BİLGİSAYAR MİMARİSİ VE ORGANİZASYONU

Ders Kodu : BM303
L+T+P : 0-3-1
AKTS : 4

Dersi Veren :
Prof. Dr. Ahmet Akbaş

Mühendislik Fakültesi
Bilgisayar Müh. Bölümü

13.ncü HAFTA DERSİ

Paralel Çalışma ve Performans

BİLECİK ŞEYH EDEBALİ ÜNİVERSİTESİ

2025-26 GYY DERSİ

BM303

BİLGİSAYAR MİMARİSİ ve ORGANİZASYONU

Prof. Dr. A. Akbaş

Ders İçeriği :

- Çoklu iş parçacığı
- Genel amaçlı ve grafik işlemcilerde vektör işleme
- Paylaşılan bellekli çoklu işlemciler
- Paylaşılan veriler için tutarlı önbellek
- Dağıtılmış bellekli sistemlerde mesaj geçirme
- Paralel programlama
- Performansın matematiksel modellemesi

13.ncü HAFTA DERSİ

1- Bilgisayar Sistemlerine Giriş (Fonksiyonel Birimler, Sistem Mimarisi, Performans, Teknolojik Arka Plan);

2- Programcı Bakış Açısıyla bir Sistem Turu;

3- Bilgisayar Sistemlerinde Bilginin (information) Temsili ve Manüplasyonu;

4- Komut Seti Mimarisi, Programların Makine Düzeyinde Temsili-I (RISC ve CISC mimariler);

5- Komut Seti Mimarisi, Programların Makine Düzeyinde Temsili-II (IA-32 ve x86-64 mimarileri);

6- İşlemci ve Giriş/Çıkış Birimleri Arasında Veri Transferi (Program Kontrollü G/Ç, Kesmeli G/Ç);

7- Yazılım (Program Hazırlama ve Çalıştırma Süreçleri, C ve Assembly Dilleri Arasındaki Etkileşim, İşletim Sistemleri);

8- İşlemci (CPU) Birimi (Fonksiyonel Birimler, Komut Getirme vr Yürütme Adımları, Kontrol Sinyalleri, HDL);

9- Program Performansının Optimizasyonu (Komut Düzeyinde Paralelleme, Eş-zamanlılık, Süperskaler İşlemciler);

10- Giriş/Çıkış Organizasyonu (Yol Yapısı, Yol Hakimliği, Arayüz Devreleri, Ara-bağlantı Standartları);

11- Bellek Sistemi (Temel Bellek Devreleri, Ana Bellek Organizasyonu, Bellek Teknolojileri);

12- Aritmetik (Tam Sayı ve Kayan Noktalı Sayılar için Aritmetik İşlem Devreleri);

13- Paralel Çalışma ve Performans (Çoklu İş Parçaları, Vektörel Süreçler, Paylaşımlı Bellekli Çoklu İşlemciler);

14- ARM İşlemcili SoC Örnekleri.

BİLECİK ŞEYH EDEBALİ ÜNİVERSİTESİ

2025-26 GYY DERSİ
BM303
BİLGİSAYAR MİMARİSİ ve ORGANİZASYONU
Prof. Dr. A. Akbaş

ENG. TERMINOLOGY & ACRONYMS

DONANIMSAL ÇOKLU İŞ PARÇACIĞI

Performans optimizasyonu için 9.ncu hafta dersinde, program yürütme süresini azaltan **boru hattı** ve **süperskalar** işlemlerini; 11.nci hafta dersinde de, komut ve verilere erişim süresini azaltan **önbellek kullanımı**nı görmüştük. Bu derste, performansı daha da iyileştirmek için **çoklu iş parçacığı** (**multi-threading**), **vektör işleme** ve **çoklu işleme** (**multi-processing**) yöntemlerini ele alacağız. Bu yöntemler, genel amaçlı bilgisayarlarda kullanılan **çok çekirdekli işlemci yongalarında** uygulanmaktadır. Bu yöntemlerle performans artışı, kaynak kullanımı iyileştirilerek ve paralel olarak daha fazla işlem gerçekleştirilerek sağlanır.

Her proses, **program içinde bağımsız bir yürütme yolu**na karşılık gelen bir veya daha çok **iş parçacığı** (**thread**) içerir. İş parçacıklarının durumu, **program sayacı** ve diğer **işlemci kaydedicilerinin içeriği** ile belirlenir. İki veya daha fazla iş parçacığı, farklı işlemcilerde çalışabileceği gibi; bir programın aynı bölümünü farklı veriler üzerinde yürütme veya bir programın farklı bölümlerini yürütmek üzere de atanabilir. Farklı programların iş parçacıkları da farklı işlemcilerde yürütülebilir. **Tek bir programın parçası olan tüm iş parçacıkları aynı bellek adres alanında çalışır ve aynı işlemle ilişkilendirilir**.

İşletim sistemi (**OS**), programlar arasında bağlam geçişleri gerçekleştirerek aynı işlemcinin farklı programlar için **çoklu görev** (**multitasking**) yapmasını sağlar. Bir program **OS** tarafından, mevcut yürütme durumu bilgisiyle beraber bir **proses** (**işlem**) olarak kabul edilir. **İşletim sistemi** bu kapsamda **her prosesle birlikte ona tahsis edilen bellek ve diğer kaynak bilgilerini tutar**. Örneğin bilgisayarda açılan Web tarayıcı, kelime işlemci ve müzik programı gibi her bir uygulamayı ayrı bir prosesle ilişkilendirebilir.

Bu derste, **aynı işlemcide çalışan** ve her biri tek **bir iş parçacığına sahip olan** iki veya daha fazla programla oluşan **çoklu görevler**e odaklanacağız. İşletim sistemi, engellenmemiş proseslerden birini zaman dilimleme yöntemi ile seçerek kısa bir süre çalışmasına izin verir. Bu sürede yalnızca seçilen prosese karşılık gelen iş parçacığı etkindir. İşletim sistemi bu süre sonunda bağlam değiştirir ve bir sonraki zaman diliminde ilgili iş parçacığını etkin hale getiren farklı bir prosesi seçer. Bunun için, işletim sisteminde bir zamanlayıcı kesme hizmet rutinini çağırır.

BİLECİK ŞEYH EDEBALİ ÜNİVERSİTESİ

2025-26 GYY DERSİ
BM303
BİLGİSAYAR MİMARİSİ ve ORGANİZASYONU
Prof. Dr. A. Akbaş

ENG. TERMINOLOGY & ACRONYMS

DONANIMSAL ÇOKLU İŞ PARÇACIĞI

İşlemci, birden fazla iş parçacığıyla çalışmak için, birden çok sayıda program sayacı ve özdeş kaydedici kümesi kullanır. Her kaydedici kümesi farklı bir iş parçacığına tahsis edilebilir. Böylece, bir bağlam geçişi esnasında kaydedici içerikleri kendi üzerine kaydedilerek geri yüklemede zaman kaybının önüne geçilir. Buna **donanımsal çoklu iş parçacığı** denir. Birden çok kaydedici kümesi, bağlam geçişini basitleştirir ve hızlandırır. Sonraki komutları almak ve yürütmek için farklı bir kaydedici kümesi kullanmak üzere sadece işlemcideki bir **donanım işaretçisi** değiştirilir. Farklı bir iş parçacığına geçiş, bir saat döngüsü içinde tamamlanabilir. Önceki iş parçacığının durumu kendi kaydedici kümesinde saklanır.

Farklı bir iş parçacığına geçiş, sabit bir zaman aralığının sonunda değil, belirli bir olayın meydana gelmesiyle herhangi bir anda da tetiklenebilir. Örneğin, etkin iş parçacığı için bir **Load** veya **Store** komutu yürütülürken bir **önbellek ıskalaması** meydana gelebilir. İşlemci arada geçen bu süreyi değerlendirmek için, daha yavaş olan ana belleğe erişirken duraklamak yerine, daha hızla farklı bir iş parçacığına geçerek diğer komutları alıp yürütmeye devam edebilir. Buna **kaba taneli (coarse grained) çoklu iş parçacığı** denir

Böylece, önbellek ıskalaması gibi bir olayda başka bir iş parçacığına geçmeden önce bir iş parçacığının daha çok sayıda komutu yürütülebilir.

İş parçacıkları arasında geçiş yapmanın bir alternatifi de, belirli olaylarda her komut alındıktan sonra geçiş yapmaktır. Buna **ince taneli (fine grained)** veya **iç içe geçmiş** (**interleaved**) çoklu iş parçacığı denir. Burada amaç işlemci verimini artırmaktır. Her yeni komut, diğer iş parçacıklarından gelen öncüllerden bağımsızdır.

Diğer taraftan, veri bağımlılığı nedeniyle oluşan duraklamaları azaltılmalıdır. Bu kapsamda, **birden çok iş parçacığından gelen komutlar iç içe geçirilerek** verim artırılabilir. Bu durumda belirli bir iş parçacığının tüm komutlarını tamamlaması daha uzun sürer. Intel IA-32 mimarisine sahip işlemcilerde yalnızca iki iş parçacığı için sahip bir iç içe geçmiş çoklu iş parçacığı biçimi kullanılır

Vektör İşleme: Birden çok sayıda hesaplama gerektiren bir uygulama, tamsayı veya kayan noktalı sayılar dizisi gibi veri vektörleri üzerinde işlemler gerçekleştirmek için döngüler kullanan programlar içerir.

BİLECİK ŞEYH EDEBALİ ÜNİVERSİTESİ

2025-26 GYY DERSİ
BM303
BİLGİSAYAR MİMARİSİ ve ORGANİZASYONU
Prof. Dr. A. Akbaş

ENG. TERMINOLOGY & ACRONYMS

VEKTÖR (SIMD) İŞLEME

Bir işlemci bu tür bir döngüdeki komutları yürüttüğünde, işlemler vektör elemanları üzerinde tek tek gerçekleştirilir. Sonuç olarak, tüm vektör elemanlarını işlemek için birçok komutun yürütülmesi gerekir. Bunun için işlemci birden fazla **ALU** içerecek şekilde geliştirilebilir.

Böyle bir işlemcide, tek bir komut kullanarak birden fazla veri öğesi üzerinde paralel işlem yapmak mümkündür. Bu komutlara **tek komutla çoklu veri** (**SIMD**) işleme veya **vektör komutları** denir. Bunlar paralel gerçekleştirilen işlemlerin bağımsız olması halinde kullanılabilir: **veri paralelliği**.

Vektör komutları için veriler, her biri birkaç veri öğesini tutabilen kaydedicilerde tutulur. Her vektör kaydedicisindeki öğe sayısına (**L**) **vektör uzunluğu** denir. Bu birden fazla **ALU** üzerinde paralel olarak gerçekleştirilebilecek işlem sayısıdır.

Aynı vektör kaydedicileri kullanılarak farklı boyutlardaki veri öğeleri için vektör komutları sağlanırsa, **L** değişebilir. Örneğin, Intel **IA-32** mimarisi, **L = 2**'den **L = 16**'ya kadar değişen vektör uzunlukları için komutlar tarafından kullanılan **128** bitlik vektör kaydedicilere sahiptir ve bu da **64** bitten **8** bite kadar değişen boyutlardaki tam sayı veri öğelerine karşılık gelir.

Vektör kaydedicilerinin nasıl kullanıldığını gösteren tipik vektör komutu örnekleri aşağıda verilmiştir. **OP** kodunun her veri öğesinin boyutunu belirten bir **S** eki içerdiği varsayılırsa, bir vektördeki öğe sayısı, **L**, belirlenir. Belleğe erişen komutların, etkin adres hesaplanırken geleneksel bir kaydedicinin içerikleri kullanılır. Vektör komutu

VectorAdd.S V*i*, V*j*, V*k*

Vj ve ***Vk*** vektör kaydedicilerindeki öğeleri kullanarak **L** toplamını hesaplar ve elde edilen toplamları ***Vi***'ye yerleştirir. Diğer aritmetik işlemleri gerçekleştirmek için benzer komutlar kullanılır.

Bir vektör kaydedici ile bellek arasında birden fazla veri öğesini aktarmak için özel komutlara ihtiyaç vardır.

VectorLoad.S V*i*, X(R*j*)

komutunun, ***X + [Rj]*** bellek konumundan başlayan **L** ardışık öğenin vektör kaydedici ***Vi***'ye yükler. Bunun gibi,

VectorStore.S V*i*, X(R*j*)

komutunun, vektör kaydedici ***Vi***'nin içeriğinin bellekte **L** ardışık öğe olarak depolanmasını sağlar.

BİLECİK ŞEYH EDEBALİ ÜNİVERSİTESİ

2025-26 GYY DERSİ
BM303
BİLGİSAYAR MİMARİSİ ve ORGANİZASYONU
Prof. Dr. A. Akbaş

ENG. TERMINOLOGY & ACRONYMS

VEKTÖRLEŞTİRME

Yüksek seviyeli dilde yazılmış bir kaynak programında, tam sayı veya kayan nokta sayılarından oluşan dizilerde işlem yapan döngülerin, her geçişte gerçekleştirilen işlemleri diğer geçişlerden bağımsızsa vektörleştirilebilir. Vektör komutlarının kullanılması, yürütülmesi gereken komut sayısını azaltır ve işlemlerin birden fazla **ALU**'da paralel olarak gerçekleştirilmesini sağlar. Bir vektörleştirici derleyici, çok karmaşık değillerse bu tür döngüleri tanıyabilir ve vektör komutları üretebilir.

Döngünün vektörleştirilmesi, aşağıdaki **C**-kodlu basit örnekle gösterilmiştir. **A**, **B** ve **C** dizileri için bellekteki başlangıç konumlarının **R2**, **R3** ve **R4** kaydedicilerinde olduğu varsayılırsa, derleyici, geleneksel makine dili komutlarını kullanarak yandaki şekilde gösterilen döngüyü üretebilir. Döngü gövdesinde dokuz komut vardır, dolayısıyla döngüden **N** geçiş için toplam **9N** komut yürütülür.

```
for (i = 0; i < N; i++)
    A[i] = B[i] + C[i];
```

Ref. [1]

Döngüyü vektörleştirmek için derleyici, döngüdeki her geçişteki hesaplamanın diğer geçişlerden bağımsız olduğunu ve aynı işlemin birden fazla eleman üzerinde aynı anda gerçekleştirilebileceğini kabul etmelidir. Basitleştirmek için, geçiş sayısı olan **N**'nin vektör uzunluğu olan **L**'ye eşit olarak bölünebildiğini varsayalım. Döngünün başındaki **Load**, **Add** ve **Store** komutları, aynı anda **L** eleman üzerinde işlem yapan karşılık gelen vektör komutlarıyla değiştirilir.

Sonuç olarak, vektörleştirilmiş döngü dizilerdeki tüm verileri işlemek için yalnızca **N/L** geçişi gerekir. Döngüdeki her geçişte **L** eleman işlendiğinde, **R2**, **R3** ve **R4** kaydedicilerindeki adres işaretçileri **4L** artırılır ve **R5** kaydedicisindeki sayım **L** azaltılır.

```
        Move            R5, #N          R5 is the loop counter.
LOOP:   Load            R6, (R3)        R3 points to an element in array B.
        Load            R7, (R4)        R4 points to an element in array C.
        Add             R6, R6, R7      Add a pair of elements from the arrays.
        Store           R6, (R2)        R2 points to an element in array A.
        Add             R2, R2, #4      Increment the three array pointers.
        Add             R3, R3, #4
        Add             R4, R4, #4
        Subtract        R5, R5, #1      Decrement the loop counter.
        Branch_if_[R5]> 0 LOOP          Repeat the loop if not finished.
```

Ref. [1]

VEKTÖRLEŞTİRME

Vektörleştirilmiş döngü aşağıdaki şekilde gösterilmiştir. Derleyicinin, **Add** komutlarında doğrudan operant olarak verilen ifade için **4L** değerini hesapladığını varsayıyoruz. Döngü gövdesinde hala dokuz komut vardır, ancak geçiş sayısı artık **N/L** olduğundan, yürütülmesi gereken toplam komut sayısı yalnızca **9N/L**'dir.

Vektörize edilebilir döngüler, bilgisayar grafikleri ve dijital sinyal işleme gibi uygulamalar için programlarda bulunur. Bu tür döngüler, birkaç veri öğesi üzerinde aynı anda gerçekleştirilebilen birçok bağımsız hesaplama yapar. Bir uygulama için yürütme süresinin büyük bir kısmıda bu tür döngüler yürütülüyorsa, bu döngüler vektörleştirilerek toplam yürütme süresi önemli ölçüde azaltılabilir. Performans iyileştirmesi, paralel olarak çalışabilen **ALU** sayısını belirleyen vektör uzunluğu **L** ile sınırlanır. Daha yüksek performans, başka tür bir vektör işleme desteği ile uygulanabilir.

	Move	R5, #N	R5 counts the number of elements to process.
LOOP:	VectorLoad.S	V0, (R3)	Load L elements from array B.
	VectorLoad.S	V1, (R4)	Load L elements from array C.
	VectorAdd.S	V0, V0, V1	Add L pairs of elements from the arrays.
	VectorStore.S	V0, (R2)	Store L elements to array A.
	Add	R2, R2, #4*L	Increment the array pointers by L words.
	Add	R3, R3, #4*L	
	Add	R4, R4, #4*L	
	Subtract	R5, R5, #L	Decrement the loop counter by L.
	Branch_if_[R5]> 0	LOOP	Repeat the loop if not finished.

Ref. [1]

Grafik İşleme Birimleri: Bilgisayar grafikleri için işlemeye yönelik artan talepler, grafik işleme birimi (**GPU**) adı verilen özel yongaların geliştirilmesine yol açmıştır. **GPU**'ların temel amacı, video oyunları gibi yüksek çözünürlüklü üç boyutlu grafiklerde ihtiyaç duyulan çok sayıda kayan nokta hesaplamasını hızlandırmaktır. Bu hesaplamalarda yer alan işlemler genellikle bağımsız olduğundan, büyük bir **GPU** yongası, bunları paralel olarak gerçekleştirmek için kayan nokta **ALU**'larına sahip yüzlerce basit çekirdek içerir.

GPU yongası ve bunun için ayrılmış **bellek** bir **video kartı**nda bulunur. Böyle bir kart, **PCIe** standardı gibi bir ara bağlantı standardı kullanılarak bir ana bilgisayarın genişletme yuvasına takılır. **GPU** yongasındaki işleme çekirdekleri için küçük bir program yazılır. Çok sayıda çekirdek bu programı paralel olarak yürütür.

BİLECİK ŞEYH EDEBALİ ÜNİVERSİTESİ

2025-26 GYY DERSİ
BM303
BİLGİSAYAR MİMARİSİ ve ORGANİZASYONU
Prof. Dr. A. Akbaş

ENG. TERMINOLOGY & ACRONYMS

GRAFİK İŞLEME BİRİMLERİ (GPU)

Çekirdekler aynı komutları yürütür, ancak farklı veri öğeleri üzerinde çalışır. Ayrı bir kontrol programı, ana bilgisayarın genel amaçlı işlemcisinde çalışır ve gerektiğinde **GPU** programını çağırır.

GPU hesaplaması başlamadan önce, ana bilgisayar programı önce **GPU** programının ihtiyaç duyduğu verileri ana bellekten ayrılmış **GPU** belleğine aktarır. Hesaplama tamamlandıktan sonra, ayrılmış bellekteki sonuç çıktı verileri ana belleğe geri aktarılır.

Bir **GPU** çipindeki işlem çekirdekleri, genel amaçlı bir işlemcide kullanılanlardan farklı olan özel bir komut setine ve donanım mimarisine sahiptir. Bir örnek, **NVIDIA**'nın **GPU** çiplerindeki çekirdekler için kullandığı Hesaplama Birleştirilmiş Aygıt Mimarisi'dir (Compute Unified Device Architecture, **CUDA**). Genel amaçlı bir işlemci ve bir **GPU** içeren programların yazılmasını kolaylaştırmak için **NVIDIA** tarafından **CUDA C** adı verilen bir **C** programlama dili uzantısı geliştirilmiştir. Bu uzantı, **GPU** çipindeki işlem çekirdekleri tarafından yürütülen işlevleri etiketlemek için kullanılan özel anahtar sözcüklerle tek bir programın **C** dilinde yazılmasını sağlar.

Derleyici ve ilgili yazılım araçları, son nesne programını ana bilgisayar ve **GPU** çipi için makine komutlarına çevrilen bölümlere otomatik olarak böler. Kütüphane rutinleri, **GPU** tabanlı bir video kartının ayrılmış belleğinde depolama alanı tahsis etmek ve ana bellek ile ayrılmış bellek arasında veri aktarmak için sağlanır. Ayrıca, herhangi bir satıcıdan **GPU** yongaları içeren sistemler için bir programlama çerçevesi olarak endüstri tarafından **OpenCL** adlı açık bir standart önerilmiştir .

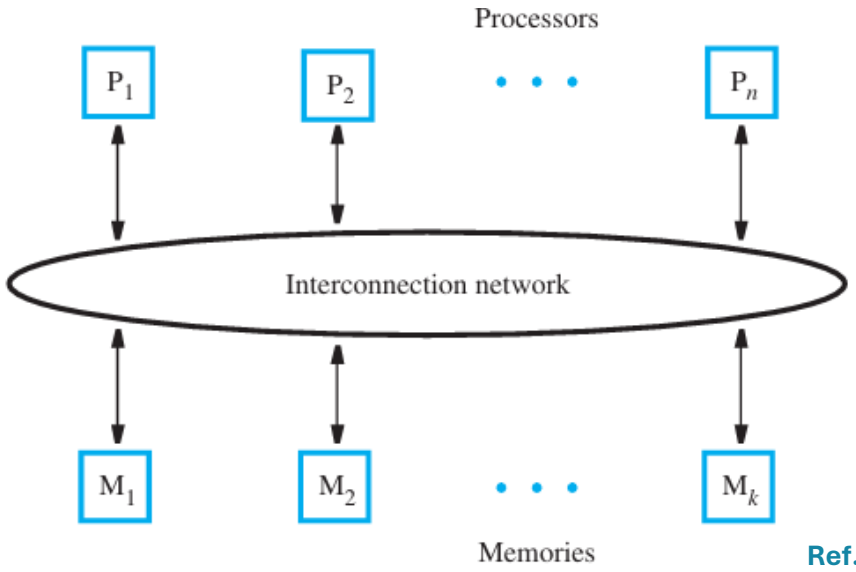
Paylaşımlı Bellekli Çok İşlemcili Sistemler: Çok işlemcili bir sistem, aynı anda bağımsız görevleri yürütebilen bir dizi işlemciden oluşur. Bu görevlerin ayrıntı düzeyi önemli ölçüde değişebilir. Bir görev, bir döngüden bir geçiş için birkaç komut veya bir alt rutinde yürütülen binlerce komut içerebilir.

Paylaşımlı bellekli çok işlemcili sistemde, tüm işlemciler aynı belleğe erişebilir. Farklı işlemcilerde çalışan görevler, aynı adresleri kullanarak bellekteki paylaşımlı değişkenlere erişebilir. Paylaşımlı belleğin boyutu büyük olabilir. Tek bir modülde büyük bir bellek uygulamak, birçok işlemcinin belleğe aynı anda erişmek için istekte bulunması durumunda bir darboğaz yaratır.

BELLEK PAYLAŞIMLI ÇOK İŞLEMCİLİ SİSTEMLER

Bu sorun, belleği birden fazla modüle dağıtarak hafifletilir, böylece farklı işlemcilerden gelen aynı anda gelen isteklerin, adreslerine bağlı olarak farklı bellek modüllerine erişme olasılığı daha yüksek olur.

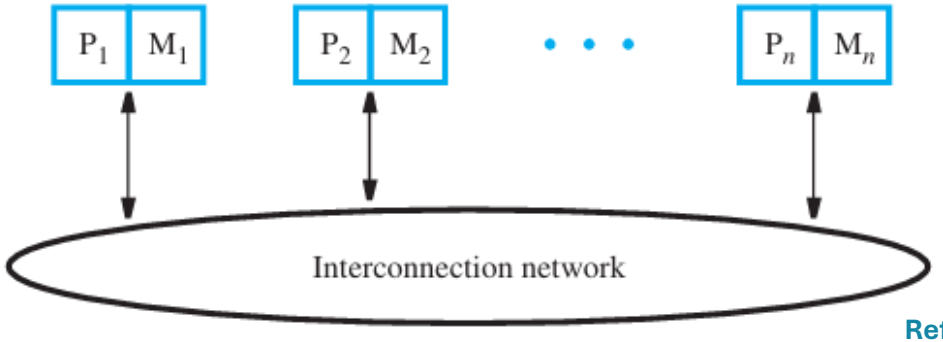
Bir bağlantı ağı, herhangi bir işlemcinin paylaşımlı belleğin bir parçası olan herhangi bir modüle erişmesini sağlar. Bellek modülleri işlemcilerden fiziksel olarak ayrı tutulduğunda, belleğe erişim isteklerinin tümü ağdan geçmek zorundadır ve bu da gecikmeye neden olur. Şekil böyle bir düzenlemeyi göstermektedir.



Ref. [1]

İşlemcilerden bellek modüllerine erişim için aynı ağ gecikmesine sahip böyle bir sisteme Tekdüzen Bellek Erişimi (Uniform Memory Accsess, **UMA**) çoklu işlemcisi denir. Gecikme tekdüze olsa da, birçok işlemci ve bellek modülünü birbirine bağlayan bir ağ için büyük olabilir.

Daha iyi performans için, her işlemciye yakın bir bellek modülü yerleştirmek tercih edilir. Sonuç, her biri bir işlemci ve bir bellek modülünden oluşan bir düğüm koleksiyonudur. Düğümler daha sonra Şekildeki gibi ağa bağlanır. Bir işlemci yerel belleğine erişim isteğinde bulunduğunda ağ gecikmesi önlenir. Ancak, uzak bir bellek modülüne erişim isteği ağdan geçmelidir. Paylaşılan belleğin yerel ve uzak bölümlerine erişim için gecikmelerdeki fark nedeniyle, bu tür sistemlere Non-Uniform Memory Accsess (**NUMA**) çoklu işlemcileri denir.



Ref. [1]

BİLECİK ŞEYH EDEBALİ ÜNİVERSİTESİ

2025-26 GYY DERSİ
BM303
BİLGİSAYAR MİMARİSİ ve ORGANİZASYONU
Prof. Dr. A. Akbaş

ENG. TERMINOLOGY & ACRONYMS

BELLEK PAYLAŞIMLI ÇOK İŞLEMCİLİ SİSTEMLER / Arabağlantı Ağları

Arabağlantı ağı, sistemdeki herhangi bir düğüm çifti arasında bilgi aktarımına izin vermelidir. Ağ ayrıca bir düğümden diğer birçok düğüme bilgi yayınlamak için de kullanılabilir. Ağdaki trafik, isteklerden (okuma ve yazma gibi) ve veri aktarımlarından oluşur.

Belirli bir ağın uygunluğu maliyet, bant genişliği, etkili çıktı ve uygulama kolaylığı açısından değerlendirilir. **Bant genişliği** terimi, bir iletim bağlantısının veri aktarma kapasitesini ifade eder ve saniyede bit veya bayt olarak ifade edilir. **Etkin çıktı**, gerçek veri aktarım hızıdır. Bu hız, mevcut bant genişliğinden daha azdır çünkü belirli bir bağlantı, veri aktarımını koordine eden kontrol bilgilerini de taşımalıdır.

Ağ üzerinden bilgi aktarımı genellikle sabit uzunlukta ve belirtilen formatta paketler şeklinde gerçekleşir. Örneğin, bir okuma isteği muhtemelen bir işlemciden bir bellek modülüne gönderilen tek bir pakettir. Paket, kaynak ve hedef için düğüm tanımlayıcılarını, okunacak konumun adresini ve ne tür bir okuma işleminin gerektiğini belirten bir komut alanını içerir.

Bir bellek modülüne bir kelime yazan bir yazma isteği, yazılacak verileri içeren tek bir paket olma olasılığı da yüksektir. Öte yandan, bir okuma yanıtı, veri aktarımı için birkaç paket gerektiren tüm bir önbellek bloğunu içerebilir.

İdeal olarak, tam bir paket, ağdaki herhangi bir düğüm veya anahtarda bir saat döngüsünde paralel olarak işlenir. Bu, birçok kabloyu içeren geniş bağlantılara sahip olmak anlamına gelir. Ancak, maliyeti ve karmaşıklığı azaltmak için bağlantılar genellikle önemli ölçüde daha dardır. Bu gibi durumlarda, bir paket, her biri bir saat döngüsünde iletilebilen daha küçük parçalara bölünmelidir. **Çoklu işlemcilerde** yaygın olarak kullanılan **ara bağlantı ağı** çeşitleri aşağıda açıklanmıştır.

Yol (bus): Bilgi aktarımı için tek bir paylaşımlı yol sağlayan bir dizi hatdır. Yollar, genellikle UMA çoklu işlemcilerinde bir dizi işlemciyi birkaç paylaşımlı bellek modülüne bağlamak için kullanılır. Herhangi bir anda birçok olası istekçiden yalnızca birine yolun kullanımına izin verilmesini sağlamak için tahkim gereklidir. Yol, yola erişim için çekişme ve birçok işlemci bağlandığında elektriksel yüklemenin neden olduğu artan yayılma gecikmeleri nedeniyle nispeten az sayıda işlemci için uygundur.

BELLEK PAYLAŞIMLI ÇOK İŞLEMCİLİ SİSTEMLER / Arabağlantı Ağları

Basit bir yol, geçerli istek için yanıt sağlanana kadar yolda yeni bir isteğin görünmesine izin vermez. Ancak, yanıt gecikmesi yüksekse, yolun önemli bir boşta kalma süresi olabilir. Bir isteğin ve karşılık gelen yanıtının ayrı olaylar olarak ele alındığı bölünmüş işlem yolunu kullanarak daha yüksek performans elde edilebilir. Bunlar arasında başka aktarımlar da gerçekleşebilir.

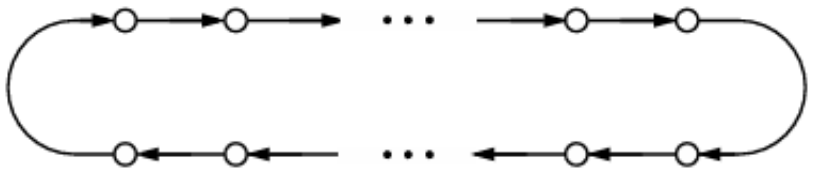
Birden fazla işlemcinin belleğe okuma istekleri yapması gereken bir durumu düşünün. **Tahkim (yol hakemliği)**, isteği için veri yolunun kullanımına izin verilecek ilk işlemciyi seçmek için kullanılır. İstek yapıldıktan sonra, yolu boşta bırakmak yerine isteğini yapmak üzere ikinci bir işlemci seçilir. Bu isteğin farklı bir bellek modülüne yapıldığı varsayılırsa, iki okuma erişimi paralel ilerler.

Hiçbir modül erişimini tamamlamamışsa, isteğini yapmak üzere üçüncü bir işlemci seçilir ve bu böyle devam eder. Sonunda, bir bellek modülü okuma erişimini tamamlar. Verileri istekte bulunan işlemciye aktarmak için yolu kullanma izni verilir. Diğer modüller erişimlerini tamamladıkça, yol yanıtlarını aktarmak için kullanılır.

Her istek ile karşılık gelen yanıtı arasındaki gerçek zaman uzunluğu, bellekle farklı işlemler için istekler ve yanıtlar, mevcut bant genişliğinin verimli bir şekilde kullanılması için yolda iç içe geçirildiğinden değişebilir.

Bölünmüş işlem yolu daha karmaşık bir yol protokolü gerektirir. Karmaşıklığın ana kaynağı, her yanıtı karşılık gelen isteğiyle eşleştirme gereksinimidir. Bu genellikle yolda görünen her istekle benzersiz bir etiket ilişkilendirilerek ele alınır. Daha sonra her yanıt, kaynağın orijinal isteğiyle eşleştirilebilmesi için uygun etiketle birlikte görüntülenir.

Ring (Halka): Düğümler arasında noktadan noktaya bağlantılarla bir halka ağı oluşturulur. Şekilde tek bir halka gösterilmiştir. Uzun tek bir halka, herhangi iki düğüm arasındaki iletişim için yüksek ortalama gecikmeyle sonuçlanır.



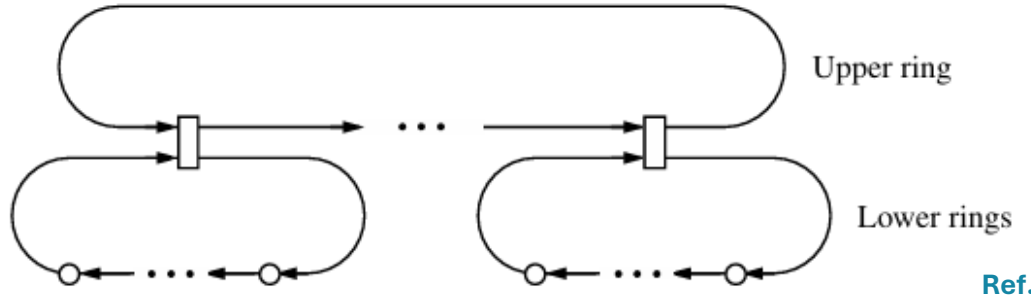
Ref. [1]

BELLEK PAYLAŞIMLI ÇOK İŞLEMCİLİ SİSTEMLER / Arabağlantı Ağları

Bu yüksek gecikme iki farklı şekilde azaltılabilir. Düğümleri zıt yönde bağlamak için ikinci bir halka eklenebilir. Ortaya çıkan çift yönlü halka, ortalama gecikmeyi yarıya indirir ve bant genişliğini iki katına çıkarır. Ancak, iletişimlerin işlenmesi daha karmaşıktır.

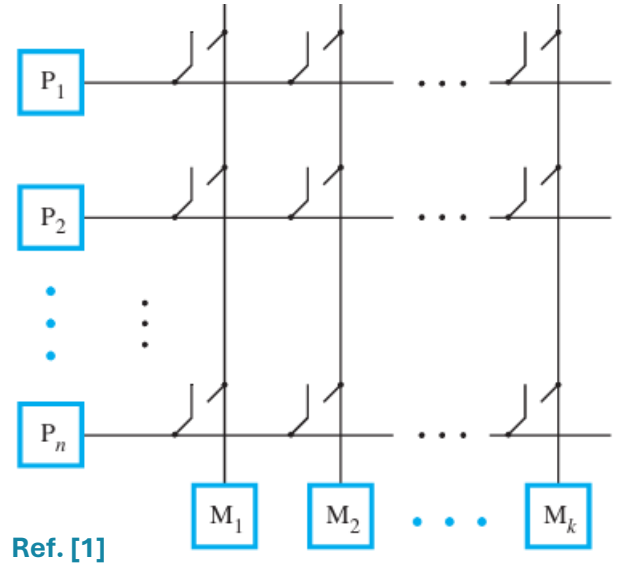
Başka bir yaklaşım, halkaların bir hiyerarşisini kullanmaktır. Şekilde iki seviyeli bir hiyerarşi gösterilmiştir. Üst seviye halka, alt seviye halkaları birbirine bağlar. Alt seviye halkalardaki herhangi iki düğüm arasındaki iletişim için ortalama gecikme, bu düzenlemeyle azaltılır. Aynı alt seviye halkadaki düğümler arasındaki transferler, üst seviye halkayı geçmek zorunda değildir. Farklı alt seviye halkalardaki düğümler arasındaki transferler, üst seviye halkanın bir kısmında bir geçişi içerir.

Hiyerarşik şemanın dezavantajı, farklı alt seviye halkalardaki birçok düğümün birbirleriyle sık sık iletişim kurması durumunda üst seviye halkanın bir darboğaz haline gelebilmesidir.



Ref. [1]

Crossbar (Çapraz çubuk): Çapraz çubuk, ağa bağlı herhangi bir birim çifti arasında doğrudan bağlantı sağlayan bir ağdır. Genellikle UMA çoklu işlemcilerinde işlemcileri bellek modüllerine bağlamak için kullanılır. Aynı hedef birden fazla isteğin hedefi değilse birçok eş zamanlı aktarıma olanak tanır. Örneğin, Şekildeki gibi bir anahtar koleksiyonundan oluşan bir çapraz çubuk kullanarak yukarıda sunulan UMA dizenindeki yapı uygulanabilir.
Burada ***n*** işlemci ve ***k*** bellek için ***n* × *k*** anahtara ihtiyaç vardır.



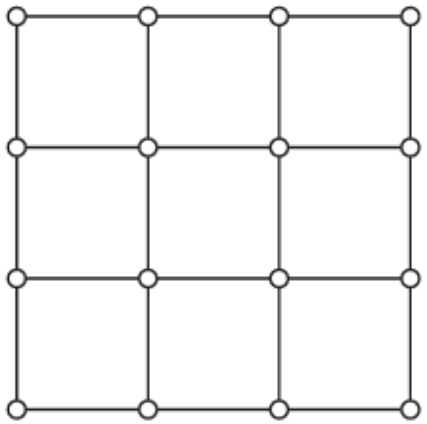
Ref. [1]

BELLEK PAYLAŞIMLI ÇOK İŞLEMCİLİ SİSTEMLER / Arabağlantı Ağları

Mesh (örgü) ağ: Çok sayıda düğümü birbirine bağlamak için şekildeki gibi 2 boyutlu bir örgülü ağ kullanılabilir. Mesh'in her bir dahili düğümü, yatay ve dikey komşularının her birine bir tane olmak üzere dört bağlantıya sahiptir.

Mesh'in sınırları ve köşelerindeki düğümler daha az komşuya ve dolayısıyla daha az bağlantıya sahiptir. Aksi takdirde mesh'te birbirinden çok uzakta olacak düğümler arasındaki iletişimin gecikmesini azaltmak için, ağın zıt sınırlarında bulunan düğümler arasında sarmal bağlantılar oluşturulabilir. Bu tür bağlantılara sahip bir ağa **torus** denir.

Bir torustaki tüm düğümlerin dört bağlantısı vardır. Ortalama gecikme azalır, ancak bir torus üzerinden istekleri ve yanıtları yönlendirmenin uygulama karmaşıklığı, basit bir mesh durumunda olduğundan biraz daha yüksektir.



Ref. [1]

Önbellek Tutarlılığı (CACHE COHERENCE): Paylaşımlı bellekli çok işlemcili bir sistemin programlanması kolaydır. Bir programdaki her değişkenin bellekte özel bir konumu vardır. Bu konuma herhangi bir işlemci erişebilir. Her işlemcinin kendi önbelleği vardır. Bu nedenle, paylaşılan verilerin kopyalarının birden fazla önbellekte bulunma olasılığı sorun teşkil eder. Öyle ki, herhangi bir işlemci kendi önbelleğindeki paylaşılan bir değişkene yeni bir değer yazdığında, bu değişkenin bir kopyasının olduğu tüm önbellekler eski, yanlış değere sahip olacaktır. Bu nedenle değişikliklerin bildirilmesi gerekir, böylece kopyaları yeni değere güncellenebilir veya geçersiz kılınabilir. Buna birden çok önbellekte paylaşılan verilerin tutarlı bir görünümüne sahip olmayı gerektiren **önbellek tutarlılığını koruma** denir.

Bir önbellekteki veriler üzerinde yazma işlemleri gerçekleştiren iki temel yaklaşımdan biri olan **yazma-geçiş** yaklaşımı hem önbellekteki hem de ana bellekteki verileri değiştirir. **Geri yazma** yaklaşımı ise, yalnızca önbellekteki verileri değiştirir; önbellekteki değiştirilmiş bir veri bloğunun değiştirilmesi gerektiğinde ana bellek kopyası güncellenir. Çok işlemcili bir sistemde önbellek tutarlılığını ele almak için benzer yaklaşımlar kullanılabilir.

ÖNBELLEK TUTARLILIĞI (CACHE COHERENCE)

Yazma Protokolü : Yazma protokolü iki şekilde uygulanabilir. Bir versiyon, diğer önbelleklerdeki değerleri güncellerken, ikinci versiyon diğer önbelleklerdeki kopyaları geçersiz kılar.

Önce **güncelleme protokolü**nü ele alalım. Bir işlemci önbelleğindeki bir veri bloğuna yeni bir değer yazdığında, yeni değer aynı zamanda değiştirilen bloğu içeren bellek modülüne de yazılır. Bu bloğun kopyaları diğer önbelleklerde mevcut olabileceğinden, bu kopyalar **yazma** işleminin neden olduğu değişikliği yansıtacak şekilde güncellenmelidir. Bunu yapmanın en basit yolu, **yazılan verileri sistemdeki tüm işlemcilerin önbelleklerine yayınlama**ktır. Her işlemci yayınlanan verileri aldığında, bu blok önbelleğinde mevcutsa etkilenen önbellek bloğunun içeriğini günceller.

Yazma protokolünün ikinci versiyonu kopyaların geçersiz kılınmasına dayanır. Bir işlem önbelleğine yeni bir değer yazdığında, **bu değer de bellekteki uygun konuma gönderilir ve diğer önbelleklerdeki tüm kopyalar geçersiz kılınır**. Bunun ardından yine, geçersiz kılma isteklerini sistem genelinde göndermek için yayınlanabilir.

Geri Yazma Protokolü : Geri yazma protokolüyle tutarlılığın sürdürülmesi, **bellekteki bir veri bloğunun sahipliği** kavramına dayanır. Başlangıçta, bellek tüm blokların sahibidir ve bellek, önbelleğine bir kopyasını yerleştirmek için bir işlemci tarafından okunan herhangi bir bloğun sahipliğini korur.

Bir işlemci önbelleğindeki bir bloğa yazmak isterse, önce bu bloğun özel sahibi olmalıdır. Bunu yapmak için, diğer önbelleklerdeki tüm kopyalar önce bir yayın isteğiyle geçersiz kılınmalıdır. Bloğun yeni sahibi daha sonra başka bir işlem yapmak zorunda kalmadan içeriği istediği gibi değiştirebilir.

Başka bir işlemci değiştirilmiş bir bloğu okumak istediğinde, blok için istek geçerli sahibine iletilmelidir. Veriler daha sonra geçerli sahibi tarafından istekte bulunan işlemciye gönderilir. Veriler ayrıca, sahipliği yeniden edinen ve bellekteki bloğun içeriğini güncelleyen uygun bellek modülüne de gönderilir. Önceki sahibi olan işlemcinin önbelleği, bloğun bir kopyasını tutar. Bu nedenle, blok artık iki önbellekteki ve bellekteki kopyalarla paylaşılır. Diğer işlemcilerden aynı bloğu okumak için gelen sonraki istekler, bloğu içeren bellek modülü tarafından karşılanır.

ÖNBELLEK TUTARLILIĞI (CACHE COHERENCE)

Geri Yazma Protokolü : Başka bir işlemci değiştirilmiş bir bloğa yazmak istediğinde, geçerli sahip verileri istekte bulunan işlemciye gönderir. Ayrıca bloğun mülkiyetini istekte bulunan işlemciye aktarır ve önbelleğe alınmış kopyasını geçersiz kılar. Blok yeni sahip tarafından değiştirildiği için, bellekteki bloğun içerikleri güncellenmez. Aynı blok için bir sonraki isteği yeni sahip karşılar.

Geri yazma protokolünün, yazma-geçiş protokolünden daha az trafik oluşturma avantajı vardır. Bunun nedeni, bir işlemcinin bu bloğa başka bir işlemci tarafından ihtiyaç duyulmadan önce bir önbellek bloğuna birkaç yazma gerçekleştirme olasılığının yüksek olmasıdır. Geri yazma protokolüyle, bu yazmalar yalnızca önbellekte gerçekleştirilir ve geçersiz kılma isteğiyle sahiplik edinilir. Yazma-geçiş protokolüyle, her yazma işlemi uygun bellek modülünde de gerçekleştirilmeli ve diğer önbelleklere yayınlanmalıdır.

Şimdiye kadar, bu protokollerdeki güncelleme ve geçersiz kılma isteklerinin ara bağlantı ağı üzerinden yayınlandığını varsaydık. Bu tür yayınların uygulanmasının kolay olup olmadığı büyük ölçüde ara bağlantı ağının yapısına bağlıdır.

Yayıncılığı desteklemek için en doğal ağ tek yoludur. Az sayıda işlemciyi tek bir yol kullanarak bellek modüllerine bağlayan çoklu işlemcilerde, önbellek tutarlılığı snooping olarak bilinen bir şema kullanılarak gerçekleştirilebilir.

Snoopy (meraklı) Önbellekler: Tek yol sisteminde, işlemciler ve bellek modülleri arasındaki tüm işlemler yoldaki istekler ve yanıtlar aracılığıyla gerçekleşir. Aslında, yola bağlı tüm birimlere yayınlanırlar. Her işlemci önbelleğinin yoldaki tüm işlemleri gözlemleyen veya gözetleyen bir denetleyici devresi olduğunu varsayalım. Şimdi geri yazma protokolü ve önbellek tutarlılığının nasıl uygulandığına ilişkin bazı senaryoları açıklayalım.

Daha önce bellekten önbelleğine bir bloğun kopyasını okuyan bir işlemci, bu bloğa ilk kez yazmadan önce, işlemci diğer tüm önbelleklere bir geçersiz kılma isteği yayınlamalıdır. Bu isteği önbellek denetleyicileri kabul eder ve aynı bloğun tüm kopyalarını geçersiz kılar. Bu eylem, istekte bulunan işlemcinin bloğun yeni sahibi olmasına neden olur. İşlemci daha sonra bloğa yazabilir ve onu değiştirilmiş olarak işaretleyebilir. Aynı işlemcinin değişmiş önbellek bloğuna yazmak için başka yayınına gerek yoktur.

ÖNBELLEK TUTARLILIĞI (CACHE COHERENCE)

Snoopy (meraklı) Önbellekler: Şimdi, başka bir işlemci aynı blok için yolda bir okuma isteği yayınlarsa, bellek yanıt vermemelidir çünkü bloğun geçerli sahibi değildir. İstenen bloğa sahip olan işlemci, yoldaki okuma isteğini dinler. Önbelleğinde istenen bloğun değiştirilmiş bir kopyasını tuttuğu için, belleğin yanıt vermesini önlemek için yola özel bir sinyal verir. Ardından, bloğun bir kopyasını yayınlar ve kopyasını değiştirilmemiş olarak işaretler. Yoldaki yanıtı, okuma isteğini yayınlayan işlemcinin önbelleği kabul eder. Bellek te, yanıtı, bloğun kopyasını güncellemek için kabul eder. Bu durumda, bellek bloğun sahipliğini yeniden kazanır ve bloğun kopyaları iki işlemcinin önbelleklerinde olduğundan bloğun paylaşımlı bir durumda olduğu söylenir. Önbelleğe alınan iki kopya ve bellekteki bloğun kopyası aynı verileri içerdiğinden tutarlılık korunur. İşlemcilerden gelen sonraki istekleri bellek karşılar.

Şimdi, iki işlemcinin kendi önbelleklerinde aynı bloğun kopyalarının bulunduğu ve her iki işlemcinin de aynı anda aynı önbellek bloğuna yazmaya çalıştığı duruma bakalım. Blok paylaşımlı olduğundan, bellek bloğun sahibidir. Bu nedenle, her iki işlemci de geçersiz kılma mesajını yayınlamak için yolu kullanmayı talep eder.

Yolun kullanımı önce işlemcilerden birine verilir. Bu işlemci geçersiz kılma talebini yayınlar ve bloğun yeni sahibi olur. Dinleme yoluyla, diğer işlemcinin önbelleğindeki bloğun kopyası geçersiz kılınır. Diğer işlemciye daha sonra yolun kullanımı verildiğinde, yalnızca okunabilir bir talep yayınlar. Bu talep, aynı blok için bir okuma talebini ve geçersiz kılma talebini birleştirir. İlk işlemcinin denetleyicisi yalnızca okunabilir talebi dinler, yolda bir veri yanıtı sağlar ve önbelleğindeki kopyayı geçersiz kılar. Bu nedenle bloğun mülkiyeti, talebi yapan ikinci işlemciye aktarılır. Blok tekrar değiştirildiği için bellek güncellenmez. İki işlemciden gelen talepler sırayla işlendiğinden, önbellek tutarlılığı korunur. Bu şema, önbellek denetleyicilerinin yolun etkinliğini gözleme ve uygun eylemi gerçekleştirme yeteneğine dayanır. Bu tür şemalara **snoopy-cache yöntemleri** denir.

Performans nedenleriyle, snooping işlevinin bir işlemcinin ve önbelleğinin normal çalışmasına müdahale etmemesi önemlidir. Çoğu kez, önbellek bir istekle ilgili bloğun geçerli bir kopyasını içermez. Gereksiz müdahale yapmamak için, her önbelleğe önbellekteki bloklar hakkında aynı durum bilgilerini koruyan ancak dinleme devresi tarafından ayrı ayrı erişilebilen bir dizi yinelenen etiket sağlanabilir.

BİLECİK ŞEYH EDEBALİ ÜNİVERSİTESİ

2025-26 GYY DERSİ
BM303
BİLGİSAYAR MİMARİSİ ve ORGANİZASYONU
Prof. Dr. A. Akbaş

ENG. TERMINOLOGY & ACRONYMS

DİZİN TABANLI ÖNBELLEK TUTARLILIĞI / MESAJ GEÇİREN ÇOKLU BİLGİSAYARLAR

Dizin Tabanlı Önbellek Tutarlılığı: Snoopy önbellekleri kavramının tek yol sistemlerinde uygulanması kolaydır. Büyük bellek paylaşımlı çok işlemcili sistemler, halka ve mesh gibi ara bağlantı ağlarını kullanır. Bu tür sistemlerde, her bir isteği tüm işlemcilerin önbelleklerine yayınlamak verimsizdir. Bu soruna ölçeklenebilir ancak daha karmaşık bir çözüm, hangi düğümlerin paylaşımlı durumda belirli bir bloğun kopyalarına sahip olabileceğini belirtmek için her bellek modülündeki dizinleri kullanır.

Bir blok değiştirilirse, dizin geçerli sahibi olan düğümü tanımlar. Bir işlemciden gelen her istek önce ilgili bloğu içeren bellek modülüne gönderilmelidir. Bu bloğa ait dizin bilgileri, gerçekleştirilen eylemi belirlemek için kullanılır. Blok değiştirilirse, okuma isteği geçerli sahibine iletilir. Paylaşılan bir blok için bir yazma isteği durumunda, tek tek geçersiz kılmalar yalnızca söz konusu bloğun kopyalarına sahip olabilecek düğümlere gönderilir. Önbellek tutarlılığını uygulamaya yönelik dizin tabanlı yaklaşımın maliyeti ve karmaşıklığı, kullanımını büyük sistemlerle sınırlar.

Günümüzdeki çok çekirdekli yongalar da dahil olmak üzere küçük çok işlemcili cihazlar genellikle dinlemeyi kullanır.

Mesaj Geçiren Çoklu Bilgisayarlar: Çoklu işlemcileri kullanmanın farklı bir yolu, sistemdeki her düğümü kendi belleğine sahip tam bir bilgisayar olarak basitleştirmektir. Sistemdeki diğer bilgisayarların bu belleğe doğrudan erişimi yoktur. Paylaşılması gereken veriler, bir bilgisayardan diğerine mesaj gönderilerek değiştirilir. Bu tür sistemlere **mesaj geçiren çoklu bilgisayarlar** denir.

Paralel programlar, mesaj geçiren çoklu bilgisayarlar için paylaşımlı bellekli çoklu işlemcilerden farklı şekilde yazılır. Düğümler arasında veri paylaşmak için, verinin kaynağı olan bilgisayarda çalışan program, hedef bilgisayara veriyi içeren bir mesaj göndermelidir. Hedef bilgisayarda çalışan program mesajı alır ve veriyi o düğümün belleğine kopyalar.

Mesaj geçişini kolaylaştırmak için, her düğümdeki özel bir iletişim birimi genellikle gönderilen ve alınan mesajların biçimlendirilmesi ve yorumlanmasının düşük seviyeli ayrıntılarından ve mesaj verilerinin düğümün belleğine kopyalanmasından sorumludur. Her düğümdeki bilgisayar, iletişim birimine komutlar verir. Bilgisayar daha sonra diğer hesaplamaları yapmaya devam ederken, iletişim birimi mesaj gönderme ve alma ayrıntılarını halleder.

BİLECİK ŞEYH EDEBALİ ÜNİVERSİTESİ

2025-26 GYY DERSİ
BM303
BİLGİSAYAR MİMARİSİ ve ORGANİZASYONU
Prof. Dr. A. Akbaş

ENG. TERMINOLOGY & ACRONYMS

ÇOKLU İŞLEMCİLER İÇİN PARALEL PROGRAMLAMA

Çoklu İşlemciler İçin Paralel Programlama:

Önceki bölümlerde, uygulama programlarında paralellikten yararlanabilen bellek paylaşımlı çoklu işlemciler için donanım düzenlemeleri açıklandı. Mevcut paralellik, bağımsız geçişlere sahip döngülerde ve ayrıca bağımsız üst düzey görevlerde bulunabilir.

Üst düzey bir dilde yazılmış bir kaynak program, bir programcının istenen hesaplamayı anlaşılması kolay bir şekilde ifade etmesine olanak tanır. Derleyici tarafından makine diline çevrilmesi gerekir. İşlemcinin donanımı, programcının istediği hesaplamayı gerçekleştirmek için makine dili komutlarını doğru sırayla yürütmek üzere tasarlanmıştır. Paralel olarak yürütülebilecek bağımsız üst düzey görevleri otomatik olarak belirleyemez.

Derleyicinin paralelliği algılama ve kullanma konusunda da sınırlamaları vardır. Bu nedenle, **kaynak programdaki genel hesaplamayı açıkça görevlere bölmek ve bunların birden fazla işlemcide nasıl yürütüleceğini belirtmek programcının sorumluluğundadır**.

Bellek paylaşımlı çok çekirdekli işlemci için programlama, tek işlemcili system programlamanın doğal bir uzantısıdır.

Yüksek seviyeli bir kaynak programı, tek bir işlemci tarafından yürütülen görevler kullanılarak yazılır. Ancak belirli görevlerin farklı işlemcilerde aynı anda yürütüleceğini belirtmek de mümkündür. Verilerin paylaşımı, atanan görevlerini gerçekleştirirken farklı işlemciler tarafından okunan ve yazılan **global değişkenleri tanımlayarak** sağlanır. Intel **IA-32** mimarisini uygulayanlar gibi genel amaçlı bilgisayarlarda şu anda kullanılan çok çekirdekli yongalar bu şekilde programlanır.

Paralel programlamayı göstermek için, her biri **N** sayı içeren iki vektörün skaler)dot) çarpımını hesaplama örneğini ele alalım. Bu görev için bir **C** dili programı şekilde gösterilmiştir. Burada iki vektörün içeriklerinin başlatılmasının ayrıntıları, paralel programlamayla ilgili yönlere odaklanmak için atlanmıştır.

Döngü, N çarpımın toplamını biriktirir. Her geçiş, önceki geçişte hesaplanan kısmi toplama bağlıdır ve son geçişte hesaplanan sonuç skaler çarpımdır. Bağımlılığa rağmen, toplamanın ilişkisel özelliğinden yararlanarak programı eş zamanlı yürütme için bağımsız görevlere bölmek mümkündür. Her görev kısmi bir toplam hesaplar ve nihai sonuç kısmi toplamların toplanmasıyla elde edilir.

ÇOKLU İŞLEMCİLER İÇİN PARALEL PROGRAMLAMA

Çoklu İşlemciler İçin Paralel Programlama:
Skaler çarpımı hesaplamak için paralel bir program uygulamada iki sorunun yanıtlanması gerekir:
• Kısmi toplamları hesaplamak için birden fazla işlemcinin paralel yürütmeye katılmasını nasıl sağlanır?
• Skaler çarpımın nihai sonucu hesaplanmadan önce her işlemcinin kısmi toplamını hesapladığından nasıl emin olunur?

```
#include    < stdio.h>      /* Routines for input/output. */

#define     N     100       /* Number of elements in each vector. */

double      a[N], b[N];     /* Vectors for computing the dot product. */

void        main (void)
{
            int i;
            double dot_product;

            < Initialize vectors a[], b[] – details omitted.>
            dot_product = 0.0;
            for (i = 0; i <  N; i++)
               dot_product = dot_product + a[i] * b[i];
            printf ("The dot product is %g\ n", dot_product);
}
```

Ref. [1]

İş Parçacığı Oluşturma: İlk soruyu cevaplamak için, farklı işlemcilere atanan görevlerin nasıl tanımlandığına ve bu görevlerin birden fazla işlemcide nasıl yürütülmeye başlatıldığına bakalım. İşlemci sayısı, **P** ve her vektördeki eleman sayısı, **N** için parametreler kullanarak skaler çarpım programının paralel bir sürümünü yazabiliriz. Basitleştirmek için **N**'nin **P**'ye eşit olarak bölünebildiğini varsayalım. Genel hesaplama, **N** çarpımın toplamını içerir. **P** işlemci için, her görevin **N/P** çarpımın kısmi toplamının hesaplanması olduğu **P** bağımsız görev tanımlarız.

Bir program tek bir işlemcide yürütüldüğünde, yürütme denetiminin etkin bir iş parçacığı vardır. Bu iş parçacığı, programın yürütülmesi başladığında işletim sistemi (OS) tarafından örtük olarak oluşturulur. Paralel bir program için, bağımsız görevlerin her işlemci için bir tane olmak üzere birden fazla yürütme denetimi iş parçacığı tarafından ayrı ayrı işlenmesini gerektirir. Bu iş parçacıkları açıkça oluşturulmalıdır. Tipik bir yaklaşım, paralel programlamayı destekleyen bir kütüphanede **create_thread** adlı bir rutin kullanmaktır. Kütüphane rutini, yeni oluşturulan iş parçacığı tarafından yürütülecek bir alt rutine işaretçi olan bir giriş parametresini kabul eder.

BİLECİK ŞEYH EDEBALİ ÜNİVERSİTESİ

2025-26 GYY DERSİ
BM303
BİLGİSAYAR MİMARİSİ ve ORGANİZASYONU

Prof. Dr. A. Akbaş

ENG. TERMINOLOGY & ACRONYMS

ÇOKLU İŞLEMCİLER İÇİN PARALEL PROGRAMLAMA

Kütüphane rutini tarafından, farklı bir yığınla yeni bir iş parçacığı oluşturmak için bir işletim sistemi hizmeti çağrılır, böylece diğer alt rutinleri çağırabilir ve kendi yerel değişkenlerine sahip olabilir. Tüm genel değişkenler tüm iş parçacıkları arasında paylaşılır.

İş parçacıklarını birbirinden ayırmak gerekir. Bir yaklaşım, **0** ile **P –1**f arasındaki her iş parçacığı için özgün bir tam sayı döndüren **get_my_thread_id** adlı başka bir kütüphane rutini sağlamaktır. Bir iş parçacığı bu bilgiyle, sorumlu olduğu hesaplama alt kümesini belirleyebilir.

İş Parçacığı Senkronizasyonu: İkinci soru, iş parçacıklarının görevlerini ne zaman tamamladığının belirlenmesini içerir, böylece nihai sonuç doğru bir şekilde hesaplanabilir. Bu nedenle birden fazla iş parçacığının senkronizasyonu gerekir. Birkaç senkronizasyon yöntemi vardır ve bunlar genellikle paralel programlama için ek kütüphane rutinlerinde uygulanır. Burada, **bariyer** adı verilen bir yöntemi ele alacağız.

Bir bariyerin amacı, iş parçacıklarının bariyer için kütüphane rutinine bir çağrının yapıldığı programdaki belirli bir noktaya ulaşana kadar beklemelerini sağlamaktır.

Bariyer rutinini çağıran her iş parçacığı, son iş parçacığı rutini çağırana ve tüm iş parçacıklarının yürütmelerine devam etmesini sağlayana kadar yoğun bekleme döngüsüne girer. Bu, iş parçacıklarının bariyer çağrısından önceki ilgili hesaplamalarını tamamladığından emin olur.

Örnek Paralel Program: İş parçacığı oluşturma ve senkronizasyonuyla ilgili sorunları ve iş parçacığı yönetimi için sağlanan tipik kütüphane rutinlerini açıkladıktan sonra, şimdi bir örnek olarak paralel skaler çarpım programını sunabiliriz. Aşağıdaki şekil, bir ana rutini ve paralel yürütme için bağımsız görevleri tanımlayan **ParallelFunction** adlı başka bir rutini gösterir. Program yürütülmeye başlayınca, ana rutini yürüten yalnız bir iş parçacığı vardır. Bu iş parçacığı vektörleri, ve ardından **bariyer senkronizasyonu** için gerekli bir paylaşılan değişkeni başlatır. Paralel yürütmeyi başlatmak için, **create_thread** rutini, her biri **ParallelFunction**'ı yürüten ek iş parçacıkları oluşturmak için ana rutinden **P–1** kez çağrılır. Ardından, ana rutini yürüten iş parçacığı **ParallelFunction**'ı doğrudan çağırır, böylece toplam **P** iş parçacığı genel hesaplamaya dahil olur. İşletim sistemi yazılımı, iş parçacıklarını paralel yürütme için farklı işlemcilere dağıtmaktan sorumludur.

BİLECİK ŞEYH EDEBALİ ÜNİVERSİTESİ

2025-26 GYY DERSİ
BM303
BİLGİSAYAR MİMARİSİ ve ORGANİZASYONU
Prof. Dr. A. Akbaş

ENG. TERMINOLOGY & ACRONYMS

ÇOKLU İŞLEMCİLER İÇİN PARALEL PROGRAMLAMA

Örnek Paralel Program: Her iş parçacığı, **0** ile **P-1** aralığında özgün bir tam sayı tanımlayıcısı elde etmek için **ParallelFunction**'dan **get_my_thread_id**'yi çağırır. Bu bilgileri kullanarak, iş parçacığı o iş parçacığının kısmi toplamını üreten döngünün başlangıç ve bitiş dizinlerini hesaplar. Döngüyü yürüttükten sonra, sonucu özgün tanımlayıcısını dizi dizini olarak kullanarak paylaşılan **partial_sums** dizisinin ayrı bir öğesine yazar. Ardından, iş parçacığı diğer iş parçacıklarının hesaplamalarını tamamlamasını beklemek için bariyer senkronizasyonu için kütüphane rutinini çağırır.

Hesaplamasını tamamlayan son iş parçacığı bariyer rutinini çağırdıktan sonra, tüm iş parçacıkları **ParallelFunction**'a geri döner. **ParallelFunction**'da gerçekleştirilecek başka bir hesaplama yoktur, bu nedenle ana rutinde kütüphane çağrısı tarafından oluşturulan **P -1** iş parçacıkları sonlanır. **ParallelFunction**'ı doğrudan ana rutinden çağıran iş parçacığı, **partial_sums** dizisindeki değerleri kullanarak nihai sonucu hesaplamak için geri döner.

```
#include   < stdio.h>          /* Routines for input/output. */
#include   "threads.h"         /* Routines for thread creation/synchronization. */

#define    N    100            /* Number of elements in each vector. */
#define    P    4              /* Number of processors for parallel execution. */

double     a[N], b[N];         /* Vectors for computing the dot product. */
double     partial_sums[P];    /* Array of results computed by threads. */
Barrier    bar;                /* Shared variable to support barrier synchronization. */

void       ParallelFunction (void)
{
           int my_id, i, start, end;
           double s;

           my_id = get_my_thread_id ();     /* Get unique identifier for this thread. */
           start = (N/P) * my_id;           /* Determine start/end using thread identifier. */
           end = (N/P) * (my_id + 1) – 1;   /* N is assumed to be evenly divisible by P .* /
           s = 0.0;
           for (i = start; i <= end; i++)
              s = s + a[i] * b[i];
           partial_sums[my_id] = s;         /* Save result in array. */
           barrier (&bar, P);               /* Synchronize with other threads. */
}

void       main (void)
{
           int i;
           double dot_product;

           < Initialize vectors a[], b[] – details omitted.>
           init_barrier (&bar);
           for (i = 1; i <  P; i++)         /* Create P – 1 additional threads. */
              create_thread (ParallelFunction);
           ParallelFunction();              /* Main thread also joins parallel execution. */
           dot_product = 0.0;               /* After barrier synchronization, compute final result. */
           for (i = 0; i <  P; i++)
              dot_product = dot_product + partial_sums[i];
           printf ("The dot product is %g\ n", dot_product);
}
```

Ref. [1]

BİLECİK ŞEYH EDEBALİ ÜNİVERSİTESİ

2025-26 GYY DERSİ
BM303
BİLGİSAYAR MİMARİSİ ve ORGANİZASYONU
Prof. Dr. A. Akbaş

ENG. TERMINOLOGY & ACRONYMS

PERFORMANS MODELLEME

Önceki şekildeki program, iş parçacığı oluşturma ve senkronizasyonunu göstermek için genel kütüphane rutinlerini kullanır. **C** dilinde paralel programlama için büyük bir rutin koleksiyonu **IEEE 1003.1** standardında tanımlanmıştır. Bu koleksiyon ayrıca **POSIX** iş parçacıkları veya **P** iş parçacığı kütüphanesi olarak da bilinir. Çeşitli iş parçacığı yönetimi ve senkronizasyon mekanizmaları sağlar. Bu kütüphanenin uygulamaları, çok işlemcili programlamayı kolaylaştırmak için yaygın olarak kullanılan işletim sistemleri için mevcuttur.

Performans Modelleme: Bilgisayarın en önemli performans ölçüsü, programları yürütme hızdır. Bir işlemciye komutların getirilip yürütüldüğü hız, komut seti mimarisi ve donanım tasarımı ile belirlenir. Yürütülen toplam komut sayısı da, derleyici ve komut seti mimarisi tarafından belirlenir. Bir işlemci için hususları belirleyen düşük seviyeli bir matematiksel performans modelini daha önce görmüştük.

Bu modeldeki terimler, yürütülen komut sayısı, komut başına ortalama döngü sayısı ve saat frekansıdır. Bu model, yeterince ayrıntılı bilgi ile yürütme süresini tahmin edebilir.

Daha az ayrıntılı bilgiye dayanan daha yüksek seviyeli bir model, performanstaki olası iyileştirmeleri değerlendirmek için kullanılabilir. Bilgisayarlarda yürütme süresi T_{orig} olan bir programı ele alalım. Amacımız, paralel işleme gibi bir performans iyileştirmesi sunulduğunda yürütme süresinin ne ölçüde azaltılabileceğini değerlendirmektir.

Yürütme süresinin bir kısmının f_{enh} 'in iyileştirmeden etkilendiğini varsayalım. Kalan kesir, $f_{unenh} = 1 - f_{enh}$, değişmez. ***p***, zamanın $f_{enh} \times T_{orig}$ kısmının performans artışı nedeniyle azaldığı faktörü temsil etsin. Yeni yürütme süresi şöyle olur: $T_{new} = T_{orig}\,(f_{unenh} + f_{enh}/p)$

Hızlanma, T_{orig}/T_{new} veya $1/\,(f_{unenh} + f_{enh}/p)$ oranıdır

Bu hızlanma ifadesi Amdahl Yasası olarak bilinir. Belirli bir performans artışının, yürütme süresinin daha büyük bir kısmını etkilemesi durumunda faydasının arttığına dair sezgisel gözlemi ifade etmenin bir yoludur.

Orijinal yürütme süresinin bir dökümü belirlendikten sonra, olası hızlanma için bir üst sınır belirlemek genellikle yararlıdır. Bunu yapmak için, p →∞'nin yürütme süresinin fenh kesrinin sıfıra ideal ancak gerçekçi olmayan azalmasını yansıtmasını sağlarız.

PERFORMANS MODELLEME

Sonuçtaki hızlanma $1/f_{unenh}$'dir, bu da geliştirilmeyen yürütme süresi kısmının performans üzerindeki sınırlayıcı faktör olduğu anlamına gelir.

f_{unenh}'in daha küçük bir değeri hızlanmada daha büyük bir sınır verir. Örneğin, $f_{unenh} = 0.1$, 10'luk bir üst sınır verir, ancak $f_{unenh} = 0.05$, 20'lik daha büyük bir sınır verir. Ancak, gerçekçi bir ***p*** değeri kullanılarak beklenen hızlanma normalde üst sınırın oldukça altındadır. Örneğin, $f_{unenh} = 0.05$ ile ***p*** = 16 kullanıldığında, yalnızca 1/(0,05 + 0,95/16) = 9,1'lik bir hızlanma elde edilir, bu da 20'lik üst sınırın oldukça altındadır.

Bu tartışmadan çıkan önemli sonuç, geliştirilmiş kısım keyfi olarak büyük bir faktörle iyileştirilmiş olsa bile, orijinal yürütme süresinin geliştirilmemiş kısmının elde edilebilir hızlanmayı önemli ölçüde sınırlayabileceğidir. Bir programcı, belirli bir geliştirmeyi uygulamadan önce yürütme süresinin f_{unenh} ve f_{enh} kesirlerini yaklaşık olarak bile belirleyebiliyorsa,
Amdahl Yasası beklenen iyileştirmeye ilişkin yararlı bir içgörü sağlayabilir.

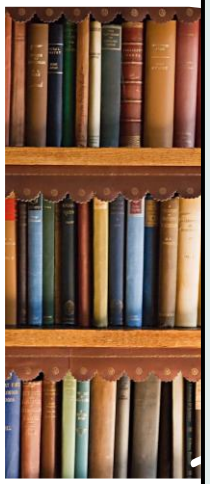
Bu bilgi, performanstaki beklenen kazanımın geliştirmeyi uygulamak için harcanan çaba ve masrafı haklı çıkarıp çıkarmadığını belirlemek için kullanılabilir.

BİLECİK ŞEYH EDEBALİ ÜNİVERSİTESİ

2025-26 GYY DERSİ
BM303
BİLGİSAYAR MİMARİSİ ve ORGANİZASYONU
Prof. Dr. A. Akbaş

Kaynaklar

REFERANSLAR



1- Hamacher, C., Vranesic, Z., Zaky, S. and Manjikian, N. (2012). *Computer Organization and Embedded Systems.* McGraw Hill, New York. ISBN 978–0–07–338065–0

2- Bryant, R.E. and O'Hallaron, D.R. (2016). *Computer Systems, A Programmer's Perspective.* Pearson Education Limited, Malaysia. ISBN 10: 1-292-10176-8

3- Mono, M.M. (2001). *Bilgisayar Sistem Mimarisi*. Literatür Yayınları, İstanbul. ISBN 975-843-31-5

4- Stallings, W. (2005). *Computer Organisation and Architecture, Designing for Performance.* Pearson, Prentice Hall, NJ. ISBN 0-13-607373-5

5- Abd-El-Barr, M. and El-Rewini, H. (2005). *Fundamentals of Computer Organization and Architecture.* Wiley-Interscience, Hoboken-New Jersy. ISBN 0-471-46741-3

6- Reynolds, C. and Tymann, P. (2008). *Principles of Computer Science.* McGraw-Hill Schaum's Outline Series, New York. 0-07-151037-0

7- https://www.wikipedia.org v.b internet siteleri

13.ncü HAFTA
DERSİNİN SONU