1. Paralel ve Dağıtık Hesaplama

Paralel hesaplama ve dağıtık hesaplama, büyük ölçekli hesaplamaları verimli bir şekilde gerçekleştirmek için kullanılan iki temel yaklaşımdır.

Paralel Hesaplama vs. Dağıtık Hesaplama:

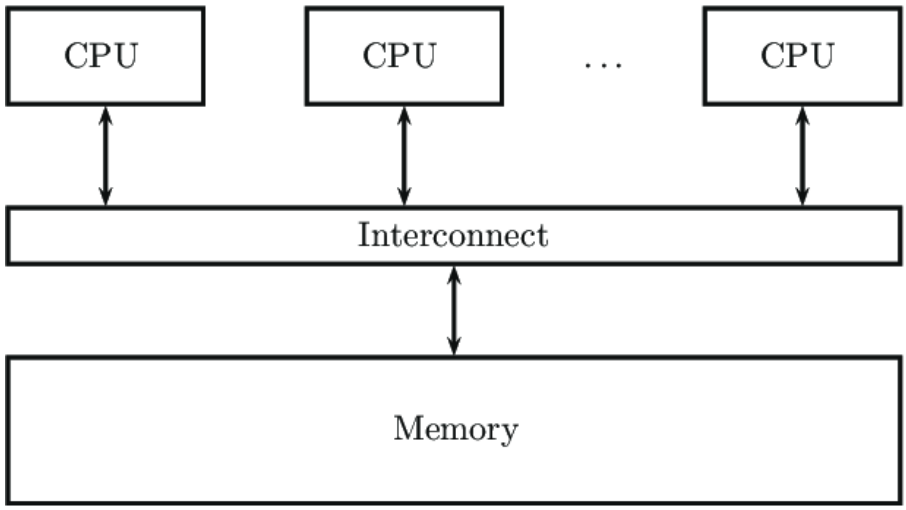
Özellik	Paralel Hesaplama	Dağıtık Hesaplama
Temel Yapı	Tek bir sistem içinde işlemcilerin koordineli çalışması	Birden fazla bağımsız bilgisayarın bir ağ üzerinde işbirliği yapması
Donanım	Çok çekirdekli CPU'lar, GPU'lar	Birden fazla düğüm (node), veri merkezleri
İletişim	Paylaşılan bellek veya yüksek hızlı veri yolu kullanımı	Ağ üzerinden mesaj tabanlı iletişim
Örnek Kullanım	Bilimsel hesaplamalar, yapay zeka, simülasyonlar	Bulut bilişim, büyük veri işleme, web hizmetleri

1.1 Paralel Hesaplama

Paralel hesaplama, birden fazla işlem biriminin aynı anda çalışarak hesaplamaları hızlandırmasıdır. Bu yaklaşımda görevler eşzamanlı çalıştırılarak işlem süresi azaltılır. Bu yaklaşımda iki farklı model öne çıkmaktadır. Bunlardan biri tüm işlemcilerin aynı belleğe eriştiği “Paylaşılan Bellek Modeli”, diğeri ise “Dağıtılmış Bellek Modeli”dir .

1.1.1 Paylaşılan Bellek Modeli

Paylaşılan bellek modeli, tüm işlemcilerin aynı bellek alanına eriştiği bir paralel hesaplama modelidir. Bu modelde, işlemciler arasında bellek paylaşımı sayesinde hızlı veri erişimi sağlanır ve iletişim gecikmeleri minimize edilir.



Özellikleri:

- Ortak Bellek Kullanımı: Tüm işlemciler aynı bellek havuzuna erişir.
- Düşük Gecikme: İşlemciler arasında mesajlaşma gerekmez, doğrudan bellek erişimi vardır.
- Senkronizasyon Mekanizmaları: Bellek çakışmalarını önlemek için mutex, semafor ve bariyerler kullanılır.
- Kolay Programlama: Mesaj tabanlı iletişime kıyasla daha kolay programlanabilir.

Senkronizasyon Mekanizmaları:

Paralel hesaplamada birden fazla işlemcinin ortak belleğe erişirken veri tutarsızlığını ve çakışmaları önlemek için senkronizasyon mekanizmaları kullanılır:

- Mutex (Mutual Exclusion - Karşılıklı Dışlama): Aynı anda sadece bir işlemcinin belirli bir kaynağa erişmesini sağlar. Örneğin, kritik bir bölgeye girerken bir mutex kilidi kullanılır ve işlem tamamlandığında kilit serbest bırakılır.
- Semafor: Birden fazla işlemciye aynı anda erişim hakkı tanıyabilir. Sayısal bir değer içerir ve belirli bir kaynak için kaç işlemin aynı anda çalışabileceğini sınırlar.
- Bariyer (Barrier): Tüm işlemcilerin belirli bir noktaya ulaşıp senkronize olmasını sağlar. Bir işlemci bariyer noktasına ulaştığında, diğerleri de bu noktaya gelene kadar bekler.

Not: Kolay Programlama (Örneği)

Kolay programlama, bir programın yazılması, anlaşılması ve bakımının yapılmasını daha az çaba gerektirecek şekilde düzenlenmesi anlamına gelir. Bu genellikle yüksek seviyeli programlama dilleri, kütüphaneler ve framework'ler kullanılarak sağlanır.

Paralel programlamada kolay programlama, mesaj geçişli (MPI) yerine paylaşılan bellek modelini kullanmak veya düşük seviyeli thread yönetimi yerine OpenMP gibi kütüphaneleri tercih etmek gibi yaklaşımlar ile mümkündür.

Örnek Kod:

```c
#include <stdio.h>
#include <omp.h>

int main() {
    int i;
    int toplam = 0;

    #pragma omp parallel for reduction(+:toplam)
    for (i = 0; i < 1000; i++) {
        toplam += i;
    }

    printf("Toplam: %d\n", toplam);
    return 0;
}
```

Paylaşılan Bellek Modelinin Avantajları ve Dezavantajları:

Avantajları:

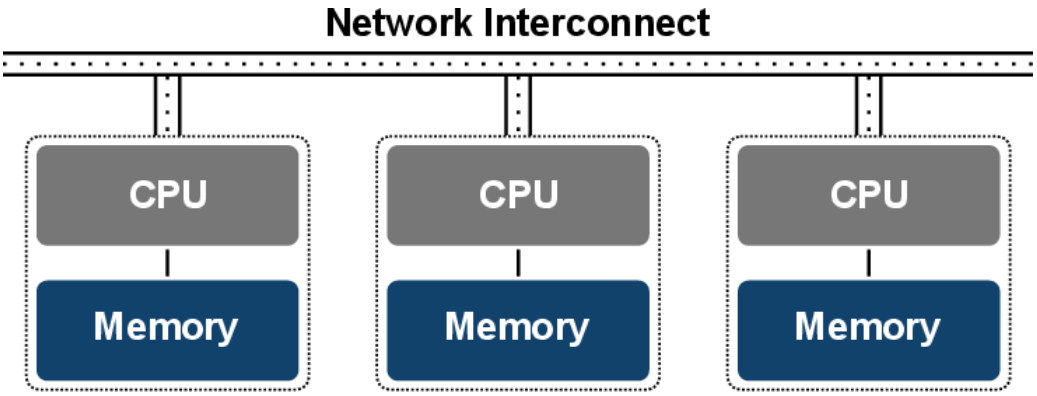
- Belleğe doğrudan erişim sayesinde yüksek hız.
- Paralel programlama dillerinde (OpenMP gibi) kolay uygulama.
- Veri paylaşımı ve iletişim basittir.

Dezavantajları:

- Bellek çakışmaları nedeniyle ölçeklenebilirlik sınırlıdır.
- Büyük sistemlerde bellek erişim noktası tıkanıklık yaratabilir.
- Paylaşılan bellek yapısı büyük ölçekli sistemler için uygun olmayabilir.

1.1.2 Dağıtılmış Bellek Modeli

Dağıtılmış bellek modeli, her işlemcinin veya düğümün (node) kendi özel belleğine sahip olduğu ve diğer işlemcilerle mesaj geçişli iletişim (Message Passing) yoluyla haberleştiği bir paralel hesaplama modelidir. Bu model, büyük ölçekli hesaplamalarda ve dağıtık sistemlerde kullanılır.



Özellikleri

- Bağımsız Bellek: Her işlemci veya düğüm, kendi yerel belleğine sahiptir ve diğerleriyle doğrudan paylaşım yapmaz.
- Mesaj Geçişli İletişim: İşlemciler, veri paylaşmak için mesajlar göndererek haberleşir (ör. MPI – Message Passing Interface).
- Ölçeklenebilirlik: Daha fazla düğüm eklenerek sistemin işlem gücü artırılabilir.
- *Ağ Bağlantısı Gerekir: İşlemciler arasındaki iletişim, Ethernet, InfiniBand gibi ağ protokolleri üzerinden sağlanır.*

Dağıtılmış Bellek Modeli Nasıl Çalışır?

Dağıtılmış bellek modelinde, işlemciler doğrudan ortak bir belleğe erişmezler. Bunun yerine, işlemciler birbirleriyle mesaj alışverişi yaparak veri paylaşır. Bir dağıtılmış bellek modelinin tipik işleyişi şöyledir:

1. Bir düğüm işlem yapar ve veriye ihtiyacı olursa başka bir düğümden talep eder.
2. İlgili düğüm veriyi hazırlar ve mesaj ile istekte bulunan düğüme gönderir.
3. Alıcı düğüm mesajı alır ve işlemine devam eder.

Dağıtılmış Bellek Modelinde Kullanılan Programlama Modelleri

Dağıtılmış bellek sistemlerinde programlama için bazı yaygın modeller ve araçlar şunlardır:

- MPI (Message Passing Interface): İşlemciler arasında mesaj geçişi ile haberleşmeyi sağlayan bir protokoldür.
- MapReduce: Büyük veri kümelerinin paralel olarak işlenmesini sağlar (Google tarafından geliştirilmiştir).
- Apache Spark: Büyük veri analitiğinde kullanılan bir dağıtık hesaplama framework'üdür.
- Hadoop: Büyük ölçekli veri işleme için yaygın kullanılan dağıtılmış sistemdir.

MPI kullanarak iki işlemci arasında mesaj gönderme ve alma işlemi:

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        int data = 100;
        MPI_Send(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Gönderici: Veriyi gönderdim: %d\\n", data);
    } else if (rank == 1) {
        int received_data;
        MPI_Recv(&received_data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Alıcı: Veriyi aldım: %d\\n", received_data);
    }

    MPI_Finalize();
    return 0;
}
```

Dağıtılmış Bellek Modelinin Avantajları – Dezavantajları:

Avantajları:

- Büyük ölçekli sistemler için uygundur.
- Bellek çakışmalarını önler. Çünkü her düğüm kendi belleğini yönetir.
- Ölçeklenebilirlik yüksektir. Yeni düğümler eklenerek sistemin kapasitesi artırılabilir.
- Hata toleransı iyidir. Bir düğüm arızalansa bile sistem çalışmaya devam edebilir.

Dezavantajları:

- İletişim gecikmeleri olabilir. Düğümler arasında mesaj geçişi gerektiği için ağ gecikmeleri yaşanabilir.
- Programlama daha karmaşıktır. Verilerin mesajlaşma ile paylaşılması gerektiğinden senkronizasyon zor olabilir.
- Ağ altyapısına bağımlıdır. Güçlü bir ağ bağlantısı gerektirir, aksi halde performans düşebilir.

1.2 Dağıtık Hesaplama

Dağıtık hesaplama, birden fazla bağımsız bilgisayarın veya düğümün (node) bir ağ üzerinden iletişim kurarak ortak bir görevi gerçekleştirmesidir. Her düğüm belirli bir görevi üstlenerek büyük ölçekli işlemleri tamamlar.

Dağıtık Hesaplamanın Temel Özellikleri:

1) Özerk Sistemler
2) Ölçeklenebilirlik
3) Hata Toleransı
4) Paralellik ve Verimlilik

Özerk Sistemler: Düğümler (node'lar) bağımsız olarak çalışır ve gerektiğinde birbirleriyle iletişim kurarak veri alışverişi yapar. Özerk sistemlerde (bağımsız düğümler) veri alışverişi, düğümlerin birbirleriyle doğrudan veya dolaylı olarak iletişim kurması ile sağlanır. Dağıtık hesaplama sistemlerinde düğümler, mesaj geçişli iletişim (Message Passing) veya paylaşılan veri depoları gibi yöntemler kullanarak veri transferi yapar.

a) Mesaj Geçişli İletişim (Message Passing):

- Düğümler doğrudan mesaj gönderme (send) ve mesaj alma (receive) işlemleri ile haberleşir.
- MPI (Message Passing Interface) protokolü kullanılarak düğümler arasında veri aktarımı yapılabilir.
- Düğümler arası güçlü izolasyon sağlar. Bununla birlikte ağ trafiği oluşturur, senkronizasyon gerektirir.

MPI ile mesaj gönderme (Düğüm 0 veriyi Düğüm 1'e gönderiyor, düğüm 1 bu veriyi alıyor):

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        int data = 100;
        MPI_Send(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Düğüm 0: Veriyi gönderdim: %d\n", data);
    } else if (rank == 1) {
        int received_data;
        MPI_Recv(&received_data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Düğüm 1: Veriyi aldım: %d\n", received_data);
    }

    MPI_Finalize();
    return 0;
}
```

b) Paylaşılan Veri Deposu (Shared Data Store)

- Düğümler ortak bir veri tabanında veya bellek alanında veri saklayabilir.
- Örnek: Apache Kafka, Redis, Cassandra gibi sistemler, verileri düğümler arasında paylaşmak için kullanılır.
- Düğümler asenkron olarak veri okuyabilir ve yazabilir. Bununla birlikte veri tutarsızlıkları ve gecikmeler yaşanabilir.

Apache Kafka ile veri paylaşımı:

```python
from kafka import KafkaProducer

producer = KafkaProducer(bootstrap_servers='localhost:9092')
producer.send('veri_kanali', b'Bu bir mesajdır')
producer.flush()
```

c) RPC (Remote Procedure Call - Uzak Prosedür Çağrısı)

- Bir düğüm, uzaktaki bir düğümün fonksiyonunu çağırarak işlem yapmasını sağlar.
- gRPC, REST API, SOAP gibi protokoller kullanılır.
- Düğümler modüler çalışabilir. Bununla birlikte ağ gecikmeleri yaşanabilir.
-

gRPC ile Uzaktan Fonksiyon Çağırma Örneği

```
import grpc

channel = grpc.insecure_channel('server_address:50051')
stub = MyServiceStub(channel)
response = stub.RemoteFunction(MyRequest(data="Merhaba Düğüm"))
```

Hangi Yöntem Ne Zaman Kullanılır?

Yöntem	Ne Zaman Kullanılır?	Örnek Kullanım
Mesaj Geçişli (MPI)	Yüksek hızlı düğüm arası veri alışverişi gerektiğinde	Süper bilgisayarlar
Paylaşılan Veri Deposu	Büyük veri kümelerinin senkronize edilmesi gerektiğinde	Apache Spark, Hadoop
RPC (gRPC, REST API)	Uzaktan işlem çağırma gerektiğinde	Mikro hizmetler (Microservices)

Ölçeklenebilirlik: Bir sistemin iş yükü arttıkça performansını koruyarak genişleyebilme yeteneğidir. Dağıtık hesaplamada ölçeklenebilirlik, yeni düğümler (node'lar) eklenerek işlem gücünün artırılabilmesi anlamına gelir. Bir sistem ölçeklenebilir ise, donanım veya yazılım değişikliği yapılmadan daha fazla iş yükünü verimli bir şekilde işleyebilir. Dağıtık hesaplama sistemlerinde iki ana ölçeklenebilirlik türü vardır:

a) Yatay Ölçeklenebilirlik (Horizontal Scaling - Scale Out)

- Yeni düğümler ekleyerek sistemin kapasitesini artırmaktır.
- Örneğin, 10 sunucu ile çalışan bir sistemin, 20 sunucuya çıkarılması yatay ölçeklenebilirliktir.

b) Dikey Ölçeklenebilirlik (Vertical Scaling - Scale Up)

- Mevcut donanımın işlemci (CPU), bellek (RAM) veya depolama (HDD/SSD) kapasitesini artırmaktır.
- Örnek: Bir sunucunun 16 GB RAM'den 64 GB RAM'e yükseltilmesi dikey ölçeklenebilirliktir.

Dağıtık Hesaplamada Ölçeklenebilirliğin Önemi: Daha Fazla İşlem Gücü: Büyük veri ve yüksek hesaplama gerektiren işlemler için kapasite artırılabilir. Ağır iş yükleri daha fazla düğüm eklenerek dengeli bir şekilde dağıtılabilir. Bir düğüm çökse bile sistem çalışmaya devam eder. İhtiyaca göre kaynak artırılıp azaltılabilir.

Ölçeklenebilirliği Etkileyen Faktörler:

- Ağ İletişimi: Düğümler arası mesaj trafiği, sistemin performansını etkileyebilir. Bunun için düşük gecikmeli ağ altyapısı (InfiniBand, RDMA) kullanılır.
- İşYükü Dengesi (Load Balancing): Tüm düğümlerin eşit iş yükü alması gerekir. Load balancer (nginx, HAProxy) kullanımı gerekir.
- Veri Tutarlılığı (Data Consistency): Çok sayıda düğümde veri tutarlılığını korumak zor olabilir. CAP Teoremi, Quorum tabanlı replikasyon uygulanmalıdır.
- Dağıtık Depolama: Büyük verileri etkili saklamak ve erişmek gereklidir. HDFS, Cassandra, Amazon S3 gibi ölçeklenebilir sistemler kullanılmalıdır.

Hata Toleransı: Bir sistemin donanım veya yazılım hatalarına rağmen çalışmaya devam edebilme yeteneğidir. Dağıtık hesaplama sistemlerinde hata toleransı, sistemin bireysel bileşenlerde oluşan arızalardan etkilenmemesi ve işleyişin sorunsuz bir şekilde devam etmesi için kritik öneme sahiptir.

Hata Toleransı Teknikleri

1. Fiziksel Yedeklilik (Redundancy): Verilerin veya bileşenlerin birden fazla kopyası tutulur. Örnek: RAID disk yapıları, yedek güç kaynakları.
2. Checkpointing (Kontrol Noktası): Sistem belirli aralıklarla çalışma durumunu kaydeder. Hata durumunda en son kontrol noktasından devam edilir. Örnek: Süper bilgisayarlarda kullanılan MPI checkpointing.
3. Replikasyon (Replication): Aynı veri farklı düğümlerde saklanır. Bir düğüm başarısız olursa, diğer düğümler bu veriyi sağlar. Örnek: HDFS (Hadoop Distributed File System), Google Spanner.
4. Load Balancing (Yük Dengeleme): İş yükü eşit olarak dağıtılır, böylece bir düğüm aşırı yüklenmez. Örnek: Nginx, HAProxy yük dengeleme sistemleri.
5. Failover ve Recovery (Yedekleme ve Kurtarma): Hata tespit edildiğinde otomatik olarak yedek düğüme geçiş yapılır. Örnek: Bulut sistemlerinde (AWS, Google Cloud) otomatik failover mekanizmaları.

Paralellik ve Verimlilik: Dağıtık hesaplamada paralellik, iş yükünün birden fazla bağımsız bilgisayar (düğüm) arasında bölünerek eşzamanlı işlenmesi anlamına gelir. Verimlilik, bu düğümlerin işlem gücünü maksimum seviyede kullanarak iş yükünü en kısa sürede tamamlamasıyla sağlanır.

a) Görev Paralelliği (Task Parallelism): Farklı düğümlerin, farklı görevleri yerine getirmesi.

- Avantaj: Her düğüm özel bir işlev üstlendiğinden iş yükü bölünerek sistem hızlandırılır.
- Dezavantaj: Görevlerin birbirine bağımlılığı arttıkça paralelleşme zorlaşır.
- Örnek: Bir web sunucusunda, bir düğüm veritabanı sorgularını işlerken, diğer düğüm web isteklerini yönetebilir.

b) Veri Paralelliği (Data Parallelism): Aynı işlem, farklı düğümler üzerinde farklı veri parçaları için eşzamanlı yürütülür.

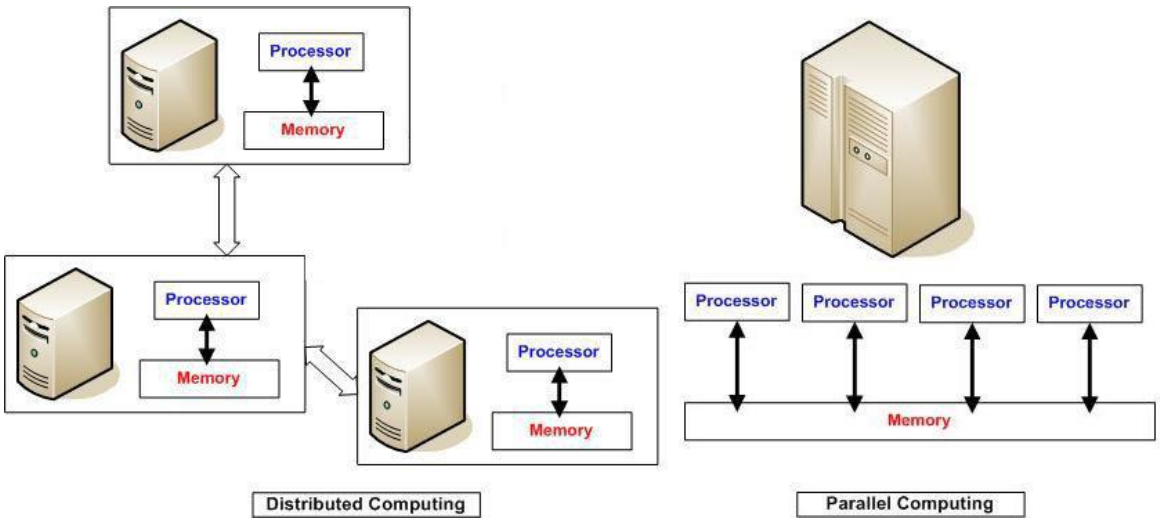
- Avantaj: Büyük veri kümeleri parçalanarak işlenebilir, böylece işlem süreleri ciddi oranda kısalır.
- Dezavantaj: Düğümler arasındaki veri aktarımı gecikmelere neden olabilir.
- Örnek: Hadoop MapReduce, büyük veri setlerini küçük parçalara bölerek her bir parçayı farklı düğümlerde işler.

c) Boru Hattı Paralelliği (Pipeline Parallelism): İşlem aşamalara ayrılır, her aşama tamamlandığında bir sonraki düğüme iletilir.

- Avantaj: Uzun süren işlemler bölündüğünde, işlem süresi azalır.
- Dezavantaj: Aşamalar bağımlı olduğundan, herhangi bir düğümde gecikme olursa tüm işlem yavaşlar.
- Örnek: Derleyiciler, kodu tokenize etme, sözdizimi analizi ve kod oluşturma gibi aşamalara ayırır.

1.3 Paralel ve Dağıtık Hesaplama Arasındaki Temel Farklar

- Paralel hesaplama genellikle tek bir sistem içinde çalışırken, dağıtık hesaplama birden fazla fiziksel sistem içerir.
- Paralel hesaplama daha düşük gecikmeli ve yüksek hızlı hesaplamalara odaklanırken, dağıtık sistemler ölçeklenebilirlik ve hata toleransı sağlar.
- Dağıtık hesaplama, ağ tabanlı iletişim gerektirirken, paralel hesaplama bellek veya veri yolu üzerinden işlem yapar.



Yukarıdaki şekilde sol tarafta dağıtık hesaplama modeli görülmektedir. Farklı düğümler (node'lar) kendi işlemcileri ve bellekleri ile bağımsız çalışarak ağ üzerinden iletişim kurmaktadır. Sağ tarafta paralel hesaplama modeli yer almaktadır. Tek bir sistem içinde birden fazla işlem birimi, ortak bir bellek üzerinden aynı anda çalışmaktadır.

2. Dağıtık Algoritmaların Sınıflandırılması

Dağıtık algoritmalar, sistemdeki düğümlerin nasıl iletişim kuracağını, karar alacağını ve senkronize olacağını belirler. Bu algoritmalar, temel olarak aşağıdaki şekilde sınıflandırılabilir:

a) Merkezi (Centralized) ve Dağıtık (Decentralized) Algoritmalar:

- Merkezi Algoritmalar: Bu tür algoritmalarda, sistemdeki tüm kararlar tek bir merkezi düğüm tarafından alınır. Merkezi algoritmalar, basit ve etkili olmalarına rağmen, merkezi düğümün hata yapması durumunda tüm sistemin çalışması olumsuz etkilenebilir.
- Dağıtık Algoritmalar: Bu algoritmalarda, kararlar sistemdeki tüm düğümler tarafından ortaklaşa alınır. Dağıtık algoritmalar, hata toleransı ve ölçeklenebilirlik açısından daha avantajlıdır.

b) Asenkron ve Senkron Algoritmalar:

- Senkron Algoritmalar: Bu tür algoritmalarda, düğümler belirli zaman adımlarında senkronize olur. Senkron algoritmalar, analiz edilmesi kolay olmasına rağmen, gerçek dünya uygulamalarında senkronizasyon sağlamak zor olabilir.
- Asenkron Algoritmalar: Asenkron algoritmalarda, düğümler herhangi bir senkronizasyon olmaksızın çalışır. Bu tür algoritmalar, gerçek dünya uygulamalarında daha yaygın olarak kullanılır.

c) Koordinasyon, Konsensus ve Seçim Algoritmaları:

- Koordinasyon Algoritmaları: Dağıtık sistemlerde, düğümlerin birbirleriyle uyumlu bir şekilde çalışmasını sağlar. Örneğin, dağıtık kilitleme algoritmaları, kaynaklara erişimi koordine eder.
- Konsensus Algoritmaları: Bu algoritmalar, sistemdeki düğümlerin ortak bir karara varmasını sağlar. Özellikle blockchain teknolojisinde kullanılan konsensus algoritmaları, sistemin güvenilirliğini artırır.
- Seçim Algoritmaları: Dağıtık sistemlerde bir lider seçmek için kullanılır. Lider seçimi, sistemdeki düğümler arasında koordinasyonu sağlamak için önemlidir.

2.1 Örnek Algoritmalar

a) Lider Seçimi Algoritması:

Bully Algoritması: En yüksek kimlik numarasına sahip düğümü lider olarak seçer.

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define ELECTION 1
#define OK 2
#define COORDINATOR 3

int main(int argc, char** argv) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    int leader = -1;
    if(rank == 2) { //Seçim başlatan düğüm, örneğin, 2 numaralı düğüm seçimi başlatsın
        printf("Düğüm %d seçim başlattı.\n", rank);
        for(int i = rank + 1; i < size; i++) {    // Daha büyük ID'ye sahip düğümlere seçim mesajı gönder
            MPI_Send(NULL, 0, MPI_INT, i, ELECTION, MPI_COMM_WORLD);
        }
        int ok_received = 0;
        MPI_Status status;
        for(int i = rank + 1; i < size; i++) {   // OK mesajı bekle
            int flag = 0;
            MPI_Iprobe(i, OK, MPI_COMM_WORLD, &flag, &status);
            if(flag) {
                MPI_Recv(NULL, 0, MPI_INT, i, OK, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                ok_received = 1;
            }
        }
        if(!ok_received) { // Eğer OK gelmediyse kendini lider ilan et
            leader = rank;
            for(int i = 0; i < size; i++) {
                MPI_Send(&leader, 1, MPI_INT, i, COORDINATOR, MPI_COMM_WORLD);
            }
            printf("Düğüm %d lider oldu.\n", leader);
        }
    }
    if(rank > 2) { // Diğer düğümler davranışı
        MPI_Recv(NULL, 0, MPI_INT, MPI_ANY_SOURCE, ELECTION, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Düğüm %d seçim mesajı aldı ve OK gönderdi.\n", rank);
        MPI_Send(NULL, 0, MPI_INT, 2, OK, MPI_COMM_WORLD);
    }
    MPI_Bcast(&leader, 1, MPI_INT, MPI_ANY_SOURCE, MPI_COMM_WORLD); // Lider ilanını tüm düğümler dinler
    if(leader != -1) {
        printf("Düğüm %d, liderin %d olduğunu öğrendi.\n", rank, leader);
    }
    MPI_Finalize();
    return 0;
}
```

Dağıtık Kilitleme Algoritması:

→Lamport'un Dağıtık Kilitleme Algoritması: Zaman damgaları kullanarak kaynaklara erişimi düzenler.

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define REQUEST 0
#define REPLY 1
#define RELEASE 2

int main(int argc, char** argv) {
    int rank, size, clock = 0;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (rank == 0) { // Broadcast Talebi
        clock++;
        printf("Düğüm %d, clock=%d ile kilit talep ediyor.\n", rank, clock);
        for (int i = 1; i < size; i++) {
            MPI_Send(&clock, 1, MPI_INT, i, REQUEST, MPI_COMM_WORLD);
        }
        for (int i = 1; i < size; i++) {   // Cevapları bekle
            MPI_Recv(&clock, 1, MPI_INT, i, REPLY, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
        printf("Düğüm %d, kilidi aldı ve kritik bölgeye girdi.\n", rank);
        printf("Düğüm %d kritik bölgeyi bitirdi.\n", rank);

        for (int i = 1; i < size; i++) {   // broadcast yayınla
            MPI_Send(&clock, 1, MPI_INT, i, RELEASE, MPI_COMM_WORLD);
        }

    } else {
        int recv_clock;
        MPI_Recv(&recv_clock, 1, MPI_INT, 0, REQUEST, MPI_COMM_WORLD, MPI_STATUS_IGNORE); // Talep dinle
        clock = (clock > recv_clock) ? clock + 1 : recv_clock + 1;
        printf("Düğüm %d, clock=%d ile kilit talebini aldı.\n", rank, clock);
        MPI_Send(&clock, 1, MPI_INT, 0, REPLY, MPI_COMM_WORLD); // Talebi kabul et (REPLY gönder)
        MPI_Recv(&recv_clock, 1, MPI_INT, 0, RELEASE, MPI_COMM_WORLD, MPI_STATUS_IGNORE); // yayını bekle
        printf("Düğüm %d, clock=%d ile kilit bırakıldığını öğrendi.\n", rank, clock);
    }

    MPI_Finalize();
    return 0;
}
```

3. Moore Yasası ve Amdahl Yasası

3.1 Moore Yasası

Moore Yasası, Intel'in kurucularından Gordon Moore tarafından 1965 yılında öne sürülen bir gözleme dayanır. Bu yasa, transistör yoğunluğunun her 18-24 ayda bir ikiye katlanacağını öngörür. Bu durum, işlemci performansının sürekli olarak artacağı anlamına gelir. Ancak, günümüzde transistör boyutlarının fiziksel sınırlarına yaklaşması nedeniyle, Moore Yasası'nın geçerliliği sorgulanmaktadır.

3.2 Amdahl Yasası

Amdahl Yasası, paralel hesaplamada bir programın hızlanmasının, seri ve paralel bileşenlerin oranına bağlı olduğunu ifade eder. Bu yasa, paralel işlemci sayısı artsa bile, programın seri kısımlarının performansı sınırlayacağını gösterir. Amdahl Yasası, paralel hesaplamanın sınırlarını anlamak için önemli bir araçtır.

Amdahl Yasası Formülü:

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

Burada:

- S: Hızlanma faktörü

- P: Programın paralelleştirilebilen kısmı
- N: İşlemci sayısı

3.3 Amdahl Yasası ile Hızlanma Analizi

Amdahl Yasası kullanılarak, bir programın paralel hesaplama ile ne kadar hızlanabileceği analiz edilebilir. Örneğin, bir programın %90'lık kısmı paralelleştirilebilir ve 10 işlemci kullanılırsa, hızlanma faktörü yaklaşık 5 kat olacaktır.

4. Paralel Programlama Modelleri

4.1 Paylaşımlı Bellek Modeli

Bu modelde, birden fazla iş parçacığı aynı bellek bölgesine erişebilir. Bir iş parçacığı tarafından yapılan değişiklikler diğer iş parçacıkları tarafından da görülebilir. İş parçacıkları, değişkenler, veri yapıları veya bellek blokları üzerinde işlem yaparak birbirleriyle iletişim kurar.

Örnek:

Bir sayacı (counter) artıran iki iş parçacığı düşünelim:

```c
#include <stdio.h>
#include <pthread.h>

#define NUM_THREADS 2
#define NUM_INCREMENTS 1000000

int counter = 0;                // Paylaşılan kaynak
pthread_mutex_t lock;           // Mutex nesnesi

void* increment(void* arg) {
    for(int i = 0; i < NUM_INCREMENTS; i++) {
        pthread_mutex_lock(&lock);   // Kilitleme
        counter++;                   // Kritik bölge
        pthread_mutex_unlock(&lock); // Kilit açma
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];
    pthread_mutex_init(&lock, NULL); // Mutex başlatma

    // İş parçacıklarını oluştur
    for(int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, increment, NULL);
    }

    // İş parçacıklarının tamamlanmasını bekle
    for(int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("Final counter value: %d\n", counter);

    pthread_mutex_destroy(&lock); // Mutex temizleme
    return 0;
}
```

Bu kodda pthread_mutex_lock kullanarak iki iş parçacığının aynı anda değişkeni değiştirmesini önlüyoruz.

4.2 Paylaşımlı Bellek Modelinde Kullanılan Araçlar

Paylaşımlı bellek modelinde veri tutarlılığını sağlamak için çeşitli araçlar kullanılır:

1. PThread (POSIX Threads):
    - C ve C++ dillerinde kullanılan iş parçacıkları kütüphanesidir.
    - pthread_create, pthread_join, pthread_mutex_lock gibi fonksiyonlar içerir.
2. OpenMP (Open Multi-Processing):
    - Paralel programlamayı kolaylaştıran bir direktif setidir.
    - #pragma omp parallel for gibi yapılarla paralel döngüler oluşturulabilir.

5.Mesaj Geçişli Model (MPI)

Mesaj geçişli modelde, bağımsız düğümler arasında veri aktarımı mesajlar aracılığıyla gerçekleştirilir. MPI (Message Passing Interface), bu modelin en yaygın kullanılan araçlarından biridir. MPI_Send ve MPI_Recv gibi temel işlemler, düğümler arasında veri göndermek ve almak için kullanılır.

OpenMP ile Çok İş Parçacıklı Programlama

```c
#include <stdio.h>
#include <omp.h>

#define NUM_INCREMENTS 1000000

int main() {
    int counter = 0;

    // OpenMP paralel bölge
    #pragma omp parallel
    {
        // Her iş parçacığı aşağıdaki döngüyü çalıştıracak
        for(int i = 0; i < NUM_INCREMENTS; i++) {
            #pragma omp critical
            {
                counter++; // Kritik bölge
            }
        }
    }

    printf("Final counter value: %d\n", counter);

    return 0;
}
```

6 Flynn'ın Sınıflandırması

Flynn'ın sınıflandırması, paralel işlemci mimarilerini temel alınarak Michael J. Flynn tarafından 1966 yılında ortaya atılmıştır. Bu sınıflandırma, komut (instruction) ve veri (data) akışı üzerinden dört temel mimariyi tanımlar. Flynn'ın sınıflandırması, komut akışı (Instruction Stream) ve veri akışı (Data Stream) olmak üzere iki temel bileşene dayanır. Buna göre dört temel işlemci mimarisi tanımlanır:

6.1. SISD (Single Instruction, Single Data) - Tek Komut, Tek Veri

Bu model, klasik tek çekirdekli bilgisayarlar için kullanılan ardışık (serial) işlem modelidir. Bir işlemci, tek bir komut akışı ile bir veri üzerinde işlem yapar. Bu yapı, geleneksel Von Neumann mimarisi ile büyük benzerlik gösterir.

Özellikleri:

- Tek bir işlemci kullanılır.
- Tek bir komut akışı ile tek bir veri işlenir.
- Seri (sıralı) işlem yapar.
- Bellek ve işlemci arasında tek bir veri yolu vardır.

Kullanım Alanları:

- Geleneksel bilgisayarlar (örneğin: PC'ler, dizüstü bilgisayarlar).
- Eski nesil işlemciler (Intel 8086 gibi).
- Mikrodenetleyiciler ve gömülü sistemler.

6.2. SIMD (Single Instruction, Multiple Data) - Tek Komut, Çoklu Veri

Bu modelde tek bir komut, birden fazla veri elemanı üzerinde aynı anda işlem yapar. Vektör işlemleri ve paralel hesaplamalar için yaygın olarak kullanılır. Grafik İşlem Birimleri (GPU'lar) ve multimedya uygulamaları için uygundur.

Özellikleri:

- Bir işlemci birimi aynı anda çok sayıda veri öğesi üzerinde işlem yapar.
- Veri paralelliği sunar.
- Yüksek performans gerektiren uygulamalar için idealdir.

Kullanım Alanları:

- Grafik İşlemciler (GPU'lar) → Oyunlar, 3D modelleme
- Multimedya İşlemleri → Ses ve görüntü işleme
- Bilimsel ve Mühendislik Hesaplamaları
- Yapay Zeka ve Derin Öğrenme Modelleri

Örnek Mimariler:

- GPU'lar (NVIDIA CUDA, AMD Radeon)
- Intel MMX (Multimedia Extensions) ve SSE (Streaming SIMD Extensions)
- ARM Neon teknolojisi

Örnek Kod: Aşağıdaki OpenMP ile yazılmış SIMD kodu, bir dizinin tüm elemanlarına aynı işlemi uygular:

```c
#include <stdio.h>
#include <omp.h>

#define N 8

int main() {
    float A[N], B[N];

    // Diziyi başlat
    for(int i = 0; i < N; i++) {
        A[i] = i * 1.0f;
    }

    // OpenMP ile SIMD tipi paralel döngü
    #pragma omp parallel for
    for(int i = 0; i < N; i++) {
        B[i] = A[i] * A[i];  // Her elemanın karesini al
    }

    // Sonuçları yazdır
    printf("A dizisi ve karesi (B dizisi):\n");
    for(int i = 0; i < N; i++) {
        printf("A[%d] = %.1f, B[%d] = %.1f\n", i, A[i], i, B[i]);
    }

    return 0;
}
```

6.3. MIMD (Multiple Instruction, Multiple Data) - Çoklu Komut, Çoklu Veri

Bu model, farklı işlemcilerin (ya da çekirdeklerin) farklı komutları çalıştırarak, farklı veri kümeleri üzerinde paralel işlem yapmasını sağlar. Modern çok çekirdekli CPU'lar ve süper bilgisayarlar bu yapıdadır.

Özellikleri:

- Birden fazla işlemci, birden fazla komutu aynı anda çalıştırır.
- Farklı veri kümeleri üzerinde işlem yapılır.
- İşlemci çekirdekleri bağımsız çalışabilir.
- Paylaşımlı veya dağıtılmış bellek yapılarıyla kullanılabilir.

Kullanım Alanları:

- Çok çekirdekli işlemciler (Multi-Core CPUs)
- Süper Bilgisayarlar
- Büyük veri analizleri (Big Data)
- Paralel veri tabanı işlemleri
- Dağıtık sistemler (Distributed Systems)

Örnek Mimariler:

- Intel Core i7, AMD Ryzen (çok çekirdekli CPU'lar)
- Süper bilgisayarlar (IBM Blue Gene, Cray XT)
- Dağıtık sistemler (Hadoop, Spark)

Örnek Kod: Aşağıdaki OpenMP kodu, farklı çekirdeklerin farklı görevleri çalıştırmasını sağlar. Burada, iki ayrı görev farklı iş parçacıkları tarafından eş zamanlı olarak çalıştırılmaktadır:

```c
#include <stdio.h>
#include <omp.h>

int main() {

    // Paralel bölge başlatılıyor
    #pragma omp parallel sections
    {
        // İlk görev
        #pragma omp section
        {
            printf("Görev 1: Thread %d tarafından yürütülüyor.\n", omp_get_thread_num());
            for(int i = 0; i < 5; i++) {
                printf("Görev 1 - Adım %d\n", i);
            }
        }

        // İkinci görev
        #pragma omp section
        {
            printf("Görev 2: Thread %d tarafından yürütülüyor.\n", omp_get_thread_num());
            for(int i = 0; i < 5; i++) {
                printf("Görev 2 - Adım %d\n", i);
            }
        }
    }

    return 0;
}
```

6.4. SPMD (Single Program, Multiple Data) - Tek Program, Çoklu Veri

Bu model, tek bir programın birden fazla işlemci veya çekirdek tarafından, farklı veri kümeleri üzerinde çalıştırılmasıdır. MIMD'nin özel bir hali olarak düşünülebilir.

Özellikleri:

- Tek bir program, farklı veri parçaları üzerinde çalıştırılır.
- Paralel hesaplamalar için yaygın olarak kullanılır.
- Genellikle MPI (Message Passing Interface) ile kullanılır.

Kullanım Alanları:

- Büyük ölçekli paralel işlemler
- Yüksek başarımlı hesaplamalar (HPC)
- Süper bilgisayarlar ve dağıtık sistemler

Örnek Mimariler:

- MPI (Message Passing Interface) kullanan süper bilgisayar sistemleri
- HPC (High-Performance Computing) sistemleri
- Paralel veri işleme yazılımları (Hadoop, Spark)

Örnek Kod (MPI ile SPMD Kullanımı):

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv) {
    int rank, size;

    // MPI ortamını başlat
    MPI_Init(&argc, &argv);

    // Düğüm kimliğini (rank) ve toplam düğüm sayısını (size) al
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Her düğüm kendi işini yapıyor
    printf("Merhaba! Ben %d numaralı düğümüm. Toplam %d düğüm var.\n", rank, size);

    // Örneğin, her düğüm sadece kendi numarasını 2 ile çarpıp sonucu yazdırsın
    int result = rank * 2;
    printf("Düğüm %d: Hesapladığım sonuç %d.\n", rank, result);

    // MPI ortamını kapat
    MPI_Finalize();

    return 0;
}
```

6.5 Uygulama: SIMD ve MIMD ile Paralel İşlemler

```c
#include <stdio.h>
#include <omp.h>

#define N 8

int main() {

    float A[N], B[N], C[N];

    // SIMD Tarzı Paralel İşlem (Veri Paralelliği)
    // A dizisinin karelerini alıp B dizisine yazıyor
    #pragma omp parallel for
    for(int i = 0; i < N; i++) {
        A[i] = i * 1.0f;
        B[i] = A[i] * A[i];
    }

    // MIMD Tarzı Paralel İşlem (Görev Paralelliği)
    // İki farklı görev tanımlanıyor
    #pragma omp parallel sections
    {
        // Görev 1: B dizisinin toplamını hesapla
        #pragma omp section
        {
            float sum = 0;
            for(int i = 0; i < N; i++) {
                sum += B[i];
            }
            printf("Görev 1: B dizisinin toplamı = %.2f\n", sum);
        }

        // Görev 2: B dizisinin ortalamasını hesapla
        #pragma omp section
        {
            float avg = 0;
            for(int i = 0; i < N; i++) {
                avg += B[i];
            }
            avg /= N;
            printf("Görev 2: B dizisinin ortalaması = %.2f\n", avg);
        }
    }

    return 0;
}
```

5. GPU vs. CPU Hesaplama

Bilgisayar mimarisinin belki de en sık karşılaştırılan iki temel yapısı CPU ve GPU'dur. Bu iki işlemci tipi, aynı işi yapmaya çalışmaz; tam tersine, farklı görevler için optimize edilmişlerdir. Çoğu zaman paralel sistemler derslerinde öğrencilere "CPU genelci, GPU uzman" gibi kısa bir cümleyle anlatılsa da bu konunun altında çok daha derin bir dünya yatar.

CPU: Genel Amaçlı İşlemci

CPU, yani Merkezi İşlem Birimi, bilgisayarın beynidir. Karmaşık komut kümelerini işlemek, kontrol akışı, dallanma (branching) ve seri işlemler gibi geniş bir yelpazede görev alır. CPU tasarımında az sayıda, ancak karmaşık ve güçlü çekirdekler bulunur. Bu çekirdekler genellikle yüksek saat frekanslarında çalışır ve geniş bir önbellek yapısıyla desteklenir. CPU'nun tasarımı, bellek erişim gecikmesini minimize etmeye ve komut akışlarını esnek bir şekilde kontrol etmeye odaklanır. Bu nedenle CPU'lar:

- Kestirilemeyen koşullu dallanmalar,
- İşlemci içi karar mekanizmaları,
- Seri olarak ilerlemesi gereken algoritmalar için idealdir.

CPU'nun temel avantajı, düşük gecikme süresi ve yüksek saat hızıdır. Fakat aynı anda binlerce iş parçacığını yürütebilecek bir mimariye sahip değildir. Modern bir CPU genellikle 4 ile 32 arasında çekirdek barındırır.

GPU: Özel Amaçlı Yüksek Paralel İşlemci

Grafik İşlem Birimi (GPU) başlangıçta 3D grafiklerin işlenmesi ve görselleştirme için tasarlanmış olsa da zamanla yüksek paralel iş yüklerini işlemek için optimize edilmiş bir mimari halini almıştır. GPU'nun temel felsefesi basittir: Aynı işlemi aynı anda birçok veriye uygula. Bu yaklaşım, özellikle matris işlemleri, vektör çarpımları, sinyal işleme ve derin öğrenme gibi yüksek veri paralelliği gerektiren alanlarda çok etkilidir.

GPU'nun yapısında yüzlerce hatta binlerce basit çekirdek bulunur. Bu çekirdekler, düşük saat frekansında çalışsalar da, birlikte çalıştıklarında büyük bir hesaplama kapasitesi oluştururlar. GPU'ların SIMD (Single Instruction Multiple Data) felsefesine dayalı bir tasarımı vardır. Yani tek bir komut seti, bir anda çok sayıda veri üzerinde çalışır.

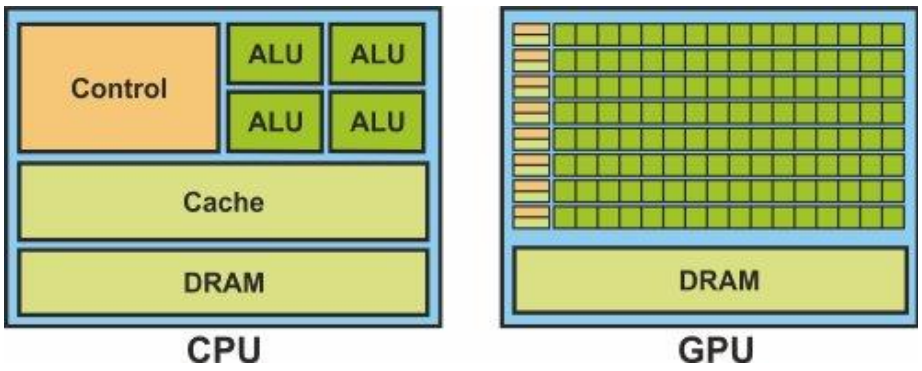
GPU'nun avantajı tam olarak burada devreye girer. Örneğin:

- Bir resimdeki tüm piksellerin aynı anda işlenmesi,
- Bir sinyalin her parçasına aynı işlemin uygulanması,
- Bir sinir ağının katmanlarındaki ağırlıkların paralel olarak güncellenmesi gibi işlemler GPU'lar sayesinde devasa hızlarda gerçekleştirilebilir.

CPU ve GPU Mimari Farklılıkları

CPU ve GPU mimarisi arasındaki temel fark, paralellik derecesidir. CPU, az sayıda güçlü ve esnek çekirdekle seri işlerde uzmanlaşırken, GPU binlerce daha basit çekirdekle veri paralelliğini ön plana çıkarır.

Bunu şekil üzerinden hayal etmek daha kolaydır:



Şekil: CPU ve GPU Mimari Yapısı

- CPU: Büyük ve karmaşık çekirdeklerden oluşan bir yapı, her biri güçlü ama az sayıda.
- GPU: Küçük, basit ama çok sayıda çekirdekten oluşan bir yapı.

Daha teknik bir gözle bakarsak, bir GPU'nun FLOPS (Floating Point Operations Per Second) değeri, aynı dönem CPU'larına göre birkaç kat daha fazladır. Ancak bu hesaplamalar genellikle çok paralel ve benzer işler içindir. Eğer görev dallanmalı ve seri bir yapıdaysa, GPU'lar verimsizleşebilir.

Matematiksel olarak bakıldığında, GPU'lar genellikle:

$$T_{GPU} = \frac{T_{CPU}}{P}$$

şeklinde bir hızlanma sunar. Burada P, GPU'daki paralel iş parçacığı sayısını temsil eder.

GPU Tabanlı Paralel İşleme: CUDA ve OpenCL

GPU'nun gücünden faydalanabilmek için CPU'nun bir işi GPU'ya aktarabilmesi gerekir. Bunun için geliştirilen en bilinen araçlardan biri NVIDIA'nın CUDA platformudur. CUDA, geliştiricilere GPU üzerinde paralel programlama imkanı sunar. CUDA dışında platformdan bağımsız bir alternatif olan OpenCL de mevcuttur.

GPU programlamada üç temel adım vardır:

1. Veri, CPU'dan (host) GPU'ya (device) kopyalanır.
2. GPU, veriyi paralel olarak işler.
3. Sonuçlar GPU'dan CPU'ya geri taşınır.

Burada dikkat edilmesi gereken önemli bir detay vardır: Veri aktarımı, özellikle büyük veri setlerinde, hesaplamanın toplam süresini etkileyen bir faktördür. Bu yüzden CUDA tabanlı optimizasyonlarda sadece hesaplama değil, veri aktarım maliyeti de dikkate alınmalıdır.

Nerede CPU Nerede GPU Kullanılır?
Bu sorunun cevabı aslında tasarım tercihine göre değişir.

Genel olarak:
- Eğer işlemler seri ve çok dallanmalı ise CPU daha uygundur.
- Eğer işlemler aynı anda çok sayıda veri üzerinde yapılacaksa, GPU daha avantajlıdır.

Örneğin:
- Bir web sunucusunun HTTP isteklerini işleyen kısmı CPU üzerinde çalışır.
- Bir makine öğrenmesi modelinin matris çarpımlarını yapan kısmı ise GPU'ya verilir.