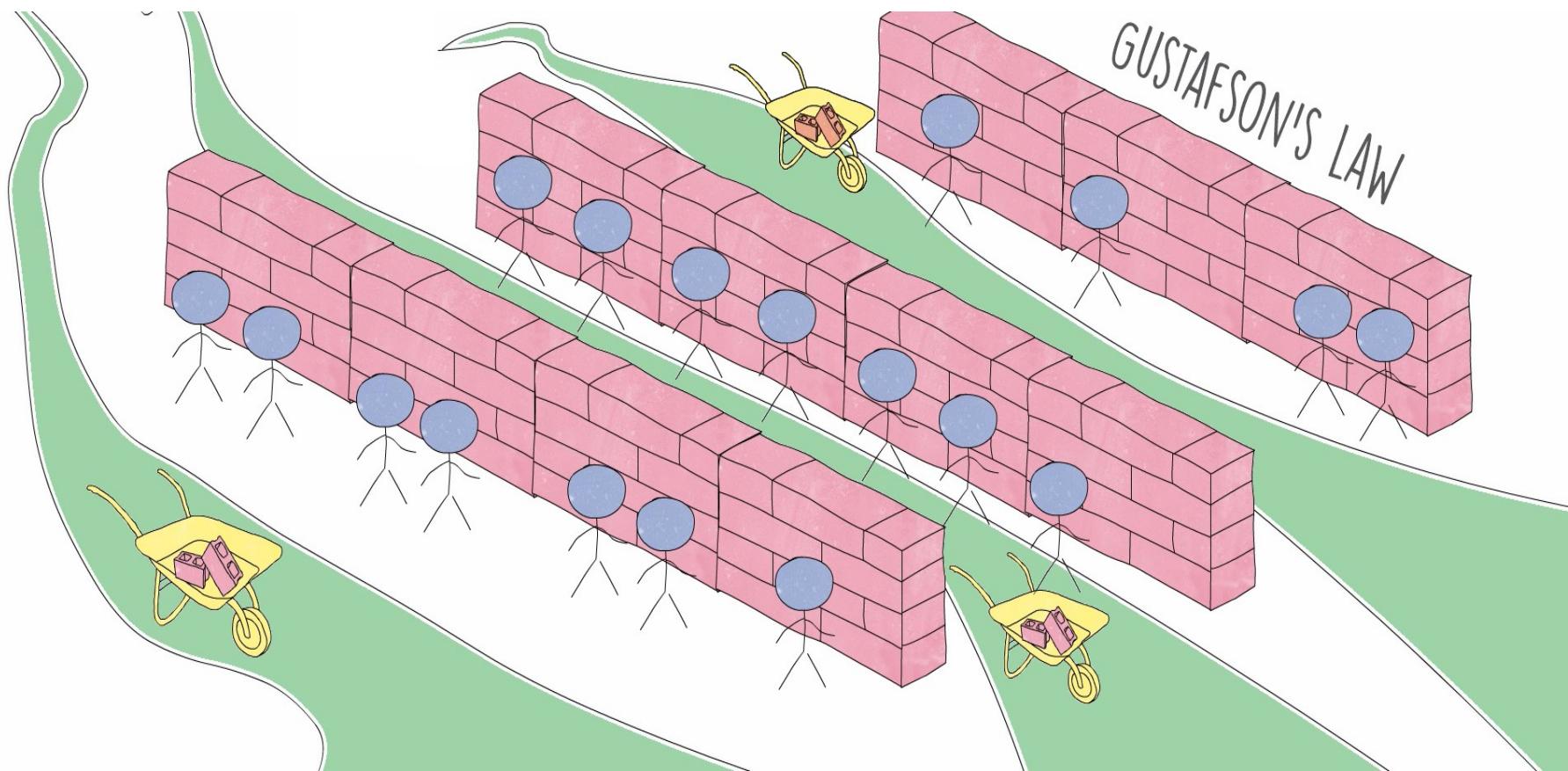
AMDAHL KANUNU



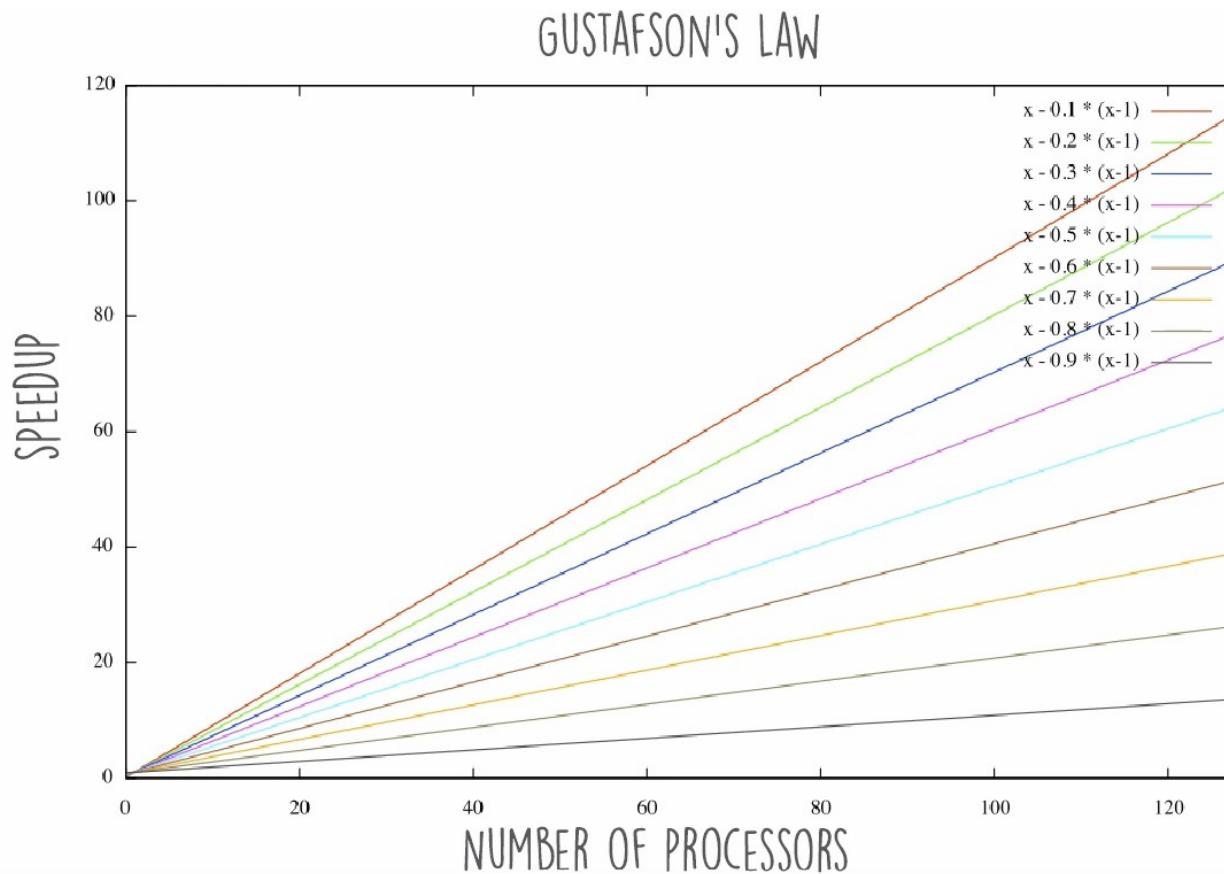
GUSTAFSON KANUNU

- Gustafson kanunu problem boyutu arttıkça teorik olarak hızın da doğrusal bir şekilde artacağını söyler.
- Problemin paralel olarak işlenebilecek ve sıralı olarak işlenebilecek kısımlarının halen önemli olduğu ve Amdahl Kanunu ile örtüşen bir şekilde olacağı unutulmamalıdır.

GUSTAFSON KANUNU



GUSTAFSON KANUNU



$$S(P) = P - \alpha(P - 1)$$

Burada;

$S(P)$: P sayıda işlemci ile elde edilen hızlanma

P : İşlemci sayısı

α : Seri kısmın oranı(0 ile 1 arasında bir değer)

Örneğin bir programın %10'u seri olarak çalışıyor ve %90 i paralel olarak çalışıyor. Bu durumda $\alpha=0.1$ olur. Eğer 10 işlemci kullanılırsa

$$S(10) = 10 - 0.1(10 - 1) = 9.1$$

Yani 10 işlemci ile yaklaşık 9.1 kat hızlanma elde edilebilir.

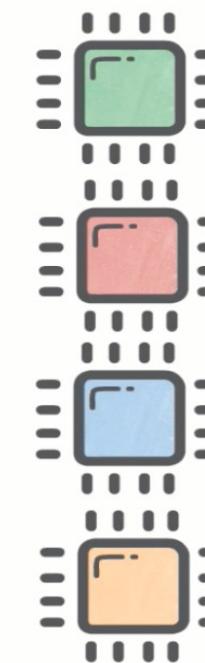
KARŞILAŞTIRMA

- **Amdahl Kanunu:** Problem boyutunun sabit olduğu varsayımlıyla çalışır ve seri kısıtlamalar nedeniyle maksimum hızlanmanın sınırlı olduğunu vurgular.
- **Gustafson Kanunu:** Problem boyutunun artırılabileceği varsayımlıyla çalışır ve büyük ölçekli paralel sistemlerde daha iyi bir ölçeklenebilirlik tahmini sunar.

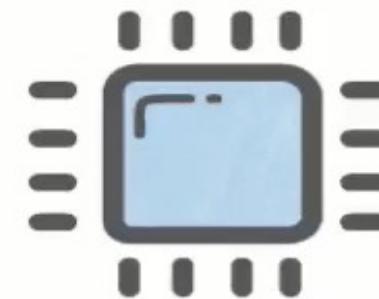
RESPONSIVENESS(YANIT HIZI)

- Responsiveness bir yazılımın veya uygulamanın kullanıcı veya dış etkenlere hızlı bir şekilde yanıt verebilme yeteneğini ifade eder.
- Örneğin bir web servis için servisin isteği aldıktan sonra, yanıtı ne kadar hızlı gönderebildiğini ölçülmesi olarak düşünülebilir.

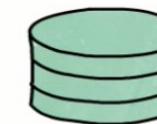
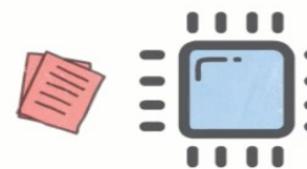
RESPONSIVENESS(YANIT HIZI)



RESPONSIVENESS(YANIT HIZI)



RESPONSIVENESS(YANIT HIZI)



DRIVE

PROCESS(SÜREÇ) ve THREAD(İŞ PARÇACIĞI)

- Processler ve Threadler paralel programlama ve çoklu görevlerin yönetilmesi için kullanılan temel kavamlardır.
- Genel olarak Processler daha fazla izolasyon sağlar, daha çok kaynak tüketir ve daha uzun süre alırlar. Buna karşılık Threadler daha az izolasyon sağlar, daha az kaynak tüketir ve daha kısa süre alırlar.

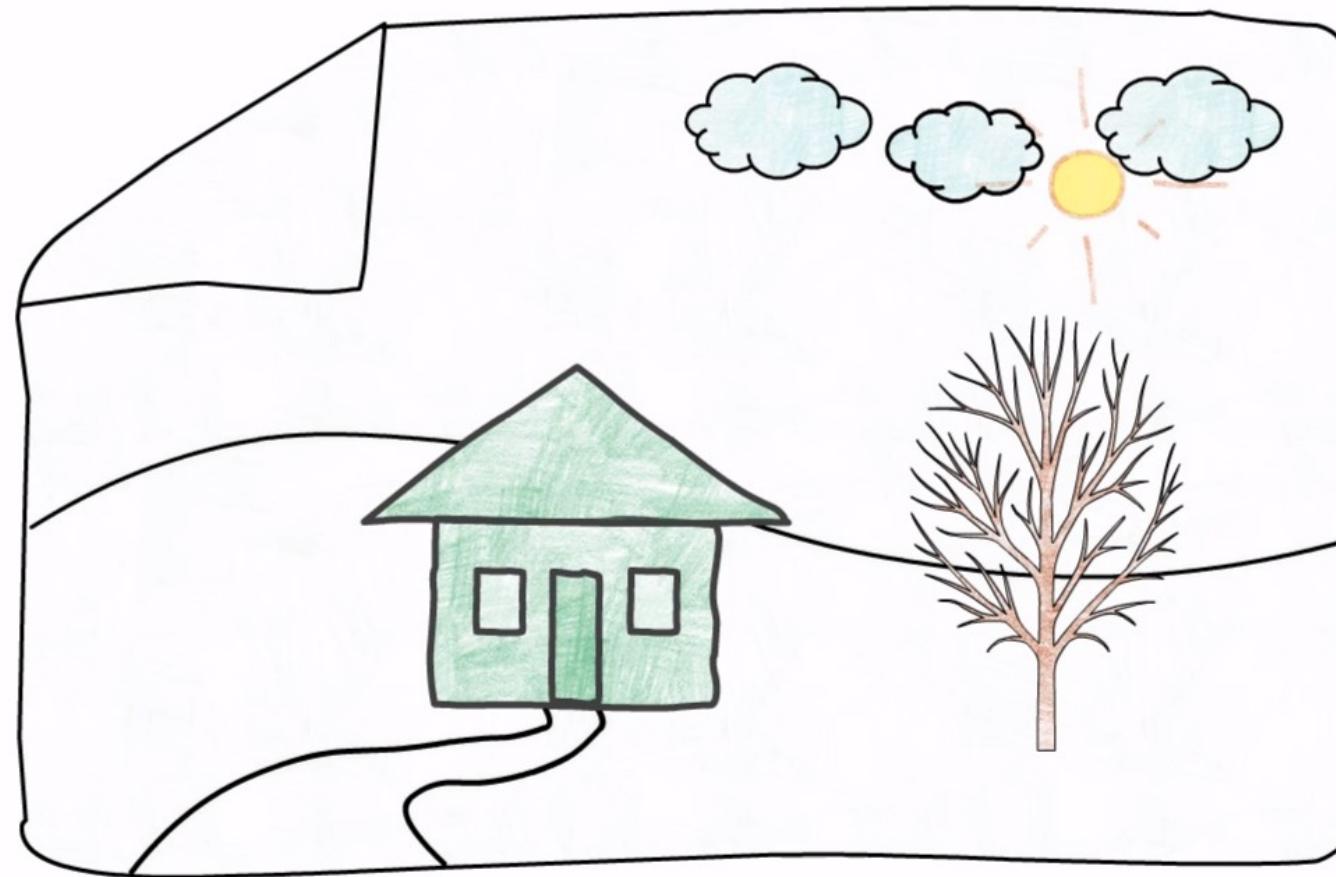
PROCESS ve THREAD ÖRNEK

- İyi bir ressam olduğunuzu düşünün.
- Kısa bir sürede bitmesi gereken bir iş aldınız ve çizmeniz gereken bir resim var.
- Kendi başına görevi tamamlayamayacağınızı düşündünüz ve arkadaşlarınızdan yardım istediniz.

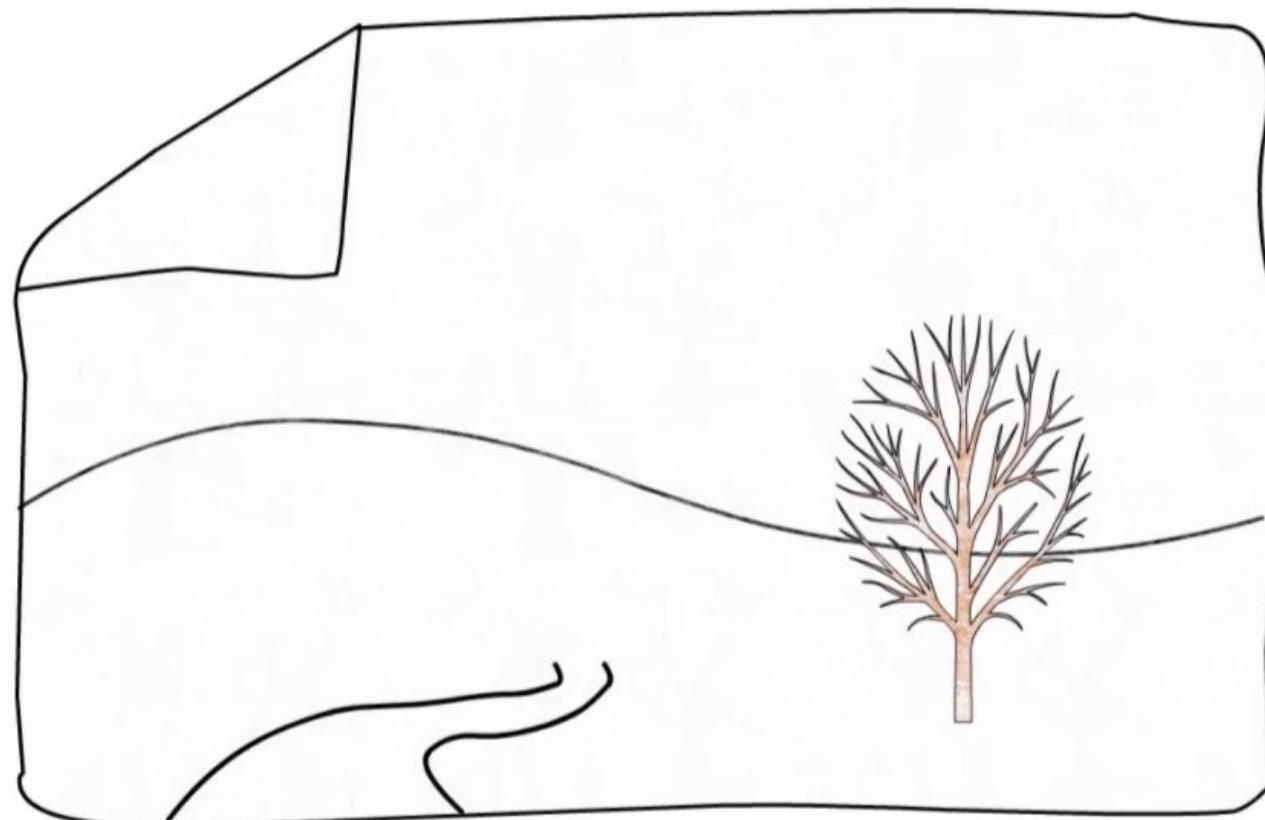
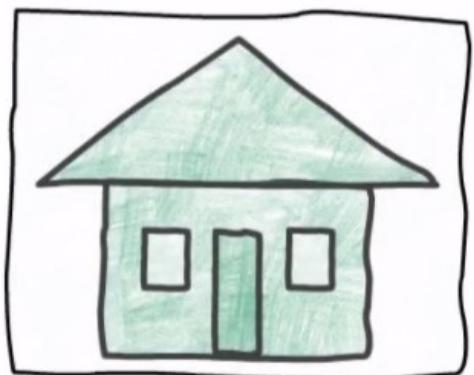
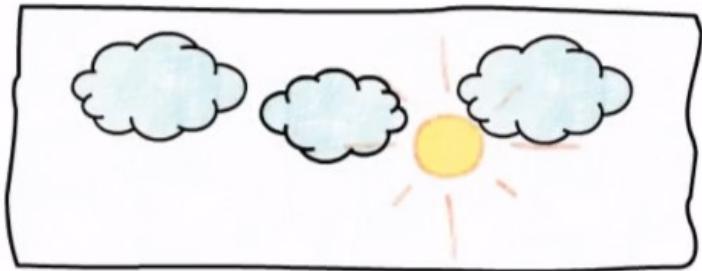
PROCESS ve THREAD ÖRNEK

- Arkadaşlarınıza da birer kağıt verdiniz ve resmin bazı bölümlerini onların çizmesini istediniz. En son tüm parçaları birleştirip asıl resmi oluşturduğunuz. Bu durum Process'e bir örnektir.
- Arkadaşlarınıza kağıt vermek yerine bir kağıt üzerinde hep beraber çizmeye başladınız. Birbirinizle iletişim kurarak ve iletişim halinde olarak resmi tamamladınız. Bu durum Thread'e bir örnektir.

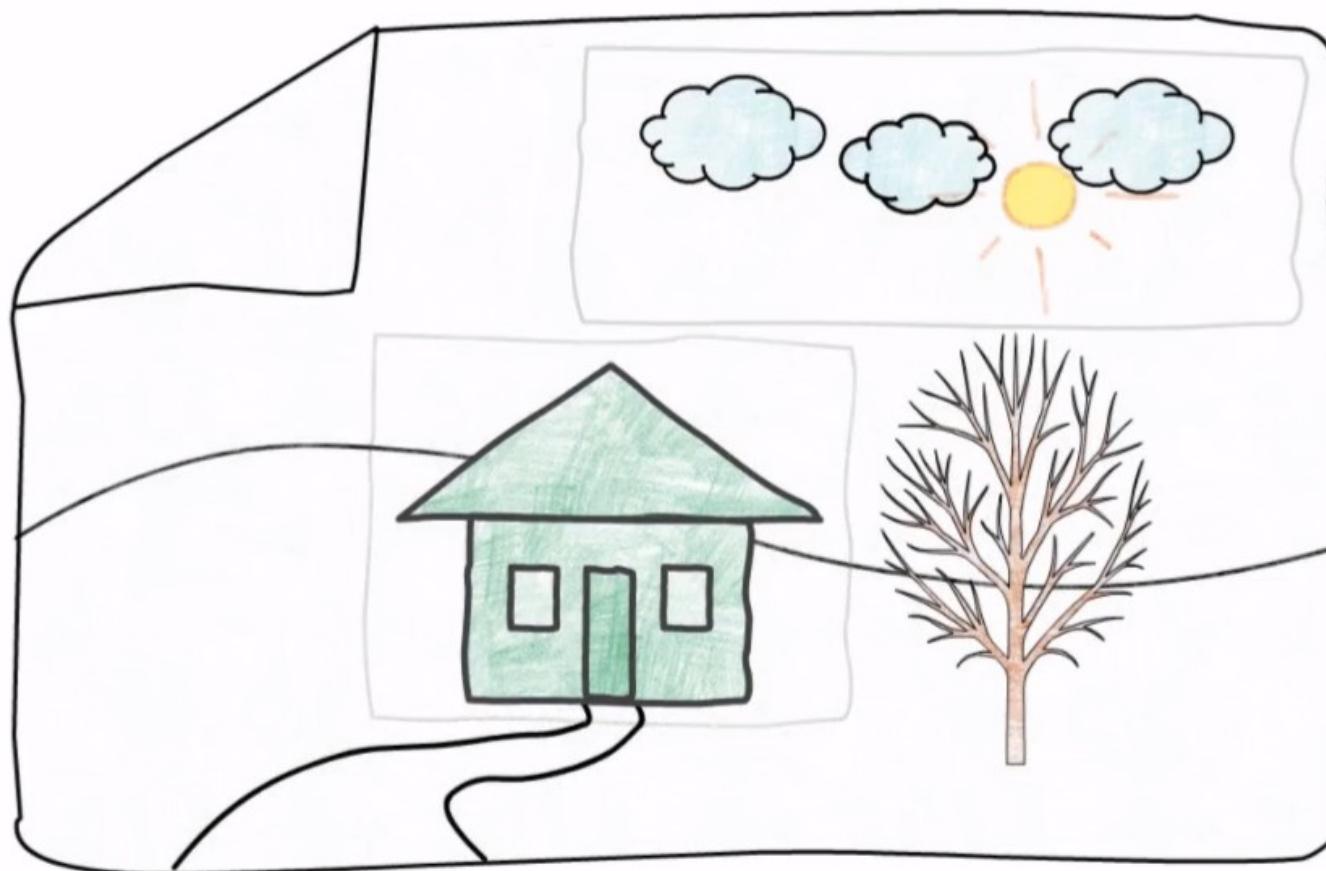
PROCESS



PROCESS



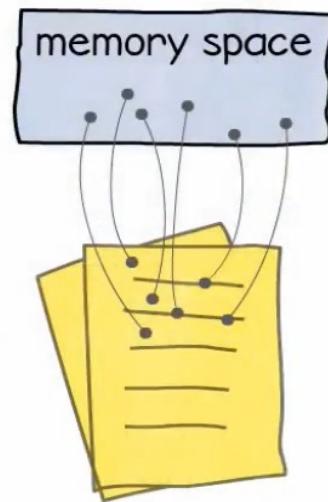
PROCESS



PROCESS

- Bu örnektenden anlaşılabileceği üzere Processlerin kendi memory(bellek) alanları vardır ve bunlar üzerinde diğer Processler ile herhangi bir çalışma ve müdahale olmadan çalışabilirler.

PROCESS



THREAD



THREAD

- Bu örnekten anlaşılabileceği üzere Thread ler ortak bir bellek üzerinde çalışırlar ve birbirleri arasında doğrudan bilgi paylaşımı yaparlar.

THREAD

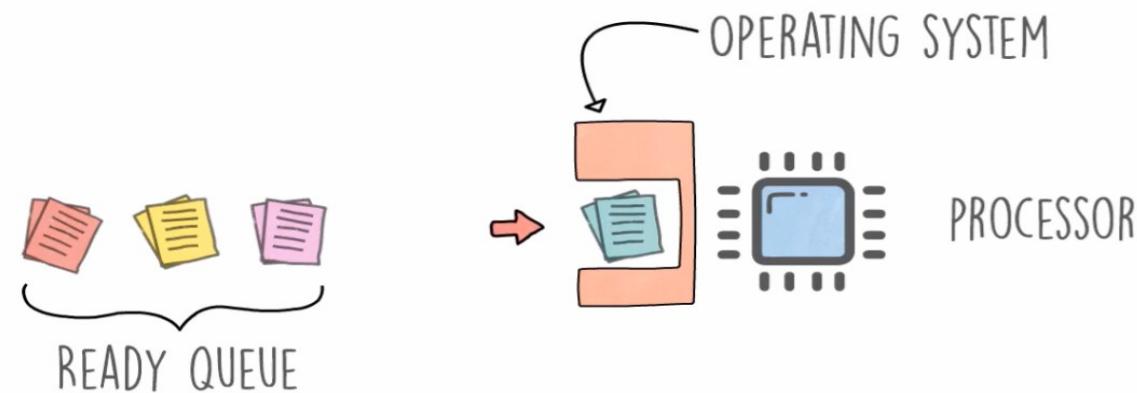
memory space



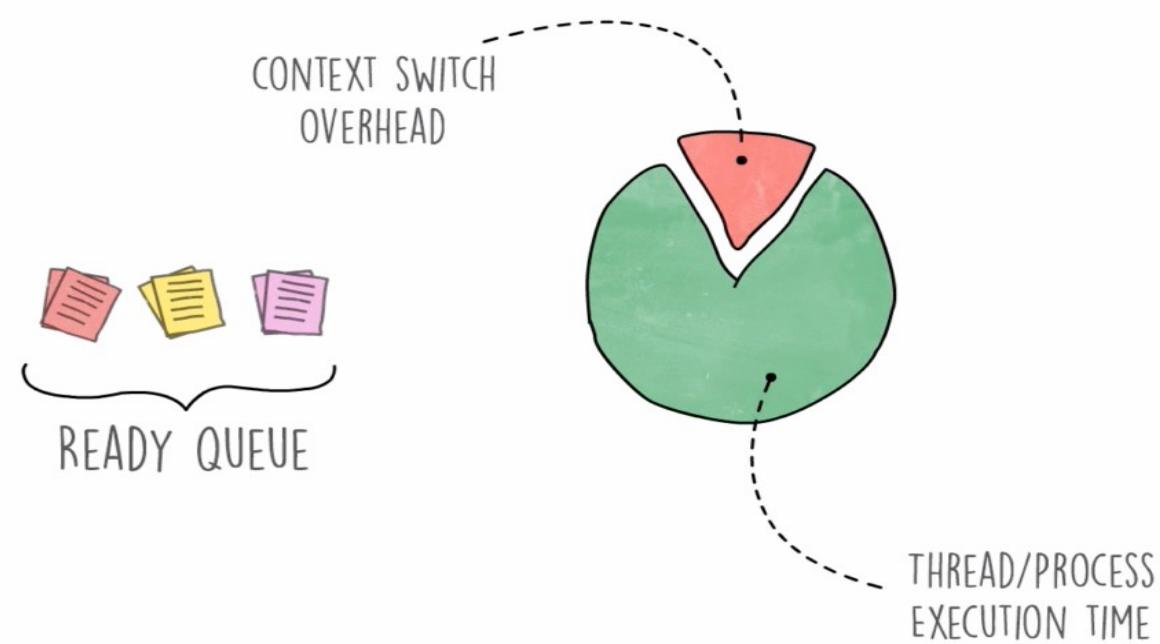
PROCESS – THREAD ÇALIŞMA SIRASI

- Hangi Process veya Thread'in çalışacağı temel olarak işletim sistemi tarafından yapılır ve programcının bu konuda endişelenmesi gerekmez.

PROCESS – THREAD ÇALIŞMA SIRASI



PROCESS – THREAD ÇALIŞMA SIRASI



CONTEXT SWITCH MALİYETİNE DONANIMSAL ÇÖZÜM: HYPER-THREADING(SMT)

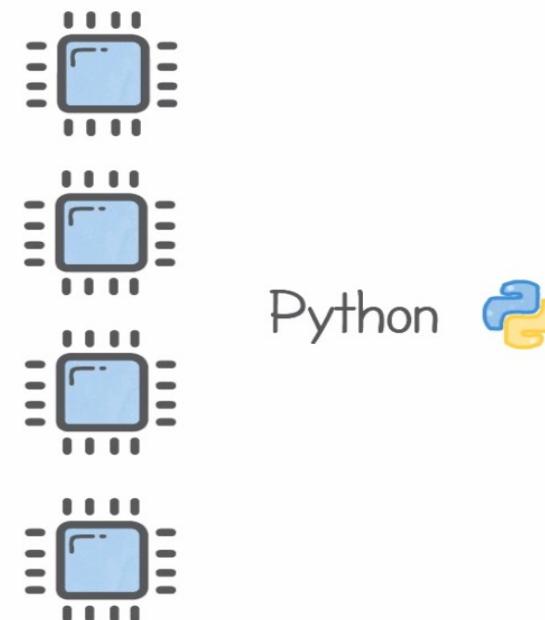
- "Context Switch Overhead" , İşletim Sisteminin bir işi durdurup diğerini başlatmasının yazılımsal maliyetidir.
- Fiziksel işlemciler (çekirdekler), özellikle "bekleme" (I/O) anlarında (örn: diskten okuma, ağdan veri beklemeye) boş zaman harcar.
- Donanımsal Thread (Intel'de Hyper-Threading, AMD'de SMT) teknolojisi, bu "bekleme" anlarındaki verimsizliği çözmek için geliştirilmiştir.

CONTEXT SWITCH MALİYETİNE DONANIMSAL ÇÖZÜM: HYPER-THREADING(SMT)

- Çalışma Prensibi:
- Tek bir fiziksel çekirdek, İşletim Sistemine (OS) kendisini iki mantıksal işlemci gibi gösterir.
- İşletim Sistemi bu çekirdeğe iki yazılımsal thread (iş) atar.
- Çekirdek, 1. thread üzerinde çalışırken bir "bekleme" (I/O) anına geldiğinde, yazılımsal bir Context Switch beklemeden, 1 nanosaniye gibi donanımsal bir hızda diğer 2. thread'i çalışmaya geçer.
- Bu sayede, işlemcinin o "bekleme" anları (I/O Bound işler) bile verimli bir şekilde doldurulmuş olur.

PYTHON'DA THREAD VE GLOBAL INTERPRETER LOCK(GIL)

PYTHON'DA THREAD VE GLOBAL INTERPRETER LOCK(GIL)



PYTHON'DA THREAD VE GLOBAL INTERPRETER LOCK(GIL)

CPU Bound(CPU Sınırı)(⌚)

Oyun/Grafik

Şifreleme/Şifre Çözme

Vdeo/ses düzenleme

Makine Öğrenmesi

IO Bound(IO Sınırı)(😊)

Dosya transferi

Arama robotu(Webpage Crawler)

Web sunucular

PYTHON'DA THREAD VE GLOBAL INTERPRETER LOCK(GIL)

- GIL'in Python'a özgü olduğu ve sadece Thread kullanılırken geçerli olduğu unutulmamalıdır. Diğer programlama dillerinde böyle bir kısıt olmayıabilir.
- CPU Bound alanında bulunan bir işlemin hızlandırılması için Thread kullanılması muhtemelen ya çok az bir fayda sağlayacak ya da hiç fayda sağlamayacaktır.
- IO Bound alanında bulunan bir işlemin hızlandırılması için Thread kullanılması uygundur.

PYTHON'DA THREAD VE GLOBAL INTERPRETER LOCK(GIL)

- CPU Bound bir işlemin hızlandırılması isteniyorsa Thread yerine Process kullanılması gerekecektir(!Python için).
- Python'un sahip olduğu bu kısıttan kurtulmak için çeşitli Python ortamları kullanılabilir. Bunlara Cython, Jython ve IronPython örnek olarak verilebilir.

EŞZAMANLILIĞIN ÜÇÜNCÜ YOLU: ASYNCIO (İŞ BİRLİKÇİ ÇOKLU GÖREV)

- Python'da IO-Bound (bekleme-yoğun) işler için threading'e modern ve güçlü bir alternatiftir.
- threading ve asyncio modelleri, "Ressam" analojimizde farklı çalışır:
 - threading Modeli (Preemptive - Zorlayıcı):
 - Birden fazla "Ressam" (Thread) vardır .
 - Hepsi aynı anda çalışmaya çalışır ama tek bir "Fırça" (GIL) için kavga ederler.
 - İşletim Sistemi (Patron), fırçayı ressamlar arasında zorla değiştirir.

EŞZAMANLILIĞIN ÜÇÜNCÜ YOLU: ASYNCIO (İŞ BİRLİKÇİ ÇOKLU GÖREV)

- asyncio Modeli (Cooperative - İş Birlikçi):
 - Sadece TEK BİR "DÂHİ RESSAM" (tek bir thread) vardır.
 - Bu ressam, aynı anda 10 farklı tuval (task) üzerinde çalışır.
 - Ressam, bir tuvalde "boyanın kurumasını" (I/O beklemesi) beklerken, o işi await anahtar kelimesiyle gönüllü olarak bırakır ve hemen diğer tuvale geçer.

EŞZAMANLILIĞIN ÜÇÜNCÜ YOLU: ASYNCIO (İŞ BİRLİKÇİ ÇOKLU GÖREV)

- Avantajları:
 - Sadece 1 thread olduğu için GIL asla bir sorun olmaz.
 - "Context Switch Overhead" yoktur (iş değişikliğini patron değil, ressamın kendisi yapar).
- Dezavantajları(Neden threading hala var?):
 - "Kırılgandır": async bir fonksiyonda await kullanmayı unutan tek bir bloklayıcı çağrı (örn: time.sleep()), o "tek dâhi ressamı" dondurur ve tüm programı kilitler.
 - "Bulaşıcıdır": Tüm kütüphane ekosisteminizin async uyumlu olması gereklidir (örn: requests yerine aiohttp kullanmak zorundasınız).

THREAD ÖRNEK AÇIKLAMASI