

19253.

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

DEĞİŞKEN BAĞIMLILIĞI VE DÖNGÜLERDE
PARALELLİK

YÜKSEK LİSANS TEZİ

Müh. Gül den ÇEVİK

Tezin Enstitüye Verildiği Tarih: 10 Haziran 1991

Tezin Savunulduğu Tarih : 4 Eylül 1991

Tez Danışmanı : Doç. Dr. Bülent ÖRENCİK

Diğer Jüri Üyeleri : Prof. Dr. M.Nadir YÜCEL

Doç. Dr. Nadia ERDOĞAN

EYLÜL 1991

T. G.
Yükseköğretim Kurulu
Dokümantasyon Merkezi
Yükseköğretim Kurulu
Dokümantasyon Merkezi
T. G.

ÖNSÖZ

Bu tez çalışmasında ele aldığım konunun bilgilerime katkısı fazla olmuştur. Konuyu veren ve katkılarıyla çalışmalarına destek olan, değerli hocam Sayın Doc. Dr. Bülent ÖRENCİK' e teşekkürü bir borç bilirim.

Ayrıca bana rahat bir çalışma ortamı sağlayan, SIEMENS-NIXDORF Teknik Müdürü Sn. Sadi ABALI'ya teşekkür ederim.

Haziran-1991

Gülten ÇEVİK

İÇİNDEKİLER

ÖZET	v
SUMMARY	vi
BÖLÜM 1. GİRİŞ	1
1.1 Paralel Bilgi İşlemeye Genel Bakış	1
1.2 Paralel Bilgi İşlemenin Tarihi	2
BÖLÜM 2. DEĞİŞKEN BAĞIMLILIKLARI	4
2.1 Temel Yaklaşım	4
2.2 Değişken Bağımlılık Tipleri	6
2.3 Basit ve Dizi Değişkenleri	7
2.4 Dizi Değişkenlerinde Veri Bağımlılıkları	8
2.4.1 Dizi Değişkenlerinde Bağımlılık Testi ..	9
2.5 İleriye ve Geriye Doğru Bağımlılık	13
2.6 Değişken Bağımlılık Grafi	13
BÖLÜM 3. SÜREÇLER VE ARALARINDAKİ SENKRONİZASYON MEKANİZMALARI	16
3.1 Temel Kavramlar	16
3.2 Paralel Çalışmaya Yönelik Komutlar	17
3.3 Süreçler Arasındaki Haberleşme Mekanizmaları	21
BÖLÜM 4. PARALEL DÖNGÜLER İÇİN SENKRONİZASYON ...	26
4.1 Temel Yaklaşım	26
4.2 Programlardaki Paralellik Düzeyleri	27
4.3 Döngülerde Paralellik	27
4.3.1 Döngü İterasyonlarını İşlemcilere Dağıtarak Paralellik	27
4.3.1.1 Senkronizasyon Yöntemleri	29
4.3.2 Döngüdeki Deyimlerin Bloklara Ayrılması Prensipleri	35
4.4 Döngü İçin Zaman ve İşlemci Üst Sınırı ..	39
4.5 Senkronizasyon Noktalarının Kaldırılması ..	42
BÖLÜM 5. UYGULAMA	47
5.1 Paralel Çeviricili Derleyiciler	47
5.2 XENIX-C Derleyicisinin Yapısı	48
5.3 Paralel Çevirici Sistemin Tasarımı	48
5.3.1 Çevirici Sistemin Modülleri	49

5.3.2 Çevirici Sistemin Çalışmasının Örnek Üzerinde Gösterilmesi.....	58
5.3.3 Sınırlamalar ve Kabuller	62
SONUÇLAR VE ÖNERİLER	66
KAYNAKLAR	68
DZGEÇMİŞ	70



ÖZET

Çok işlemcili sistemlerin kullanma alanlarından biri, büyük kapsamlı işlerin yürütülmesi ve sistemin tek bir programa tahsis edilmesidir. Bu durumda, tüm işlemciler aynı programın alt süreçlerini yürütürler.

Ana problem, ardışıl yapıdaki programlardaki paralellizmi algılayan bir yöntemin geliştirilmesidir. Bunun için, deyimler arasındaki değişken bağımlılık kurallarından faydalanarak, değişken bağımlılık grafi elde edilir. Bu graf programdaki veri akışını gösterir. Elde edilen değişken bağımlılık grafindan, eş anlı yürüyebilecek deyimleri belirleyen yöntemlerden biri seçilir. Her alt süreç, ürettiği sonuçları diğerlerine aktarabilmesi için süreçler arasındaki senkronizasyon mekanizmaları uygulanır. Çok işlemcili sistemlerin bu kullanım şeklinde, senkronizasyonun donanım düzeyinde çözmek hız kazandırır.

Çalışma nümerik yapıdaki programlar üzerinde yoğunlaştırılmıştır. Nümerik programlarda, en büyük zaman aritmetik ifadelerden oluşan döngülerde harcandığından, hız kazancı elde etmek için döngü parçalanarak işlemcilere dağıtılır.

SUMMARY

DATA DEPENDENCY AND PARALLELISM ON LOOPS

Several commercial multiprocessors are now available, ranging from computers with multiple microcomputers to computers with multiple vector processors. One use for these multiprocessors is multiprogramming. In multiprogramming processors work on separate jobs. This is effective in interactive environments, where the most jobs are small and response time is critical to user satisfaction.

Another use is to improve the turnaround of large jobs. This requires the application of several processors to a single program.

Because humans tend to think sequentially rather than concurrently, program development is mostly done in a sequential language such as Algol. While the resulting programs are very quickly on scalar machines, they are often incapable of directly making effective use of parallel processors.

The only language support for parallel processing is a set of very simple language primitives or system calls. As a result the programmer is responsible for explicitly handling all synchronization. The problem with this approach is that concurrent programming is for scientific programmers. In addition to write optimal programs on parallel architectures it is necessary to understand the architecture. There is another method for using the parallel architecture by programming, the automatic translation. Parallel programming is simply too hard to be done extremely in parallel. Instead a good system should free the programmer from having to understand the intricacies of programming with synchronization constructs. If the programmer writes sequential code where the parallelism is fairly straightforward, the programming system should discover that and rewrite the program using system synchronization primitives. We suggest the programmer uses algorithms that are fairly well tailored to parallel machines. The programming manual should document all points for this translation.

There are several methods for program decomposition. The resulting sub jobs are executed in several processors. Several vendors (such as CRAY and Sequent) have software that will accept user directives to identify the DO loops that should be executed concurrently. Such a loop is executed by assigning each iteration to a different processor. Another vendor (Alliant) has compiler that automatically detects concurrent loops and generates parallel code.

There are three levels of parallelism for a single job :

1) Parallelism at subroutine level will execute multiple subroutine calls concurrently.

2) Parallelism at the basic block level. This executes several operations from a single block of assignment statements in parallel. This type of parallelism is realized by compilers for computers with multiple functional units (Control Data 6600).

3) Parallelism at loop level will execute several iterations from a DO loop in parallel. Vectorizing compilers use this to generate vector code.

This work focuses on loops.

When a loop is completely parallel, the generation of concurrent code is easy. Each iteration is independent of other iterations. So no synchronization between iterations is necessary. However, data is often passed from one iteration to another. This requires synchronization. Because synchronization between iterations will affect the speed of the generated code, the placement of synchronization points must be done carefully. The concept of data dependence determine when concurrent execution of a loop requires synchronization.

Basically there are three types of data dependence .

1) True dependence : The value of A computed in S1 is used in S2. So S2 cannot be executed before S1. S2 depends on S1.

```
S1: A=B+C  
S2: D=A
```

2) Anti dependence : The value used in S1 must be fetched before the variable A is assigned in S2. S2 cannot be

executed before S1 and S2, depends on S1.

```
S1: D=A
S2: A=B+C
```

3) Output dependence : The assignment to D in S1 must occur before assignment in S3. S1 cannot be executed after S3, so S3 depends on S1.

```
S1: D=A
S2: if(E>0) then
S3:   D=A+E
    endif
```

Data dependence relations between statements are often represented by a data-dependence graph. Nodes of the graph represent statements and the arcs represent order constraints. This doesn't only present a convenient graphical representation of the data dependence relations, it also provides a theoretical framework for discovering parallelism. Many graph algorithms are applicable to this task.

A data dependence graph shows the data flow between statements. One important point is, data may flow between loop iterations.

Example :

```
Do 10 I=2,N
S1: B(I)=B(I)*2
S2: A(I)=A(I-1)*B(I)+C(I)
10 CONTINUE
```

The element of A computed in iteration I, is used during the next iteration, I+1.

To find loops that can be computed in concurrent mode, the compiler looks for a graph where no dependency crosses an iteration. The synchronization-free cases occur frequently enough that the simplest detection mechanisms are often sufficient. If there are dependencies that cross from one iteration to another, synchronization between iterations is needed to execute the loop concurrently.

Dependency can be classified as forward and backward. A lexically forward dependency is a dependency from a statement S_r to a statement S_k that occurs later in the loop ($k > r$). A lexically backward dependency goes to an earlier statement or to the same statement ($k \leq r$). Some dependency relations can be changed from lexically backward to forward by statement reordering.

Compilers that automatically detect concurrency in loops use one of several synchronization methods. Some of these methods are listed below.

1) One method is to synchronize on every data dependency (random synchronization). In this method the compiler looks for data dependence that cross from one iteration to another and add an appropriate synchronization primitive for each such dependency. Random placement of synchronization is very flexible but may require many synchronization points. Depending on how the system implements synchronization, this may require too many synchronization registers.

There are two methods to improve the speedup of random synchronization. One is to reorder the statements to improve overlap and the other is to eliminate covered dependencies.

2) Another method is to divide the loop into segments. Synchronization is added so segment *S* of iteration *i* is executed only after segment *s* of iteration *i-1* is completed. The iterations are pipelined through the processors. The segments are created so all data dependency relationships stay in the same pipe-line segment or go to a lexically later segment. Pipe-line method only allows lexically forward dependencies.

Random is synchronization is very flexible but optimizing this strategy is very difficult in the general case. Pipeline synchronization is more restricted, but is easier to implement. The number of synchronization points is controlled. With random synchronization the number of synchronization points may grow uncontrollably. However because of segmentation, pipelining may unnecessarily break code that need not to be synchronized.

3) A third method is to place barriers at various points of the loop. No iteration can pass beyond the barrier until all iterations reach the barrier. This strategy also allows lexically forward dependencies. When the only dependence relation is lexically forward, barrier synchronization will allow more parallelism than pipelining.

Problems with barrier synchronization occur when temporary variables are used in the loop. If a temporary variable is used in a statement and this statement is executed for all the iterations the value for the single iteration is lost. This problem can be used by using iteration local dimensional variables. A compiler that translates serial loops into concurrent loop will recognize the need for such iteration local variables.

4) A fourth method is to find sections of loop where communication is concentrated and make this sections in to critical sections. An advantage of critical sections is they handle backward dependencies. The compiler's goal is to insert as few critical sections as necessary and make them as small as possible. Because speedup is limited by the size of the largest critical section.

When there are only a few lexically backward dependencies, critical sections provide as much parallelism as random synchronization.

5) Another method is to devide the loop according to a specific algorithm in to blocks and execute them for all iterations on seperate processors. This algorithm specifies the blocks which can execute concurrently. This method also handles backward dependencies.

Powerful compiler systems, such as Parafrase Analyzer at University of Illinois , PFC at Rice University, Ptran at IBM use any of these mechanisms.

BÖLÜM 1. GİRİŞ

Günümüzde, çok işlemcili bir dizi bilgisayar sistemleri bulunmaktadır. Bu sistemlerin kullanılması alanlarından biri çoklu programlamadır. Yani işlemcilere farklı işler dağıtılır. Bu daha çok, işlerin küçük ve kullanıcıya cevap süresinin önem kazandığı durumlarda etkindir. Halbuki bu çok işlemcili sistemler, büyük kapsamlı işlerin yürütülme süresinin kısaltılmasında da kullanılırlar. Bu çalışma yönteminde, tüm işlemciler aynı programın farklı alt süreçlerini yürütürler. Bu tez çalışmasında, çok işlemcili sistemlerin bu çalışma yöntemi üzerinde durulmuş ve seri yapıdaki programların işlemcilere dağıtma problemi incelenmiştir.

1.1 Paralel Bilgi İşlemeye Genel Bakış

İnsanların düşünme tarzları paralelden çok seriye daha yatkın olduğundan, program geliştirme de doğal olarak FORTRAN, ALGOL gibi ardışıklık özelliği gösteren dillerde yapılır. Sonuçta oluşan programlar, skaler yapıdaki, sistemlerde oldukça hızlı iken çok işlemcili bir sistemde paralellik özelliğini kullanamadıklarından dolayı etkin olamazlar. Bazı dillerde paralel işleme ile ilgili komutlar bulunur. Ancak bu komutlar pek gelişmiş olmadığından programcı senkronizasyon ile doğrudan ilgilenmek zorunda kalır. Bu durumda uzun ve karışık programlar, ancak uzmanlar tarafından yazılabilir.

Bu tip programların yazılmasındaki problemler açıktır : Ölümçül kilitlenme, Kaynak yarışları, senkronizasyon gecikmeleri vb. Bunun dışında optimum program yazabilmek için paralel mimariyi iyi anlamak gerekir. Programları paralel çalıştırmanın diğer bir yöntemi de otomatik

paralelleştirme teknikleridir. Ancak otomatik teknikler dendiğinde, kullanıcının paralellliği göz ardı edeceği anlamına gelmez. Paralel programların tam otomatik tekniklerle yapılması şu anki tekniklerle çok zordur. Otomatik sistemlerin getirdiği en önemli özellik kullanıcının senkronizasyon noktaları ile ilgilenmek zorunda kalmaması. Böyle bir sistem, yazılan programlardaki belirgin paralellliği ortaya çıkararak kaynak koduna senkronizasyon primitiflerini ekler. Yani kullanıcı, örneğin bilinen FORTRAN dilini kullanarak paralellliğe uygun algoritmalar yazar. Paralel derleyici bu kaynak kodunu inceleyerek paralel bir kod üretir. Bu durumda kullanıcı nasıl program yazması gerektiği konusunda bilinçlendirilmelidir. Bu yaklaşım CRAY RESEARCH kurumu tarafından uygulanarak, CRAY 1 için vektörel FORTRAN derleyicisi geliştirildi. Nümerik çalışmalarda en büyük zaman döngülerde geçtiğinden, geliştirilen derleyicide daha çok döngülerin paralelleştirilmesi üzerinde duruldu. Bu derleyicinin kullanma kitabında, FORTRAN dili yanında derleyicinin vektörel biçime dönüştürebileceği döngü tipleri hakkında ayrıntılı bilgiler yer alıyordu. Kullanıcı bu kurallara uygun kodlar yazdığında hızlı vektörel donanımı kullanmış oluyordu. CRAY RESEARCH kurumu dışında ALLIANT isimli üretici firma , otomatik olarak kaynak kodundaki paralellliği algılayarak, döngüleri paralelleştiren bir derleyici geliştirdi.

1.2 Paralel Bilgi İşlemenin Tarihi

Belirli bir anda birden fazla işlemin yürütülmesi fikri yaklaşık 130 yıldır mevcut olduğu söylenebilir [1]. Yakın tarihte, 1940 yılları sonlarında Stibitz ve Williams tarafından geliştirilen "BELL TEL. LABORATORIES MODEL V" sisteminin iki işlemcisi vardı. 1950 yılında, kısmi diferansiyel eşitsizliklerin çözümü için geliştirilen makinada , aynı anda birden fazla işlem yapabilme yeteneği mevcuttu. 1952 yılında Leondes ve

Rubinoﬀ drum bellek ile alıřan, oklu iřleme zellięi bulunan bir iřlemci geliřtirmiřti. Paralel iřleyen dięer bir bilgisayar SOLOMON bilgisayarıydı (1962). Bu sistem birbirine eřit ve bir kontrol birimi tarafından ynetilen bir dizi kk bilgisayardan oluřuyordu. Bu kk bilgisayarlara iřlem modl deniliyordu. iřlem modlleri kendi aralarında haberleřme yetenekleri olduęu gibi, herbirinin kendi belleęi ve merkezi iřlem birimi vardı. Belirli bir zamanda, iřlem modl ya hareketsiz yada ynetici birimden daęıtılan komutlardan birini iřlemekteydi. Bu sistem birbirinden baęımsız bir dizi iřlemin, rneęin diferansiyel eřitsizliklerin zmnde kullanılıyordu.



BÖLÜM 2. DEĞİŞKEN BAĞIMLILIKLARI

Ardışıl yapıdaki programlarda paralellüğün algılanmasının temeli değişken bağımlılığıdır. En genel halde iki deyim arasındaki bağımlılık, bu iki deyimin birbirine göre varsa, yürütülme sırasını gösterir. Değişken bağımlılık kuralları uygulanarak iki deyimin eş anlamlı yürüyebileceği sonucu da çıkar. Dizi ve basit değişkenlerin bağımlılıklarının araştırılmasında farklı yaklaşımlar uygulanır.

2.1 Temel Yaklaşım

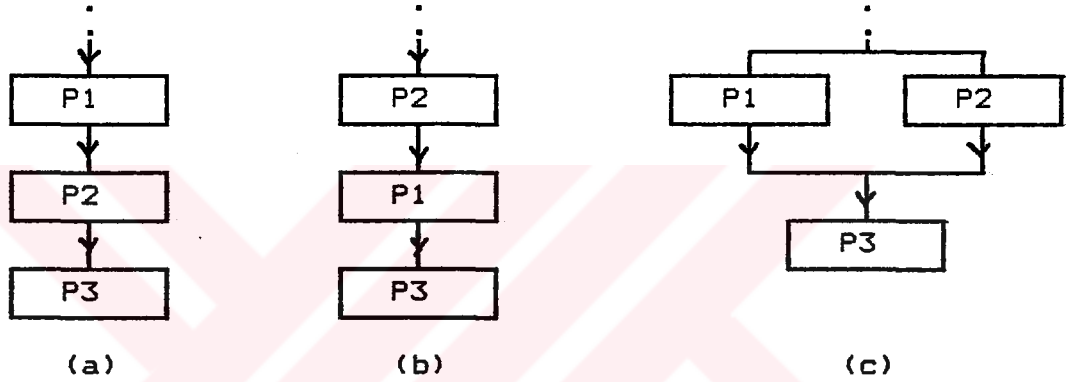
Şekil 2.1 (a) da ardışıl olarak organize edilmiş bir programın akış şeması bulunur. Bu şekil P1 altprogramı yürütüldükten sonra, P2 ardından da P3 altprogramının yürütülmesi gerektiğini gösterir [2]. Bazı durumlarda programın doğru olarak çalışabilmesi için, bu yürütülme sırası takip edilmelidir. Yani P2 program parçasının, P1 programının ürettiği sonuçlara ihtiyacı vardır. Bu gibi programlara ardışıl programlar denir. Diğer bir durumda P1 ve P2 tamamen birbirinden bağımsız program blokları olabilir. Böyle bir bağımsızlık söz konusu ise, P1 ve P2 aynı anda paralel olarak yürütülebilir ve sürecin çalışması şekil 2.1 (c) deki gibi sembole edilir. P1 ve P2 farklı işlemcilerde koşturulur ve sonuçta elde edilen veriler, P1 ve P2 nin aynı işlemcide koşturulması durumu ile aynıdır. Bir üçüncü durum olarak P1 ve P2, ne ardışıl ne de paraleldir. P1 ve P2 hem şekil 2.1 (a) daki hemde şekil 2.1 (b) deki sırada çalıştırılabilir. Programı ürettiği sonuçlar değişmeksizin program bloklarının yürütülme sırası değişebilir. Bu tip programlara komutatif program blokları denir.

Veri bağımlılığı sıkı bağlı sistemler üzerinde

incelenenecektir.

Bir P program bloğu yada bir komut, bir bellek bölgesini aşağıda açıklandığı gibi dört şekilde kullanabilir.

- 1) P çalıştığında bellek bölgesi yalnız okunur.
- 2) P çalıştığında bellek bölgesine yazılır.
- 3) P çalıştığında ilk olarak bellek bölgesine okunarak erişilir. P' nin sonraki işlemlerinde bu bellek bölgesine yazılır.



Şekil 2.1 Program Parçalarının Farklı Yürütülme Sıraları

- 4) P çalıştığında ilk olarak bellek bölgesine yazılır. P' nin sonraki işlemlerinde bu bellek bölgesi okunur. W_i , X_i , Y_i ve Z_i sırayla yukardaki dört şekle karşı gelen bellek bölgeleri olsunlar.

Sıkı bağlı sistemde P1 ve P2 nin şekil 2.1 (c) deki gibi paralel çalışabilmeleri için :

$$(W_1 \cup Y_1) \cap (X_2 \cup Y_2 \cup Z_2) = \emptyset \quad (2.1)$$

P2' nin , P1' in sonrakı erişmek üzere hesapladığı sonuçları bozmaması için :

$$Z_1 \cap (X_2 \cup Y_2 \cup Z_2) = \emptyset \quad (2.2)$$

(2.1) ve (2.2) birleştirilerek :

$$(W_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2) = \emptyset \quad (2.3)$$

Bunun dışında P2' nin , P1' in hesapladığı sonuçlara ihtiyacı olmaması gerekir :

$$(X_1 \cup Y_1 \cup Z_1) \cap (W_2 \cup Y_2) = \emptyset \quad (2.4)$$

P1' in yazdıklarına daha sonra P2' nin erişmemesi için :

$$(X_1 \cup Y_1 \cup Z_1) \cap Z_2 = \emptyset \quad (2.5)$$

(2.4) ve (2.5) birleştirilerek :

$$(X1 \cup Y1 \cup Z1) \cap (W2 \cup Y2 \cup Z2) = \emptyset \quad (2.6)$$

P1 ve P2 'nin paralel veya seri yürütülmesine bağlı olmaksızın, P3 'e girerken, tüm bellek bölgelerinin içeriğinin aynı kalması için P3 'ün okuyarak eriştiği bellek bölgeleriyle P1 ve P2 'nin yazarak eriştiği bellek bölgelerinin ortak kesişim kümeleri boş küme olmalıdır :

$$(X1 \cup Y1 \cup Z1) \cap (X2 \cup Y2 \cup Z2) \cap (W3 \cup Y3) = \emptyset \quad (2.7)$$

P programının, şekil 2.1 (a) ve (c) deki yürütüldüğünde aynı sonuçları verebilmesi için gerekli koşullar (2.3), (2.6) ve (2.7) deki koşullardır.

Gevsek bağlı çok işlemcili bir sistemde, süreçlerin paralel çalışabilirliğini incelemek için bir kabul yapılması gerekir. Bu kabul, P1 'in çalıştığı işlemci birimine bağlı belleğin içeriği, P2 'nin çalıştığı belleğe göre ana belleğe daha önce aktarılmasıdır. Gevsek bağlı sistemde, herbir işlemcinin lokal belleği olduğu için P1 ve P2 birbirinin eriştikleri verileri bozamazlar. (2.4) koşulu, (2.3) ve (2.6) nın yerini alır. (2.7) koşuluna gerek kalmaz.

Komutatiflik için , (2.2) ve (2.5) 'e gerek olmayıp (2.1), (2.4) ve (2.7) deki koşullara halen ihtiyac vardır.

Su durumda komutatiflik için aşağıdaki koşullar geçerli:

$$(W1 \cup Y1) \cap (X2 \cup Y2 \cup Z2) = \emptyset$$

$$(W2 \cup Y2) \cap (X1 \cup Y1 \cup Z1) = \emptyset$$

$$(X1 \cup Z1) \cap (X2 \cup Z2) \cap (W3 \cup Y3) = \emptyset$$

Yukarda verilen koşullara BERNSTEIN koşulları denir.

2.2 Değişken Bağımlılık Tipleri

Deyimler arası geçerli olmak üzere üç tip değişken bağımlılığı tanımlanmıştır [3].

Tanım : Bir programın çalışması sırasında, S2 deyimini yürütmeden önce S1 deyimini yürütmek gerekiyorsa S2, S1'e bağımlıdır denir ve bu iki deyim arasında bağımlılık ilişkisi vardır.

İki deyim arasında bağımlık ilişkisi varsa aşağıdaki üç

koşuldan biri vardır. Bunlar Bernstein koşullarının özeti şeklindedir.

1) S2, S1'in çıkış verilerini kullanır. Bu tip bağımlılığa gerçek bağımlılık (true dependence) ve δ ile gösterilir.

S1: $X = \dots + \dots$

S2: $\dots = X - \dots$

2) S1 veriyi kullandıktan sonra S2 bu veriye yazar. Bu bağımlılığa ters bağımlılık (antidependence) denir ve δ ile gösterilir.

S1: $\dots = \dots + X$

S2: $X = \dots / \dots$

3) S2, S1'in yazdığı veriye tekrar yazar. S1 ve S2'nin yürütülme sırası değiştirildiğinde, sonraki deyimler yanlış çıkış verisi kullanırlar. Bu bağımlılığa çıkış bağımlılığı (output dependence) denir ve δ ile gösterilir.

S1: $X = \dots$

S2: $X = \dots$

Deyimlerin paralelleştirilmesi konusunda, bu üç tip bağımlılık göz önüne alınmalı ve bunların varlığı test edilmelidir. Veri bağımlılığı deyimlerin yürütülme sırasını belirler.

Veri akışı analizinde bağımlılık, herhangi bir deyimden diğer bir deyimden önce yürütülerek, o deyim için gerekli doğru verileri hazırlaması anlamına gelir. Ters ve çıkış bağımlılığı yalnız deyimlerin yürütülme sırasını belirlediğinden, veri akışı analizinde bunların anlamları yoktur. Bu yüzden bu tip bağımlılıklara yapay bağımlılık denir.

2.3 Basit ve Dizi Değişkenleri

Bağımlılık ilişkileri üzerinde çalışırken değişkenler, dizi ve basit değişkenler olarak ayrı ele alınacaktır. Basit değişkenlerin hiçbir komponenti yoktur. FORTRAN veya diğer dillerde bunlar A, B, VERI vs. olarak gösterilirler. Dizi değişkenleri, boyutlu

olup indis içerirler. Basit değişkenler bunların komponenti olurlar. Örnek : $A(I)$, $B(2*I+3)$, $C(4)$

2.4 Dizi Değişkenlerinde Veri Bağımlılıkları

Veri bağımlılıklarını, basit değişkenler üzerinde belirlemek kolaydır. Ancak, indisli değişkenlerde yani dizi değişkenlerinde ($A(I)$, $B(I+1)$ vs.) veri bağımlılığını araştırmak daha karmaşıktır.

Bu sorunu bir örnekle açıklamak gerekirse :

```
DO 10 J=1,N
  X(J) = X(J) + C
10 CONTINUE
```

Bu döngüdeki deyim kendisine bağlı değildir. Yani belirli bir iterasyonda erişilen ve yazılan X değerlerinin indisleri aynıdır.

Buna karşıt bir örnek :

```
DO 10 J=1,N-1
  X(J+1) = X(J) + C
10 CONTINUE
```

Bu deyim kendisine bağlıdır. Çünkü $i+1$. iterasyonda erişilen X değeri, i . iterasyonun sonuç değeridir. Görüldüğü gibi iterasyonlar arası veri bağımlılığı söz konusudur. İterasyonlar arası veri bağımlılığını daha iyi açıklamak için bir genelleştirme yapma yerinde olacaktır.

```
DO 10 I=1,N
S1: X(f(I))=.....
S2:      = F(X(g(I)))
10 CONTINUE
```

$f(I)$ ve $g(I)$, X değişkeninin indis ifadeleridir. F , X değişkeninden oluşan bir fonksiyondur.

Tanım : Deyimlerde iterasyonlar arası veri bağımlılığı olabilmesi için öyle I_1, I_2 değerleri bulunmalıdır ki $1 \leq I_1 < I_2 \leq N$ ve $f(I_1) = g(I_2)$ olsun.

Diğer bir deyişle $f(I_1) - g(I_2) = 0$ denkleminin tam sayı çözümleri bulunmalıdır. Bu durum S_1 & S_2 ile gösterilir. f ve g fonksiyonları, rastgele herhangi özellikteki fonksiyonlar olsa, veri bağımlılığını test etmek son derece güç bir problemdir. Ancak bu fonksiyonlar, lineer olarak sınırlandırıldığında problemin çözümü belirli kurallarla sağlanabilir. Aşağıda bu kurallar verilecektir.

2.4.1 Dizi Değişkenlerinde Bağımlılık Testi

Yukardaki genelleştirilmiş döngü göz önüne alınsın .

$$f(i) = a_0 + a_1 * i \quad \text{ve}$$

$$g(i) = b_0 + b_1 * i$$

ise S_1 ve S_2 ' nin birbirine bağlı olabilmesi için

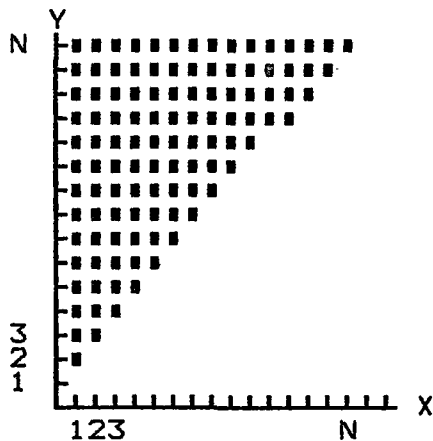
$$a_1 * x - b_1 * y = b_0 - a_0 \quad (x, y \text{ pozitif tam sayılar})$$

olmalıdır.

TEOREM 1 : $ax + by = n$ denlemin çözümü ancak ve ancak $\text{EBOB}(a, b) \mid n$ koşulu sağlanırsa vardır.

SONUC 1 : $f(i) = a_0 + a_1 * i$ olan S_1 deyimi, $g(i) = b_0 + b_1 * i$ olan S_2 deyimine ancak ve ancak $\text{EBOB}(a_1, b_1) \mid b_0 - a_0$ koşulu sağlanırsa bağlıdır.

SONUC 1, bağımlılık için gerekli , ancak yeterli değildir. Çözüm şekil 2.2 de belirtilen alan içersinde kalmalıdır.



Şekil 2.2 Çözüm Bölgesi

TEOREM 2 : $h(x,y)=f(x)-g(y)=0$ ve

R alanı $1 \leq x \leq N-1$ ve $2 \leq y \leq N$ ve $x \leq y-1$ olsun.

$h(x,y)$, R alanı içinde sürekli ise $h(x,y)=0$ denklemi

$\min h(x,y) \leq 0 \leq \max h(x,y)$ koşulunda sağlanır.

SONUC 2 : $f(x)$ ve $g(y)$ sürekli ise, $S1$ ve $S2$ deyimleri ancak aşağıdaki koşul gerçekleşirse birbirine bağlıdır:

$$\min(f(x)-g(y)) \leq 0 \leq \max(f(x)-g(y))$$

Teorem 2 de bağımlılık için gerekli ancak yeterli değildir.

TEOREM 3 : Teorem 2 'nin hızlı olarak uygulanması için bölge içersinde fonksiyonun minimum ve maksimum noktalarının kolay bulunması gerekir. Teorem 3 bu noktaların bulunması ile ilgilidir.

$f(x)=a_0 + a_1*x$ ve $g(y)=b_0 + b_1*y$ ise

$$\max(f(x)-g(y))=a_0 + a_1 - b_0 - 2*b_1 + (a_1 - b_1) (N-2)$$

$$\min(f(x)-g(y))=a_0 + a_1 - b_0 - 2*b_1 - (a_1 - b_1) (N-2)$$

Tanım : t reel bir sayı olmak üzere, bunun pozitif kısmı t ve negatif kısmı aşağıdaki gibi tanımlanır.

$$t^+ = \{ t, t \geq 0 ; 0, t < 0 \}$$

$$t^- = \{ -t, t < 0 ; 0, t \geq 0 \}$$

Teorem 2 ve teorem 3 , aşağıdaki SONUC 3 'ü getirir.

SONUC 3 (Banerjee Eşitsizliği) : $f(x)=a_0+a_1*x$ ve

$g(y)=b_0+b_1*y$ olsun. $S1$ ve $S2$ birbirine aşağıdaki eşitsizlik sağlandığında bağlıdır.

$$-b_1-(a_1+b_1) (N-1) \leq b_0+b_1-a_0-a_1 \leq -b_1+(a_1-b_1) (N-2)$$

Sonuc 1 ve 3, bağımlılık için gerekli birer testtir.

Bağımlılık testi aşağıdaki algoritmaya göre uygulanır.

1) f ve g 'nin lineer olup olmadığı belirlenir. Bunlar lineer ise a_0 , b_0 , a_1 , b_1 hesaplanır.

2) a- $EBOB(a_1,b_1) \nmid b_0-a_0$ gerçekleşmezse veya

b- Banerjee eşitsizliği tutmazsa

$S1$ ve $S2$ deyimleri birbirine bağlı değildir. a ve b koşulları doğrulanırsa deyimler birbirine bağlı olabilir. Ancak bu sonuç kesin değildir, deyimler birbirine bağlı olmayabilir.

Örnek 1 :

```
DO 2 I=0,2
S1: A(4-3*I)=B(I)
S2: C(I)=A(2*I)
2 CONTINUE
```

S1 deyimi S2 deyimine bağılı mı (S2 & S1) ?

$g(x)=2x$ ve $f(y)=4-3y$ olduğuna göre f ve g lineerdir.

$a_0=0$, $a_1=2$, $b_0=4$, $b_1=-3$

1) $EBOB(a_1, b_1) \nmid b_0 - a_0$

$EBOB(-3, 2)=1$ ve $b_0 - a_0 = 4 - 0 = 4$ olduğuna göre Sonuç 1 sağlanır.

2) $3 - (2 + (-3)) * 2 \leq 4 + (-3) - 0 - 2 \leq 3 + (2 - (-3)) * 2$

$3 \leq -1 \leq 17$

Banerjee eşitsizliği sağlanmadığından S1, S2'ye bağılı değildir. (S2 & S1)

Örnek 2 :

```
DO 2 I=1,10
S1: A(I)=B(I)
S2: C(I)=A(I+1)
2 CONTINUE
```

S1 & S2 testi yapılsın.

$f(x)=x$ ve $g(y)=y+1$ olduğundan f ve g lineerdir.

$a_0=0$, $a_1=1$, $b_0=1$, $b_1=1$

1) $EBOB(a_1, b_1) \nmid b_0 - a_0$

$EBOB(1, 1)=1$ ve $b_0 - a_0 = 1 - 0 = 1$ olduğundan Sonuç 1 sağlanır.

2) $-1 - (0 + 1) * 8 \leq 1 + 1 - 0 - 1 \leq -1 + (1 - 1) * 8$

$-9 \leq 1 \leq -1$

Banerjee eşitsizliği sağlanmadığından S2, S1'e bağılı değildir. Ancak S2 S1 testi yapılırsa bunun her iki koşulu sağladığı görülür. Bu durumda $f(x)=x+1$, $g(y)=y$ alınır. Gerçekten de $f(1)=g(2)$ ve $1 \leq 2$ olduğundan S2 & S1 denir. Koşullar sağlansa bile $f(x) \neq g(y)$ denkleminin çözümü aranmalıdır.

Görüldüğü üzere yukarıda anlatılan iki koşul, ancak belirli türdeki dizi elemanlarının bağımlılıklarını test etmek için kullanılabilir. Yani değişkenlerin bulunduğu döngünün N gibi belirli bir üst sınırı olmalı ve indis

fonksiyonları lineer olmalıdır.

Yukardaki açıklamalardan da anlaşılacağı üzere, bir döngüde bağımlılık iki farklı şekilde ortaya çıkabilir:

1) Bir deyim belirli bir iterasyonda bir bellek bölgesine yazarken, diğer bir deyim sonraki bir iterasyonda bu bellek bölgesini okur. Bu bağımlılık iterasyonlar arası bağımlılıktır (loop carried dependence) .

Örnek :

```
DO 10 I=1,N
S1: A(I)=B(I)
S2: C(I)=A(I+1)+3
10 CONTINUE
```

2) Diğer bir olasılık, bir deyim bir iterasyonda bir bellek bölgesine yazarken aynı iterasyonda diğer bir deyim bu bellek bölgesini okur. S2 deyiminin, S1 deyimine bağımlılığı döngüden bağımsızdır (loop independent dependence).

Tanım : S1 ve S2 aynı döngüde bulunan iki deyim olsun. S2, S1 'den sonra yürütülecekse S2 S1' i takip eder denir ve $S1 < S2$ olarak gösterilir.

İki deyim düşünölsün :

```
S1: X(f(i))=F( ..... )
S2: A = G(X(g(i)))
```

Tanım : $1 \leq i1 < i2 \leq N$ ve $f(i1)=g(i2)$ koşulu sağlandığında S1 ve S2 deyimi arasında iterasyonlar arası bağımlılık vardır denir. (S1 & S2)

Tanım : $1 \leq i \leq N$ (i pozitif tamsayı) ve $f(i)=g(i)$ ($S1 < S2$) koşulu sağlandığında S2, S1'e döngüden bağımsız olarak bağılıdır. (S1 & S2)

Deyimin kendi kendine bağılı olması, iterasyonlar arası bağımlılığın özel bir durumudur.

```
DO 2 I=1,N
S1: X(I+1)=X(I)+1
2 CONTINUE
S1 deyimi kendisine bağılıdır.
```

iterasyonlar arası bağımlılık, döngü indisleri nedeniyle ortaya çıkan bir bağımlılıktır. Döngüden bağımsız olan bağımlılıkların iterasyon indisleri ile bir ilgisi yoktur. Bu tip bağımlılık, deyimlerin döngü içindeki pozisyonlarından ileri gelir ve aynı iterasyon adımı içersinde ortaya çıkar.

2.5 İleriye Ve Geriye Doğru Bağımlılık

Değişken bağımlılığını ileriye ve geriye olmak üzere iki grupta toplanabilir [4].

İleriye doğru olan bağımlılık (Lexically forward dependency) Sv deyiminden , Sw deyimine olan bağımlılıktır öyleki $w > v$ dir.

Geriye doğru olan bağımlılıkta yine Sv 'den Sw 'e olan bağımlılık söz konusudur. Ancak $w \leq v$ yani Sw deyimi Sv deyiminden önce yer alır.

Bağımlılıkların bu şekilde gruplandırılmasının önemi daha sonra ortaya çıkacaktır. Bazı durumlarda, geriye doğru bağımlılık deyimlerin yürütülme sırası değiştirilerek ileriye doğru çevrilebilir.

2.6 Değişken Bağımlılık Grafı

Program deyimleri arasındaki bağımlılık, yönlendirilmiş bir graf yardımı gösterilebilir. Bu grafın düğümleri deyimler, arkları bağımlılığı daha doğrusu deyimlerin yürütülme sırasını gösterir [5]. Bu şekilde oluşan grafa değişken bağımlılık grafı (Data Dependency Graph) denir. Graf, programdaki veri akışını gösterir.

Bu grafı oluştururken temel alınacak yaklaşım, aynı isimlerdeki değişkenleri bulmak. Bu değişkenler basit değişkenler ise, bağımlılık ters, gerçek veya çıkış bağımlılığıdır. Buna göre ilgili düğümler arasında ark çizilir. Değişken dizi değişkeni ve aralarındaki

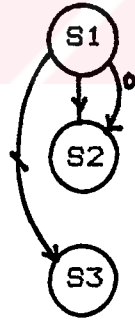
bağımlılık döngüden bağımsız ise, bunlar aynı basit değişkenler gibi işlem görür. Aksi halde iterasyonlar arası bağımlılık söz konusudur ki çizilecek arkın yönü bölüm 2.4.1 deki kurallara göre belirlenir.

Değişken bağımlılık grafi, programlardaki paralellizmin ortaya çıkarılmasında teorik bir taban teşkil etmektedir.

Örnek 1 :

```
DO 2 I=1,N
S1: A(I)=B(I)+C(I)
S2: A(I)=-A(I)/2
S3: C(I)=C(I)/2
2 CONTINUE
```

Deyimler incelendiğinde S1 & S2, S1 & S3, S1 & S2 bulunur. Değişken bağımlılık grafi şekil 2.3 de yer almaktadır. Herbir bağımlılık ilişkisine karşılık olmak üzere bir ark çizilir. Bu döngüde, döngüden bağımsız (loop independent dependence) ve ileriye doğru (forward dependency) bağımlılık vardır.



- > Gerçek Bağımlılık
- +—> Ters Bağımlılık
- o—> Çıkış Bağımlılığı

Şekil 2.3 Değişken Bağımlılık Grafi (Örnek 1)

Örnek 2:

```
DO 2 I=1,N
S1: A(I)=B(I)+C(I-1)
S2: D(I)=A(I)*2
S3: C(I)=A(I-1)+C(I)
S4: E(I)=D(I)+C(I-2)
2 CONTINUE
```

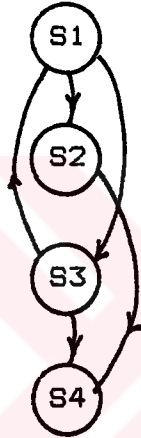

İndis fonksiyonları birbirinden farklı olan aynı isimdeki deyimleri bölüm 2.4.1 de verilen kurallar uygulandığında aşağıdaki şu sonuçlar çıkar : $S1 \delta S3$, $S3 \delta S1$, $S2 \delta S4$, $S3 \delta S4$, $S1 \delta S2$.

β bağımlılık fonksiyonu ise : $S1 \beta S3$, $S3 \beta S1$, $S2 \beta S4$, $S3 \beta S4$, $S1 \beta S2$. Herbir bağımlılık ilişkisine karşı grafta yönlendirilmiş bir ark çizilir (şekil 2.4).

İterasyonlar arası bağımlılıklar : $S1 \delta S3$, $S3 \delta S1$, $S3 \delta S4$

Döngüden bağımsız bağımlılıklar : $S1 \delta S2$, $S2 \delta S4$

Bu döngüde aynı zamanda geriye doğru bağımlılık ($S3 \rightarrow S1$) vardır.



Şekil 2.4 Değişken Bağımlılık Grafı (Örnek 2)

BÖLÜM 3. SÜREÇLER VE ARALARINDAKİ HABERLEŞME MEKANİZMALARI

Süreçlerin çalıştırılması ve yönetilmesi ile ilgili kullanıcıya bir takım komutlar sunulmuştur. Bu komutların işleyişini anlamak, işletim sisteminin süreç yönetimi hakkında fikir verecektir. Bu bölümde temel komutlar tanıtılıp işletim sisteminin bu komutları nasıl işlediği konusu üzerinde durulmuştur.

3.1 Temel Kavramlar

Süreç (Process) ve program birbirine yakın kavramlardır. Program sabit disk üzerinde yer alan, genellikle bir programlama dilinde yazılmış komutlardan oluşan kütüktür [6]. Program derleyici tarafından makina diline çevrilir ve koşturulur. Bir programın çalıştırılmasıyla süreç oluşur. Programın çalışması sonlandığında süreç de sonlanır. Program bu işleyişin statik, süreç ise dinamik elemanıdır.

Süreç çalıştığı zaman, işletim sistemi kendisine bir tanıma numarası (PID Process Identification) verir. Bunun yanında sürecin kendisine özgü bilgileri vardır. Bunlar kullanılan kütükler, bellek yerleşimi, saklayıcılar vs. dir.

Sürecin ana bellekte kapladığı alan üç bölüme ayrılır :

- 1) Text bölümü : Program kaynak kodunun yer aldığı bölge.
- 2) Veri bölümü : Sürecin çalışması için gerekli verilerin yer aldığı bölge. Diğer süreçler bu bölgeye erişemezler.
- 3) Yığın bölümü

Cok işlemcili bir sistemde, süreçler gerçek anlamda paralel çalışır. Süreçlerin paralel olması, bunların

çalışmalarının zaman içersinde çakışması anlamına gelir. Bu tez çalışmasında, çok işlemcili sistemlerin işletim sistemlerinden biri olan UNIX incelenmiştir.

Süreçlerin sistemdeki yapısı hiyerarsiktir. Yani süreçler, yine diğer süreçler tarafından başlatılırlar. Süreçler arasında baba-oğul ilişkisi vardır. Bir süreç diğer bir süreci başlatırsa, başlatan süreç baba,başlayan süreç oğul süreç olur. Baba-oğul süreçleri asenkron çalışırlar. Oğul çalışmaya başladığında, baba onun bitimini bekler ve ardından tekrar aktif duruma geçer. Baba süreç birden fazla oğul süreç yaratabilir. Bu durumda oğullar arka planda (background), baba ile birlikte senkron çalışırlar.

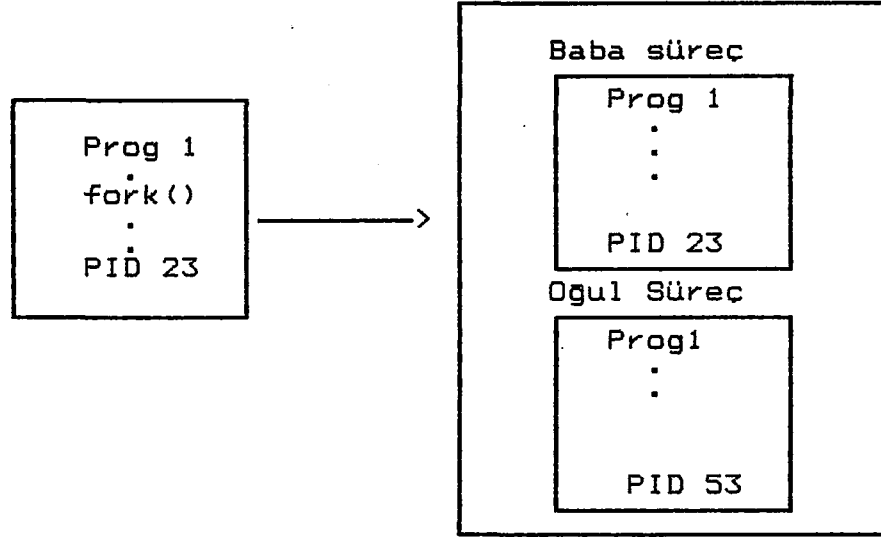
Süreçlerin çalışabilmeleri için bellekte yere ihtiyaç vardır. Belleğin işletim sistemi tarafından yönetimi, süreçlere yer tahsisi işletim sisteminin "scheduler" modülünün denetimi altında gerçekleşir.

3.2 Paralel Çalışmaya Yönelik Komutlar

FORK Komutu :

FORK yeni bir süreç yaratır. Süreçte yer alan FORK komutu yürütüldüğü anda, sistemde PID numaraları farklı, ancak birçok özellikleri birbirine eşit iki süreç bulunur (şekil 3.1). İşletim sistemi bu ikinci süreci yaratmadan bir takım işlemler yapmıştır :

- Öncelikle süreç yönetim sistemi (Process Management System), süreç tablosunda yeni süreç için bir yer açmıştır ve ona yeni bir PID numarası tahsis etmiştir. PMS tahsis edilecek PID 'nin sistemde tek olmasını



Sekil 3.1 Fork Komutunun İşleyişi

sağlamak zorundadır. Çünkü sistemdeki PID numarası en yüksek değerine ulaştığı zaman, PID sayacı tekrar sıfırlanır. Bu durumda yeni tahsis edilen PID'nin çift olma ihtimali vardır.

- Baba sürecin bellek alanı, oğul için yeni açılan bellek alanına kopyalanır.
- Oğul süreç, babanın açtığı tüm kütükleri de kullanabilmesi için, açık kütük sayacı ilgili değer kadar artırılır.
- Son olarak, fork komutu oğul sürece sıfır değerini, baba sürece oğulun PID' sini gönderir. Bu aşamadan sonra baba ve oğul birbirinden bağımsız olarak çalışabilir.

Fork sonucu oluşan oğul sürecinin babadan aldıkları :

- Çevre ve normal değişkenlerin değerleri
- Baba tarafından belirlenen kesme hizmet rutinleri
- Sürec öncelik değerleri
- Aktuel dosya
- Fork'tan önce açık bulunan kütüklere ilişkin işaretçiler

Oğul ile baba arasındaki farklar :

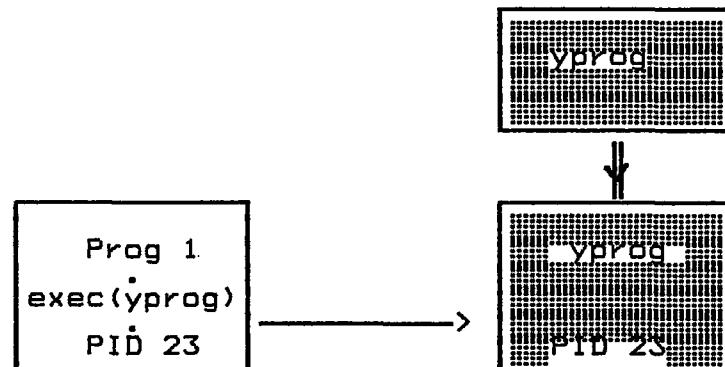
- PID numarası
- Sürec zamanı. Oğul sürecin sistem ve kullanıcı zaman sayacı sıfırlanır.

FORK komutunun algoritması kaba olarak aşağıdaki gibidir.

```
integer fork()  
begin  
    Yeterli sistem kaynakları mevcut mu ?  
    PID tahsisi  
    Kullanıcının süreç sayısını kontrol etmek  
    Oğul süreç durumunu "being created" a set etme  
    Babanın bellek bölgesini kopyalamak  
    if(baba)  
        begin  
            Oğul süreç durumunu "ready to run"a set etme  
            return (Oğul sürecin PID)  
        end  
    else  
        begin  
            Proses bölgesini yeniden oluşturma  
            return (0)  
        end  
    fi end  
end
```

EXEC Komutu :

Sabit diskten bir programı, EXEC komutunu çağıran sürecin bellek alanına yükler. Yani EXEC komutunu kullanan sürecin, data ve text segmanlarına yeni program yüklenir. Sürecin diğer çevre parametreleri (süreç öncelik değeri, PID, açılmış kütüklerin işaretçileri gibi) aynı kalır. Çağıran sürecin, eski data ve text segmanları kaybolur. EXEC ile yeni süreç oluşmaz (şekil 3.2).



Şekil 3.2 Exec Komutunun İşleyişi

- Sistem Exec komutu ile karşılaştığı zaman, işletim sistemi çekirdek moduna (kernel) moduna geçer.
- Yüklenmesi istenen kütük aranır. Bu kütük bulunmazsa Exec komutu hata kodu ile geri döner. Aksi halde , bu kütük dosya yönetim sistemi (File Management System) tarafından açılır . İlgili kütüğün erişim hakları kontrol edildikten sonra, kütüğün büyüklüğü kadar bellekte yer açılır. Kütük çağıran sürecin bellek alanına yüklenir.

EXIT ve WAIT Komutları :

EXIT, sürecin sonlanmasını sağlayan komuttur. EXIT ile süreci çalıştıran baba sürece bitiş kodu gönderilir. Baba, oğul sürecin bitimini WAIT komutu ile bekliyorsa bu bitiş kodunu alır ve oğul sürecin düzgün veya beklenmedik bir biçimde sonlanıp sonlanmadığını anlar.

İşletim sistemine Exit komutu geldiğinde yapılacak bir dizi iş vardır :

- Sürecin açık bulunan kütükleri varsa bunlar düzgün olarak kapanır.
- Sürecin semafor, ortak bellek alanı veya mesaj kuyruğu ile ilişkisi varsa bunlar kesilir.
- Son olarak sürecin kapladığı bellek bölgesi serbest bırakılır.

Baba süreç, wait komutu süresince bekleme durumundadır. Baba, oğul sürecin bitiminden önce durursa, sistemdeki ilk yani PID=1 olan süreç baba olur.

SIGNAL Komutu :

Süreçlerin kesmeleri yakalayıp, kesme hizmet programlarına dallanmalarını sağlayan komuttur.

3.3 Süreçler Arasındaki Haberleşme Mekanizmaları

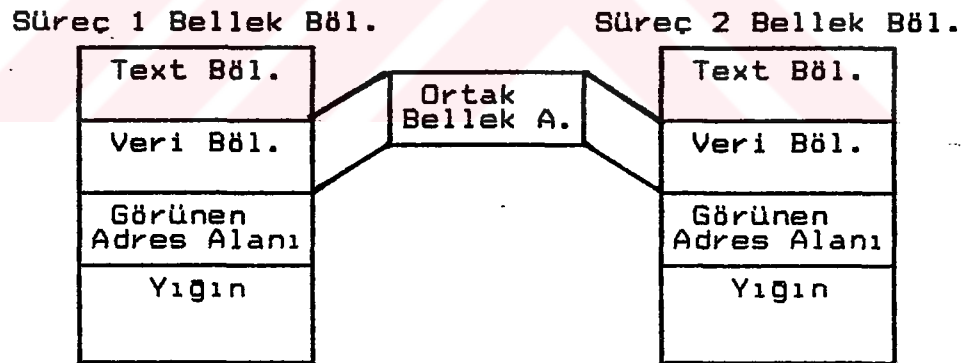
Çok işlemcili bir sistemde, süreçler çoğunlukla haberleşir ve birbirleriyle senkronize olurlar. UNIX işletim sisteminde bu haberleşmeyi sağlayan dört yöntem bulunur.

- a) Ortak Bellek Alanı (Shared Memory)
- b) Mesajlar
- c) Semaforlar
- d) Pipe

Aşağıda bu yapılar açıklanmaktadır.

Ortak Bellek Alanı :

İki veya daha fazla süreç, ortak bellek alanına erişerek buradaki bilgileri paylaşırlar. Süreçlerin baba-oğul ilişkisi içinde bulunmaları gerekmez. Farklı kullanıcıların süreçlerinin de erişim izinleri varsa bu ortak bellek alanına erişirler. Ortak bellek alanı, ana bellekte süreçlerin adres bölgeleri içersinde yer almayıp globaldir (şekil 3.3).



Şekil 3.3 Ortak Bellek Alanı

- Öncelikle herhangi bir süreç ortak bellek alanını yaratır.
- Diğer süreçler, bu alanı kullanmak istediklerini işletim sistemine bildirirler. İşletim sistemi bu süreçlere ortak bellek alanının işaretçisini verir. Böylece global alan süreçlerin adres bölgesine bağlanmış olur ve süreçler bu alana yazma/okuma yapabilirler.

Semaforlar :

İlk olarak Dijikstra, birçok süreç tarafından paylaşılan P ve V primitiflerini incelemiştir. Bu primitifler ortak semafor değişkeni üzerinde işlem yaparlar. P primitifi semafor değişkeninin değerini eksiltirken, V bir artırır.

P ve V primitiflerin fonksiyonları aşağıda görülmektedir [7].

```
var s (s semafor değişkeni)
P(s) : begin
    s=s-1
    if s < 0 then
        begin
            P(s) primitifini çağıran süreci bloke
            edip ilgili semafor değerini kuyruğa
            sok
        end
    end
end
V(s) : begin
    s=s+1
    if s ≤ 0 then
        begin
            s değerli semaforu bulunan aktif
            olmayan süreç varsa, en yüksek
            öncelikli süreci uyandırıp "hazır"
            listesine ekle.
        end
    end
end
```

Semaforlar, herhangi bir kaynağa erişmek isteyen iki veya daha fazla süreci senkronize etmeye yarar. Süreçlerin erişmek istediği kaynak; kütük, çevre cihazları veya ortak bellek alanı vs. olabilir. Semaforlar, bir sürecin tek olarak bir kaynağa erişirken yazma veya okuma sırasında diğer süreçlerle çakışmasını önler. Diğer süreçler kaynağın kullanılıp kullanılmadığını semaforun değerinden anlarlar.

Semaforların çalışma tarzlarını bir trafik ışığına benzetmek yerinde olacaktır. Işık yeşilse (semaforun değeri 0 ise) süreç kaynağı kullanabilir. Ancak bu kaynak kullanılmadan önce semaforun değeri bir artırılır yani ışığın rengi kırmızıya çekilir. Diğer süreçler semaforun değerine bakarak, kaynağın kullanılmakta olduğunu (ışık kırmızı) anlarlar. Kaynağı kullanmakta olan sürecin işi bittiğinde semaforu tekrar sıfıra çeker.

Mesaj Kuyrukları :

Süreçler arası bilgi transferinin diğer bir yöntemi de mesaj kuyruklarıdır. Bu prensipte, süreçler göndermek istedikleri bilgiyi doğrudan ilgili sürece göndermeyip bir mesaj kuyruğuna atarlar. Diğer süreçler istedikleri zaman bu mesajları okuyabilirler. Mesaj kuyrukları bilinen FIFO mantığında çalışırlar.

Süreçlerin mesaj kuyruğu ile haberleşmeleri için, aralarında baba-oğul ilişkisi bulunmasına gerek yoktur.

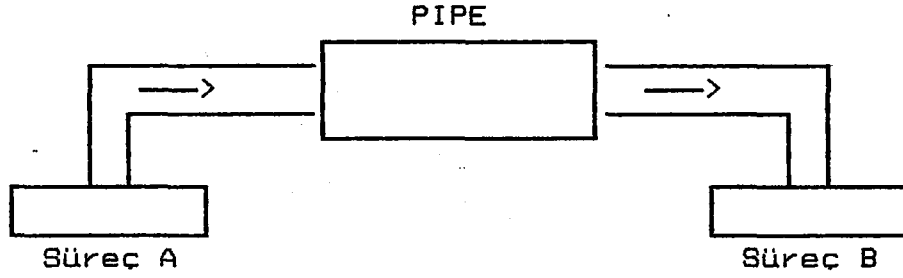
- Öncelikle bir sürecin, bir mesaj kuyruğu yaratması gerekir.
- Diğer süreçler, bu mesaj kuyruğuna erişebilmeleri için işletim sisteminden talepte bulunurlar. Ancak bu aşamadan sonra , süreçler mesaj kuyruğuna yazma/okuma yapabilirler.

Pipe :

Pipe mekanizmasının diğer yöntemlerden farkı, pipe üzerinden haberleşen süreçlerin arasında baba-oğul ilişkisinin bulunmasının gerektigidir.

Pipe, FIFO mantığında çalışır. Süreç A tarafından ilk olarak yazılan bilgiler, Süreç B tarafından da ilk olarak okunurlar. Pipe, Süreç A ve B arasında tek yönlü bir kanaldır (şekil 3.4).

Pipe mekanizmasında, veri haberleşmesi dışında süreçlerin senkronizasyonu da sağlanır. Yazan süreç pipe'a yeteri kadar hızlı bilgi yazamıyorsa, okuyan süreç



Şekil 3.4 Pipe Yapısı

bu bilgileri bekler. Aynı şekilde , okuyan süreç yeteri kadar hızlı okuyamıyorsa, yazan süreç pipe'in boşalmasını bekler. Pipe, lokal olarak süreçlerde yaşar ve işaretçisi ancak oğul sürece aktarılabilir.

İşletim sisteminde pipe'lar iki şekilde oluşturulabilir:

- a) İsmi bulunan pipe (named pipe)
- b) İsmi bulunmayan (unnamed pipe)

İsmi bulunmayan pipe'lar için dosya sisteminde özel bir kütük (pipe device) açılır. Süreçlerin çalışması bittiğinde bu özel kütük otomatik olarak silinir.

İsmi bulunmayan pipe, doğrudan süreç tarafından dosya sisteminde yaratılır. Bunların birer ismi ve erişim hakları bulunur. İsmi bulunmayan pipe'lar süreçlerin aktif olduğu müddetçe yaşamalarına karşı, ismi bulunan pipe'lar süreç tarafından doğrudan silme komutu ile silinmezse, dosya sisteminde varlıklarını korurlar. İsmi bulunan pipe'lar işletim sisteminin diğer kütükleri gibi işlem görürler. Pipe'a yazma ve okuma, bir kütüğe yazma ve okuma gibi olur. Tek fark, bir kütüğün istenilen yerine rastgele erişme imkanı varken, pipe'a ancak FIFO düzeninde erişilir.

Yukarda verilen senkronizasyon mekanizmaları haricinde paralel çalışan süreçler için özel bir durum daha vardır. İki veya daha fazla paralel çalışan süreç, yazılabilir bir bellek bölgesi paylaşıyorlarsa önemli sorunlar ortaya çıkabilir. Süreç A bellek bölgesini okurken, aynı anda Süreç B bu bölgeye yazıyorsa hatalı sonuçlar ortaya çıkabilir. Bundan dolayı süreçlerin bu

bölgelere tek olarak erişimleri sağlanır. Bu tip bölgelere kritik kısım denir. Kritik kısımda belirli bir anda tek bir süreç bulunabilir.



BÖLÜM 4. PARALEL DÖNGÜLER İÇİN SENKRONİZASYON

Bu bölümde döngülerin paralel çalışabilen süreçlere parçalanması, bu süreçlerin çok işlemcili bir sistemde koşturulması ve bu süreçler arasındaki senkronizasyon konuları incelenecektir.

4.1 Temel Yaklaşım

Amac programdan paralel çalışan süreçler elde etmek olduğu için, bir dizi işlemci tek bir programa hizmet verecektir. Yani bilgisayarın tümü tek bir işe (job) ayrılacaktır. Birçok çok işlemcili sistem, bu çalışma tarzında kullanılmaz. Çok işlemcili sistemlerde genellikle işlemciler, çoklu programlama (multiprogramming) sonucu daha fazla programı çalıştırmak ve birim zamanda biten program sayısını artırmak (throughput) için kullanırlar. Bilgisayar çoklu programlama özelliğinde ise bir işi paralel kısımlara ayırarak koşturmak, işi tek işlemcili sistemde ardışıl olarak koşturmaktan daha uzun sürecektir. Senkronizasyon gerektiren süreç o an işlemcide çalışmama olasılığı bulunduğundan ve işletim sistemi yönetimi gerektiğinden senkronizasyon süreleri uzayacaktır.

Belirtilen nedenlerden, programın paralelleştirilmesi ile performans artırımı beklenmesi için bilgisayar sisteminin tek bir programa tahsis edilebilir yapıda olup, senkronizasyonun donanım düzeyinden çözülmesi gereklidir. Bu tez çalışmasında böyle bir sistemi kullanma olanağı bulunmadığından, performans artımı beklemeyip çalışmayı bir modelleme şeklinde düşünmek gerekir.

4.2 Programdaki Paralellik Düzeyleri :

Herhangi bir programda üç farklı paralellik olabilir.

- a) Altprogram düzeyinde paralellik. Bu durumda bir dizi alt program paralel çalışır.
- b) İsteki atama deyimlerinden oluşan blokların paralel çalıştırılması. Bu tip paralellik CONTROL DATA 6600 sisteminde bulunur.
- c) Döngü düzeyinde paralellik. Bu tez çalışmasında üzerinde durulan paralellik, bu düzeyde olmalıdır.

4.3 Döngülerde Paralellik :

Döngülerde paralellik iki şekilde olabilir :

- a) Döngü her bir işlemciye bir iterasyon adımı düşecek şekilde derlenir. Örneğin P1 işlemcisi iterasyonun birinci adımını, P2 işlemcisi ikinci adımı yürütür.
- b) Döngüdeki deyimler, birbirinden bağımsız bloklara ayrılır ve herbir blok döngünün tüm iterasyonları için farklı işlemcide yürütülür [8].

4.3.1. Döngü iterasyonlarının işlemcilere Dağıtılarak Paralellik :

Çoğunlukla iterasyon sayısı, işlemci sayısını geçer. Bunun çözümü iki şekilde olur :

- a) İşlemciler, yürütecekleri iterasyon adımını kritik kısma girerek alırlar. Iterasyon adımı N olan bir döngü göz önüne alınsın.

```
enter critical section
global_I=global_I+1
I=global_I
exit critical section
if I > N then exit
```

- b) Döngü deyiminde değişiklik yapılarak, herbir işlemcinin yürüteceği iterasyon adımı önceden belirlenir. P işlemci sayısı, N iterasyon sayısı ve NUMP sistemdeki

toplam işlemci sayısı olsun :

DO I=P,N,NUMP

Aşağıdaki döngü göz önüne alınsın .

DO 10 I=1,N

S1: A(I)=B(I)+C(I)

S2: if (A(I-1) > 0) then

S3: D(I)=A(I)+A(I-1)

endif

10 CONTINUE

Bu iterasyonu ardışıl yürütülsün :

S1: A(2)=B(2)+C(2)

S2: if (A(1).....

S3: D(2)=A(2)+A(1)

S1: A(3)=B(3).....

S2: if (A(2).....

S3: D(3)=A(3)+A(2)

Görüldüğü gibi A(2)'ye birinci iterasyonda yazılır.
İkinci iterasyonda bu değer okunur.

Bu döngüyü iterasyonlara dağıtalım :

	<u>P1</u>	<u>P2</u>	<u>P3</u>
S1	A(2)=..	A(3)=..	A(4)=..
S2	if(A(1)..	if(A(2)..	if(A(3)..

Döngünün bu şekilde işlemcilere ayrılmasında, hiçbir senkronizasyon eklenmezse ikinci işlemci eski A(2) değerini kullanacaktır. Halbuki ardışıl çalışmada, ikinci iterasyon adımı birincide değiştirilmiş A değeri kullanılır.

Döngünün doğru bir şekilde çalışması için aşağıdaki senkronizasyon primitifleri eklenmelidir :

S1: A(I)=B(I)+C(I)

send(ASYNC,I)

if(I>2) wait(ASYNC,I-1)

S2: if((A(I-1)>0) then

S3: D(I)=A(I)+A(I-1)

endif

WAIT deyimi, aynı isimli ve indisli senkronizasyon değişkeni gönderilene kadar sürecin çalışmasını durdurur.

4.3.1.1 Senkronizasyon Yöntemleri

Döngülerin çalışmasında doğru sonuç elde edebilmek için, derleyicinin döngüye senkronizasyon primitifleri eklemesi gerekir. Bu primitifleri eklerken izlenecek çeşitli yöntemler vardır :

a) Rastgele Senkronizasyon :

Bu yöntemde derleyici iterasyonlar arası bağımlılık gördüğü yerlere senkronizasyon komutları ekler.

```
DO 10 I=1,N
S1: A(I)=B(I)+C(I-1)
S2: D(I)=A(I)*2
S3: C(I)=A(I-1)+C(I)
S4: E(I)=D(I)+C(I-2)
10 CONTINUE
```

Rastgele senkronizasyonda döngü aşağıdaki şekli alır :

```
DO 10 I=1,N
wait(CSYNC,I-1)
S1: A(I)=B(I)+C(I-1)
send(ASYNC,I)
S2: D(I)=A(I)*2
wait(ASYNC,I-1)
S3: C(I)=A(I-1)+C(I)
send(CSYNC,I)
wait(CSYNC,I-2)
S4: E(I)=D(I)+C(I-2)
10 CONTINUE
```

Bu döngünün hızlanması çok düşüktür. Iterasyonların işlemcilere dağılımı aşağıdadır :

I=1	I=2	I=3	I=4
S1			
S2			
S3			
S4			
	S1		
	S2		
	S3		
	S4		
		S1	
		S2	

Görüldüğü gibi, deyimlerin ancak dörtte biri paralel yürüyebiliyor.

Hızı artırmanın iki yöntemi var : Deyimlerin yürülme sırasını değiştirmek ve tekrarlanan senkronizasyon primitiflerini ortadan kaldırmak.

Yukardaki döngünün S3 ve S2 deyimleri yer değiştirebilir. Döngünün bağımlılık grafi incelendiğinde (bölüm 2.6 örnek 2), S2 ve S3 deyimlerinin birbirine bağlı olmadığını ve yer değiştirmenin sonucu etkilemediği görülür.

```
DO 10 I=1,N
  wait(CSYNC,I-1)
S1: A(I)=B(I)+C(I-1)
  send(ASYNC,I)
  wait(ASYNC,I-1)
S3: C(I)=A(I-1)+C(I)
  send(CSYNC,I)
S2: D(I)=A(I)*2
  wait(CSYNC,I-2)
S4: E(I)=D(I)+C(I-2)
10 CONTINUE
```

iterasyonların işlemcilere yeni dağılımı :

I=1	I=2	I=3	I=4
S1			
S3			
S2	S1		
S4	S3	S1	S1
	S2	S3	S3
	S4	S2	:
		S4	:

Bu şekilde deyimlerin yarısı paralel yürür.

Sv[i] -> Sv deyiminin 1. iterasyonu olmak üzere
Sv[i]<Sw[j] Sv[i], Sw[j] deyiminden önce çalıştığını gösterir.

Döngüde S1[i]<S3[i] ve CSYNC değişkeninden dolayı tablodan da görüldüğü gibi S3[i]<S1[i+1] dir.

S1[i]<S3[i]<S1[i+1]<S3[i+1] -> S1[i]<S3[i+1]

Deyimlerin doğal akışından dolayı A değişkeninin senkronizasyonuna gerek yoktur (ACSYNC,I-1) .

Aynı şekilde

$S3[I] < S1[I+1] < S3[I+1] < S1[I+2] < S4[I+2] \rightarrow S3[I] < S4[I+2]$

Sonuçta S4 deyiminden önceki (CSYNC,I-2) senkronizasyonuna gerek yoktur.

Görüldüğü gibi programın doğal akışından dolayı, (ASYNC,I-1) ve (CSYNC,I-2) senkronizasyonlarına gerek yoktur. Bu değerler zaten ilgili deyimlere taşınmaktadır.

Döngü aşağıdaki şekle dönüşür :

```
DO 10 I=1,N
  wait(CSYNC,I-1)
  S1: A(I)=B(I)+C(I-1)
  S3: C(I)=A(I-1)+C(I)
      send(CSYNC,I)
  S2: D(I)=A(I)*2
  S4: E(I)=D(I)+C(I-2)
10 CONTINUE
```

Rastgele senkronizasyon yöntemi oldukça esnek, ancak fazla senkronizasyon noktaları içerir. Uygulaması kolaydır ancak bu yöntemin optimizasyonu güçtür.

b) Pipe-line yöntemi :

Rastgele senkronizasyon yöntemine göre daha kısıtlıdır, ancak uygulaması daha kolaydır.

Bu yöntemde deyimler segmanlara gruplanır. Bu segmanların yürütülmesi pipe-line yöntemiyle olur. Deyimler öyle gruplandırılırki, veri bağımlılığı ya aynı segmanda kalır yada ileriye (lexically forward dependency) doğrudur.

Örnek :

```
DO I=1,N
seg1:S1: A(I)=B(I)+C(I-1)
seg1:S2: C(I)=A(I-1)+C(I)
seg2:S3: D(I)=A(I)*2
seg2:S4: E(I)=D(I)+C(I-2)
seg3:S5: F(I)=E(I)+F(I)
seg3:S6: G(I)=F(I)**2+D(I)**2
      2  CONTINUE
```

Döngünün bağımlılık grafi incelendiğinde yalnız ileriye doğru bağımlılık bulunduğu görülür. S1 ve S2 arasındaki bağımlılıktan dolayı S1 ve S2 aynı segmanda bulunur.

Segmanların işlemcilerle dağılımı :

I=1	I=2	I=3	I=4
Seg1 Seg2 Seg3	seg1 seg2 seg3	seg1 seg2 seg3	seg1 seg2 seg3

Rastgele senkronizasyonda olduğu gibi, deyimlerin yürütülme sıralarını değiştirilmesi segman büyüklüğünü küçültür. Daha küçük segmanlar, daha fazla paralellik anlamına gelir. Buna karşın daha fazla ve çoğunlukla gereksiz senkronizasyon noktaları bulunur.

Rastgele senkronizasyona göre, senkronizasyon noktaları kontrol altındadır. Rastgele senkronizasyonda, senkronizasyon noktaları oldukça artabilir. Ancak pipeline senkronizasyonda, kod gereksiz yere bölünebilir yani gereksiz senkronizasyon noktaları doğabilir. Yukardaki örnekte S4 ve S5 arasında senkronizasyon olmasına gerek yoktur. Ancak bu senkronizasyon noktası olmasaydı segman sayısı yalnız iki olacaktı ve ancak iki segman paralel yürüyecekti. Bu yöntemin diğer bir dezavantajı tüm deyimler arasında bağımlılık olursa, tüm deyimler aynı segmanın içinde yer alır ve döngü seri yürütülür.

Bu yöntem, yalnız ileriye doğru bağımlılık bulunan döngülerde uygulanabilir. Geriye doğru bağımlılık

bulunan döngülerde , bu bağımlılık deyim yer değiştirilmesiyle ileriye döndürülemezse bu yöntem uygulanamaz.

İterasyonlar işlemcilere dağıtılabildiği gibi, segmanlarda işlemcilere dağıtılabılır.

c) Bariyer Senkronizasyonu :

Bu yöntem de yalnız ileriye doğru bağımlılık bulunan döngülerde uygulanır. Döngü, pipe-line senkronizasyonda olduğu gibi segmanlara bölünür. Ancak bir sonraki segmana geçmeden, önceki segman tüm iterasyonlar için yürütülmüş olmalıdır.

Örnek:

```
DO 10 I=1,N
S1: A(I)=B(I)**2
    bariyer
S2: C(I)=A(I-1)+C(I)
S3: D(I)=A(I)*2
    bariyer
S4: E(I)=D(I)+C(I-2)
S5: F(I)=E(I)+F(I)
S6: G(I)=F(I)**2+D(I)**2
```

S2 den S4'e senkronizasyon gereklidir. Senkronizasyon S3'den önce veya sonra yerleştirilebilir.

Geriye bağımlılık, deyim yürütülme sırasının değiştirilmesiyle ileriye bağımlılığa çevrilebilir. Bariyer senkronizasyonu, pipe-line 'a göre daha fazla paralellik getirir. Çünkü, segmanların iterasyonu tüm işlemcilere dağıtılabılır ve daima tam anlamıyla paralellik söz konusudur.

Bariyer senkronizasyonunun sorunu basit değişkenlerdir.

Örnek:

```
DO I=1,N
S1: B=C(I)+D(I)/2
    bariyer
S2: E(I)=B+2
```

Bu döngüde, 1. iterasyonda birinci deyim tarafından hesaplanan B değeri, aynı iterasyonda ikinci deyime aktarılır. Halbuki bariyer senkronizasyonda, S1 tüm iterasyonlar için yürütüldükten sonra S2 deyimine geçilir ve B'nin ara iterasyondaki değeri kaybolur. Bunun çözümü basit değişkenlere dizi boyutuna çevrilmesidir. Konu daha sonraki bölümlerde açıklanacaktır. Bariyer senkronizasyonunda çalışan derleyiciler bu tip değişkenleri fark edip, gereken önlemi almak zorundalar. Bu bellek ve performans kayıplarına neden olur.

d) Kritik Kısım :

Döngüde, iterasyonlar arası veri bağımlılığı bulunan bölgeler kritik kısım olacak şekilde düzenlenir. Belirli bir anda, kritik kısımda ancak bir işlemci bulunabilir. Kritik kısım dışındaki kod, doğrudan tüm işlemcilere dağıtılarak yürütülür. Yöntemin en olumlu yönü, geriye doğru bağımlılık bulunan döngülerde de uygulanabilir olmasıdır.

Örnek :

```
DO 10 I=1,N
  begin critical section
S1: A(I)=A(I-1)*B(I)+C(I)
  end critical section
S2: C(I)=A(I)+C(I)
S3: D(I)=A(I)*2
S4: E(I)=D(I)+C(I)
10 CONTINUE
```

S1 deyiminin kendisine iterasyonlar arası bağımlılığı vardır. S1 deyimi kritik kısma alınarak döngü paralelleştirilir. İşlemciler sıra ile kritik kısmı yürütürler.

İki tip kritik kısım vardır :

- Sıralı kritik kısım
- Sırasız kritik kısım

Sıralı kritik kısımda, i. iterasyonu yürüten işlemcinin ardından i+1. iterasyonu yürüten işlemci kritik kısma

girer. Sıralı olmayan kritik kısımda, işlemcilerin takip edeceği sıra yoktur, işlemciler kritik kısmı boş buldukları anda girerler. Bu tip kritik kısımlar toplam alan deyimlerde uygulanabilir. Toplam alındığı için iterasyon adımının önemi yoktur. Ancak genel durumda, sıralı olmayan kritik kısımlar hatalı sonuçlara neden olabilirler.

Derleyicinin hedefi, kaynak koda en az sayıda kritik kısım eklemektir. Çünkü yürütme hızı, kritik kısmın büyüklüğü kadar düşer. Deyimlerin yürütme sıralarının değiştirilmesi, bağımlılıkları aynı kritik kısma toplamak için uygulanabilir.

4.3.2 Döngüdeki Deyimlerin Bloklara Ayrılması Prensipli

Bu yöntemi açıklamak için, veri bağımlılık grafi üzerinde bir takım tanımlar yapmak gerekir.

DO 10 I1=0,u1 I-> indis vektörü olsun ve I1,
DO 10 I2=0,u2 I2,..Id nin lineer
 fonksiyonu olsun.

..

DO 10 I3=0,ud

S1(I)

S2(I)

.

.

SN(I)

10 CONTINUE

- G, bu döngünün veri bağımlılık grafi
- S1,S2,.....SN bu grafin düğümleri
- Sr' den St' ye olan yönlü ark, $Sr\beta St$ (β bağımlılık fonksiyonu) yani St' nin Sr' ye bağlı olduğunu gösterir.
- Zincir, bir dizi düğüm ($Sq1,Sq2,....Sqk$) öyleki $Sq1\beta Sq2, Sq2\beta Sq3,.....Sqk-1\beta Sqk$ dır.
- $Sqk\beta Sq1$ olması durumunda zincir çevreye dönüşür.
- içinde en az bir çevre bulunan graf çevreseldir (cyclic).

- içinde çevre bulunmayan G grafi çevresel değildir (acyclic).
- Başka bir çevrenin alt çevresi olmayan çevreye maksimum çevre denir.
- Herhangi bir çevrenin elemanı olmayan düğüme isole nokta denir.
- Maksimum çevre veya isole noktaya π blok denir. π bloklarının kümesi, graftaki tüm elemanları verir.
- $Sq_1, Sq_2, \dots, Sq_k$ bir zincir olsun.
 $Sq_1 \in \pi_r$ ve $Sq_k \in \pi_t$ ise
veya $\pi_r = \pi_t$ ise $\pi_r \cap \pi_t$ dir.
 Γ, π blokları arasında tanımlanan kısmi sıra bağıntısıdır. $\pi_r \cap \pi_t$ ise, π_t bloğu π_r bloğuna bağlıdır denir.
- π blokları $(\pi_1, \pi_2, \dots, \pi_k)$ arasında aşağıdaki gibi bağıntılar mevcutsa π blokları bir zincir oluşturur.
 $\pi_1 \cap \pi_2, \pi_2 \cap \pi_3, \dots, \pi_{k-1} \cap \pi_k$
Tanımlardan da anlaşılacağı üzere π blokları çevre oluşturmaz.
- π blokları kümesinde yer alan blokların hiçbirinde kısmi sıra bağıntısı yoksa bu bloklar bağımsızdır denir.
- h, π blokları zincirindeki eleman sayısı olsun. Bu durumda, π blokları h bağımsız sıraya kümelendir. Dyleki,
 - * Her sıra bağımsızdır. Yani içerdigi bloklar arasında kısmi sıra bağıntısı olmaz.
 - * k . sıradaki bloklar, t . sıradaki bloklara bağlı olmaz ($k \leq t$).
 - * k . sıradaki en az bir blok, $k-1$. sıradaki bir bloğa bağlı olur ($k > 1$).

Tanımlardan da anlaşılacağı üzere sıralar, birbirinden bağımsız π blokları içerir. Bundan dolayı birinci sıranın π bloklar işlemciye dağıtılarak paralel yürütülür, ardından ikinci sıranın π blokları paralel yürütülür. Bu şekilde döngü, paralel yürüyebilen deyimlere ayrılmış olur. Yöntem, döngülerin parçalanmasında oldukça sistematik bir yol izlemektedir.

Hem ileriye hem de geriye doğru bağımlılık içeren döngülerde uygulanabilir. Yöntem bariyer senkronizasyonu ile benzerlik gösterir.

Örnek 1:

```
DO 10 I=0,100
```

```
S1: A(I)=D(I)*2
```

```
S2: B(I+1)=A(I)+C(I+1)
```

```
S3: C(I+4)=B(I)+A(I+1)
```

```
S4: X(I+2)=X(I)+X(I+1)
```

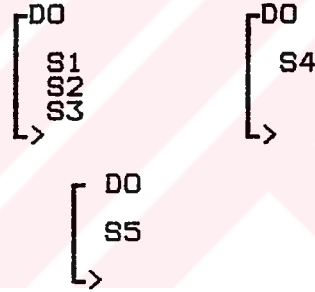
```
S5: E(I+3)=C(I)-5
```

```
10 CONTINUE
```

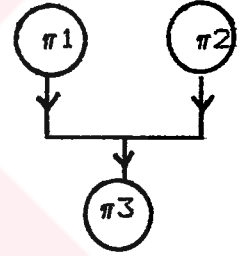
S1δS2, S3δS1, S2δS3, S3δS2, S4δS4, S3δS5 bağımlılık ilişkilerinden dolayı S1βS2, S3βS1, S2βS3, S3βS2, S4βS4 ve S3βS5 olur. İlgili bağımlılık grafi (DBG) şekil 4.1(a)'da görülmektedir.



(a)



(b)



(c)

Şekil 4.1 DBG ve Programın Yeni İşleyişi (Örnek 1)

İsole nokta : S5

Çevreler : {S1,S2,S3}, {S2,S3}, {S4}

Maksimum çevreler : {S1,S2,S3}, {S4}

π1={S1,S2,S3} π2={S4} π3={S5}

sıra1={π1,π2} sıra2={π3}

Bu yöntemle döngü, üç farklı alt döngüye ayrılmış oldu.

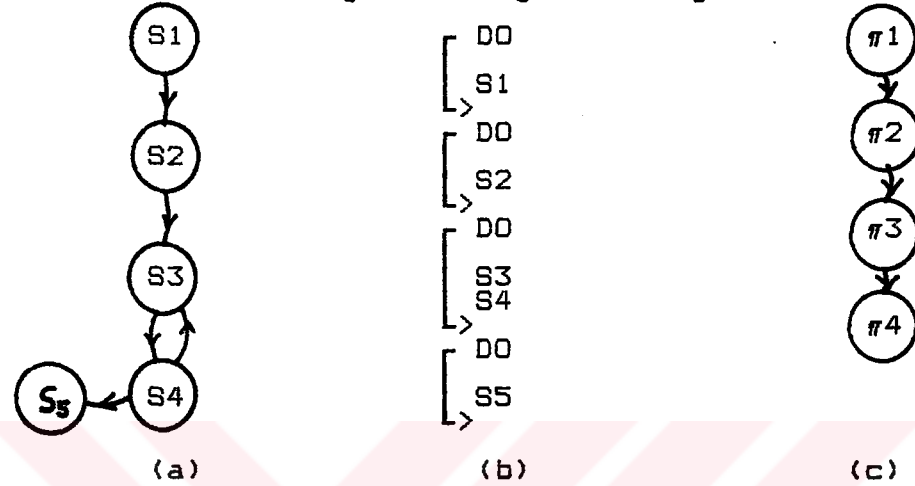
D1=π1 D2=π2 D3=π3

Sıra tanımından dolayı, sıradaki bloklar birbirinden bağımsız olup bunlar paralel yürüyebilir.

Örnekte, $\pi 1$ bloğu 1. işlemcide, $\pi 2$ bloğu 2. işlemcide paralel yürür. Her iki bloğun çalışması bittikten sonra, $\pi 3$ bloğundaki deyim herhangi bir işlemcide yürütölür.

Örnek 2:

Şekil 4.2'teki değışken bağımlılık grafi ele alınsın.



Şekil 4.2 DBG ve Programın Yeni İşleyişi (Örnek 2)

$\pi 1 = \{S1\}$ $\pi 2 = \{S2\}$ $\pi 3 = \{S3, S4\}$ $\pi 4 = \{S5\}$

Sıra1 = $\{\pi 1\}$ Sıra2 = $\{\pi 2\}$ Sıra3 = $\{\pi 3\}$ Sıra4 = $\{\pi 4\}$

Özel Durumlar :

Döngünün bloklara ayrılması yönteminde, derleyicinin fark etmesi gereken iki özel durum vardır. Bu özel durumlar örneklerle açıklanacaktır.

Durum 1:

DO 10 I=1,100

S1: T=A(I)

S2: B(I)=T

10 CONTINUE

$\pi 1 = \{S1\}$ $\pi 2 = \{S2\}$

Sıra1 = $\{\pi 1\}$ Sıra2 = $\{\pi 2\}$

Derleyici döngüyü aşağıdaki şekle çevirecektir :

DO 10 I=1,100

S1: T=A(I)

10 CONTINUE

DO 20 I=1,100

S2: B(I)=T

20 CONTINUE

Bu bölünme, T'nin basit değişken olmasından dolayı yanlış sonuç üretecektir. Derleyici T basit değişkenine indis atayarak, T' yi dizi değişkenine çevirmek zorunda.

```
DO 10 I=1,100
S1: T(I)=A(I)
10 CONTINUE
DO 20 I=1,100
S2: B(I)=T(I)
20 CONTINUE
```

Durum 2:

Döngüde iki deyim arasında, döngüden bağımsız ters bağımlılık varsa bu iki deyimi aynı bloğa alınması gerekir.

```
DO 10 I=1,100
S1: B(I)=A(I)+C(I)
S2: A(I)=D(I)+3
10 CONTINUE
```



Şekil 4.3 DBG

A değişkeninden dolayı, döngüde ters bağımlılık vardır. S1 ve S2 aynı bloğa yerleştirilmelidir (şekil 4.3).

$\pi_1 = \{S1, S2\}$ Sıra1={ π_1 }

Tüm deyimler, tek bir maksimum blok içinde yer aldığı anda, döngünün yürütülmesi seri çalışmaya eş değer olacaktır.

4.4 Döngü için Zaman Ve İşlemci Üst Sınırı

Bu bölümde verilen üst sınırlar, döngünün bölüm 4.2.2'de açıklanan prensibe göre paralelleştirilmesi durumunda geçerlidir.

Tüm işlemcilerin paralel çalışabileceği ve herbirinin toplama, çıkarma, bölme gibi aritmetik işlemleri yapabileceği varsayılmıştır. Bellek ve kontrol işlemleri için harcanan zaman ihmal edilmiştir.

Aritmetik işlemlerin, belirli bir işlemcide

yürütülmesi için gereken zaman dilimine, birim zaman denir.

$T[L]$: L döngüsünün yürütülmesi için gereken zaman

$P[L]$: L döngüsünün yürütülmesi için gereken işlemci sayısı

Bu bölümde amaç, $P[L]$ sayısında herhangi bir kısıtlama getirmeksizin, $T[L]$ süresini minimize etmektir. Verilen teorem, L döngüsünün yürütülme süresinin ve gerekli işlemci sayısının belirli bir üst sınırın altında kaldığını gösterir. Bu üst sınırlar $T[E\langle e \rangle]$, $P[E\langle e \rangle]$, $T[R\langle n, m \rangle]$, $P[R\langle n, m \rangle]$ ve L döngüsünün bazı parametreleri ile ifade edilir.

Aşağıda ilgili bir takım tanımlar verilecektir :

Atom : Bir değişken veya sabittir.

Aritmetik ifade : Atom ve $+$, $-$, $*$, $/$ vs. operatörlerden oluşan bir katardır.

$E\langle e \rangle$: Genel anlamda, e atomlu bir aritmetik ifade.

Atama deyimi : $x=E$ şeklindeki ifadedir. x değişken, E aritmetik ifadedir.

E'nin uzunluğu : E aritmetik ifadesinin atom sayısıdır.

x : Çıkış yani sonuç değeri

Giriş Değişkenleri : E aritmetik ifadesinde yer alan değişkenlerdir.

$T[E\langle e \rangle]$: Aritmetik ifadenin yürütülmesi için gerekli süre

$P[E\langle e \rangle]$: Aritmetik ifadenin yürütülmesi için gerekli işlemci sayısı.

Aşağıda $T[E\langle e \rangle]$ ve $P[E\langle e \rangle]$ ile ilgili eşitsizlikler verilmektedir. Bunların sonuçları önemli olduğundan kanıtları verilmeyecektir.

$$- T[E\langle e \rangle] \leq 4 \log e, \quad P[E\langle e \rangle] \leq 3(e-1)$$

- d, $E\langle e \rangle$ ifadesindeki iç içe parantezlerin sayısı olsun.

$$T[E\langle e \rangle] \leq \log e + 2d + 1, \quad P[E\langle e \rangle] \leq (e-2d)/2$$

$T[R, n, m]$, $P[R\langle n, m \rangle]$ lineer, geriye doğru bağımlılık gösteren bir ifadenin yürütülme zamanı ve gerekli işlemci sayısıdır.

$$R\langle n, m \rangle \quad 1 \leq m \leq n$$

$$X_k = 0 \quad k \leq 0$$

$$X_k = C_k + \sum A_{kt} * X_{kt} \quad 1 \leq k \leq n$$

C, A birer sabit olup X değişkendir.

$R\langle n, m \rangle$, toplam n işlemden oluşan bir sistemde, bu ifadenin hesaplanabilmesi için m deyim öncesi elde edilen değere ihtiyaç olduğunu gösterir.

$$T[R\langle n, m \rangle] \leq (2 + \log m) \log n - 1/2 (\log^2 m + \log m)$$

$$P[R\langle n, m \rangle] \leq \begin{cases} nm^2/2 + O(m, n) & m \ll n \\ n^3/68 + O(n^2) & m \leq n \end{cases}$$

L döngüsünün parametreleri :

- M : iterasyon sayısı
- N : Deyim sayısı
- c : Bir π bloğundaki maksimum düğüm sayısı
- h : Sıra sayısı
- z : G grafindaki maksimum çevre sayısı
- Q : iterasyonlar arasındaki bağımlılıklarda maksimum mesafe

TEOREM 1 : L döngüsünün bağımlılık grafi G, çevre içermesin.

$$T[L] \leq h \cdot T[E\langle e \rangle]$$

$$P[L] \leq (N-h+1) \cdot M \cdot P[E\langle e \rangle]$$

Kanıt : G grafi çevre içemediğinden, herbir π blok bir elemandan oluşur. h tane sıra olduğuna göre ve bu sıraların herbiri N bloktan en az bir tane içermek zorunda olduğundan, bir sırada $(N-h+1)$ 'den fazla blok olamaz. $T[E\langle e \rangle]$, bir bloğun yürütülme sırasıdır. h sıra varsa, döngünün yürütülme sırası $T[L] \leq h \cdot T[E\langle e \rangle]$ olur.

Herbir sıradaki blok ayrı bir işlemcide çalışacağı için, en fazla $(N-h+1)$ işlemciye ihtiyaç olacaktır. Herbir iterasyon da farklı bir işlemcide yürüyeceğinden işlemci sayısı için $P[L] \leq (N-h+1) \cdot M \cdot P[E\langle e \rangle]$ üst sınırı oluşur.

Sonuç 1 : L döngüsü , hiçbir deyim arasında bağımlılık bulunmayan bir döngü olsun.

$$T[L] \leq T[E\langle e \rangle]$$

$$P[L] \leq M \cdot N \cdot P[E\langle e \rangle]$$

Kanıt : Bu durumda bir sıra vardır. Bu sırada N blok vardır. Bunların herbiri ve her iterasyonu farklı işlemcilerde yürüyecektir. Tam bir paralellik söz konusu olduğundan işlem süresi, en uzun atama ifadesinin işlem süresi kadar olacaktır. Gerekli olan işlemci sayısı, deyim sayısı x iterasyon sayısı kadardır.

TEOREM 2 : L döngüsü lineer ve bağımlılık grafi çevresel olsun.

$$T[L] \leq T[R<MN, MN>] + T[E<e>]$$

$$P[L] \leq \max \{ P[R<MN, MN>], MN \cdot P[E<e>] \}$$

Kanıt : Döngüde en kötü durumu, yani tek bir π bloğu bulunduğunu düşünelim. Bu L döngüsünün parçalanmadığı anlamına gelir.

Döngüde N eleman bulunduğuna ve iterasyon sayısı M olduğuna göre toplam $M \cdot N$ işlem yapılacaktır. En kötü durumu göz önüne aldığımızdan, bir ifadenin işlenmesi için $M \cdot N$ ilerdeki bilgiye ihtiyaç olsun ($n = M \cdot N$, $m = M \cdot N$). Yani döngüdeki tüm yada bazı ifadelerin işlenmesi için $M \cdot N$ ilerdeki bir deyimın hesaplanması gerekir.

Bu $M \cdot N$ adet deyimın önceden işlenmesi için gerekli süre $T[E<e>]$, işlemci sayısı $M \cdot N \cdot P[E<e>]$ dir. Bu süre, ifadenin işlenmesi için gereken süre ile toplanırsa teoremdaki üst sınır ortaya çıkar. Aynı şekilde, gerekli işlemci sayısı, lineer $R<n, m>$ sisteminin hesaplanması için gerekli işlemci adedi ile önceden işlem için gerekli işlemci adedinin en büyüğüdür.

4.5 Senkronizasyon Noktalarının Kaldırılması

İterasyonlar arası bağımlılıklar ortadan kaldırıldığında, işlemciler kendi aralarında senkronizasyona gerek duymadıkları için işlem hızlanır. Mümkün olan durumlarda senkronizasyon noktaları kaldırılmalıdır.

a) Senkronizasyon noktalarını kaldırmada kullanılan yöntemlerden biri döngü düzenlenmesidir (loop alignment) [9].

Aşağıdaki döngü göz önüne alınsın :

```
DO 10 J=1,100
S1: A(J+1)=B(J)+C
S2: X(J)=X(J)/A(J)
S3: B(J)=A(J+1)*D
10 CONTINUE
```

S1 deyimi i. iterasyonda, S2 deyiminin i+1. iterasyonda kullanacağı bir deyimi hesaplıyor. Kontrol deyimlerinin eklenmesiyle senkronizasyon noktaları kaldırılabilir. Döngünün bu şekle getirilmesinde maliyet, kontrol deyimleri ve ek bir iterasyon adımıdır.

```
DO 10 J=1,101
S1: if(J≥2) A(J)=B(J-1)+C
S2: if(J≤100) X(J)=X(J)/A(J)
S3: if(J≥2) B(J-1)=A(J)*D
10 CONTINUE
```

b) Senkronizasyon noktalarının kaldırılmasına ikinci bir yöntem olarak işlem dalgasını (Computation wavefront) verilebilir. Bu yöntemin genelleştirmek ve uygulamak oldukça zordur.

iki indis değişkeni bulunan döngüler ele alınacaktır.

```
DO 10 I1=1,Q1,1
DO 10 I2=1,Q2,1
S1
S2
...
10 CONTINUE
```

Bu döngü ardışıl olarak çalıştığı zaman, $Q1 \times Q2$ alanı boyunca i1 ve i2 tam sayı çifti için şekil 4.4(a) da gösterildiği üzere çalışır ($1 \leq i1 \leq Q1$ ve $1 \leq i2 \leq Q2$)

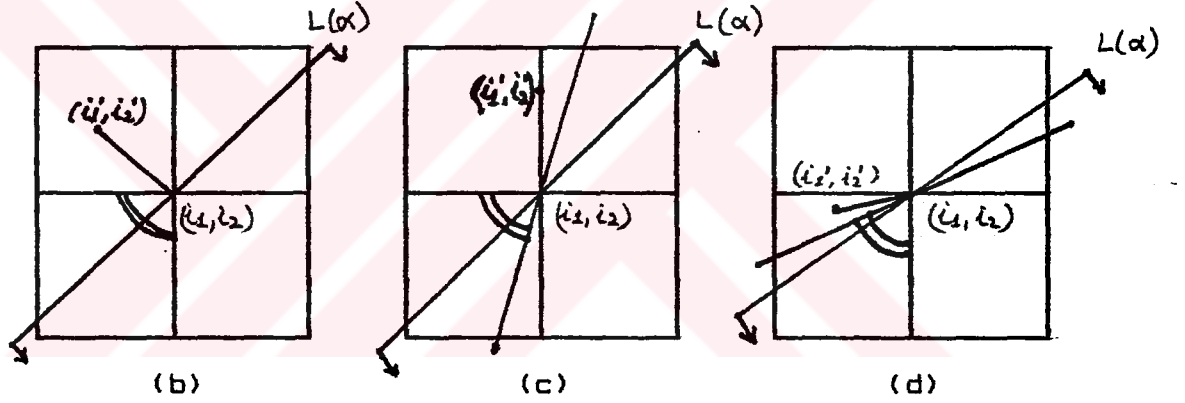
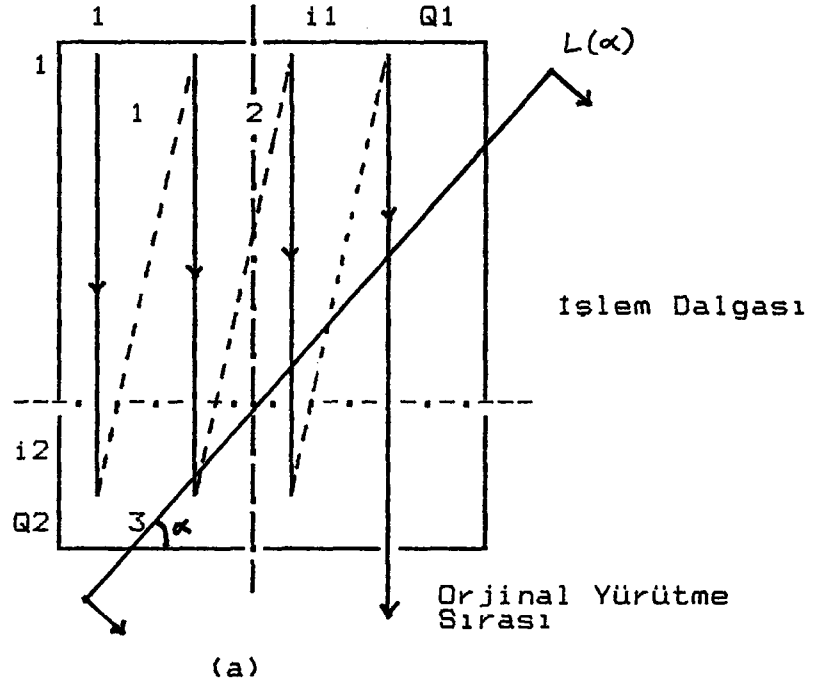
- a) $i1 < i1'$ veya $i1 = i1'$ ve

b) $i2 < i2'$ ise $(i1, i2) < (i1', i2')$

O halde orjinal yürütme sırası $<^o$ tanımı ancak

$(i1, i2) < (i1', i2')$ ise $S(i1, i2) <^o S(i1', i2')$ olarak verilir.

- $(i1, i2)$ çiftini göz önüne alalım ve herhangi $(i1', i2')$ çifti için $S(i1', i2') \leq S(i1, i2)$ bağıntısı gerçekleşsin.



Şekil 4.4 İşlem Dalgası

Yani $(i1', i2')$ indislerini yürüten S deyiminin çıkış verileri, $(i1, i2)$ indislerini yürüten S deyiminin giriş verileri olsun.

- $(i1, i2)$ için bir bağımlılık vektörü tanımlayalım. Böyleki $V = (i1', i2') (i1, i2)$

Genel halde $(i1, i2)$ için birden fazla bağımlılık vektörü bulunur. $(i1', i2')$ yanında $(i1'', i2'')$ değerleri vardır ve bu değerler $S(i1'', i2'')$ $S(i1, i2)$ bağıntısını gerçekleştirir. Herbir $(i1, i2)$ noktası için bir bağımlılık vektörü tanımlayabiliriz.

$$V = \{(i1', i2') (i1, i2) | S(i1', i2') S(i1, i2)\}$$

Genel durumda bu küme tek değildir. Ancak bundan sonraki açıklamalarda bu kümenin tek olduğu düşünülecektir. Yani

bu küme $(i1, i2)$ değerlerine bağlı olmayacak ve yalnız tek bir elemanı olacaktır. Bu durumda küme :

$$V = \{(0,0) (d1, d2)\}$$

$d1 = i1 - i1'$, $d2 = i2 - i2'$ $d1$ ve $d2$ sabit değerlerdir.

Örnek :

$$A(I1+a1, I2+a2) = A(I1+b1, I2+b2) + B(I1+c1, I2+c2)$$

$$V = \{(0,0) (a1-b1, a2-b2)\}$$

V kümesinin tek bir elemanı olduğu için

$$V = (0,0) (a1-b1, a2-b2) \text{ yazılabilir.}$$

$L(\alpha)$ olarak bir işlem hattı tanımlanır. Buna göre S deyimi, işlem doğrusu üzerinde yer alan tüm $(i1, i2)$ tamsayı çiftleri için paralel yürütülür. $L(\alpha)$ doğrusu, soldan sağa doğru hareket edip tüm $Q1 \times Q2$ alanını tarar.

$L(\alpha)$ işlem doğrusunun denklemi, $a*i1 + b*i2 + c = 0$ olarak verilebilir.

$\alpha = \tan^{-1} (a/b)$, c değeri doğru hareket ettikçe artar. Bu doğru ile tanımlanan işlem sırası $\langle [L(\alpha)] \rangle$ olarak gösterilir.

Verilen bir bağımlılık vektörü için dört farklı paralel yürütme düşünülebilir. Aşağıda bu algoritma açıklanmaktadır.

Algoritma :

$V = (i1', i2') (i1, i2)$, $d1 = i1 - i1'$, $d2 = i2 - i2'$ olsun. Şekil 4.4 de çift yay ile gösterilen açılar, 1., 2. ve 3. bölge için α açılarını verir.

Durum 1: $d1 > 0$ ve $d2 > 0$ yani $(i1', i2')$ 1. bölgede yer alır. $S(i1', i2') \delta S(i1, i2)$ ilişkisi mevcuttur ve $0 \leq \alpha \leq 90$ (şekil 4.4(b)).

Durum 2: $d1 = 0$ ve $d2 > 0$ yani $(i1', i2')$ 2. bölgede yer alır. $S(i1', i2') \delta S(i1, i2)$ ilişkisi mevcuttur ve $0 \leq \alpha < 90$ (şekil 4.4(c)). Çift yay ile gösterilen açı düşey doğru üzerindeki haric olmak üzere tüm noktaları kapsar.

Durum 3: $d1 > 0$ ve $d2 \leq 0$ yani $(i1', i2')$ 3. bölgede yer alır. $S(i1', i2') \delta S(i1, i2)$ ilişkisi vardır (şekil 4.4 (d))

$$h = \tan^{-1} \frac{|d_2|+1}{d_1} \quad \text{ise} \quad h \leq \alpha \leq 90 \text{ diyebiliriz.}$$

Durum 4 : Her üç durum yoksa $S(i_1', i_2')$ & $S(i_1, i_2)$ ve $0 \leq \alpha \leq 90$ dir.

$\alpha=90$ alınır, S deyimini i_2 'nin tüm değerleri için eş anlı yürütürken i_1 değeri ardışıl olarak değişir. Aynı şekilde $\alpha=0$ alınır, S deyimini i_1 tüm değerleri için paralel yürütürken, i_2 ardışıl değişir. Durum 4 de S deyimini, i_1 ve i_2 nin tüm değerleri için tek bir adımda eş anlı yürütülebilir.

Sonuç : Aşağıdaki değerler döngü, işlem dalgası yöntemi ile çalıştırılırsa elde edilen hızlanmayı verir.

$$\alpha=0 \quad \text{ise} \quad Q_2$$

$$0 \leq \alpha \leq 45 \quad \text{ise} \quad Q_1 + \frac{Q_2}{\tan \alpha}$$

$$45 \leq \alpha < 90 \quad \text{ise} \quad Q_2 + Q_1 \cdot \tan \alpha$$

$$\alpha=90 \quad \text{ise} \quad Q_1$$

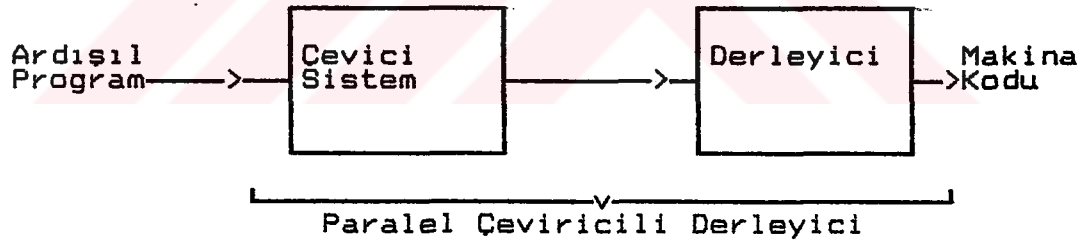
Halbuki döngünün ardışıl yürütme zamanı $Q_1 \times Q_2$ dir. Genelde durum uygunsa α değerinin 0 veya 90 alınması kolaylık sağlar.

BÖLÜM 5. UYGULAMA

Uygulama çalışması olarak, paralel çevirici sistem geliştirilmiştir. Paralel çevirici , C kaynak kodunu inceleyip buradaki FOR döngüsünde yer alan deyimler arasındaki paralellliği algılamaktadır. Kaynak koduna döngü yerine, gereken senkronizasyon deyimleri eklenerek bu döngüyü paralel alt süreçler şeklinde çalıştırmaktadır.

5.1 Paralel Çeviricili Derleyiciler

Bu tip derleyiciler, belirli ardışıl dilde yazılmış programı eş anlı paralel yürüyebilen parçalara ayırarak derler (şekil 5.1). Paralel süreçler farklı derleyicilerde kosturularak hız kazancı sağlanır.



Şekil 5.1 Paralel Çeviricili Derleyicinin Yapısı

Bu mantıkta Parafrase Analyzer (Illinois University), PFC-Parallel Fortran Converter (Rice University), PTRAN (IBM) ve KAP (Kuck & Associates) gibi güçlü derleyiciler geliştirilmiştir.

PFC tasarımının temeli, Parafrase derleyicisinin temeline dayanır. Bu derleyici bölüm 4.3.2 de açıklanan ve deyimlerin # bloklara ayrılması prensibiyle çalışır. Bu derleyicinin ilk versiyonu oldukça güçsüz iken, geliştirilen ikinci versiyonu birincisine göre yaklaşık on kat daha hızlıdır.

5.2 XENIX - C DERLEYİCİSİNİN YAPISI

Uygulama XENIX ortamında yapılmıştır ve XENIX-C derleyicisinin bir takım özellikleri kullanılmıştır.

Bu derleyici iki ana kısımdan oluşur [10].

a) cc komutu ile çağırılan sürücü program. Bu sürücü programın, geçişleri çalıştırmak, derleme opsiyonlarını geçişlere dağıtma, bağlayıcıyı (linker) çalıştırma gibi fonksiyonları vardır.

b) Derleyicinin diğer kısmı geçişlerdir. XENIX C derleyicisi dört geçişli bir derleyicidir.

0. Geçiş : Bu geçiş ön derleyici (pre-processor) olarak adlandırılır. Kütükleri program koduna dahil etme (file inclusion), ön derleyici ile tanımlanan sabitleri programda yerine koyma gibi işlemler yapar.

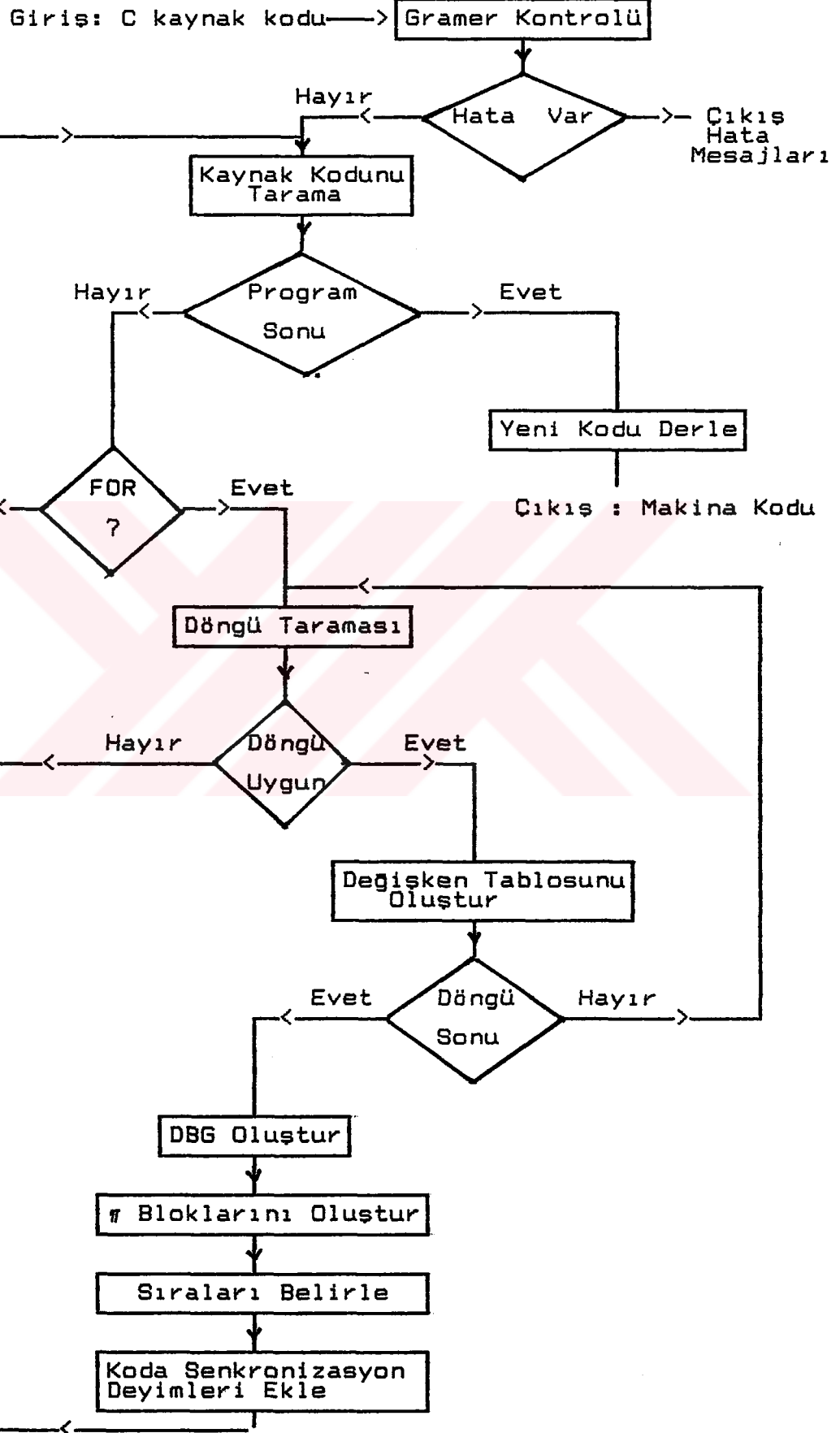
1. Geçiş : Derleyicinin tarama (parser) yapılan geçişidir. İki foksiyonu vardır . a) Gramer ağacını yaratma b) Sembol tablosunu hazırlama.

2. Geçiş : Kod oluşturur. Birinci geçişte hazırlana gramer ağacını tarayarak ikili kod oluşturur.

3. Geçiş : Bu geçişte optimizasyon yapılır. İkinci geçişte hazırlanan kod incelenerek, daha iyi performans elde etmek için kod üzerinde değişiklikler yapılır. Bu geçiş sonucunda obje kod oluşur.

5.3 Paralel Çevirici Sistemin Tasarımı

Bu tez çalışmasında, uygulama olarak tasarlanan çevirici sistem (translator), C dilinde yazılmış kaynak kodunu analiz etmektedir. Çevirici sistem, C kaynak kodundaki FOR döngülerinde yer alan aritmetik ifadeler arasındaki paralellliği ortaya çıkarmaktadır. FOR döngüsünde aritmetik ifade yerine başka deyimler yer alıyorsa bu döngüde işlem yapılmaz. Sistemin fonksiyonu kaba olarak akış diyagramda (şekil 5.2) gösterilmektedir.



5.3.1 Çevirici Sistem Modülleri

Aşağıda tasarlanan çevirici sistemin modülleri açıklanmaktadır.

A) Sürücü Modül :

Bu sistemde yer alan diğer modülleri çalıştırıp aradaki koordinasyonu sağlayan modüldür. Döngü desteklenmeyen yapıda ise alt modülleri çalıştırmaz.

B) Gramer Kontrol Modülü :

Çevirici sistemin tarama modülleri, kaynak kodunun gramere uygun olduğunu varsayar. Bu yüzden, tarama modülü çalıştırılmadan önce kodun gramer kontrolünün yapılması gerekir. Çevirici sistem, gramer kontrolü için C standart derleyicisinden faydalanır.

Derleyicinin 0. ve 1. geçişleri çalıştırılır. Çevirici sisteme giriş olarak verilen, C kaynak kodunda syntax hatası varsa bunlar bu iki geçiste anlaşılır. Hata mesajları ekranda görüntülenip çevirici sistemden çıkılır.

C) Tarama Modülleri :

Tarama modülleri, döngüde yer alan değişkenleri bir tabloda listeler. Bu değişken tablosunun sekiz sütunu bulunup, her değişken için bu sütunlar doldurulur.

1.Sütun (identification) : Değişken

- a) FOR deyiminde yer alıyorsa ident=0,
- b) aritmetik ifadede yer alıyorsa ident=1,
- c) indis ifadesinde yer alıyorsa ident=2.

2.Sütun(name): Değişken isminin yer aldığı sütun.

3.Sütun(variable type) : Değişken,

- a) basit değişken ise varitype=0
- b) dizi değişken ise varitype=1 olur.

4.Sütun(variable status): Değişken,

- a) eşitliğin solunda yani üzerine yazma yapılıyorsa varistat=1,
- b) eşitliğin sağında yani okunuyorsa varistat=0 olur.

5.Sütun(number of index) : Değişken dizi tipindense noindex=dizi boyutu.

6.Sütun(expression number) : Değişkenin yer aldığı ifadenin kaçınıcı deyim olduğunu gösterir.

7.Sütun(variable declaration) : Değişkenin tipi (tamsayı, karakter, uzun tamsayı,ondalık veya uzun ondalık) bu sütunda tutulur.

8.Sütun(variable index exp.) : Değişken dizi tipindense, indis ifadesinde yer alan değişkenlerin katsayıları bu alanda tutulur.

Örnek : $2*I1+3*I2+5$ indis ifadesinde varia alanında 5,2,3 tutulur.

Örnek : $-2+2*I1-4*I1+5$ indis ifadesinde varia alanında $3(5-2=3)$, $-2(-4+2=-2)$ tutulur. Görüldüğü gibi katsayılar hesaplanarak varia alanına yerleştirilir.

Beş tane tarama modülü vardır :

C-1)Kaynak Kodunda FOR deyimini arayan modül : Bu tarama modülü, kaynak kodunu karakter bazında okuyarak FOR deyimini arar. FOR döngüsü bulunduğunda, kodun bu pozisyonuna ilişkin bir işaretçi üretir.

C-2) FOR Deyimini Tarama Modülü : FOR deyiminde yer alan değişkenleri ortaya çıkarıp bunları değişken tablosuna ekler.

C-3) Deyim Tarama Modülü : Aritmetik ifadeleri tarayıp, bu ifadelerdeki değişkenleri, değişken tablosuna ekler.

C-4) Indis ifadesini Tarama Modülü : Bu modül indis ifadesini tarayıp, bu ifadenin Postfix notasyonunu üretir ve indis katsayılarını değişken tablosuna yerleştirir. Postfix notasyonu, derleyicilerin aritmetik ifadeleri hesaplamak için kullandıkları notasyondur [11].

Örnek 1 : $5+2*I1-3*I2$ ifadesini ele alalım.

Bu ifadenin Postfix notasyonu : $5I0*+2I1*+3I2*-$

Örnek 2 : $-2+2*I2-5*I1+4*I2$

Bu ifadenin Postfix notasyonu : $2I0*-6I2*+5I1*-$

Görüldüğü gibi, indis ifadesinde değişkenlerin birden fazla katsayıları bulunabilir. Ancak bu katsayılar tabloya yerleştirilmeden önce, toplanmalı/çıkarılmalıdır ve tek bir katsayı haline getirilmelidir.

C-5) Deklarasyon Modülü : Bu modül, kaynak kodunu baştan tarayarak, değişken tablosunda yer alan değişkenlerin tipini (karakter, tamsayı, ondalık sayı, uzun tamsayı gibi) belirler. Değişkenlerin tipinin belirlenmesi, daha sonra anlatılacak olan süreçler arasındaki haberleşmede önem kazanır.

D) Değişken Bağımlılık Matrisini Oluşturma Modülü :

Bu modül, bir önceki aşamada oluşturulan değişken tablosunu analiz eder veri bağımlılığını bir matris şeklinde ortaya koyar. Değişken tablosu tarama modülleri tarafından oluşturulur ve döngüde yer alan tüm değişkenlerin gerekli özellikliklerini içerir.

Değişkenler arasındaki bağımlılık ilişkisi (doğru, ters ve çıkış bağımlılığı) tespit edilir. Bu tespiti yapmak için, değişken tablosunda yeterli bilgiler (değişkenin ismi, eşitliğin solunda/sağında yer alışı, dizi/sabit değişken vs.) bulunur. Herhangi iki deyim arasında (örnek x ve y deyimi) bağımlılık bulunduğunda (SxBSy), bağımlılık matrisinin x.satırı ve y. sütununa "1" sabiti konur. Bağımlılık grafında bu ilişki, x düğümünden y düğümüne bir ark ile gösteriliyordu. Satır ve sütunlarda "0" bulunuyorsa, bu satır ve sütun numaralı deyimler arasında bağımlılık yoktur.

$$DBM = \begin{matrix} & y \\ & \vdots \\ x & \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix} \\ & \dots 1 \end{matrix}$$

- Deyimler arasında ters bağımlılık varsa, değişken bağımlılık matrisinde (DBM) x. satır, y. sütuna "1" konulduğu gibi y. satır, x.sütuna da "1" konur. Yani değişken bağımlılık grafında x-y arasında çevre oluşturulur.

- Dizi tipindeki değişkenler arasındaki bağımlılık incelenirken, değişken tablosunun 8. sütunundan (varia) faydalanılır.

Örnek 1 :

S1: $A(i+1)=\dots\dots\dots$

S2: $\dots\dots=A(i)+\dots\dots$

S1 deyimindeki A değişkeni için $varia[0]=1$, $varia[1]=1$
S2 deyimindeki A değişkeni için $varia[0]=0$, $varia[1]=1$
Bölüm 2.4.1 de verilen kurallara göre S1S2. Bu $varia$ 'nın 0. elemanı ile anlaşılır. Yani S1 deyimi A değişkenine, S2 deyimine göre daha önce erişir. Her iki deyimde de $varia[1]=1$ olması, aynı indis katsayıları bulunduğunu gösterir. $DBM[1][2]=1$

Örnek 2 :

DO 10 I=1,20

S1: $\dots\dots=\dots\dots+A(I+1)$

S2: $A(I+1)=\dots\dots+$

10 CONTINUE

S1 deyimindeki A değişkeni için $varia[0]=1$, $varia[1]=1$
S2 deyimindeki A değişkeni için $varia[0]=1$, $varia[1]=1$
Her iki deyimdeki $varia$ 'ların aynı olması iterasyonlar arası bağımlılık olmadığını gösterir. Dizi tipindeki bu değişkenlerin bağımlılığını incelerken bunlar sabit değişler gibi ele alınabilir.

Tanımaya göre S1 ve S2 arasında ters bağımlılık bulunur.

$DBM[1][2]=1$, $DBM[2][1]=1$

Örnek 3 :

DO 10 I=1,K+1

S1: $A[2*I+3]=\dots+\dots$

S2: $\dots\dots=A[I]+\dots\dots$

10 CONTINUE

Bölüm 2.4.1 de verilen kurallar en genel haldeki bir döngüye uygulanamaz. Bu kuralların uygulanabilmesi için döngünün sabit bir üst sınırının bulunması ve iterasyon artımlarının bir olması gerekir.

Halbuki 'C' dilinde döngülerin bir üst sınırın olması şart olmayıp, döngünün bitimi bir eşitsizlikle kontrol edilebilir. Bundan dolayı bu kural uygulanmamıştır.

Bu iki deyimin bağımlılığı için,
 $f(x)=2x+3$ $g(y)=y$ olmak üzere
 $f(x)-g(y) \neq 0$ (tüm iterasyonlar için) çözümü gereklidir.
Bu çözümü bulmak için, çevirici sistemin ayrı bir
program kodu oluşturup, bunu derledikten sonra
koşturmalıdır. Bu işlem oldukça zaman alır ve
performansı olumsuz yönde etkiler. Bundan dolayı bu gibi
indis ifadeleri ortaya çıktığında DBM[1][2]=1 ve
DBM[2][1]=1 yapılır. Yani bu iki deyimin aynı blokta yer
alması sağlanır.

Örnek 4 :

```
DO 10 I=1,20
```

```
S1: A(-2*I+3*I+2)=.....+.....
```

```
S2: .....=A(I).....
```

```
10 CONTINUE
```

Her iki deyimin postfix notasyonu alındıktan sonra,

S1 deyimi için varia[0]=2, varia[1]=1

S2 deyimi için varia[0]=0, varia[1]=1

Görüldüğü gibi indis ifadesinin hesaplanması örnekteki
deyimler için anlam kazanmaktadır. Her iki deyimin
varia[0] alanlarına bakıldığında, iterasyonlar arası
bağımlılık söz konusudur. S1S2 den dolayı
DBM[1][2]=1 olur.

E) Parçalama modülü :

Bu modül DBM matrisini analiz ederek, deyimleri π
bloklarına ayırır. Bu π blokları, sıra tanımına uygun
olarak gruplandırılır ve sıralar oluşturulur. DBM 'den π
blokları arasındaki bağımlılık ilişkiler ortaya
çıkarılır. π blokları arasındaki bağımlılık grafi
çizildiğinde, bu grafta çevre olamayacağı açıktır. π
blokları arasındaki bağımlılık ilişkisinden faydalanarak,
deyimlerin yeni yürütülme sırası tespit edilir. Yeni
yürütülme sırasında, π blokları arasındaki geriye doğru
olan bağımlılıklar ortadan kaldırılmıştır. Çevirici
sistemin bu aşamasında, hangi deyimin paralel,
hangilerinin ardışıl yürüyeceği belirlenmiştir.

F)Kod Oluşturma Oluşturma Modülü:

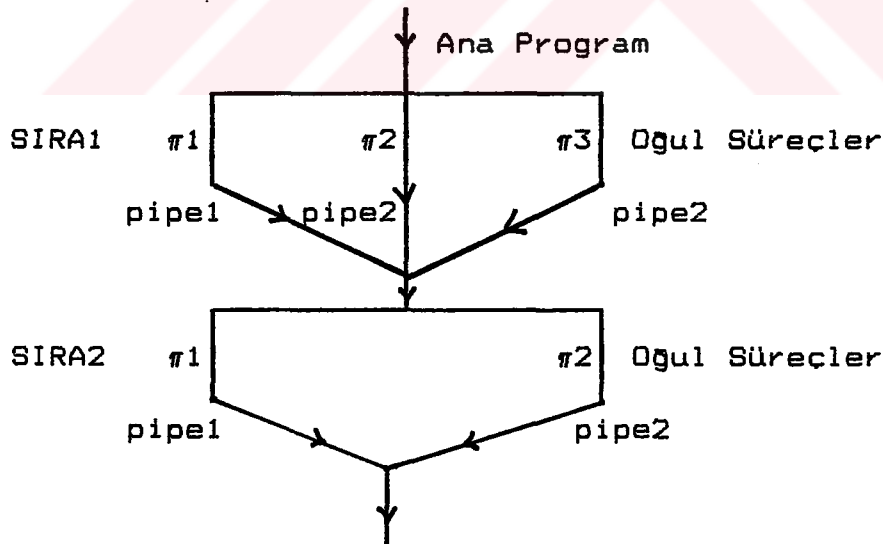
Bu modülde, kaynak koduna bir önceki aşamada belirlenen deyimlerin paralel yürüyebilmesi ve süreçler arasındaki senkronizasyonu sağlayan deyimler eklenir. Eklenen deyimler, aşağıdaki anlatılan çalışma şeklini sağlayacak şekildedir.

Eklenen senkronizasyon kodları ile , örneğin birinci sıradaki π bloklarında yer alan deyimler birer oğul süreç olarak paralel çalıştırılır. Bu deyimler, ürettikleri sonuçları pipe üzerinden baba sürece aktarırlar. Baba süreç, oğul süreçlerden tüm sonuç verileri okuduktan sonra ikinci sıraya ait olan π bloklarını paralel çalıştırır. Sonuç verileri, oğuldan babaya transfer edildikten sonra aynı mantıkta sonraki sıralar çalıştırılır.

Örnek : Bir döngü π blokları algoritmasına göre parçalandıktan sonra iki sıra oluşsun.

Sıra1={ $\pi1$, $\pi2$, $\pi3$ } , Sıra2={ $\pi4$, $\pi5$ }

Programın akışı, şekil 5.3' de olacak şekilde düzenlenir.



Şekil 5.3 Döngünün Eş Anlı Yürütülmesi

Baba, oğul süreçleri çalıştırdıktan sonra her bir oğul sürece bağlı olan pipe'dan bilgi beklemeye başlar. Oğul süreçler, deyimleri yürüttükten sonra, bunların sonuçlarını yani eşitliğin solunda yer alan değişkenlerin içeriklerini pipe'e yazarlar. Tüm değişkenler pipe'e

yazıldıktan sonra oğul süreçler sonlanırlar. Baba süreç, pipe'leri teker teker tarayarak okur. Yani pipe1 den bir değer okuduktan sonra sıra ile pipe2 ve pipe3 den okur. Okumanın bu şekilde yapılması pipe'lerde birikimi önler ve oğul süreçlerin gerçek olarak paralel çalışmasını sağlar.

Oğul süreç, babaya ürettikleri sonuçları aktaracağı için, pipe'ın yönü daima oğuldan babaya doğrudur. Yani oğul pipe'a sürekli yazar, baba pipe'den sürekli okur.

Pipe'lar kaynak kodun çalışması sırasında yaratılacağı için, koda ilgili deyimler eklenir. Tanımlanan pipe'ların sayısı, maksimum yürüyecek oğul süreçlerin sayısı kadardır. Diğer bir deyişle, sıralarda yer alan π bloklarının maksimum sayısı kadardır. Örnekte Sıra1 maksimum sayıda π bloğu içermektedir. Dolayısıyla tanımlanacak pipe'ların sayısı üçtür.

Bu modülün diğer bir fonksiyonu, bölüm 4.3.2 de açıklanan 1. özel durumdan dolayı sabit değişkenlerin gerektiğinde dizi değişkenine çevirmektir. Bu özel durumu tekrar bir örneklerle açıklamak faydalı olacaktır.

Örnek 1 :

```
DO 20 I=1,20
```

```
S1: K=A(I)+B(I)
```

```
S2: C(I)=A(I)+1
```

```
S3: D(I)=K+1
```

```
20 CONTINUE
```

```
 $\pi 1=\{S1\}$ ,  $\pi 2=\{S2\}$ ,  $\pi 3=\{S3\}$ 
```

```
Sıra1= $\{\pi 1, \pi 2\}$  Sıra2= $\{\pi 3\}$ 
```

Yönteme göre , $\pi 1$ ve $\pi 2$ birbirine paralel olarak tüm iterasyonlar için yürütüldükten sonra $\pi 3$ yürütülecektir. Halbuki döngünün doğru olarak çalışması için, örneğin 1. iterasyonda üretilen K değeri yine 1. iterasyondaki S3 deyimine verilmelidir. Bunu sağlamak için, K sabitine indis atanır ve 1. iterasyonda elde edilen K değeri, K dizisinin 1. alanında saklanır. Program, döngü dışında da K değerini kullanabilir. Bundan dolayı programda hem

sabit deęişken olarak K, hem de dizi deęişkeni olarak yeni K deęişkeni bulunmalıdır. Bu sebepten dizi deęişkeni olarak tanımlanan K' ya yeni bir isim verilmeli ve bu yeni deęişkenin kaynak kodunda deklarasyonu yapılmalıdır.

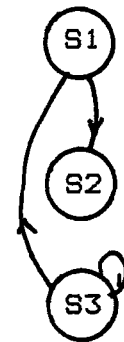
```
K_YENI(1)=K
DO 20 I=1,20
S1: K_YENI(I)=A(I)+B(I)
S2: C(I)=A(I)+1
S3: D(I)=K_YENI(I)+1
20 CONTINUE
K=K_YENI(I)
```

Örnek 2 :

```
DO 20 I=1,20
S1: A(I)=K+1
S2: C(I)=A(I)+B(I)
S3: K=K+1
20 CONTINUE
```

S3 deyimi tarafından i. iterasyonda üretilen K değeri, i+1. iterasyonda S1 deyimi tarafından kullanılacaktır. Çevirici sistem bu sorunu , uygun indisleri atayarak çözer. Döngünün yeni şekli :

```
K_YENI(1)=K
DO 20 I=1,20
S1: A(I)=K_YENI(I)+1;
S2: C(I)=A(I)+B(I)
S3: K_YENI(I+1)=K_YENI(I)+1
20 CONTINUE
K=K_YENI(I)
```



Şekil 5.4 DBG

Yeni döngünün bağımlılık grafi şekil 5.4 de gösterilmiştir.

Değişken tipinin ne olduğu bu modülde önem kazanmaktadır. Örneğin pipe'a gönderilen "23", baba süreç tarafından hem tamsayı olarak hem de karakter

katarı olarak okunabilir. Çevirici sistem bu ayırımı, değişken tablosundan bakarak yapar ve kaynak kodunda gereken düzenlemeyi yapar.

5.3.2 Çevirici Sistemin Çalışmasını Örnek Üzerinde Gösterilmesi :

Örnek 1: Aşağıdaki C programındaki döngü paralel çalışan alt gruplara ayrılacaktır.

```
main()
{
    int a[20], b[20];
    int DEG, g[30], c[30];
    int i, k;

    for(i=0; i<=19; i++) a[i]=b[i]=c[i]=g[i]=i;
    k=5;

    for(i=0; i<=20; i=i+2) {
        S1: a[i]=k+b[i];
        S2: g[i]=k+b[i];
        S3: a[i]=b[i]+1;
        S4: a[i+1]=b[i+1]*3;
    }

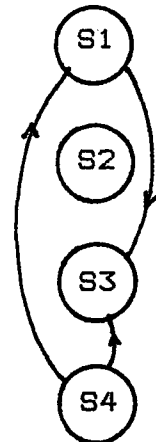
    for(i=0; i<=19; i++) printf("A=%d B=%d C=%d\n", a[i],
        b[i], c[i]);
    printf("k=%d\n", k);
}
```

Tarayıcılar sonucunda değişken tablosu (şekil 5.6) oluşur.

Bağımlılıklar : S1δS3, S4δS1, S4δS3

Değişken bağımlılık grafi şekil 5.5 de yer almaktadır.

$$DBM = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$



Şekil 5.5 DBG (Örnek 1)

İsole noktalar : S1, S2, S3, S4

$\pi_1 = \{S1\}$, $\pi_2 = \{S2\}$, $\pi_3 = \{S3\}$, $\pi_4 = \{S4\}$

π blokları arasındaki bağımlılık ilişkileri :

$\pi_1 \Gamma \pi_3$, $\pi_4 \Gamma \pi_3$, $\pi_4 \Gamma \pi_1$

π bloklarının yeni yürütme sırası : $\pi 4$, $\pi 2$, $\pi 1$, $\pi 3$
Sıra1={ $\pi 4$, $\pi 2$ }, Sıra2={ $\pi 1$, $\pi 3$ }

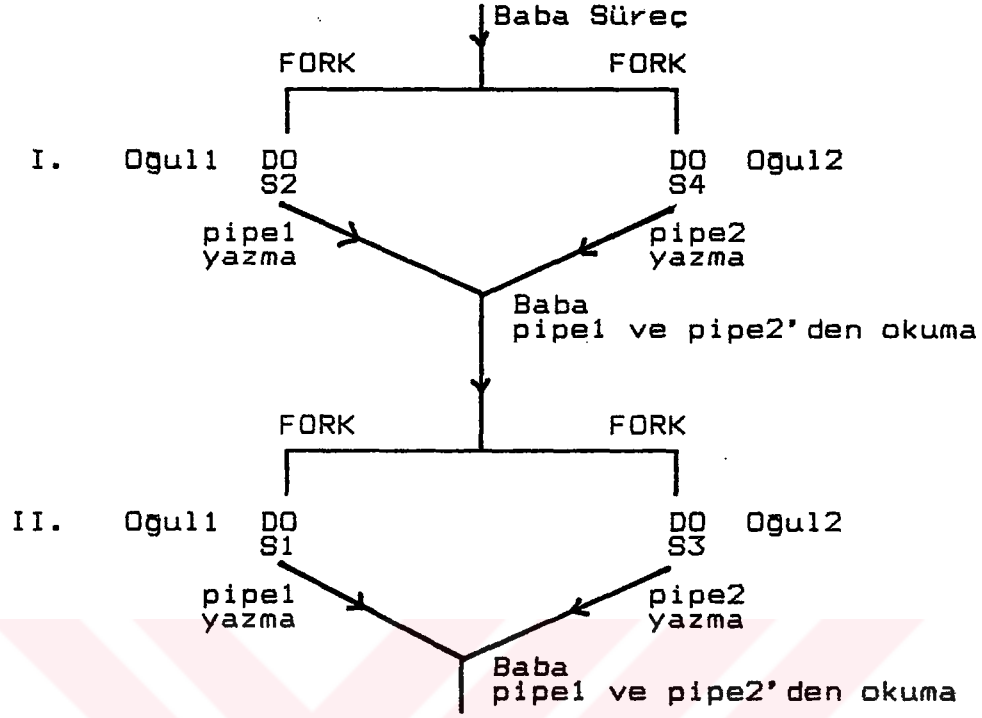
ident	name	v.type	v.stat	noindex	expno	vdecl	varia
0	i	0	1	-	-	int	-
1	a	1	1	20	1	int	0,1
2	i	0	0	-	1	int	-
1	k	0	0	-	1	int	-
1	b	1	0	20	1	int	0,1
2	i	0	0	-	1	int	-
1	g	1	1	20	2	int	0,1
2	i	0	0	-	2	int	-
1	k	0	0	-	2	int	-
1	b	1	1	20	2	int	0,1
2	i	0	0	-	2	int	-
1	a	1	1	30	3	int	0,1
2	i	0	0	-	3	int	-
1	b	1	0	20	3	int	0,1
2	i	0	0	-	3	int	-
1	a	1	1	20	4	int	1,1
2	i	0	0	-	4	int	-
1	b	1	0	20	4	int	1,1
2	i	0	0	-	4	int	-

Şekil 5.6 Değişken Tablosu (Örnek 1)

Kod oluşturma modülü iki pipe kurar. S4 ve S2 deyimleri, FORK deyimi ile iterasyonun tüm adımları için yürütülür. Herbir oğul süreç, deyimleri yürüttükten sonra a ve g dizilerinin içeriklerini pipe'a yazarlar. Aynı anda baba süreç bunları okur (şekil 5.7).I. ve II. aşamada aynı anda üç süreç çalışır : 2 oğul ve bir baba.

Kod oluşturma modülü, kaynak kodda döngü yerine bu çalışmayı sağlayacak deyimler koyar.

C programına bakıldığında, 1. FOR döngüsü tek bir deyim ve 3. FOR döngüsü aritmetik deyim olmayan bir deyim içerdiğinden her iki FOR üzerinde işlem yapılmaz.



Şekil 5.7 Programın Paralel Çalıştırılması (Örnek 1)

Örnek 2: Aşağıdaki C programındaki döngü paralel çalışan alt gruplara ayrılacaktır.

```
main()
{
    int dim[100],a[20];
    int b[20],g[20],d[40],c[20],z[20],h[20];
    int i,k;
    for(i=0;i<=19;i++) a[i]=b[i]=c[i]=g[i]=i;
    k=5;
    do{
        a[i]=k+1;
        ++k;
    }while(k<10);

    for(i=0;i<=18;i++){
        S1: a[i]=a[i+1]*2;
        S2: b[i]=k+3;
        S3: g[i]=g[i]+2*(i+1);
        S4: c[i]=d[i]+5*(d[i]/2);
        S5: h[i]=z[i]+14+d[i+1];
        S6: d[i+k]=k=h[i+1]+1;
    }
}
```

Deyimler arasındaki bağımlılık ilişkileri : S1δS1, S6δS2, S4δS6, S5δS6, S6δS5, S6δS4

$$DBM = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Değişken tablosu, şekil 5.8 de yer almaktadır.

isole noktalar: S2,S3

Cevreler: {S4,S6}, {S5,S6}, {S4,S5,S6}, {S1}

Maksimum çevreler: {S4,S5,S6}, {S1}

$\pi_1=\{S1\}$, $\pi_2=\{S2\}$, $\pi_3=\{S3\}$, $\pi_4=\{S4,S5,S6\}$

π blokları arasındaki bağımlılıklar : $\pi_4 \rightarrow \pi_2$

π bloklarının yeni yürütülme sırası: π_1 , π_4 , π_3 , π_2

Sıra1= $\{\pi_1, \pi_4, \pi_3\}$ Sıra2= $\{\pi_2\}$

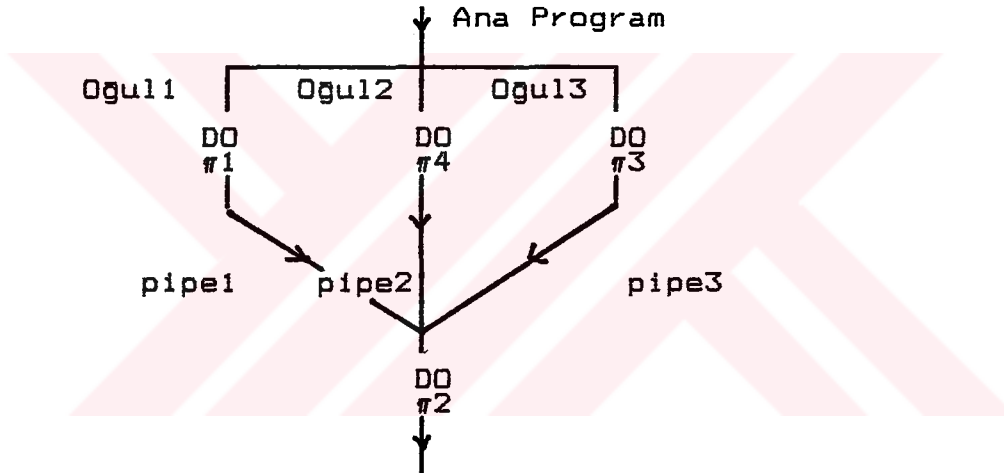
ident	name	v.type	v.stat	noindex	expno	vdecl	varia
0	i	0	1	-	-	int	-
1	a	1	1	20	1	int	0,1
2	i	0	0	-	1	int	-
1	a	1	0	20	1	int	1,1
2	i	0	0	-	1	int	
1	b	1	1	20	2	int	0,1
2	i	0	0	-	2	int	
1	k	0	0	-	2	int	-
1	g	1	1	20	3	int	0,1
2	i	0	0	-	3	int	
1	g	1	0	20	3	int	0,1
2	i	0	0	-	3	int	
1	j	0	0	-	3	int	-
1	c	1	1	20	4	int	0,1
2	i	0	0	-	4	int	-
1	d	1	0	40	4	int	0,1
2	i	0	0	-	4	int	-
1	d	1	0	40	4	int	0,1
2	i	0	0	-	4	int	-
1	h	1	1	20	5	int	0,1
2	i	0	0	-	5	int	-
1	z	1	0	20	5	int	0,1
2	i	0	0	-	5	int	-
1	d	1	0	40	5	int	1,1
2	i	0	0	-	5	int	-
1	d	1	1	40	6	int	0,1,1
2	i	0	0	-	6	int	-
2	k	0	0	-	6	int	-
1	k	0	1	-	6	int	-
1	h	1	0	20	6	int	1,1
2	i	0	0	-	6	int	

Şekil 5.8 Değişken Tablosu (Örnek 2)

Döngüdeki k değişkenine uygun indisler verilir.

```
i_yeni=0;
k_yeni[0]=k
for(i=0;i<=18;i++,i_yeni++){
a[i]=a[i+1]*2;
b[i]=k_yeni[i_yeni]+3;
g[i]=g[i]+2*(j+1);
c[i]=d[i]+5*(d[i]/2);
h[i]=z[i]+5+d[i+1];
d[i+k_yeni[i_yeni]]=k[i_yeni+1]=h[i+1]+1;
}
```

Programın akışı şekil 5.9 de olduğu gibi düzenlenir.



Şekil 5.9 Programın Paralel Çalıştırılması (Örnek 2)

5.3.3 Sınırlamalar Ve Kabuller :

Bu uygulama bir takım kabuller altında gerçekleştirilmiştir. Çevirici sistem "C" kaynak kodunu incelemektedir. "C" dili oldukça geniş kapsamlı olduğundan, "C" 'nin özellikleriyle ortaya çıkan tüm durumlara çözüm getirilememiştir.

1) Aritmetik ifadelerden oluşan döngüler üzerinde çalışılmıştır. Bu döngüler, tek bir deyimden oluşuyorsa iterasyon bazında paralellik uygulanmadığı için bu döngü üzerinde işlem yapılmaz. Aynı şekilde döngü C'nin bir deyimini veya fonksiyonunu içerdiği durumu göz önüne alınmamıştır.

2) Indis ifadeleri $a_0 \pm a_1x \pm a_2x \pm \dots \pm a_nx$ şeklinde lineer fonksiyon olması gerekir.

3) Tamsayı, karakter, uzun tamsayı, ondalık veya uzun ondalık sayı tipindeki değişkenler üzerinde işlem yapılmaktadır. Aritmetik ifadelerde karakter katarı olmayacağı açıktır. İşaretci (pointer) üzerinde işlem yapılmaz.

4) Tek boyutlu dizi değişkenleri arasındaki bağımlılık incelenmektedir. Dizi değişkeni çok boyutlu olduğu zaman, bölüm 2.4.1 de verilen kurallar geçerli değildir.

```
DO 10 I=1,100
```

```
DO 20 J=1,100
```

```
S1: A[I][J]=.....
```

```
S2: .....=A[I+1][J].....
```

```
20 CONTINUE
```

```
10 CONTINUE
```

Aynı isimdeki dizi değişkenlerinde, boyutlardan ancak biri diğerine göre değişirse bağımlılık belirlenebilir (S2&S1). Ancak genelleştirme yapılırsa bağımlılığı tespit etmek zorlaşır.

```
DO 10 I=1,100
```

```
DO 20 J=1,100
```

```
S1: A[I+1][J]=.....
```

```
S2: .....=A[2*I][J-1+I]
```

```
10 CONTINUE
```

```
20 CONTINUE
```

Boyutlardan biri aynı olmadığı için, boyutları birer birer analiz etmek sonuç getirmez. İki indis ifadesinin $1 \leq I \leq 100$, $1 \leq J \leq 100$ arasında çakışıp çakışmadığı, ancak bu aralıkta indis ifadelerinin hesaplanmasıyla anlaşılır. Bu, çevirici sistemin yeni program üretmesi, bunu derlemesi ve koşturması ile olur. Bu işlemler çok uzun olup çevirici sistemin hızını düşürür.

5) Dış döngü deyimleri dışında, döngünün içinde yer alan içiçe döngüler desteklenmemektedir. İçiçe alt döngüler bulunuyorsa bağımlılık incelemeleri gittikçe karmaşılaşmaktadır. İç döngülerdeki deyimler, artık tek bir deyim olarak düşünülmeyip, bu deyim dahil olduğu

döngü deyiminde yer alan değişkenler de göz önüne alınmalıdır. (DEG € DO1 1. döngü deyiminde (DO) yer alan değişkenler anlamına gelir.) C'nin FOR deyimleri FORTRAN dilindeki kadar basit olmayıp içersinde çeşitli atama ifadeleri, değişkenler bulunabilir.

```
DO 10..... S17ßS3 testi yapılmak istensin
DO 20..... Aşağıdaki bağımlılık testleri
S1           yapılmalı :
S2           S17ß(DEG€DO3)
DO 30..... S17ßS3
S3           (DEG€DO1)?ß(DEG€DO3)
S4           (DEG€DO2)?ß(DEG€DO3)
30 CONTINUE
S5
S6
20 CONTINUE
10 CONTINUE
```

Bu döngüyü daha genelleştirelim.

```
DO 10..... S17ßS5 testi için
DO 20..... S17ßS5
S1           S17ß(DEG€DO3)
S2           S37ß(DEG€DO4)
DO 30..... (DEG€DO1)?ß(DEG€DO3)
S3           (DEG€DO2)?ß(DEG€DO4)
S4           (DEG€DO3)?ß(DEG€DO4)
DO 40..... S57ß(DEG€DO3)
S5           (DEG€DO1)?ß(DEG€DO4)
40 CONTINUE (DEG€DO2)?ß(DEG€DO3)
30 CONTINUE
20 CONTINUE
S6
10 CONTINUE
```

Görüldüğü gibi en genel halde durum oldukça karmaşıktır.

6) Döngü deyiminde yer alan indis, iterasyon adımını belirler. Bu indisin döngü içersinde eşitliğin solunda yer alıp bunlara yeni değer atanması durumu güçleştirir. Bu durumda indis değişkeni, hem bu deyimde hem de döngü

deyiminde hesaplanır.

İterasyon adımları genellikle sabit değişkenlerle belirlenir. Döngü parçalanması metodunda sabit değişkenler dizi boyutuna çevrilirler. Böylece iterasyon adımı dizide yer almış olurki bu durumda döngü yanlış çözüm üretir.

Örnek :

```
for(i=0;i<=20;i++){  
    i=A[i]+5;  
    B[i]=D[i]+3;  
}
```

Basit değişkenlere indis ekleme algoritması bu döngüyü aşağıdaki şekle dönüştürür.

```
for(i=0;i<=20;i_yeni[i]++,i++){  
    i_yeni[i]=A[i_yeni[i]]+5;  
    B[i_yeni[i]]=D[i_yeni[i]]+3;  
}
```

Bu problemin çözümü getirilmemiştir.

SONUÇLAR VE ÖNERİLER

Çok işlemcili sistemlerin kullanılma alanlarından biri çoklu programlamadır. Çoklu programlamada herbir işlemci farklı işleri yürütür. Çok işlemcili sistemlerin diğer bir kullanılma alanı, belirli bir anda sistemin bütününün tek bir programa hizmet vermesidir. Bu tez çalışmasında ele alınan bu durumdur. Başka bir deyişle, ortada yüklü bir program vardır. Bu sistemdeki derleyici, programı birbirinden bağımsız olarak çalışabilecek alt süreçler oluşacak şekilde derler. Bu alt süreçler sistemin farklı işlemcilerinde çalışarak programın bitim süresini kısaltırlar. Bu tez çalışmasında amaç bu tip derleyicilerin, programları hangi yöntemleri kullanarak alt süreçlere ayırdığını incelemektir.

Paralelliği algılamamanın temeli deyimlerin birbiriyle olan ilişkisini bulmaktır. Bunun için ilk olarak 1966 yılında BERNSTEIN tarafından ortaya atılan değişken bağımlılık kuralları kullanılmıştır. Bu kurallar deyimlerin eş anlı yürüyebileceğini ortaya çıkardığı gibi deyimler birbirine bağlı olması durumunda bunların yürütülme sıralarını verir. Değişken bağımlılığı matematiksel olarak bir graf şeklinde gösterilir. Bu grafta, düğümler deyimlere, arklarda bağımlılık ilişkisine karşı gelir. Oluşan grafa değişken bağımlılık grafı denir. Bu graf, bağımlılık ilişkisini düzenli bir şekilde göstermekle kalmayıp, eş anlı yürüyebilecek süreçlerin bulunmasında teorik bir taban teşkil eder. Programlardaki paralelliği algılayan yöntemlerden tümü değişken bağımlılık grafını kullanır.

Programlarda alt program, deyim blokları ve döngüler düzeyinde paralellik söz konusudur. Çalışma aritmetik deyimlerden oluşan döngüler üzerinde yoğunlaştırıldı. Programlar arasındaki paralelligi algılayan bir dizi yöntem bulunur. Bu yöntemlerden biri ele alınmış ve uygulama şeklinde ortaya konulmuştur. Uygulama çoklu işleme özelliği bulunan UNIX işletim sisteminde yapılmıştır. Çalışma, yukarıda bahsedilen özellikteki çok işlemcili sistem üzerinde yapılmadığından, ortaya çıkan uygulama bir modelleme şeklinde düşünülmelidir. Tasarlanan uygulama, ardışıl dilde yazılmış bir kaynak kodunu paralel çalışabilen bir yapıya çevirdiğinden uygulamaya, çevirici sistem denmektedir. Çevirici sistem, "C" dilinde yazılmış bir kaynak kodunu incelemektedir. Bu kodda yer alan, aritmetik deyimlerden oluşan döngülerde hangi deyimlerin paralel yürüyeceği ortaya çıkarılır. Bu deyimler birbiriyle paralel yürüyebilecek şekilde, kaynak koduna deyimler eklenir. Bu şekilde, çevirici sistem tarafından oluşturulan yeni kod derlenip çalıştırıldığında yürüme esnasında döngüler eş anlı süreçler şeklinde çalıştırılır. Tasarlanan çevirici sistem, bu işlemi bir takım kısıtlamalar halinde yürütüp "C" dilinin sağladığı tüm durumlara çözüm getirememektedir.

Bu tezin kapsamına yalnız döngü şeklindeki yapılar alınmıştır. Bu çalışma genişletilip daha geniş kapsamlı döngü uygulamaları ve kontrol yapıları ele alınabilir. Değerli çalışmaların ortaya çıkacağı inancındayım.

KAYNAKLAR

- [1] KUCK, D., On the Number of Operations Simultaneously Executable in FORTRAN-Like Programs, their Resulting Speedup, IEEE Transaction on Comp. Vol. C-21, No.12, pp 668-680 ,December 1972.
- [2] BERNSTEIN, A.J., Analysis of Programs for Parallel Processing, IEEE Transaction on Computers, Vol Ec-15, No.5 , pp 757-763, October 1966.
- [3] ALLEN, J.R. and KENNEY, K., Automatic Translation of FORTRAN Programs to Vector Form, ACM Transactions on Programming Languages and Systems, Vol. 9, No.4, pp 491-542, October 1987.
- [4] WOLFE, M., Multiprocessor Synchronization for Concurrent Loops, IEEE Software pp. 34-42, January 1988.
- [5] KUCK, D., High Speed Multiprocessor and Compilation Techniques, IEEE Transaction on Computers, Vol. C-29, No. 9, pp 763-776, September 1980.
- [6] BOES, R. and REIMANN, B., UNIX Magazin SIEMENS pp 837-889, November 1989.
- [7] HWANG, K. and BRIGGS, F., Computer Architecture and Parallel Processing, Mc Graw Hill pp 533-572 1985.
- [8] BANERJEE, U., Time and Parallel Processors Bounds for FORTRAN Like Loops, IEEE Trans. on Comp. Vol. C-28, No.9, pp 660-669, September 1979.
- [9] ALLEN, J.R. and KENNEDY, K., A Parallel Programming Environment, IEEE Software, pp 21-29 July 1985.
- [10] Programmer's Reference Manual, SCD XENIX System V, Santa Cruz Operation Inc, pp 151-302, 1987.

[11] HOROWITZ, E. and SAHNI, S., Fundamentals of Data Structures, Computer Science Press USA pp 287-300, pp 91-97, 1976.



DZGEÇMİŞ

GÜlden ÇEVİK, 1965 yılında İstanbul'da doğdu. Lise öğrenimini İstanbul, Fenerbahçe Lisesinde tamamladı. 1982 yılında İ.T.Ü. Elektrik-Elektronik Fakültesi Kontrol ve Bilgisayar bölümüne girdi ve 1986'da bu bölümden mezun oldu. 1987 yılında İ.T.Ü. Fen Bilimleri Enstitüsü Elektrik-Elektronik Ana bilim dalı Kontrol ve Bilgisayar programında yüksek öğrenim görmeye hak kazandı. Halen SIEMENS-NIXDORF şirketinde, sistem yazılım uzmanı olarak görev yapmaktadır.

