EGE ÜNİVERSİTESİ

YÜKSEK LİSANS TEZİ

HİBRİT BİR KRİPTO ALGORİTMASININ PARALELLEŞTİRİLEREK ÇOK ÇEKİRDEKLİ İŞLEMCİLERİN PERFORMANSININ ANALİZ EDİLMESİ

Ecem İREN

Tez Danışmanı : Doç. Dr. Aylin Kantarcı

Bilgisayar Mühendisliği Anabilim Dalı

Bilim Dalı Kodu : 619.01.00
Sunuş Tarihi : 20.11.2014

Bornova-İZMİR
2014

EGE ÜNİVERSİTESİ FEN BİLİMLERİ ENSTİTÜSÜ

(YÜKSEK LİSANS TEZİ)

HİBRİT BİR KRİPTO ALGORİTMASININ PARALELLEŞTİRİLEREK ÇOK ÇEKİRDEKLİ İŞLEMCİLERİN PERFORMANSININ ANALİZ EDİLMESİ

Ecem İREN

Tez Danışmanı : Doç. Dr. Aylin Kantarcı

Bilgisayar Mühendisliği Anabilim Dalı

Bilim Dalı Kodu : 619.01.00
Sunuş Tarihi : 20.11.2014

Bornova-İZMİR

2014

Ecem İREN tarafından Yüksek Lisans tezi olarak sunulan "Hibrit Bir Kripto Algoritmasının Paralelleştirilerek Çok Çekirdekli İşlemcilerin Performansının Analiz Edilmesi" başlıklı bu çalışma E.Ü. Lisansüstü Eğitim ve Öğretim Yönetmeliği ile E.Ü. Fen Bilimleri Enstitüsü Eğitim ve Öğretim Yönergesinin ilgili hükümleri uyarınca tarafımızdan değerlendirilerek savunmaya değer bulunmuş ve 20/11/2014 tarihinde yapılan tez savunma sınavında aday oybirliği/oyçokluğu ile başarılı bulunmuştur.

Jüri Üyeleri:	**İmza**
Jüri Başkanı : Doç. Dr. Aylin KANTARCI	**.....**
Raportör Üye: Yrd. Doç. Dr. Radosveta SOKULLU	**.....**
Üye : Doç. Dr. Rıza Cenk ERDUR	**.....**

EGE ÜNİVERSİTESİ FEN BİLİMLERİ ENSTİTÜSÜ

ETİK KURALLARA UYGUNLUK BEYANI

E.Ü. Lisansüstü Eğitim ve Öğretim Yönetmeliğinin ilgili hükümleri uyarınca Yüksek Lisans Tezi olarak sunduğum "Hibrit Bir Kripto Algoritmasının Paralelleştirilerek Çok Çekirdekli İşlemcilerin Performansının Analiz Edilmesi" başlıklı bu tezin kendi çalışmam olduğunu, sunduğum tüm sonuç, doküman, bilgi ve belgeleri bizzat ve bu tez çalışması kapsamında elde ettiğimi, bu tez çalışmasıyla elde edilmeyen bütün bilgi ve yorumlara atıf yaptığımı ve bunları kaynaklar listesinde usulüne uygun olarak verdiğimi, tez çalışması ve yazımı sırasında patent ve telif haklarını ihlal edici bir davranışımın olmadığını, bu tezin herhangi bir bölümünü bu üniversite veya diğer bir üniversitede başka bir tez çalışması içinde sunmadığımı, bu tezin planlanmasından yazımına kadar bütün safhalarda bilimsel etik kurallarına uygun olarak davrandığımı ve aksinin ortaya çıkması durumunda her türlü yasal sonucu kabul edeceğimi beyan ederim.

.... / .... / 20..

İmzası

Adı-Soyadı

ÖZET

HİBRİT BİR KRİPTO ALGORİTMASININ PARALELLEŞTİRİLEREK ÇOK ÇEKİRDEKLİ İŞLEMCİLERİN PERFORMANSININ ANALİZ EDİLMESİ

İREN, Ecem

Yüksek Lisans Tezi, Bilgisayar Mühendisliği Anabilim Dalı
Tez Danışmanı: Doç. Dr. Aylin Kantarcı
Kasım 2014, 79 sayfa

Bu tezde, MIRACL Kriptografik Kütüphanesinin rutinleri kullanılarak güvenliği sağlamak ve kriptografik algoritmanın performansı arttırmak amacıyla RSA ve AES algoritmalarından oluşan hibrit bir uygulama geliştirilmiştir. Seri olarak yazılmış bu uygulamanın çalışma performansını daha da arttırabilmek amacıyla çok çekirdekli işlemcilerde aynı uygulamanın paralel versiyonu yazılmıştır. Paralel versiyon yazılırken OpenMP Uygulama Programlama Arayüzü'nden yardım alınmıştır.

Program paralelleştirilirken öncelikle veri bağımlılıkları manuel olarak analiz edilmiş ve daha sonra Par4All programı ile bu analizler test edilip bağımlılıkların maksimum seviyede çözülmesi sağlanmıştır. Daha sonra seri programa OpenMP kütüphanesinden birtakım direktifler eklenerek uygulama hem iki çekirdekli hem de dört çekirdekli platformlarda görev ve veri seviyesinde paralelleştirilmiştir. Paralelleştirilen programda da performans analizi yapılarak seri uygulama ile karşılaştırılmıştır. Ayrıca çok çekirdekli işlemciler için ideal konfigürasyon kaç çekirdek olmalı ve Hyperthreading olmalı mı gibi sorulara cevaplar aranmıştır. Performans analizlerinin karşılaştırılması sonucu çok çekirdekli işlemciler ile ilgili birtakım yorumlar yapılmış ve çok çekirdekli işlemciler ile ilgili bazı öneriler listelenmiştir.

Anahtar sözcükler: Çok çekirdekli işlemciler, paralel programlama, OpenMP, RSA, AES, MIRACL Kriptografik Kütüphane

ABSTRACT

ANALYZING PERFORMANCE OF MULTICORE PROCESSORS BY PARALLELIZATION OF A HIBRID CRYPTO ALGORITHM

İREN, Ecem

MSc in Computer Eng.
Supervisor: Assoc. Prof. Dr. Aylin KANTARCI
November 2014, 79 pages

In this thesis, to provide security and increase performance of an cryptographic algorithm, a hibrit implementation that consists of RSA and AES is improved by using routines of MIRACL Cryptographic Library routines. To have more improving in the performance of hibrit implementation that is written as serial, same implementation is converted to parallel version on multicore processors. While parallel version is being written, OpenMP Application Programming Interface is used.

While program is being parallelized, firstly data dependencies are analysed as manual and then analysis are tested to provide a solution for data dependencies in maximum level. After solving data dependencies, implementation is parallelized in data level and task level on multicore platforms that has two and four cores. Parallelization is achieved by adding some directives from OpenMP library to serial implementation. Performance analysis are made on parallelized implementation and parallel performance is compared with serial performance. Also,for ideal configuration how many cores should be in a multiprocessor subject hyperthreading affect is analyzed. At the end of the comparisons, some comments are written and some advices are given about multicore processors.

Keywords: Multicore processors, parallel programming, OpenMP, RSA, AES, MIRACL Cryptographic Library

TEŞEKKÜR

Bu çalışma süresince kıymetli görüşlerinden yararlandığım, yakın ilgisini esirgemeyen ve tezin biçimlenmesinde yardımcı olan sayın Doç. Dr. Aylin Kantarcı'ya, teşekkürü bir borç bilirim. Ayrıca tez çalışması boyunca bana sürekli destek veren aileme de teşekkürlerimi sunarım.

İÇİNDEKİLER

Sayfa

İÇİNDEKİLER (devam)

Sayfa

İÇİNDEKİLER (devam)

Sayfa

ŞEKİLLER DİZİNİ

Şekil Sayfa

ŞEKİLLER DİZİNİ (devam)

ŞEKİLLER DİZİNİ (devam)

ŞEKİLLER DİZİNİ (devam)

ÇİZELGELER DİZİNİ

SİMGELER VE KISALTMALAR DİZİNİ

Simgeler	Açıklama
n	Güvenlik parametresi
e_i	Şifreleme üsleri
d_i	Özel anahtar parçaları
v_i	Şifreli metin parçaları
k	Girdi
c	Girdi
b	Girdi
b	Batch genişliği
p	Asal sayı
q	Asal sayı
φ(N)	Phi fonksiyonu
N	Ortak anahtar
P	İşlemci sayısı
S_p	Hız faktörü
T_p	P tane işlemcideki program süresi
T_p	Tek işlemcideki program süresi

Kısaltmalar

CPU	Merkezi İşlem Birimi
ALU	Aritmetik Mantık Birimi
RSA	Rivest Shamir Adleman
AES	Gelişmiş Şifreleme Algoritması
API	Uygulama Programlama Arayüzü
GPU	Grafik İşleme Birimi
ECC	Eliptik Eğri Kriptografi
HECC	Haskell Eliptik Eğri Kriptografi
IDEA	Uluslararası Veri Şifreleme Algoritması
OS	İşletim Sistemi
RNG	Rasgele Sayı Üreticisi

1. GİRİŞ

1.1 Problem Tanımı

Bugünlerde bilgisayar ağları bilginin değişimi konusunda daha fazla önem kazanmaya başlamıştır. Bu ağların en önemli gereksinimlerinden birisi de bilginin bir yerden diğerine güvenli bir şekilde iletiminin sağlanmasıdır. Bilginin göndericiden alıcıya ulaşmasında en güvenli yolu öneren tekniklerden birisi de kriptografidir. Kriptografinin temel amacı hassas bilginin alıcı dışındaki diğer tüm kişilere okunamaz hale gelmesini gerçekleştirmektir (Nagendra and Sekhar, 2014)

Kriptografik algoritmalar bilindiği üzere geçmişte 8 bit, 16 bit ve 32 bit'lik işlemcilerde geliştirilmişlerdir. Şifreleme ve deşifreleme işlemleri esnasında çok büyük boyuttaki verilerin düşük performansta çalışmaları, seri olarak yazılmış kripto algoritma uygulamalarının bu işlemcilerde karşılaştığı en büyük problemdir. Artık günümüzde, asimetrik ve simetrik kripto algoritmaları daha yeni ve modern bilgisayarlar ile daha yüksek performansta ve hızlı bir biçimde tekrar yazılabilirler. Günümüzün modern bilgisayarları yüksek işlem gücüne sahip çok işlemcili bilgisayarlardır ve 64 bit, 128 bit ve hatta daha fazla olan çok çekirdekli işlemciler bu bilgisayarlara örnek olarak gösterilebilir. Yazılım ve donanım teknolojilerindeki hızlı gelişmelerle, şifreleme algoritmalarının seri uygulamaları yeterince güvenli ve hızlı değildir. Diğer yandan, paralel algoritmalar hızlılığı devam ettirmede çok önemli bir rol oynarlar. Bu nedenle artık programlarda paralelleştirme teknikleri yaygın olarak uygulanmaktadır.

Kriptografik algoritmaların paralelleştirilmesi, donanımsal paralelleştirme ve yazılımsal paralelleştirme şeklinde ikiye ayrılmaktadır. Donanımsal Paralelleştirme, RISC işlemcisiyle paralelleştirme, ASIC ile paralelleştirme ve FPGA ile paralelleştirme olarak üçe ayrılmaktadır. Yazılımsal Paralelleştirme ise çok çekirdekli işlemcilere veya grafiksel işleme birimlerine paralel kod yazılımı, ayrıştırma (decomposition) yönteminin uygulanması ve döngülerin paralelleştirilmesi yöntemleriyle üçe ayrılmaktadır (Jose and Raj, 2012).

Çalışmada kullanılan program açık anahtar (asimetrik) kriptografi ve aynı zamanda simetrik anahtar kriptografi algoritmalarını birleştiren hibrit bir uygulamadan oluşmaktadır. Açık anahtar şifreleme, deşifreleme ve anahtar üretimi evreleri yoğun hesaplamalı operasyonlardan oluşmaktadır çünkü çok büyük sayılarla fazla sayıda modüler çarpma işlemine ihtiyaç duyulur. Bu nedenle; açık anahtar uygulamaları simetrik anahtar uygulamalarından daha yavaş çalışmaktadır. Uygulamanın yavaşlamasına

sebebiyet veren bu durumu ortadan kaldırmak için hibrit algoritma kullanılmasına karar verilmiştir. Büyük boyuttaki verilerin şifrelenmesi ve deşifrelenmesi simetrik kriptografi algoritmasıyla, üretilen simetrik anahtarın şifrelenerek ve sonra tekrar deşifrelenerek gizliliğin sağlanması asimetrik kriptografi algoritmasıyla gerçekleştirilmiştir.

Sadece hibrit uygulama ile sorunlar çözülmemiş program yine yavaş çalışmıştır. Bu nedenle uygulamanın daha da hızlandırılması konusu ortaya çıkmıştır. Bu noktada, yukarıda bahsedilen paralelleştirme teknikleri devreye girmektedir. Son zamanlarda, genel amaçlı hesaplamalar için kullanılan ve GCC yapısı üzerine kurulan OpenMP, algoritmaları paralelleştirmede geniş bir kullanım alanı kazanmaktadır. Birçok yoğun hesaplamalı problem, OpenMP'nin yüksek paralel özelliklerini kullanarak önemli ölçüde performans artışı kazanmaktadır. GCC yapısı, programcılar için bu tarz uygulamaları mevcut hale getiren bir sistemden oluşmaktadır(Saxena et al., Kishore, Handa, Kapoor, 2013). OpenMP yaklaşımı, paralel programların uygulanabilirliğini daha kolay bir hale getirmektedir. Genel olarak, program performansının bilgisayarın işlemci hızı ile doğru orantılı olduğunu baz alırsak, kripto algoritmalarının çok çekirdekli işlemciye sahip bilgisayarlarda paralelleştirilerek uygulanması sonucunda önemli bir performans artışı beklenmektedir. Çok çekirdekli işlemcilerde bu algoritma önce seri olarak uygulanacak daha sonra programın paralel versiyonları çalıştırılacaktır. Ayrıca hyperthreading in performans üzerine olan etkisi de incelenecektir. Bunu yanı sıra Farklı iki platformda seri performans ile paralel performans karşılaştırılarak birtakım sonuçlara varılacaktır.

1.2 Tezin İçeriği

Tezin 2. Bölümünde temel bilgilere yer verilmiştir. Temel bilgiler, işlemciler ve çok çekirdekli işlemciler, işlemcilerde paralellik, paralel çok çekirdekli programlama, kriptografi gibi konuları içermektedir. Tezin 3. Bölümünde ise paralel kriptoloji ayrıntılı olarak anlatılmıştır. Bu bölüm kriptolojinin paralel yapılma nedenleri ve paralellik çeşitleri, konuyla ilgili yapılmış çalışmalar, hibrit algoritmalar ve çalışmada kullanılan güvenlik algoritması, AES algoritması çalışma mekanizması, MIRACL kriptografik kütüphane, çalışmada OpenMP kullanılma nedenleri, OpenMP paralelleştirme şekilleri ve çalışmada kullanılan programın paralelleştirilmesi gibi konuları içermektedir. OpenMP paralelleştirme şekilleri başlığı altında tüm paralellik çeşitleri hakkında bilgiler verilmiştir. Tezin sonuç ve değerlendirme bölümünde yürütülen çalışma sonucunda ortaya çıkan bulgular tablolandırılarak sonuçlar hakkında yorumlar yapılacaktır. Öneriler bölümünde ise çalışma sonucunda elde ettiğimiz yorumlara göre çok çekirdekli işlemciler hakkında birkaç tavsiyeye yer verilmiştir.

2. TEMEL BİLGİLER

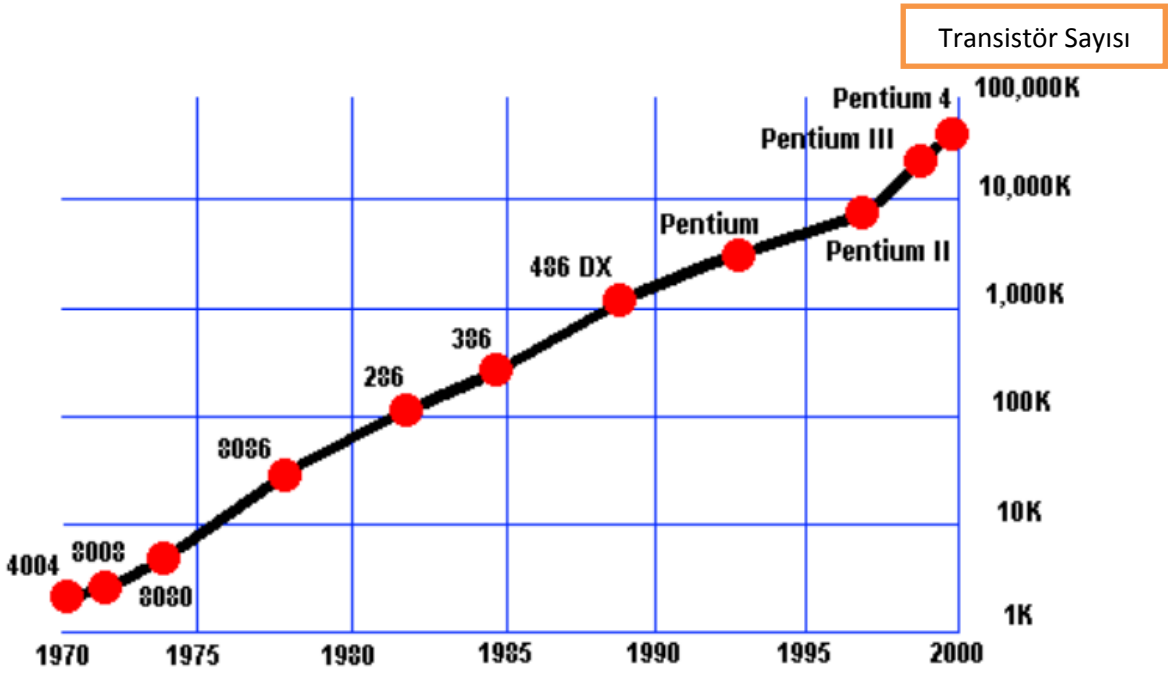
2.1 İşlemciler ve Çok Çekirdekli İşlemciler

Mikroişlemciler uzun zamandan beri kullanılmakta ve yaşadığımız dünyada köklü değişikliklere sebep olmaktadır. CISC mimariye sahip Intel Pentium işlemci ailesinin kökleri, 1978'de ortaya çıkarılan 8086'ya kadar dayanmaktadır. Intel 1970'lerin başlarında ilk mikroişlemciyi 4 bit 4004 olarak üretmiştir. 4004 çok güçlü değildi ve tek yapabildiği toplama ve çıkarma işlemleriydi. Hemen ardından her ikisi de 8 bit olmak üzere 8008 ve 8080'i geliştirmişlerdir. Motorola aynı hareketi yaparak Intel 8080'e eşit olan 6800'i tasarlamıştır. Daha sonra, şirketler 16 bit mikroişlemcileri imal etmişlerdir. Motorola 68000'i çıkarırken Intel 8086 ve 8088'i piyasaya koymuştur. İlk kullanıcı tabanlı kişisel bilgisayarlar, Intel'in yaygın olarak kullanılan Pentium dizisinden ve 32 bit 80386 işlemcisinden oluşmaktadır. Her işlemci nesli daha küçük, daha hızlı, daha fazla güç harcayacak ve daha fazla ısı dağıtacak şekilde geliştirilmiştir (Schauer, 2008).Tüm Pentium mikroişlemcileri geriye uyumluluğu esas alıp 8088'in temel tasarımı üzerine geliştirilmişlerdir. Orijinal 8088 üzerinde çalışan bir parça program 5000 kat daha hızlı bir şekilde Pentium 4 üzerinde çalışabilir (Cleveland Institute of Electronics, 2011). Daha önceki işlemciler için geliştirilmiş yazılımların desteklenmesi donanım ve ISA düzeyinde komut seti karmaşıklığına sebep olmuştur. Dolayısıyla, mimari dünyasında bu tür işlemciler CISC (Complex Instruction Set Computers) işlemciler olarak adlandırılmışlardır (Tanenbaum and Austin, 2013).

1987'de ise ilk SPARC işlemciyle birlikte RISC (Reduced Instruction Set Computers) ekolü bilgisayar dünyasına dahil olmuştur. Bu mimaride geriye uyumluluk esas alınmadığı için donanım CISC işlemcilere göre daha basittir, komut seti de daha yakındır. Performans açısından CISC ve RISC ekolleri karşılaştırıldığında RISC işlemcilerin daha hızlı ve güvenilir oldukları, daha az güç tükettikleri görülmektedir (Stallings, 2010).

İşlemcilerin performansını arttırmaya rehberlik eden ilkelerden birisi de *Moore Yasası* olarak bilinmektedir. Moore Yasası, Intel firmasının kurucularından biri olan Gordon Moore'un Electronics Magazine dergisinde yayınlanan makalesi ile teknoloji tarihine kendi adıyla geçen bir yasadır (Schauer et al., 2008;Vikipedi , 2013). 1965 yılında, Gordon Moore, çip üzerindeki transistör sayısının her yıl yaklaşık olarak iki katına çıkması gerektiğini belirtmiştir (Bu kural, 1975 yılında 2 yıl olarak düzeltilmiştir). Moore Yasası, Dave House tarafından yenilenerek bilgisayar performansının 18 ayda bir iki katına çıkmasını zorunlu kılmıştır. Şekil 2.1'den, erken yıllarda ortaya çıkarılan mikroişlemcilerin çip bazında transistör sayıları görülmektedir **(**Schauer, 2008**)**. Bu yasaya göre, her 18 ayda bir bileşen

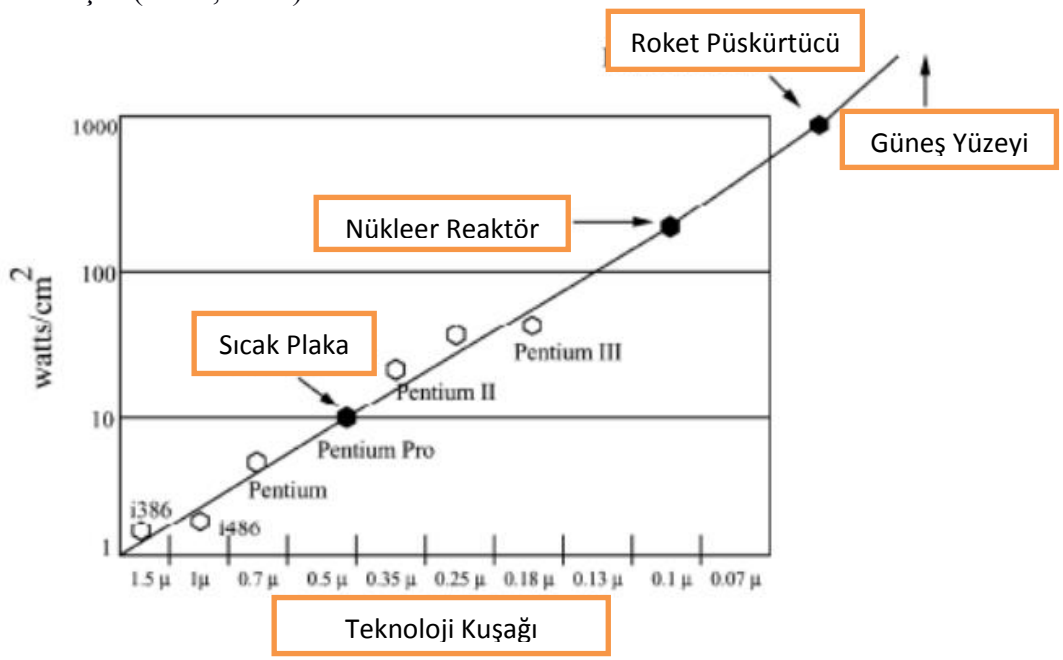
sayısı iki katına çıkarsa, bilgisayarların işlem kapasitelerinde büyük ölçüde artışlar meydana geleceği, üretim maliyetleri sabit kalacağı ve hatta maliyetlerin azalacağı öngörülmektedir. Sözün ilk olarak söylendiği 1965 yılından beri bu yasa çoğunlukla geçerli olmuştur. Yasa temel olarak bir tümleşik devrenin fiziki boyutunun devreyi oluşturan transistör sayısının karesiyle değiştiği anlamına gelmektedir. Örneğin tümleşik devre bünyesindeki transistör sayısı iki katına çıkarsa devrenin boyutu dört katına çıkmaktadır **(**Vikipedi, 2013).



Şekil 2.1. Moore Yasasının Gösterimi (Schauer, 2008).

Moore, bu yasanın sonraki on yıl boyunca geçerliliğini koruyacağını tahmin etmişti ama Intel bu yasayı günümüze kadar çiğnemeden devam ettirmeyi yukarı şekilde de görülebildiği gibi başarmıştır (Korkusuz, 2010). Anlaşıldığı üzere, her iki yılda bir transistör sayısı aşağı yukarı 2 katına çıkmıştır (Bkz. Şekil 2.1). Sonuç olarak, Moore Yasası egemenliğini sürdürmeye devam etmektedir. Örnek olarak, 2008 yılında Intel, Tukwila adı verilen ve dünyanın ilk 2 bilyon transistörlü mikroişlemcisini üretmiştir (Schauer, 2008). Bu hızlı gelişme sonucunda entegrenin maliyeti sabit kalmış, bilgisayarların lojik ve bellek devrelerinin fiyatları etkileyici bir şekilde azalmıştır. Lojik ve bellek elemanları, entegre içerisinde daha yoğun olarak sıkıştırıldığından elektriksel yollar kısalmış ve çalışma hızı da artmıştır. Bilgisayarlar küçülmüş ve farklı çevrelerde yerleştirilmesi kolaylaşmıştır. Enerji ve soğutma gereksinimlerinde azalmalar olmuştur. Bunlardan başka, entegre devreler içindeki bağlantılar lehimleme ile yapılan bağlantılardan daha güvenilirdir ve entegre içine daha fazla transistör yerleştirilebildiğinden entegreler arası bağlantılar azalmıştır (Örnek, 2013).

Mikroişlemciler daima, performans sıkıntısı çeken pazarlar tarafından performans ve maliyet faktörleri ön planda tutularak tasarlanmışlardır. Tümleşik devre işleme teknolojisi ile birlikte gelişmiş çip üretim teknolojisi, performansı arttırmak için çip üzerinde 1 bilyon transistörü entegre etmeyi mümkün kılan entegrasyon yoğunluğunu önermektedir. Bunun yanı sıra, Pollack kuralı tarafından yönlendirilen mikro-mimariler ile elde edilen performans artışı tahminen, karmaşıklık derecesi artışının karekökü ile doğru orantılı bir şekilde gerçekleşmektedir. Bu da işlemci çekirdeği üzerindeki mantık birimlerini iki katına çıkarmanın performansı sadece 40% oranında geliştireceği anlamına gelmektedir. Gelişmiş çip üretim teknikleriyle birlikte gelen bir diğer büyük engel güç harcanımı sorunudur. Çalışmalar göstermiştir ki, çip boyutunun küçülmesiyle birlikte transistör akım kaçağı da artmış ve bu durum sabit güç harcanımını Şekil 2.2'de gösterildiği gibi yüksek değerlere kadar çıkarmıştır (Venu, 2011).



Şekil 2.2. Güç Yoğunluğunun Artması (Venu, 2011).

Mikroişlemci endüstrisi, 1970'lerden beri teknolojik ilerlemelerin gidişatında büyük bir öneme sahip olmaya devam etmektedir. Büyüyen pazar ve daha hızlı performansa duyulan ihtiyaç, endüstriyi daha hızlı ve akıllı çipler üretmeye zorlamıştır (Venu, 2011). 1983-2002 yılları arasında işlemcilerin frekanslarında da artış gündeme gelmiştir. Mesela, Pentium 4, 8 yıllık ömründe hız (frekans) bakımından performansını 1.3 GHz'den 3.8 GHz'e çıkarmıştır. Çip başına düşen transistör sayısının ve frekansının artmasıyla birlikte güç tüketimi tehlikeli bir derecede artmıştır (Schauer, 2008).Böylece, batarya ömrü ve sistem maliyeti sınırlamaları, tasarım ekibini performanstan ziyade güç üzerinde düşünmeye zorlamıştır. Güç ve ısı farkındalığına sahip mimarilerde de ilerlemeler meydana gelmiştir. Güce duyarlı olan

mimarilerin iki çeşidi vardır: düşük-güç ve güç farkındalığına sahip tasarımlar. Düşük güç mimarileri, bir yandan performans kısıtlamalarını tatmin ederken diğer yandan da güç tüketimini en düşük seviyede tutmaktadır. Bu sistemlere örnek olarak düşük güç ve gerçek zamanlı performansın çok önemli sayıldığı gömülü sistemler verilebilir. Güç farkındalığına sahip mimariler bir yandan güç kısıtlamalarını karşılarken diğer yandan da performans parametrelerini en üst seviyede tutmaktadır. Isıya duyarlı olan sistem tasarımları, çip üzerinde sıcak noktaların nereye düştüğüne karar veren ve simülasyonları kullanmaktadır. Ayrıca bu tür mimariler, sıcak noktaların sayısını ve etkisini azaltmak için bu simülasyonlar kullanılarak yeniden gözden geçirilip düzeltilmişlerdir (Schauer, 2008).

Nihayetinde tasarımcılar, güç duvarı olarak adlandırılan ve bir mikroişlemcinin dağıtabileceği kadar ki güç sınırını ifade eden bir parametreye uymak zorunda kalmışlar ve işlemcinin frekansını arttırmayı durdurmuşlardır (Venu, 2011).

Saat frekansının daha yukarıya çıkamamasının diğer bir nedeni ise belli bir noktadan sonra veri yolu üzerindeki tellerin her birinden iletilen sinyallerin senkronizasyonunu bozması ve bu nedenle sinyallerde kayma olmasıdır. Bu sınırın altında bir frekans kullanıldığı zaman tüm tellerin yayılım gecikmesi aynı olmakta ve senkronizasyonu bozulmamaktadır (Tanenbaum and Austin, 2013).

İşlemci performansını arttırmaya yönelik bir sonraki adım, bir sonraki başlıkta tanıtılacak olan bir anda çok sayıda komut başlatmaya ve işletmeye yönelik donanımsal paralellik yöntemlerinin kullanılması olmuştur.

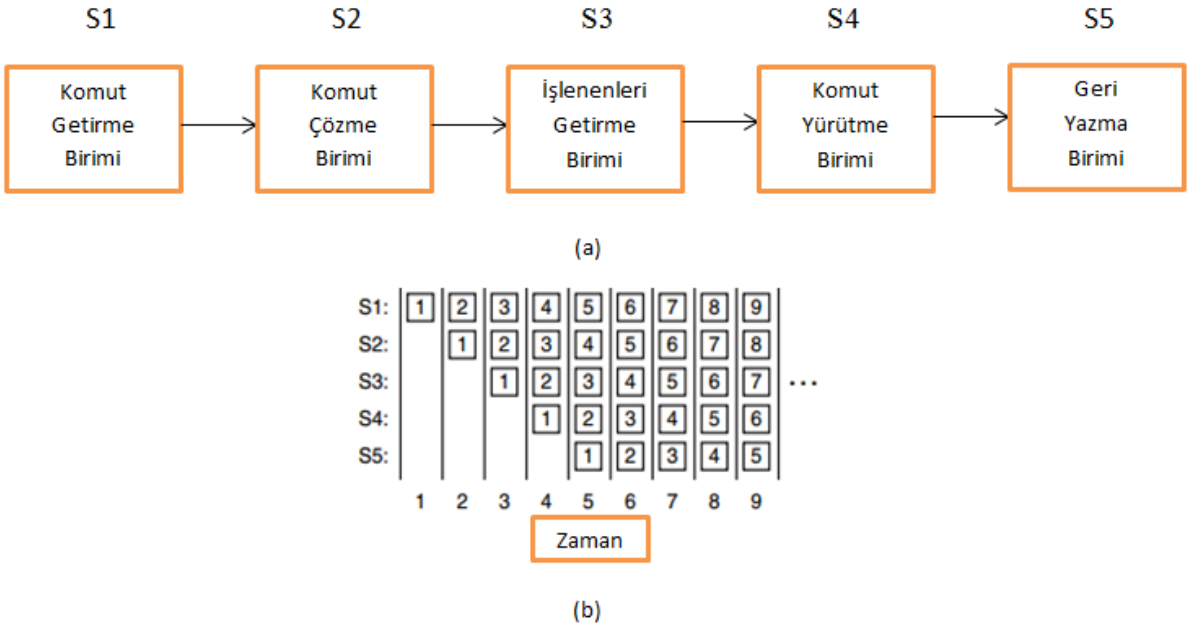
2.2 İşlemcilerde Parallellik

Paralel işletim için geliştirilen ilk donanımsal çözümler *Komut Seviyesinde* ve *İşlemci Seviyesinde Parallellik* alt başlıklarında incelenebilir. Komut düzeyi paralelleştirme, birim zamanda daha fazla komut çalıştırabilmek için birbirinden bağımsız komutların eş zamanlı olarak çalıştırılmasına dayanır. İşlemci düzeyinde paralelleştirmede ise, aynı problem üzerinde birden fazla Merkezi İşlem Birimi bir araya gelerek çalışmaktadırlar. Her bir yaklaşımın kendine özgü yararları vardır (Tanenbaum and Austin, 2013). Alt başlıklarda bu iki yaklaşım detaylı olarak incelenmektedir:

2.2.1 Komut Seviyesinde Paralellik

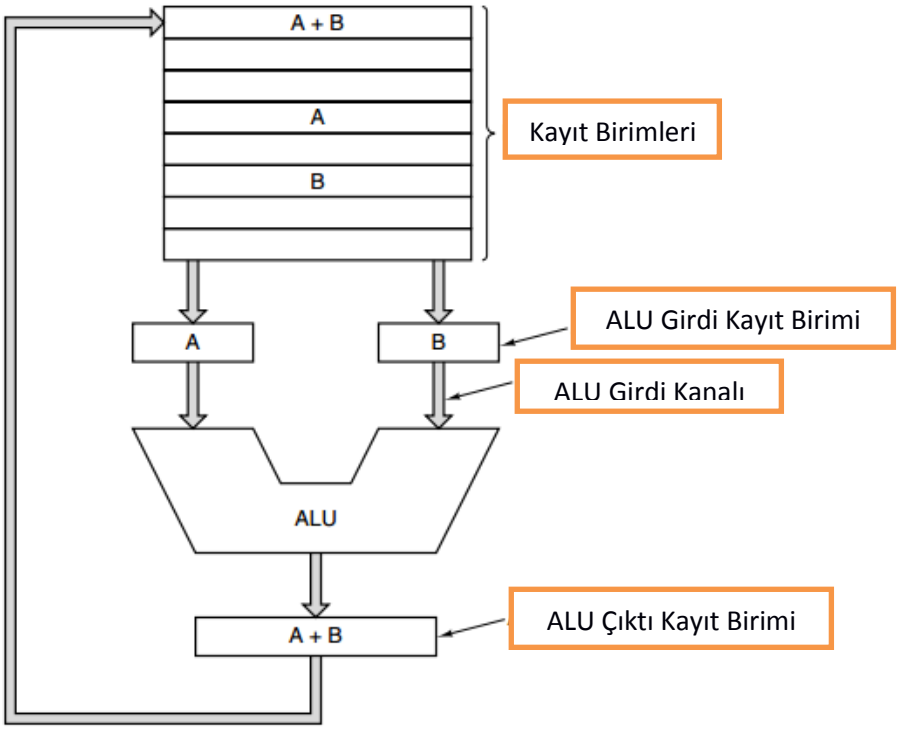
Komut düzeyinde paralellik ilk başta içinde *boru hattı* (pipeline) mimarisi şeklinde uygulanmış, daha sonra *süperskalar* işlemciler yönünde evrimleşmiştir (Tanenbaum and Austin, 2013).

Bu yöntemler bir komutun yaşam döngüsü boyunca farklı evrelerden geçmesi ve bu evrelerin her birinin farklı bir donanım kaynağı kullanması ilkesine dayanırlar. Şekil 2.3 (a)'da aşamalar (stages) adı verilen beş birimlik bir boru hattı gösterilmektedir. Aşama 1, komutları bellekten gidip getirme ve komuta ihtiyaç duyuluncaya kadar komutu tamponda saklama işlemlerini gerçekleştirmektedir. Aşama 2'de, komutların şifresi çözülerek tipine ve hangi işlenenlerin kullanılması gerektiğine karar verilmektedir. Aşama 3, işlenmesi gereken bilgilerin kayıt birimlerinden veya bellekten gidip getirilmesini ifade etmektedir. Aşama 4, işlenenleri Şekil 2.4'de gösterilen veri yolu üzerinde çalıştırarak komutun yürütülmesini sağlamakta ve son olarak Aşama 5 ise, sonucu uygun kayıt birimine geri yazmaktadır (Tanenbaum and Austin, 2013).



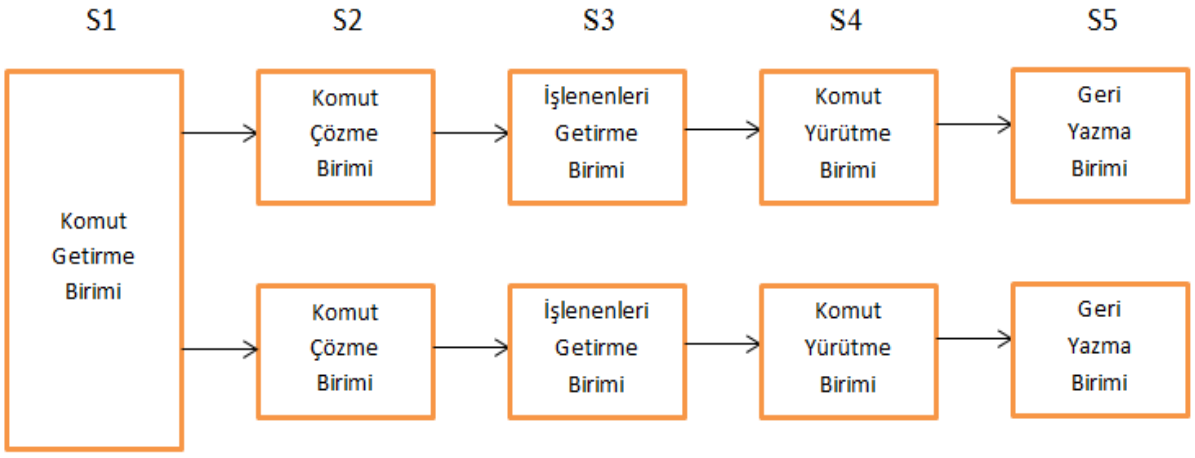
Şekil 2.3. a) Beş Birimlik Boru Hattı, b) Her Aşamanın Durumu (Tanenbaum and Austin, 2013).

Şekil 2.3 (b)'de boru hattının zamanın bir fonksiyonu olarak nasıl çalıştırıldığı gösterilmektedir. Birinci saat çevrimi boyunca, Aşama 1 komut 1 üzerinde çalışmakta ve komutu bellekten getirmektedir. İkinci saat çevrimi boyunca, Aşama 2 komut 1'i deşifre ederken bu arada Aşama 1'de komut 2'yi bellekten getirmektedir. Üçüncü saat çevrimi esnasında, Aşama 3 işlenmesi gereken bilgileri komut 1 için bellekten getirirken, Aşama 2 komut 2'yi deşifre etmekte ve Aşama 1 ise komut 3'ü bellekten getirmektedir. Dördüncü saat çevrimi esnasında, Aşama 4 komut 1'i yürütürken, Aşama 3 komut 2 için işlenmesi gereken bilgileri bellekten getirmekte, Aşama 2 komut 3'ü deşifre etmekte ve Aşama 1 ise komut 4'ü bellekten getirmektedir. Son olarak, Aşama 5 komut 1'i geri yazarken diğer işlemler yine aynı sırada gerçekleşmektedir (Tanenbaum and Austin, 2013).



Şekil 2.4. Tipik Bir Von Neumann Makinesinin Veri Yolu (Tanenbaum and Austin, 2013).

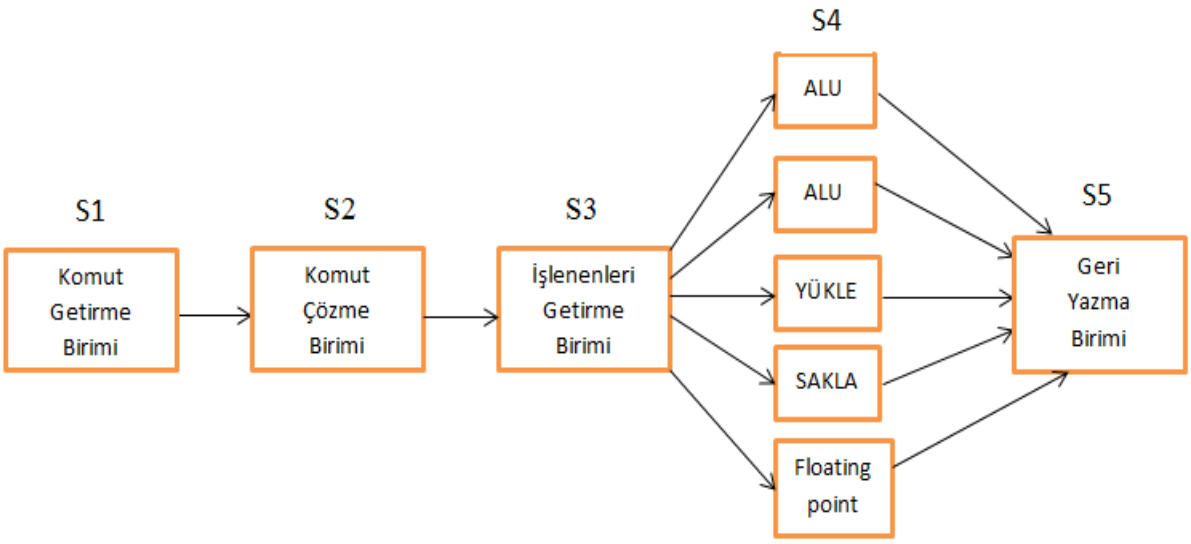
Performansı daha da arttırmak için Şekil 2.5'de gösterilen ikili boru hatları geliştirilmiş ve bunların başarılı olduğu görülmüştür. Dört boru hattı kullanıldığında ise işlemcinin çok yer kapladığı, daha pahalı hale geldiği ve daha çok güce ihtiyaç duyduğu tespit edilmiştir. Bu noktadan sonra Şekil 2.6'da verilen süperskalar mimariye geçiş yaşanmıştır (Tanenbaum and Austin, 2013).



Şekil 2.5. İki Boru Hattına Sahip Bir İşlemci Tasarımı (Tanenbaum and Austin, 2013).

Süperskalar işlemciler de boru hattı mekanizmasına bağlı olarak çalışırlar. Ancak işletim evresi birden çok sayıda ve farklı türde işletim biriminden oluşur. Şekil 2.6'daki örnek incelenecek olursa, S3 evresini tamamlayan komutlar uygun S4 birimine gönderilirler. Bu mimarinin başarılı olmasının sebebi S4 İşletim evresinin diğer evrelere göre daha fazla zaman

tüketmesidir. S3 evresinden hızla gelen komutlar uzun süre harcanan S4 evresinde uygun birime atanırlar. Bu şekilde aynı anda birden fazla komut farklı S4 birimlerinde işletilerek paralelilik sağlanmış olur (Tanenbaum and Austin, 2013).



Şekil 2.6. Beş Fonksiyonel Birimlik Süperskalar İşlemci (Tanenbaum and Austin, 2013).

Komut düzeyinde paralellik ile aynı anda işletilmekte olan komut sayısı yükseltilir, ancak bu mimarinin performanslarının olumsuz etkilendiği durumlar da söz konusudur. Örneğin, her çalışma evresinin farklı sürede tamamlanması evreler arasında komut aşamalarının bekletilmesine yol açar. Başka bir problem de if, for, while, goto gibi atlamalarda yaşanır. Boru hattına alınmaması gereken komutlar boru hattına alınmış, bu durum sonradan fark edilmiş ve bu komutlar bellekte/kayıtçılarda geri alınamaz değişikliklere yol açmış olabilirler. Bu problem, araştırmacıların uzun süre ilgisini çekmiş ve çeşitli atlama tahminleme tekniklerinin geliştirilmesine yol açmıştır (Schauer, 2008).

2.2.2 İşlemci Seviyesinde Paralellik

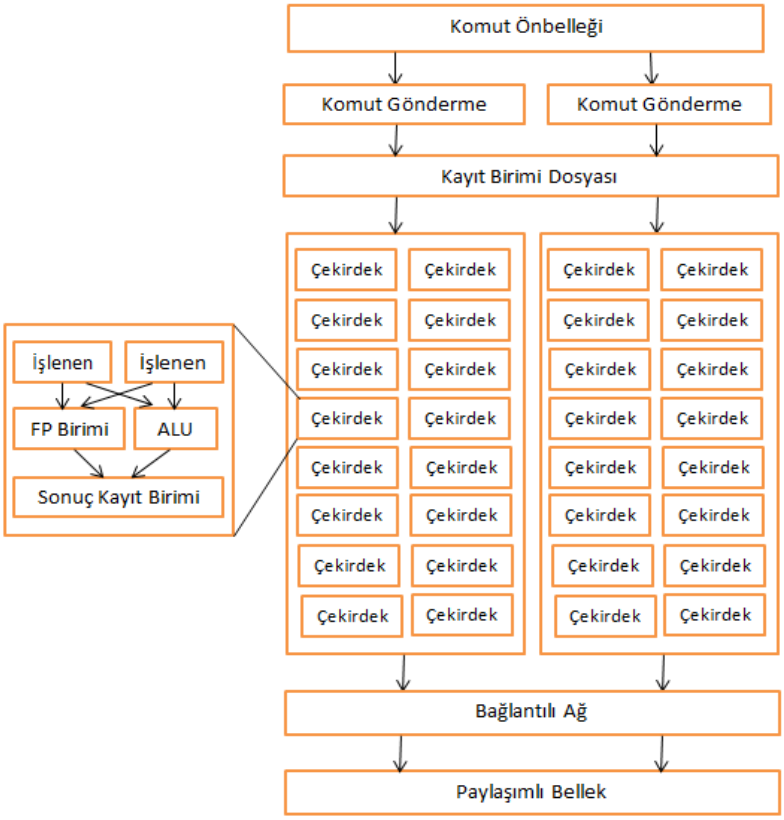
Komut seviyesinde paralellik performans artışı konusunda bir miktar yardımcı olmuştur. Ancak, boru hattı ve süperskalar işlemleriyle performansta 5 ya da 10 kattan daha fazla bir artış pek yaşanmamaktadır. 50, 100 ya da daha fazla kat bir artış için birden fazla Merkezi İşlem Birimine sahip bilgisayarlar tasarlanmalıdır. Bu, ancak İşlemci Seviyesi paralellikle sağlanabilmiştir. İşlemci seviyesi paralellik kendi içinde SIMD (Single Instruction Multiple Data) işlemciler, çoklu işlemciler ve çoklu bilgisayarlar olarak üçe ayrılmaktadır (Tanenbaum and Austin, 2013).

Fiziksel bilimler, mühendislik, bilgisayar grafikleri gibi hesaplama alanlarında var olan önemli problemler döngü ve dizileri içermekte ya da diğer yüksek düzenli yapılarla ilişkili olmaktadır. Genelde, birçok farklı veri kümeleri üzerinde tekrar tekrar aynı hesaplamalar yapılmaktadır. Bu programların düzenliliği ve yapısı, özellikle farklı veri kümelerini paralel

çalışma konusunda kolay bir hedef haline getirmiştir. SIMD işlemciler ve vektör işlemciler gibi başlıca iki yöntem, bu yüksek düzene sahip programların hızlı ve etkin şekilde çalışması için kullanılmaktadır. Bu iki kavram birbirlerine çoğu yönden oldukça benzemekle birlikte, SIMD işlemciler genelde paralel bilgisayar olarak ve vektör işlemciler ise tekli işlemcinin bir uzantısı olarak kabul edilmektedirler. Veri paralel bilgisayarlar, birçok uygulamanın başarılı olmasını olağanüstü verimliliklerinin bir sonucu olarak görmüştür. Bu tür başarılı uygulamalar, alternatif yaklaşımlara göre daha az transistörle önemli hesaplama gücü üretebilmektedir (Tanenbaum and Austin, 2013).

SIMD işlemciler tekli komut ve çoklu veri anlamına gelmektedir**.** Birden fazla veri üzerinde aynı operasyonu gerçekleştiren bilgisayarları tanımlamaktadırlar. Yani SIMD işlemciler, farklı veri kümeleri üzerinde aynı komutları kullanan büyük sayıda özdeş işlemcilerden ve bir kontrol biriminden oluşmaktadır. Bu makineler veri düzeyinde paralelliği sağlamakla beraber eş zamanlılığı desteklememektedir. Çünkü aynı anda çalışan hesaplamalar olmasına rağmen verilen bir zamanda sadece bir süreç görev yapmaktadır. Çoğu modern Merkezi İşlem Birimi tasarımı, multimedya kullanımını geliştirmek için SIMD komutlarına yer vermektedir (Vikipedi, 2014). Modern grafik işleme birimleri yoğun olarak SIMD işlemeye dayanmaktadır. Nvidia Fermi grafik işleme birimi en fazla 16 tane SIMD akışlı çoklu işlemciye sahip olmaktadır. Her bir akışlı çoklu işlemci 32 tane SIMD işlemciden oluşmaktadır (Şekil 2.7) (Tanenbaum and Austin, 2013).

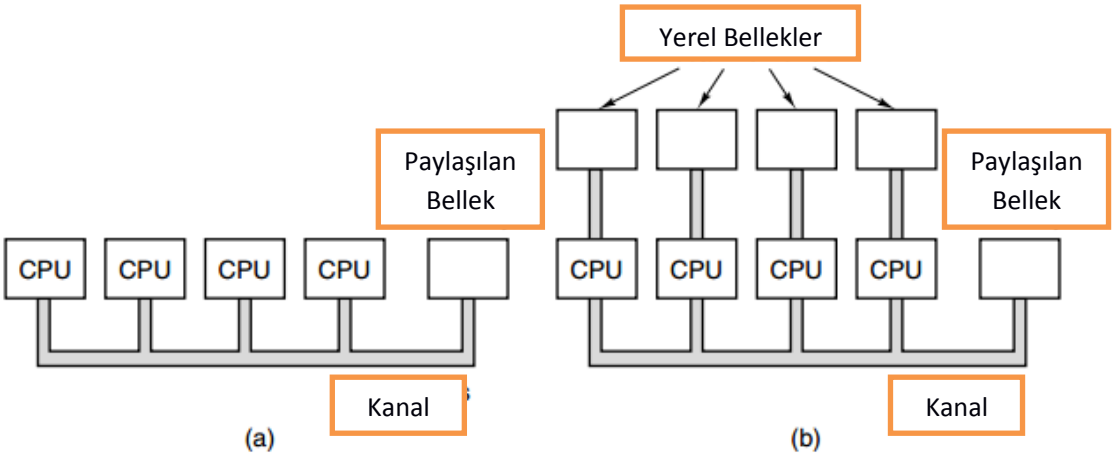
Vektör işlemciler ise, veri dizileri üzerinde işlemler yürütmede etkili olmuştur. Vektör işlemcilerde SIMD işlemciden farklı olarak tüm işlemler tek ve yoğun bir boru hattına sahip fonksiyonel birimde gerçekleştirilmektedir. SIMD işlemci ve vektör işlemcinin her ikisi de diziler üzerinde çalışmaktadır. SIMD işlemci bu işi, birden fazla fonksiyonel üniteye sahip olarak yapmaktadır. Diğer yandan vektör işlemci, bellek tarafından tek bir komutla yüklenebilen klasik kayıt birimleri setinden oluşan vektör kayıt birimi kavramını barındırmaktadır. Örneğin, vektör toplama komutu vektör kayıt birimlerinden vektör değerlerini alıp onları toplayıcı boru hattına yollayarak toplama işlemini sonuçlandırmaktadır. Intel mimarisi üzerindeki SIMD uzantılı komutlar, multimedya ve bilimsel hesaplamaları hızlandırmak için bu çalışma modelini de kullanmaktadır (Tanenbaum and Austin, 2013).



Şekil 2.7. Fermi Grafik İşleme Biriminin SIMD Çekirdeği (Tanenbaum and Austin, 2013).

Veri paralel bilgisayarlardan sonra tasarlanan ilk paralel sistem, birden fazla Merkezi İşlem Birimine sahip *çoklu işlemciler* olarak bilinmektedir. Bu sistemde birden fazla Merkezi İşlem Birimi ortak bir belleği paylaşmaktadır. Her bir Merkezi İşlem Birimi, belleğin herhangi bir parçasını okuyup yazdığı için, yazılım içerisinde birinin diğerinin işine karışmasını önlemek amacıyla koordineli bir şekilde çalışmaları gerekmektedir. İki ya da daha fazla Merkezi İşlem Birimi yakından etkileşim yeteneğine sahipse bu işlemciler sıkı bağlı olarak adlandırılmaktadır. Bu mimarisel yaklaşımda, değişik uygulama şemaları da mümkün olmaktadır. En basit olanı ise birden fazla Merkezi İşlem Birimi sahip tek bir kanal ve hepsinin bağlı olduğu tek bir bellek tasarımıdır. Bu tür kanal tabanlı çoklu işlemci Şekil 2.8 (a)'da gösterilmektedir (Tanenbaum and Austin, 2013).

Çok sayıda işlemcinin belleğe erişmek için aynı kanalı kullanması çekişmelere yol açmaktadır. Çoklu işlemci tasarımcıları bu çekişmeyi azaltmak ve performansı arttırmak için çeşitli sistemler meydana getirilmiştir. Diğer bir tasarım Şekil 2.8 (b)'de gösterilmektedir. Bu sistemde, her bir Merkezi İşlem Biriminin yerel bir belleği vardır ve bu belleğe diğer işlemciler tarafından erişim yapılamamaktadır. Bu bellek, program kodu ve paylaşılmayan veriler için kullanılmaktadır. Özel belleğe erişim ise ana kanalı kullanarak yapılmamakta ve büyük ölçüde kanal trafiğini azaltmaktadır (Tanenbaum and Austin, 2013).



Şekil 2.8. a) Tek Kanallı Çoklu İşlemci, b) Yerel Belleğe Sahip Çoklu Bilgisayar. (Tanenbaum and Austin, 2013)

Bir çoklu işlemcide kullanılabilecek işlemci sayısı sınırlıdır. Pek çok işlemciyi belleğe bağlamak işi zorlaştırmaktadır. Bu problemlerin üstesinden gelebilmek için birçok tasarımcı paylaşımlı belleğe sahip olma fikrini terk etmiş ve sadece geniş sayıda birbirine bağlı bilgisayarları içeren sistemler kurmuşlardır. Her birinin kendine özel belleği var olmakla birlikte ortak bir bellek bulunmamaktadır. Bu sistemler *çoklu bilgisayarlar* olarak isimlendirilmiştir. Birbirine sıkı olarak bağlanmış çoklu işlemci sistemlerinin aksine çoklu bilgisayarlara sahip sistemlerde Merkezi İşlem Birimleri birbirine gevşek olarak bağlanmıştır (Tanenbaum and Austin, 2013).

Burada Merkezi İşlem Birimleri, birbirlerine elektronik posta tarzı bir mesaj atarak haberleşmekte fakat bu haberleşme elektronik postaya göre daha hızlı yapılmaktadır. Çok büyük sistemler için her bir bilgisayarın başka bir bilgisayara çizgisel olarak bağlı olması pratik bir yöntem değildir bu nedenle 2D ve 3D ızgara, ağaç ve yüzük gibi topolojiler kullanılmaktadır. Bunun bir sonucu olarak, mesajlar bir bilgisayardan diğerine geçerken genellikle bir ya da daha fazla ara bilgisayarı ve anahtarı kullanmak zorundadır. Bununla beraber, mesajın geçiş süresi birkaç mikrosaniye bazında olabilmektedir. Çoklu işlemcilerin programlanması ve çoklu bilgisayarların inşa edilmesi kolay olduğundan her birinin iyi özelliklerini birleştiren karma bir sistemi tasarlamak adına birçok araştırma yapılmaktadır (Tanenbaum and Austin, 2013).

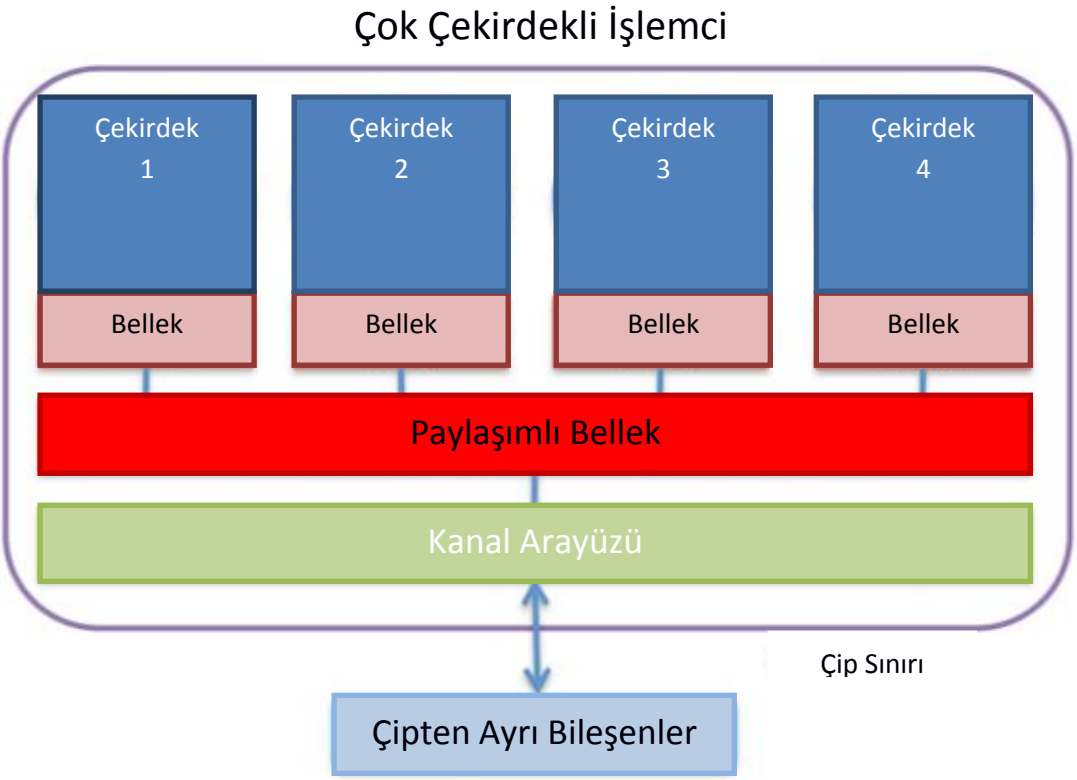
Bu noktaya kadar dile getirdiğimiz işlemcilerin tümü de tek çekirdekli işlemciler sınıfına girmektedir. Performansı kayda değer bir şekilde geliştiren başka bir teknik ise çok çekirdekli işlemcilerdir. Çok çekirdekli işlemciler, on yıldır varlıklarını sürdürmektedirler. Çok çekirdekli işlemciler genellikle daha yavaş frekanslarda çalışmaktadır (Schauer et al., 2008;Venu, 2011). Ancak, tek çekirdekli işlemcilerin bugünlerde karşılaştığı yüksek verimlilik ve yüksek enerji verimliliği ile birlikte uzun ömürlü pil gibi teknoloji

sınırlamalarından dolayı, çok çekirdekli işlemciler daha fazla önem kazanmıştır (Venu, 2011). Bir sonraki başlıkta çok çekirdekli işlemciler hakkında ayrıntılı bilgi verilmektedir.

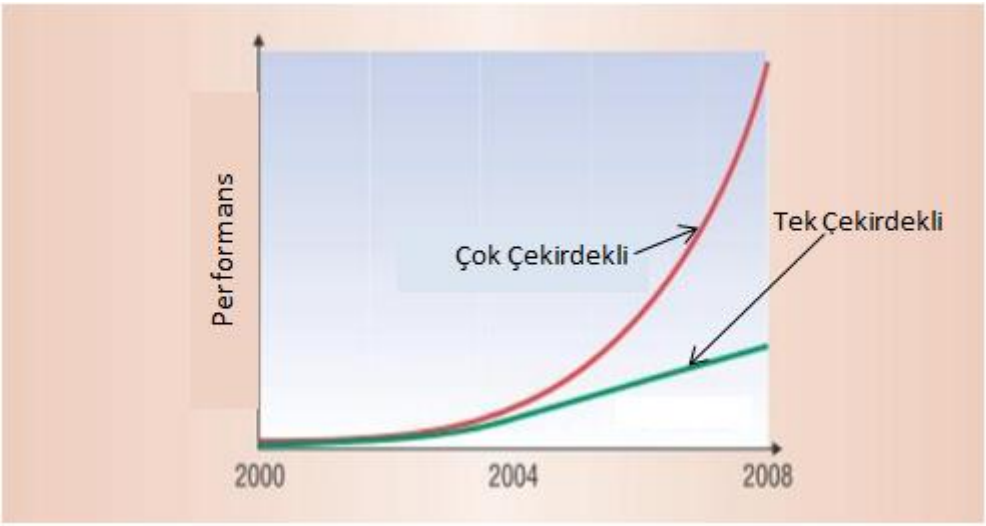
2.3 Çok Çekirdekli İşlemciler

Çok çekirdekli işlemci genel olarak iki ya da daha fazla birbirinden bağımsız işlemcinin bütünleşmiş bir devrede yer alması şeklinde tanımlanabilir. Bu terim, aynı bütünleşmiş devreye bağlı olmayan birden fazla Merkezi İşlem Birimine sahip çoklu Merkezi İşlem Birimi teriminden farklı fakat aynı zamanda bu terimle de ilişkilidir. Tekli işlemci terimi ise, sistem başına bir işlemcinin düşmesini kastetmektedir. Bu işlemci sadece bir çekirdeğe sahiptir bu nedenle çok çekirdekli ya da çoklu Merkezi İşleme Birimi gibi çoklu işleme mimarileriyle zıt düşmektedir. Şekil 2.9'da çok çekirdekli bir işlemcinin temel bileşenleri gösterilmektedir (Prinslow, 2011).

Çok çekirdekli işlemcileri daha net olarak açıklamak gerekirse, bir çip üzerinde birkaç çekirdek barındıran tek bir işlemcileri ifade etmektedir. Çekirdekler hesaplama birimleri ve ön belleklerden oluşan fonksiyonel birimlerdir. Çip üzerinde bulunan birden fazla çekirdek, işlemcinin performansını arttırmak için bir araya getirilmişlerdir (Venu, 2011). Bir işlemcide birden fazla çekirdek entegre edilerek Merkezi İşlem Biriminin iş parçacığı ya da süreç düzeyinde paralelliği desteklemesi sağlanmıştır. Genel olarak, bir Merkezi İşlem Birimi çekirdeği sadece bir süre içerisinde bir iş parçacığı ya da süreç çalıştırabilmektedir. *Hyperthreading* teknolojisi sayesinde, bir Merkezi İşlem Birimi çekirdeğinin eş zamanlı olarak iki iş parçacığı uygulamayı çalıştırabilmesi mümkün hale gelmiştir (Zhang et al.,Zhang, Bi, 2012). Çok çekirdekli işlemci üzerinde bulunan çekirdeklerin her birinin, tek çekirdekli işlemcinin gösterdiği en yüksek performansta muhakkak çalışacak diye bir kural yoktur, fakat paralel çalışarak ve birden fazla görevin üstesinden gelerek sistemin tüm performansını iyileştirmektedirler. Performans artışını gözlemleyebilmek için, tek çekirdekli ve çok çekirdekli işlemcilerin çalışma mantığının iyice anlaşılması gerekmektedir. Birden fazla program çalıştıran tek çekirdekli işlemciler, bir program üzerinde çalışmak için o programa zaman dilimi ataması (quantum) yapmaktadırlar. Daha sonra kalan programlara farklı zaman dilimleri atayarak çalışmayı sürdürmektedirler. Eğer bir süreç uzun bir sürede tamamlanıyorsa, diğer kalan tüm süreçler arka planda gecikmeye maruz kalabilmektedirler. Ancak, çok çekirdekli işlemcilerin var olması halinde, aynı anda paralel çalışabilen birden fazla görev varsa her bir görev ayrı bir çekirdekte paralel olarak çalıştırılarak sistem performansı Şekil 2.10'daki gibi arttırılabilir. Intel, AMD, IBM ve TENSILICA gibi ünlü işlemci üreticileri çok çekirdekli işlemcileri geliştirmeye başlamışlardır (Venu, 2011).



Şekil 2.9. Genel Çok Çekirdekli İşlemcilerin Basit Bir Blok Diyagramı (Prinslow, 2010).



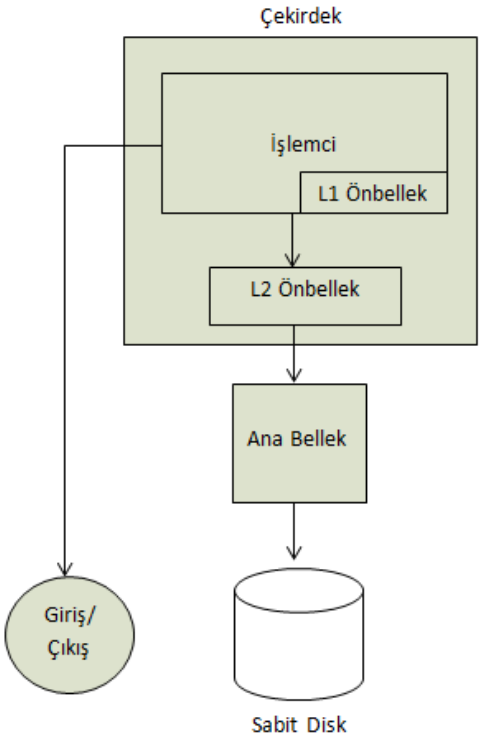
Şekil 2.10. Çok Çekirdekli Çiplerin Performans Gelişimi (Venu, 2011).

2.3.1 Çok Çekirdekli İşlemcilerin Genel Mimarisi

Şekil 2.11'de gösterilen yapı, herhangi bir çok çekirdekli tasarıma özel değildir fakat çok çekirdekli bir mimarinin temel taslağı hakkında genel bir bilgi vermektedir. Üretici tasarımları birbirinden farklı olmasına rağmen, çok çekirdekli mimarilerin belirli durumlara bağlı kalmaya gereksinimi vardır. İşlemciye en yakın şekilde bulunan birinci seviye ön bellek (L1), işlemci tarafından sıklıkla veri depolamak için kullanılan çok hızlı bir bellek türüdür. İkinci seviye ön bellek (L2) ise çipten ayrı ve birinci seviye ön bellekten daha yavaş fakat ana

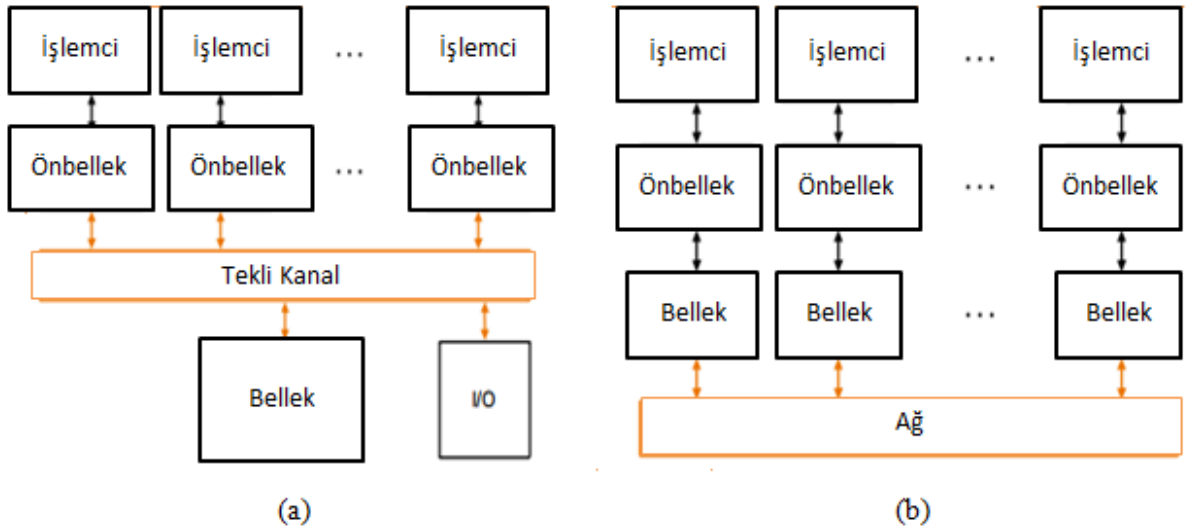
bellekten daha hızlı çalışmaktadır. İkinci seviye ön bellek, birinci seviye ön bellekten daha geniş ve aynı amaçla kullanılmaktadır. Ana bellek, tüm ön belleklerden daha büyüktür ve daha yavaş bir şekilde çalışmaktadır. Ana bellek, örnek olarak Microsoft Word programında sürekli güncellenen bir dosyayı saklamak için kullanılmaktadır. Çoğu sistem, ana hafızaya 1 GB ve 4 GB değerleri arasında yer verirken, yaklaşık olarak birinci seviye ön belleğe 32 KB ve ikinci seviye ön belleğe ise 2 MB genişliğinde yer ayırmaktadır. Sonuç olarak, ön bellek ve ana bellekte veri bulunmadığı zaman sistem veriyi sabit diskten tekrar kazanmak zorundadır. Bu işlem ana belleği okumaktan daha üssel olarak daha fazla zaman almaktadır (Schauer, 2008).

Çok çekirdekli işlemciler, uygulama gereksinimlerine bağlı olarak farklı şekillerde tasarlanmaktadırlar. Heterojen çekirdek grupları, homojen çekirdek grupları veya bunların birleşiminden oluşan sistemler oluşturulabilmektedir. Homojen çekirdek mimarisinde, Merkezi İşlem Biriminde bulunan tüm çekirdekler aynıdır ve "böl ve fethet" yaklaşımını kullanarak işlemcinin performansını arttırmaya çalışmaktadırlar. Bunu yaparken yüksek hesaplama yoğunluklu uygulamayı daha az hesaplama yoğunluklu uygulamalara bölüp bu uygulamaları paralel çalıştırmaktadır. Diğer yandan heterojen çekirdekler, özel uygulamaları çalıştırmak için tasarlanmış farklı çekirdeklerden oluşmaktadır. Bu nedenle heterojen çekirdekler, bilgisayar üzerinde farklı uygulamaların çalıştırmasında sorunlara neden olmaktadır (Venu, 2011).



Şekil 2.11. Genel Modern İşlemci Yapılandırması (Venu, 2011).

Eğer çekirdekler yan yana koyulursa, çekirdekler arasında ve çekirdek ile ana bellek arasında bir haberleşme yöntemi gereklidir. Bu haberleşme ise genellikle tek haberleşme kanalı veya bağlantılı ağ kullanılarak elde edilmektedir. Bağlantılı ağ yaklaşımı dağıtık bellek modelini kullanıyorken kanal yaklaşımı paylaşımlı bellek modelini kullanır. Yaklaşık olarak 32 çekirdek kullanan bir kanal, performansa olumsuz yönde etki eden işleme, haberleşme ve rekabet işlemleriyle aşırı dolmaktadır. Şekil 2.12'de paylaşımlı ve dağıtık bellek modelleri gösterilmektedir (Schauer, 2008).



Şekil 2.12. a) Paylaşımlı Bellek Modeli, b) Dağıtık Bellek Modeli (Schauer, 2008).

Çok çekirdekli işlemciler, tek çekirdekli işlemcilerin eksikliklerine bant genişliğini arttırarak ve tüketilen gücü azaltarak cevap vermektedir. Çizelge 2.1'de Georgia Tech Üniversitesinde bulunan Packaging Araştırma Merkezi tarafından kullanılan tek ve çok çekirdekli işlemcilerin karşılaştırılması yapılmıştır. Aynı kaynak voltajı ve düşük frekansta çalışan birden fazla çekirdekle yapılan çalışmada, toplam harcanan güçte dörtte bir oranında azalma varken bant genişliğinde neredeyse on kat bir artış söz konusudur (Schauer, 2008).

Çizelge 2.1. Tek Çekirdek ve Çok Çekirdek Karşılaştırması (Schauer, 2008).

Özellikler	Tek Çekirdekli İşlemci (45nm)	Çok Çekirdekli İşlemci (45nm)
Vdd	1.0V	1.0V
I/O pins(toplam)	1280 (ITRS)	3000 (Tahmini)
İşletim frekansı	7.8GHz	4GHz
Çip-paket veri oranı	7.8Gb/s	4Gb/s
Bant genişliği	125GByte/s	1 TeraByte/s
Güç	429.78W	107.39W
Çipteki toplam pin sayısı	3840	9000 (Tahmini)
Paketteki toplam pin sayısı	2840	4500 (Tahmini)

2.3.2 Çok Çekirdekli İşlemcilerin Yüzleştiği Problemler

Çok çekirdekli işlemcilerin birçok avantajı olmasına rağmen, bu teknolojinin önemli sorunları da bulunmaktadır. Paralel çalışacak şekilde oluşturulmamış yazılım programlarına özgü bir ana problem, çok çekirdekli işlemcilerde tek çekirdekli işlemcilere kıyasla daha yavaş çalışmalarıdır. Bir başka deyişle programlar seri olarak yazılmışsa çok çekirdekli işlemcilerde çalışma süresi uzamaktadır. Ayrıca çok çekirdekli sistemlerde çalışan uygulamaların kendiliğinden çekirdek sayısı arttıkça daha hızlı olmayacağına da işaret edilmiştir. Bunun nedeni ise, işlemcinin kontrol ünitesinin daha karmaşık olmasıdır. Programcılar, çok çekirdekli ortamda artan işlemci sayısından faydalanabilen ve süreyi uzatmayan uygulamalar yazmalıdırlar. Günümüzdeki uygulamaların çoğunluğu, tek işlemci üzerinde çalışacak ve çok çekirdekli işlemcinin yeteneklerinden yararlanamayacak şekilde yazılmıştır. Yazılım firmaları çok çekirdekli işlemcide çalışan programlar geliştirebilmelerine rağmen, yıllar önce geliştirilen yazılım programlarının şu andaki çok çekirdekli platformlara uyum sağlaması endüstrinin karşılaştığı ciddi bir mesele olarak görülmüştür. Programları yeniden tasarlamak mümkün gözükmesine rağmen, bugünün ortamında tercih edilen teknolojik bir karar olmamaktadır. Şirketler daha çok müşteri memnuniyeti, maliyet indirgemesi gibi anahtar parametreleri ön planda tutacak şekilde yazılım programlarını tasarlamaktadırlar (Venu, 2011).

Endüstri bu problemi, tek çekirdekli işlemcide çalışan yazılım programlarını çok çekirdekli işlemciye adapte edecek derleyiciler tasarlayarak çözmeye çalışmaktadırlar. Derleyiciler, kod üreten ve komutların paralel olarak çalışmasını sağlayan komutları yeniden düzenleme işlemini yapan “code reordering” yöntemini gerçekleştirmektedirler. Ayrıca derleyiciler, kendiliğinden paralel iş parçacıkları ya da süreçler üretebilecek şekilde geliştirilmektedirler. Intel, çok çekirdekli işlemcilerde programcıların paralellikten faydalanabilmeleri adına C++ ve Fortran araçları için büyük güncellemeler uygulamışlardır. C, C++ ve Fortan dillerinde çoklu işlemeyi destekleyen ve çoklu iş parçacıklı kodların verimli olmasını sağlayacak direktifler üreten OpenMP adında uygulama programlama ara yüzü ortaya çıkarılmıştır (Venu, 2011).

Diğer bir yandan, günümüzdeki işlemciler çok çekirdekli sistemlerdir ve tek bir işlemciyle bütünleştirilen çekirdek sayısı gelecekte daha fazla artacaktır. Çok çekirdekli işlemcilerin karşılaştığı sorunlardan biri de işlemci çekirdeğine yeterli bellek erişimi sağlamaktır. Gizli depolama birimleri (iç cache) bellek genişliği ihtiyacını azaltmaya yardımcı olabilir fakat çekirdek sayısı arttıkça, bellek erişimi gecikmesinin artışından kaçınmak için işlemci tasarımcıları daha yüksek bellek genişliği sağlamanın bir yolunu bulmalıdırlar. Yeni işlemci

tasarımları, bellek ara yüzünün performansını geliştirmek ve veriyi tüm çekirdeklere tedarik edebilmek için işlemci ile bellek denetleyicisini entegre etmişlerdir. Önceki bellek denetleyicisi tasarımlarıyla karşılaştırıldığında, bu çözüm artan bellek genişliği ve azalan gecikme ile bellek erişimi önermektedir. (Majo and Gross, 2011)**.**

Çip üzerindeki bağlantılar, çok çekirdekli çiplerin performansı için kritik bir engel oluşturmaktadır. Artan çekirdek sayısı, özellikle veri bellekten çok çekirdekli çipe yollanacağı zaman büyük ölçüde bağlantı gecikmelerine neden olmaktadır. İşlemcinin performansı, veri açlığı senaryolarını önlemek için ne kadar hızlı çalıştığından ziyade Merkezi İşlem Biriminin veriyi ne kadar hızlı bir şekilde getirdiğine bağlı olmaktadır. Bu sorunu çözmek için başvurulan klasik yöntemlerden birkaçı ise ara belleğe alma (cache) ve bellek ile işlemci arasında daha akıllı bir bütünleştirme sağlamadır (Venu, 2011).

Çip üzerinde bulunan bellek denetleyicisinin birtakım avantajları da vardır. Çok çekirdekli çoklu işlemcide (birkaç tane çok çekirdekli işlemcilerden meydana gelen işlemci), yerel bellek denetleyicisi tek bir merkezi bellek denetleyicisi yokmuş gibi sistem ölçeklenmesine izin vermektedir. Bunun yerine, fiziksel adres alanı işlemciler arasında bölünür ve her bir işlemcinin çekirdekleri doğrudan sadece fiziksel belleğin bir parçasına yerel bellek ara yüzü ile erişebilmektedirler. Paylaşımlı belleğe sahip çok işlemcinin yaygın modelini desteklemek için, her bir işlemci (ve çekirdekleri) sadece kendisine doğrudan bağlı olan yerel belleğe değil, aynı zamanda diğer işlemcilerin yerel belleklerine de ulaşabilme yeteneğine sahip olmalıdır. Bu uzaktan bellek erişimi, işlemcileri birbirine bağlayan çapraz çip bağlantısı ile gerçekleşmektedir. Büyük işlemci üreticileri, kendilerine özgü çapraz işlemci bağlantı teknolojisi ile ortaya çıkmışlardır (örn., Intel QuickPath Interconnect (QPI) veya AMD Hypertransport). Çapraz çip bağlantısının verimliliği, çip üzerindeki bellek denetleyicisinin verimliliğinden daha düşüktür. Çapraz çip bağlantısı üzerinden geçen uzaktan bellek erişimi, ayrıca yerel bellek erişimine göre daha fazla gecikmeye maruz kalmaktadır. Bunun nedeni, bellek ara yüzünün heterojen bir yapıya sahip olmasıdır. Bu tür çok işlemciler non-uniform bellek mimarisi (NUMA) olarak sınıflandırılmaktadır. Uzaktan bellek erişiminin performans faktörü önemlidir ve şimdiki uygulamalarda NUMA faktörü maksimum 2 olabilir (2 kat yavaşlamaya eşittir) (Majo and Gross, 2011).

NUMA sistemleri için şu ana kadar düşünülen en önemli performans optimizasyonu, sistemin veri yerelliğini arttırmaktır. (Belleği, kendisine erişmek isteyen programlara yakın bir yere tahsis etmek böylece uzaktan bağlantıyı azaltmak ve buna bağlı olarak NUMA faktöründen kaçınmak). Çeşitli araştırmacılar, NUMA sistemlerde çalıştırılan uygulama performanslarını geliştirmek için birtakım yöntemler araştırmışlardır. Tüm yaklaşımlar, belleğin

paylaştırılmasını ve hesaplamaların sistemde eşleştirilmesini amaçlayan profil tabanlı ve on-line optimizasyonları kullanarak veri yerelliğini arttırmaya odaklanmaktadırlar. Çoğu uygulaman performansının bellek sistemi performansı tarafından sınırlandırılması, NUMA çok çekirdekli çok işlemcilerin bellek sistemini basit ve gerçekçi bir şekilde anlamayı önemli bir konu haline getirmiştir. Bu sistemler üzerinde iyi sonuçlar verecek veri ve hesaplama eşleştirilmelerini bulmak son derece kritiktir (Majo and Gross, 2011).

Bellek mücadelesi, şimdiki çok çekirdekli mimarilerde önemli bir sorun olarak görülmektedir. Çok çekirdekli sistemler üzerinde yapılan ölçümler sonucunda, önceden kaynaklarda belirtildiği gibi çipten ayrılmış bellek trafiği her zaman yoğun olmadığı anlaşılmıştır. Sıkışıklık, problem boyutuna bağlıdır. Küçük problem boyutları ile iş yapıldığında belleğe çok sayıda küçük veri istekleri gitmekte ve küçük off-chip mücadeleleri ortaya çıkmaktadır. İşler küçük olduğu için arka arkaya gelen işler hemen bitivermekte bu nedenle bekleme ve tıkanıklık olmamaktadır. Bekleme ve tıkanıklığın olmaması ya da çok az olması, bellek denetleyicisinin isteği hemen karşılamasından kaynaklanmaktadır. Bunun tersine, daha büyük program boyutları bellek mücadeleleri doğurmakta ve bellek denetleyicisinin istekleri döndürmesi için geçen zaman artmaktadır. Bu sırada yeni gelen bellek ihtiyaçları da bellek önünde birikmeye ve bellek tıkanıklığına yol açmaktadır. Off-chip bellek için mücadele, aktif çekirdek sayısı ile üssel olarak artmakta fakat ek bellek denetleyicileri eklemek bellek mücadelesini azaltmaktadır(Tudor et al.,Teo, See, 2011).

Çok çekirdekli sistemlerde, işlemci çekirdek adı verilen ve paralel çalışmaya uygun olan birimlerden oluşmaktadır. Bununla birlikte gizli depolama birimi ve bellek genişliği gibi önemli kaynakların paylaşılması ve çekirdek sayısının artması, çekirdekler arasında yarışa ve kaynak mücadelesine sebep olmaktadır. Bu durum ise tasarım karmaşıklığına yol açmaktadır (Tudor et al.,Teo, See et al., 2011; Venu, 2011). Birden fazla iş parçacığının eş zamanlı olarak paylaşılan veriye ulaşmak istemesi veri yarış durumu olarak bilinen zamanlamaya bağlı bir hatadır. Çok çekirdekli bir ortamda veri yapısı bir çekirdek tarafından güncellenirken diğer tüm çekirdeklerin erişimine açıktırlar. Mesela ikincil çekirdek, birincil çekirdeğin veriyi güncellemesini beklemeden aynı veriye ulaşmaya çalışması yanlıştır. Özellikle yarış durumlarını düzeltmek zordur ve kodun kontrol edilerek bu durumların fark edilmesi imkânsızdır. Bunun nedeni ise veri yarışlarının aniden meydana gelen durumlar olmasıdır. Yarış durumlarını önlemek için karşılıklı dışlama (mutually exclusion) yöntemlerini barındıran özel donanım gereksinimlerine yer verilmelidir (Venu, 2011). Buna ek olarak, teknolojinin ilerlemesiyle çekirdek sayısı fazlalaşmakta fakat bellek genişliği daha yavaş bir oranda artmaktadır. Bunun nedeni, kablo gecikmelerinin ve güç dağıtımının diğerlerinin

arasında paylaştırılmasından kaynaklanmaktadır. Bu nedenle, çekirdeğe düşen off-chip bellek genişliği, çekirdek sayısının artmasına ayak uyduramamıştır. Diğer bir akım, dolara bağlı olan bellek kapasitesinin Moore Kuralına göre büyümeye devam etmesidir. Bu teknoloji akımları devam ettiği sürece, çok çekirdekli sistemlerdeki önemli performans sorunları off-chip bellek etrafında dönmeye devam edecektir (Tudor et al.,Teo, See, 2011).

Bellek mücadelesinin yol açtığı bir başka sorun ise bellek ve işlemci arasında artan performans uyumsuzluğudur. Ana belleğe olan erişimin gecikmesi, çok çekirdekli sisteme geçmeden önce üzerinde çalışılan önemli bir performans sorunuydu. Fakat günümüzdeki çok çekirdekli sistemlerde, çip dışındaki ana belleğin paylaşılması bellek genişliği için yeni performans sorunları ortaya çıkarmıştır. Bir programın performansı, birden fazla çekirdek çip dışındaki ana bellek genişliği için rekabete girdiğinde düşmektedir. Çipten ayrı belleğe talep eden çekirdek sayısının artması, daha uzun bellek erişim zamanıyla sonuçlanmakta ve bu da programı çalıştırmak için gereken toplam döngü sayısını arttırmaktadır. Daha düşük bellek maliyeti ile daha büyük bellek kapasitesi, çalıştırılan problemlerin boyutlarının artmasına izin vermektedir. Bu durum ise çipten ayrı bellek genişliği için talebi arttırıp mücadelelere yol açmaktadır (Tudor et al.,Teo, See, 2011).

Çok çekirdekli işlemcilerin performansını etkileyen bir diğer önemli faktör ise çip bileşenleri arasındaki etkileşimdir. Yani başka bir deyişle kanal çekişmesi ve gecikmenin anahtar problem olduğu bir sistemde çekirdek, bellek denetleyicisi ve paylaşılan bileşenler, ön bellek ve bellek arasındaki etkileşimden bahsedilmektedir. Özelleşmiş enine bağlantılar (crossbar) ya da mesh yöntemleri donanım üzerinde uygulanarak bu sorunlar giderilmektedir (Venu, 2011).

Çok çekirdekli işlemcilerin son günlerde yüzleştiği sorunlar tam olarak çözüldüğünde bu tür sistemlerde tam performans verimliliği gerçekleştirilebilir. Bu teknoloji alanında, yeni çok çekirdekli programlama dili ve çok çekirdek farkındalığına sahip yazılımlar içeren birçok teknolojik buluşlar beklenmektedir. Bu konu, adapte olunması açısından en zorlu teknolojilerden biri olmasına rağmen çok çekirdekli işlemcilerden daha verimli bir şekilde yararlanabilmek için önemli derecede araştırmalar yapılmaktadır (Venu, 2011).

Uyumluluk protokolleri ve bağlantılı ağlar bazı problemleri çözmüş olsa bile programcılar paralel uygulamalar yazmayı öğreninceye kadar çok çekirdekli işlemcilerin tam verimliliğine ve faydalarına erişemeyeceklerdir (Schauer, 2008).

2.4 Paralel Çok Çekirdekli Programlama

Paralel programlama, aynı görevin parçalara bölünmüş ve uyarlanmış bir hale getirilerek, sonuçları daha hızlı elde etmek için çoklu işlemcilerde eş zamanlı olarak işletilmesidir. Çok

çekirdekli bilgisayarlar piyasaya çıktığından bu yana, daha fazla bilgisayar paralel bir platforma sahip olmaya başlamıştır. Fakat çoğu program hala seri olarak çalışmaktadır. Paralel bilgisayarların avantajlarından yararlanabilmek için paralel bir şekilde çalışabilen bir program tasarlanmalıdır. Bir hesaplama probleminde eğer problem daha küçük problemlere ayrılabiliyorsa bu problemde eş zamanlılık var demektir. Yani başka bir deyişle; küçük problemler güvenli bir şekilde aynı anda çalışabilir (Tan and Li, 2012).

Problem, program içerisine yerleştirildikten sonra eş zamanlılık olasılıklarını görmek zor olabilir. Bundan dolayı; eş zamanlılığı tespit edebilmek için programcılar paralel çözümlerinin tasarımlarına, problemi problem alanının içerisinde analiz ederek başlamalıdırlar. Eş zamanlılığı bulmak için birtakım modeller programcılara yardımcı olabilir. Bu modeller, bir problemde eş zamanlılığı tanımlamayı ve analiz etmeyi sağlamaktadır. Tanımlama ve analiz etme işlemleri yapıldıktan sonra, algoritma uzayından bir ya da daha fazla model, uygun algoritma yapısının tasarlanması için seçilebilir. Burada amaç, belirlenen eş zamanlılıkların uygulanabilmesidir (Tan and Li, 2012). Programdaki paralelleştirme olanaklarının belirlenmesi için gerekli olan adımlar aşağıda sıralanmıştır:

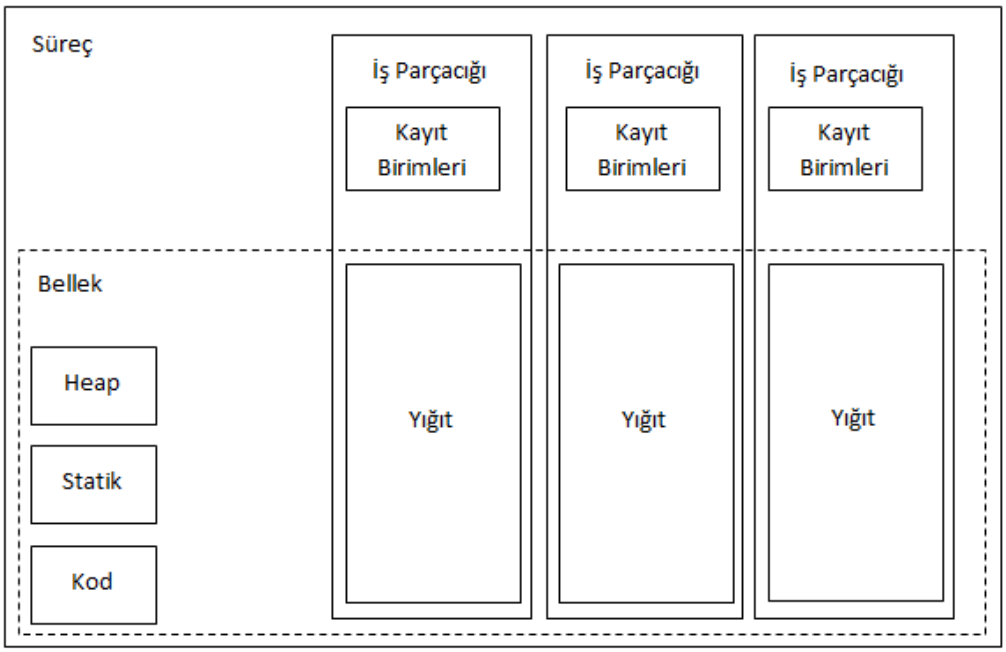
1) Uygulamanın çalışma süresine bakılıp en fazla zaman harcanan bölgeler tespit edilir.
2) Bu bölgelerde program bağımlılıklar için analiz edilir ve bağımlılıklar ayrıştırılıp birden fazla paralel göreve ya da döngüye dönüştürülebilir.
3) Ek yük ve paralelleştirme stratejisinden kazanılan performans tahmin edilir. Eğer yaklaşım, iş parçacığı sayısıyla doğru orantılı bir şekilde çalışıyorsa, bu iyi bir yaklaşım olarak kabul edilir (Gove, 2011).

Paralelleştirme olanakları belirlendikten sonra uygun programlama modelinin belirlenmesi gerekmektedir. 2 temel paralel yazılım geliştirme tekniği bulunmaktadır:

1. Kod seri yazılır. Paralelleştirilebilecek alanlar belirlenir ve çalışma sırasında bu alanlara gelindiğinde otomatik iş parçacığını yaratmak üzere derleyiciye bazı direktifler gönderilir. Günümüzde, bu yöntem için kullanılan direktiflerin sağlandığı en uygun kütüphane OpenMP kütüphanesidir. OpenMP, çalışma zamanı kütüphanesi ile derleyicinin birleşiminden oluşan bir mekanizma sağlar. OpenMP Uygulama Program Arayüzü, C/C++ ve Fortran'da çoklu platform paylaşılan bellek tabanlı paralel programlamayı Unix ve Windows NT ortamları dahil olacak şekilde tüm mimarilerde desteklemektedir. OpenMP, paylaşımlı hafızayı nitelendirmek için kullanılan derleyici direktifleri, kütüphane rutinleri ve çevre değişkenlerinden oluşmaktadır. OpenMP direktifleri seri programlama dilini Tekli Program Çoklu Veri yapıları, iş paylaşımı yapıları ve senkronizasyon yapıları ile genişleterek paylaşımlı ve

kullanılan derleyici direktifleri, kütüphane rutinleri ve çevre değişkenlerinden oluşmaktadır. OpenMP direktifleri seri programlama dilini Tekli Program Çoklu Veri yapıları, iş paylaşımı yapıları ve senkronizasyon yapıları ile genişleterek paylaşımlı ve özel veriyle çalışmayı mümkün hale getirmektedir. OpenMP ile yazılmış bir program, ana iş parçacığı adı verilen tek bir görev ile çalışmaya başlar. Paralel bir bölge ile karşılaşıldığında, ana iş parçacığı bir iş parçacıkları takımı yaratır. Paralel bölgedeki komutlar takım içerisindeki her bir iş parçacığı tarafından paralel bir şekilde çalıştırılır. Paralel bölgenin bitiminde ise, takımdaki tüm iş parçacıkları senkronize olurlar. Daha sonra tekrar ana iş parçacığı bir sonraki paralel bölge ile karşılaşana kadar tek başına çalışmaya devam eder. OpenMP'de iş parçacıkları kernel seviyesinde yaratılmaktadır (Navalgund et al**.,** Desai, Angalki, Yamanur, 2013).

2. Yazılım geliştirici bellekte kullanıcı seviyesinde çalışan bir iş parçacığı kütüphanesi kullanarak yazılımını çok iş parçacıklı olarak yazar, yazılım çalışırken her bir iş parçacığı bir çekirdek üzerine atanır. Oluşturulan her bir iş parçacığının Şekil 2.13'de gösterildiği gibi kendine ait bir program kontrolü, yığıtı ve kayıt birimi vardır. Tüm iş parçacıkları dosya sistemi ve bellek alanı gibi sistem kaynaklarını paylaşmaktadır. Paylaşılan bellek alanı iş parçacıkları arasında iletişim yolu sağlamaktadır (IBM, 2006).

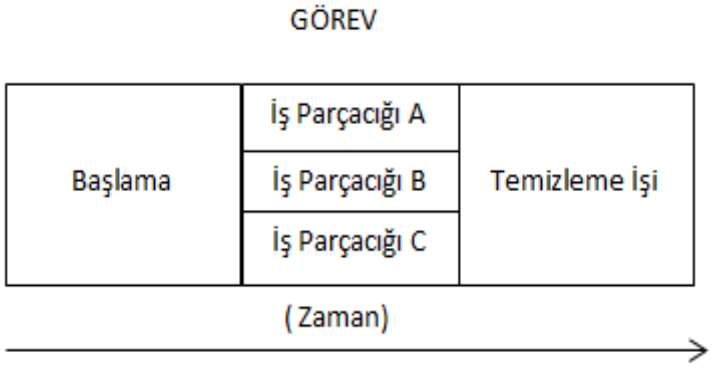


Şekil 2.13. İş Parçacıklarının Yapısı (IBM, 2006).

İş parçacıkları kullanılarak yapılan paralel programlamanın dört modeli bulunmaktadır. İlk model *patron/işçi* modelidir. Bu modelde bir iş parçacığı patron olarak işlevini yürütmektedir. Çünkü bu iş parçacığı diğer işçi olarak nitelendirilen iş parçacıklarına görevler atamaktadır.

Her işçi kendilerine atanan görevleri bitirene kadar birbirinden bağımsız olarak görevlerini sürdürürler. Görevlerini bitirdiklerinde ise patron iş parçacığına diğer göreve atanmak için hazır olduklarını bildirirler. Alternatif olarak patron, işçilerin diğer göreve hazır olup olmadıklarını görmek için işçileri periyodik bir biçimde kontrol etmektedir. Patron/işçi modelinin bir diğer varyasyonu ise iş sırası modelidir. Patron iş parçacığı, işleri bir kuyruğa koymakta ve işçiler kuyruğu denetleyip kuyruktan görevleri üstlenmektedirler. Birden fazla patron olduğunda bu durum genellikle üretici/tüketici olarak adlandırılmaktadır. Bu modele örnek olarak ofis ortamındaki sekreterliğe ait yazı görevi verilebilir. Ofis müdürü, yazılacak dokümanları sepete koyar ve sekreterler sepetten dokümanları alarak yazmaya başlarlar. Burada dokümanlar kuyruğu, sekreterler ise iş parçacıklarını simgelemektedir (Fedkiw, 2014).

İkinci model, *çalışma ekibi* modelidir. Bu modelde birden çok iş parçacığı tek bir görev üzerinde çalışmaktadır. Görev paralel yürütülecek şekilde parçalara ayrılmakta ve her bir iş parçacığı tek bir parçayla ilgilenmektedir. Bu modelin çalışma mantığı Şekil 2.14'den görülebilir. Bu modele örnek olarak bir binayı temizleyecek olan insanlar verilebilir. Binanın her bir odasını başka bir insan temizleyecektir. Burada temizleme işlemi görevi, insanlar ise iş parçacıklarını temsil etmektedir (Fedkiw, 2014).



Şekil 2.14. Çalışma Ekibi Modeli (Fedkiw, 2014).

Üçüncü model, *boru hattı* modelidir. Boru hattı modelinde, görev adımlara ayrılmaktadır. Adımlar istenilen sonucu elde etmek için belli bir sırada çalıştırılmaktadır. Her adımda yürütülen iş, bir önceki adıma bağlıdır ve bir sonraki iş için ön şart teşkil etmektedir. Ancak, amaç birden fazla sonuç üretmek ve bu nedenle adımlar paralel çalışacak şekilde tasarlanmıştır. Başka bir deyişle, bir adım bir sonucun üzerinde çalışırken bir önceki adım bir sonraki sonuç üzerinde çalışabilir. Bu modele örnek olarak otomobil montaj hattı verilebilir. Montaj hattında her bir adım sürekli olarak bir önceki adımdan ürün almakta ve kendi işini yapıp bir sonraki adıma kendi sonucunu iletmektedir (Fedkiw, 2014).

Son model, OpenMP'nin de kullandığı model olan *fork-join* modelidir. Sıralı olarak yürütülmekte olan bir programın içinde, paralel parçalar halinde işlenip tekrar sıralı hale dönülmesi genelde fork-join yapısı olarak adlandırılır (Akgün vd., Şahin, Yücedağ, Bayıroğlu, 2012). Diğer bir deyişle, ana iş parçacığı gerektiğinde iş parçacıkları takımı yaratabilir ve bunların işlerinin sonlanmasını bekleyebilir.

İlk anlatılan paralelleştirme yöntemi genellikle veri-yoğun işler için kullanılmakta, iş parçacıkları ise işlem-yoğun problemler için tercih edilmektedir. İlk yöntemde senkronizasyon, kilit mekanizması kendiliğinden, iş parçacıkları kütüphanesinde ise bizzat kullanıcı tarafından sağlanmaktadır (IBM, 2006).

2.4.1 Bağımlılıklar

Uygulamanın paralel bir şekilde yürüyebileceğini ancak uygulamada var olan bağımlılıklar belirlemektedir. Döngü ve veri tabanlı bağımlılıklar veya hafıza tabanlı bağımlılıklar şeklinde iki tür bağımlılık söz konusudur (Gove, 2011).

Döngü bağımlılığında, bir sonraki Şekil 2.15'de gösterilen hesaplamada, döngünün içerideki hesaplamaları yapabilmesi için döngü şartını sağlaması gerekmektedir. Döngüdeki her bir iterasyon, döngünün şartına bağlıdır. Döngü, ya iterasyon sayısı 1000'e eşit olduğu durumda ya da kullanıcının parametrelerde verdiği x ve y değerlerinin karelerinin toplamı 4'e eşit ve büyük olduğu durumlarda çalışmasını bitirecektir. Bu nedenle kullanıcının hangi değerleri vereceği bilinmediği için, bu döngünün ne zaman biteceğini tahmin etmek mümkün değildir. Ayrıca burada her bir döngü iterasyonuna yetersiz derecede düşen bir iş vardır. Bu nedenle döngüyü birden fazla iş parçacığına bölmeye gerek yoktur. Buradan da bu döngünün seri bir şekilde yürütülmesi gerektiği sonucuna varılmaktadır (Gove, 2011).

```
int inSet(double ix, double iy)
{
   int iterations=0;
   double x = ix, y = iy, x2 = x*x, y2 = y*y;
   while ( (x2+y2 < 4) && (iterations < 1000) )
   {
     y  = 2 * x * y + iy;
     x  = x2 - y2 + ix;
     x2 = x * x;
     y2 = y * y;
     iterations++;
   }
   return iterations;
}
```

Şekil 2.15. Noktanın Mandelbrot Setinde Olup Olmadığını Kontrol Eden Kod (Gove, 2011).

Hafıza bağımlılıkları çözümü daha zor olan bağımlılıklardır. Bu bağımlılıklar, aynı adrese yapılan bir hafıza erişiminin diğer bir hafıza erişimine göre düzenlenmesi gerektiğini belirlemektedir. Üretici-tüketici problemi hafıza bağımlılığına verilen en iyi örnektir. Bu örnekte elemanları tutan kapasitesi sınırlı bir dizi vardır. Aynı zamanda bu dizi hem üretici hem de tüketici tarafından paylaşılan bir değişkendir. Sınırlı diziler işletim sisteminde, süreçlerin paylaşılan belleği kullanabilmelerini sağlamaktadır. Şekil 2.16'dan üretici ve tüketiciye ait kod parçacıkları görülebilir. Üretici ve tüketici rutinleri ayrı ayrı çalıştırıldığında doğru sonuçlar vermesine rağmen aynı anda çalıştırıldıklarında gerçek sonucu vermeyebilir. Yine örnek olarak, sayaç değişkeni değerinin 5 olduğunu varsayalım. Üretici ve tüketici süreçleri aynı anda "++count" ve "- - count" komutlarını çalıştırdığında sayaç değişkeninin değeri 4,5 ya da 6 olabilir. Süreçler ayrı olarak çalıştırıldığında doğru sonucun 5 olduğu anlaşılmaktadır. Fakat eş zamanlı çalıştırıldığında sayaç değerinin belki de 5 olmadığı aşağıda anlatılmaktadır (Silberschatz et al**,** Galvin, Gagne, 2005).

```
//Üretici için gereken kod
while (count == BUFFER SIZE)
;// hiçbir şey yapma
// geçici diziye bir eleman ekle
buffer[in] = item;
in = (in + 1) % BUFFER SIZE;
++count;

//Tüketici için gereken kod
while (count == 0)
;// hiçbir şey yapma
// geçici diziden bir eleman çıkar
item = buffer[out];
out = (out + 1) % BUFFER SIZE;
--count;
```

Şekil 2.16. Hafıza Bağımlılığına Örnek Bir Kod (Silberschatz et al., Galvin, Gagne, 2005).

Komut "++count" makine dilinde Şekil 2.17'deki gibi ifade edilmektedir. Burada "$register_1$" değişkeni Merkezi İşlem Birimine ait yerel bir kayıt birimini temsil etmektedir. Aynı şekilde komut "- -count" makine dilinde Şekil 2.18'deki gibi yazılmaktadır. "$register_2$" değişkeni yine Merkezi İşlem Birimine ait yerel kayıt birimini temsil etmektedir (Silberschatz et al.**,**Galvin, Gagne, 2005).

$$register_1 = \text{count}$$
$$register_1 = register_1 + 1$$
$$\text{count} = register_1$$

Şekil 2.17. Arttırma Komutu (Silberschatz et al., Galvin, Gagne, 2005).

$$register_2 = \texttt{count}$$
$$register_2 = register_2 - 1$$
$$\texttt{count} = register_2$$

Şekil 2.18. Azaltma Komutu (Silberschatz et al., Galvin, Gagne, 2005).

"register$_1$" ve "register$_2$" değişkenleri belki de aynı fiziksel kayıt birimi olsalar bile bu fiziksel kayıt biriminin içeriği kesinti denetimcisi (interrupt handler) tarafından kaydedilmekte ve yeniden yüklenebilmektedir. Eş zamanlı çalıştırılan "++count" ve "- -count" komutlarının işleyişi seri uygulamanın çalışmasına benzemektedir. Yani önceden çalıştırılması gereken alt düzey komutlar, program eş zamanlı yürütülürken bazen rasgele sırayla çalıştırılmaktadır. Bunun tersine, yüksek seviye komutların sırası program işleyişi esnasında korunmaktadır. Aşağıdaki adımlar rasgele çalıştırmaya örnek olarak verilebilir. Şekil 2.19'dan görüldüğü gibi T_5 anında sayaç 4 değerini göstermektedir. T_4 ve T_5 adımları yer değiştirirse, sayaç 6 değerine atanmış olacaktır. Bu yanlış sonuca, süreçlere aynı anda "count" değişkenini değiştirme izni vererek varmaktayız. Bu tür durumlar, birden fazla sürecin aynı anda aynı veriyi değiştirmek istemesi sonucu ortaya çıkmaktadır. Yürütmenin sonucu, o an hangi süreç aktif olduysa o süreç tarafından belirlenmektedir. Bu olay *mücadele durumu* olarak isimlendirilmektedir. Verilen örnekte mücadele durumuna karşı önlem almak adına, bir zamanda sadece bir sürecin "count" değişkenini değiştirmesi sağlanmalıdır. Böyle bir garantileme için süreçlerin bir şekilde senkronize olması gerekmektedir. Çok çekirdekli sistemlerin gelişmesiyle beraber veri paylaşımını gerçekleştiren çoklu iş parçacıklı uygulamalar da artmaktadır. Bu nedenle süreç senkronizasyonu ve koordinasyonu büyük önem kazanmaktadır (Silberschatz et al.,Galvin, Gagne, 2005).

T_0:	*producer*	execute	$register_1 = \texttt{count}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \texttt{count}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\texttt{count} = register_1$	$\{count = 6\}$
T_5:	*consumer*	execute	$\texttt{count} = register_2$	$\{count = 4\}$

Şekil 2.19. Sayaç ve Kayıt Birimlerinin Son Durumları (Silberschatz et al., Galvin, Gagne, 2005).

A iş parçacığının bir görev olduğunu düşünelim. A görevi, B'nin sonucundan üretilecek veriye ihtiyaç duymaktadır. Yani A görevi, B görevine bağımlıdır. B, görevini bitirinceye kadar ve A'nın ihtiyaç duyduğu veriyi üretip A'ya bırakıncaya kadar, A görevine başlayamaz. Bu durum, *true dependency* olarak ifade edilmektedir. Genel olarak, B verinin üzerine yazmakta ve A ise B'nin güncellediği veriyi okumaktadır. Çizelge 2.2'de görevlerin sahip olabileceği dört tip bağımlılık gösterilmiştir (Gove, 2011).

→"Read after read" (RAR) operasyonunda herhangi bir bağımlılık yoktur. Her iki iş parçacığı okuma işlemini yaptığında, bu iki iş parçacığı arasında herhangi bir bağımlılık bulunmaz. Hatta iş parçacıklarının çalışma sıraları ne olursa olsun, programın sonucu değişmez.

→"Read after write" (RAW) operasyonunda *true dependency* bulunmaktadır.

→"Write after read" (WAR) operasyonunda *antidependency* vardır. Burada veri bir görev tarafından güncellenmeden önce başka bir görev tarafından okunmaktadır.

Çizelge 2.2. Bağımlılık Türleri (Gove, 2011).

		İkinci Görev	
		Oku	**Yaz**
İlk Görev	**Oku**	Read after read (RAR) No dependency	Write after read (WAR) Antidependency
	Yaz	Read after write (RAW) True dependency	Write after write (WAW) Output dependency

→"Write after write" (WAW) operasyonunda *output dependency* vardır. Burada 2 görevden biri en son sonucu güncellemektedir. Bu tür bağımlılıkta hangi görevin çalışacağının sırası kritik bir konudur. *Antidependency* ve *output dependency* kodun seri çalışmasını gerektirir (Gove, 2011).

```
void anti-dependency()
{
   result1 = calculation( data1 );   // Önce bu komut işlenir
   data1   = result2 + 1;            // Daha sonra data1 güncellenir
}
```

Şekil 2.20. Antidependency Örneği (Gove, 2011).

Şekil 2.20'deki örnekten anlaşılabildiği gibi "data1" değişkeni üzerinde bir *antidependency* vardır. Çünkü ilk satır komutunda data1 değişkeni kullanılarak (okunarak) bir sonuca varılmaktadır. Daha sonra data1'in değeri değişmektedir. Bu bağımlılığın kurtarılabilmesi için Şekil 2.21'de gösterildiği gibi ilk satırda geçici bir değişken yaratılarak data1 değişkenine eşitlenir. Daha sonra ikinci satırda bu geçici değişkenin değeri ile sonuç hesaplanır. Üçüncü satırda ise data1 değişkeninin üzerine yazılır (Gove, 2011).

```
void anti-dependency()
{
   data1_prime = data1;        // data1'in yerel kopyası
   result1 = calculation( data1_prime );
   data1   = result2 + 1;    // Antidependency ortadan kalkar
}
```

Şekil 2.21. Geçici Değişkenin Yaratılması (Gove, 2011).

Şekil 2.22'de ise data1 değerinin öncelikle result1 tarafından güncellenip daha sonra result2 tarafından değiştirilmesi söz konusudur (Gove, 2011).

```
void output-dependency()
{
   data1 = result1 + 2;
   data1 = result2 + 2; // Aynı değişkenin üzerine yazar
}
```

Şekil 2.22. Output Dependency Örneği (Gove, 2011).

2.5 Kriptografi

Bilgisayar güvenliği terimi, yıllar önce bilgisayarda depolanan gizli bilginin gerçek anlamda gizli tutulması işlemini tanımlamak için kullanılmaktaydı. Ancak günümüz itibariyle tüm dünyada kitlesel olarak yayılan bilgisayarlar ve artan internet erişimi ile bilgisayar güvenliği şu an kullanıcıların güvenliğini koruma gibi diğer ciddi sorunlarla ilgilenmektedir. Bu durum, insanları bilgi güvenliği konusuna odaklanmaya sürüklemiştir bu nedenle her bilgisayar sisteminde veya her şirkette bilgi güvenliği ilkelerine sadık kalınmalıdır. Bilgi güvenliği bilgiyi yetkisiz erişimden korumakla sorumludur (Khalifa, 2011).

Geçtiğimiz 20 yıl içerisinde, bilgi teknolojileri alanındaki hızlı gelişmelerle internette yaratıcı uygulamalar ve teknolojiler ortaya çıkmıştır. Son günlerde, dünyanın herhangi bir yerindeki bir kişiden birkaç saniye içerisinde internet aracılığıyla multimedya mesajı alabilmekte ya da biz bu kişilere mesaj yollayabilmekteyiz (Fadhil and Younis, 2014).

Kriptografinin elektronik veri işlemlerine uygulanması önemli ölçüde ilgiyi de beraberinde getirmiştir. Her gün milyonlarca kullanıcı internet yoluyla banka servislerini ya da e-ticaret işlemlerini kullanarak ve bu gibi çeşitli alanlarda büyük miktarda bilgi üretmekte ve bilgiyi karşılıklı olarak değiştirmektedir. Bu tür örnekler ya da diğer uygulama örnekleri güvenlik bakımından sadece bilginin iletiminde değil, ayrıca içeriğinin korunmasında özel bir yere sahiptir (Navalgund et al.,Desai, Angalki, Yamanur, 2013).

Kriptografi yukarıda anlatılan durumlar için, bütünlüğü, kimlik denetimini ve gizliliği sağlamaya çalışan matematiksel metotlardan oluşmaktadır. Bu metotlar, bilginin iletimi sırasında hem bilginin doğruluğunu ve bütünlüğünü hem de bilgiyi gönderen ve alan kişinin güvenliğini sağlamaya çalışmaktadırlar (Wikipedia et al., 2014;Anderson, 2008). Gizlilik, yetkisiz kişilerin bilgi edinmesini önlemek anlamına gelmektedir. Bütünlük ise verinin gönderildikten sonra ve veriye ulaşmadan önce başka kişiler tarafından değişmediğini garantilemektedir. Kimlik denetimi ya da başka bir deyişe bilginin bulunabilirliği, verinin yetkili kişiler tarafından talep edildiği sürece mevcut olduğunu garantilemektedir (Khalifa, 2011). Kriptografi bize, en modern güvenlik protokolleri altında yatan araçları sağlamakta ve dağıtık sistemleri koruyan anahtar etkinleştirme teknolojisini uygulamaktadır (Wikipedia et al.,2014;Anderson, 2008).

Kriptografi, kriptografik tekniklerin kullanılması ve uygulanması anlamına gelmektedir. Kripto analiz, güvenli haberleşmeyi analiz ederek şifreleme algoritmalarının nasıl kırılacağı konusuyla ilgilenmektedir. Kriptoloji ise kriptografi ve kripto analizin birleşmiş çalışmasından meydana gelmektedir. Modern çağa göre kriptografi, bilginin okunabilir durumdan kimsenin anlayamayacağı anlamsız bir metne dönüşmesini simgeleyen şifreleme ile eş anlamlı kabul edilmektedir. Deşifreleme ise anlamsız olan metnin yeniden izin verilen kişilerce okunabilir duruma geçmesi olarak ifade edilmektedir. Birinci Dünya Savaşı'ndan beri ve bilgisayarın gelişiyle kriptolojiyi yürütmek için kullanılan yöntemler giderek daha karmaşık ve uygulaması da daha yaygın hale gelmiştir. Kriptografik algoritmalar, şifreleme ve deşifreleme süreçlerinde kullanılan matematiksel fonksiyonları içermektedir. Şifreleme ve deşifreleme işlemleri yapılırken güvenliği sağlama amacıyla anahtarlar kullanılmaktadır (Wikipedia, 2014). Günümüzde bu amaçla yazılmış algoritmalar *simetrik* ve *asimetrik* anahtarlı kripto algoritmaları olarak ikiye ayrılmaktadır. Simetrik algoritmalarda verinin şifrelenmesinde kullanılan anahtar ile şifrelenmiş verinin kriptosunun çözülmesi için kullanılan anahtar aynıdır (Yerlikaya vd.**,**Buluş, Buluş, 2006). Simetrik anahtar algoritmaları blok şifreleme şeması ve akış şifreleme şeması olmak üzere iki kategoride incelenmektedir. Blok şifreleme şeması, mesajı belirtilen uzunluktaki bloklara bölerek her bir bloğu ayrı ayrı şifrelemektedir. En basit haliyle, blok genişliği bir olduğunda blok şifreleme şeması akış şifreleme şemasına dönüşmektedir. Akış şifreleme şeması, blok şifreleme şemasına göre daha avantajlıdır. Hata eğilimli bir iletimde, gönderici deşifreleme sürecini etkilemeden hatalı veriyi yeniden gönderebilir. Buna ek olarak, telefon konuşmaları ve kablosuz ağ iletişimlerinde kolayca kullanılmaktadır. Fakat blok şifreleme şemasına göre daha az güvenli olduğu düşünülmektedir. DES (Data Encryption Standard), AES (Advanced Encryption

Standard), 3 DES ve Blowfish algoritmaları simetrik algoritmaya örnek olarak verilebilir (Khalifa, 2011). Asimetrik algoritmaya bakacak olursak; gönderen ve alıcı tarafında kullanılan anahtarlar farklıdır. Bu nedenle veri her iki tarafta farklı anahtar ile şifrelenmekte ve çözülmektedir. Asimetrik algoritmalar ayrıca açık anahtar algoritmaları olarak bilinmektedir. RSA (Rivest Shamir Adleman) algoritması asimetrik algoritmaya örnek olarak verilebilir. Açık anahtar algoritmaları temelde, uzunluğu 128 ile 2048 bit arasında değişen çok büyük tamsayıların ağır matematiksel problemlerine (modüler çarpma ve modüler üs alma) dayanmaktadır (Fadhil and Younis, 2014).

Simetrik ve asimetrik algoritmaların yanı sıra sayısal imzalar da güvenliği sağlamaya yönelik uygulamalardır. Sayısal imza, kimlik doğrulama ve inkar edememeyi sağlayan bir yöntemdir. Ayrıca mesajların içeriğinin değişip değişmediğini sorgulamayı sağlamaktadır. Sayısal imza bir kişi tarafından üretilmekte ve sadece bu imzayı bilen kişilerce kontrol edilmektedir (Tektaş, 2014).

Şifreleme algoritmalarının performans kriterlerinden bahsedilirse, kırılabilme süresinin uzunluğu, şifreleme ve çözme işlemlerinde harcanan zaman (zaman karmaşıklığı), şifreleme ve çözme işleminde gereksinim duyulan bellek miktarı (bellek karmaşıklığı), algoritmaya dayalı şifreleme uygulamalarının esnekliği, bu uygulamaların dağıtımındaki kolaylık ya da algoritmaların standart hale getirilebilmesi ve algoritmanın kurulacak sisteme uygunluğu gibi parametreler ön planda tutulmaktadır (Tektaş, 2014).

2.5.1 Çalışmada Kullanılan Güvenlik Algoritması

RSA adını, kendisini geliştiren kişilerin soyadlarının baş harflerinden almıştır. Çalışmada güvenlik algoritması olarak kullanılan RSA elektronik olarak imzalanmış iş sözleşmeleri, elektronik denetlemeler, elektronik siparişler ve diğer kimlik doğrulaması gerektiren elektronik haberleşmeler için çok iyi bir araçtır. Aynı zamanda bir genel anahtarlı şifreleme tekniği olan RSA, çok büyük tamsayıları oluşturma ve bu sayıları işleminin zorluğu üzerine düşünülmüştür. Anahtar oluşturma işlemi için asal sayılar kullanılarak daha güvenli bir yapı oluşturulmuştur. Anahtar oluşturma algoritması aşağıda anlatılmıştır:

- ✓ P ve Q gibi iki büyük tamsayı seçilir.
- ✓ Bu iki asal sayının çarpımı N=P*Q hesaplanır.
- ✓ N sayısının Phi fonksiyonu φ(N)=(P-1)(Q-1) şeklinde hesaplanır.
- ✓ 1'den büyük φ(N)'den küçük φ(N) ile aralarında asal bir E tamsayısı seçilir.
- ✓ Seçilen E tamsayısının mod φ(N)'de tersi alınır, sonuç D gibi bir tamsayıdır.
- ✓ E ve N tamsayıları genel anahtarı, D ve N tamsayıları ise özel anahtarı oluşturur.

- ✓ Genel ve özel anahtarları oluşturduktan sonra gönderilmek istenen bilgi genel anahtar ile şifrelenir.

Şifreleme işlemi şu şekilde yapılmaktadır:

- ✓ Şifrelenecek bilginin sayısal karşılığının E' ninci kuvveti alınır.
- ✓ Sonucun mod N deki karşılığı şifrelenmiş metni oluşturmaktadır.
- ✓ Genel anahtar ile şifrelenmiş bir metin ancak özel anahtar ile açılabilir.
- ✓ Şifrelenmiş metin, yine aynı yolla, şifrelenmiş metnin sayısal karşılığının D'ninci kuvveti alınır.
- ✓ Sonucun mod N deki karşılığı orijinal metni oluşturur (Vikipedi, 2014).

Bu algoritmada iki asal sayının çarpımını kullanarak anahtar oluşturulmasının sebebi, iki asal sayının çarpımını asal çarpanlarına ayırmak asal olmayan sayıları ayırmaktan daha zorlu olmasıdır. Algoritmada, en çok zaman harcanan kısımlar, üst alma ve mod bulma işlemleridir (Vikipedi, 2014).

RSA algoritmasının gerçekleştiriminde AES isimli başka bir kriptoloji algoritması da tercih edilmektedir. AES, 128 bit uzunluğundaki blok ile 128, 192 ya da 256 bit olan anahtar kullanmaktadır. Baytların yer değiştirmesi, 4x4'lük matrisler üzerine yayılmış metin parçalarının satırlarına uygulanan kaydırma işlemleri algoritmanın işlemlerini içermektedir. 2010 yılından beri en ünlü simetrik algoritmalardan biridir. AES algoritmasında giriş, çıkış ve matrisler 128 bitliktir. Matris 4 satır, 4 sütun (4×4), 16 bölmeden oluşur. Bu matrise 'durum' adı verilmektedir. Durumun her bölmesine bir baytlık veri düşer. Her satırda 32 bitlik bir kelimeyi meydana getirir. AES algoritması, 128 bit veri bloklarını 128, 192 veya 256 bit anahtar seçenekleri ile şifreleyen bir blok şifre algoritmasıdır. Anahtar uzunluğu bit sayıları arasındaki farklılık AES tur döngülerinin sayısını değiştirmektedir. Bu durum Çizelge 2.3'den görülebilir. Anahtar uzunluğunun şifrelemenin etkinliği açısından önemi bulunmaktadır. Pratik uygulamalarda en yüksek güvenilirlik sağlaması nedeniyle anahtar uzunluğu 256 olarak seçilir (Baydar, 2014).

Çizelge 2.3. AES Anahtar Uzunluğu ile Tur Sayısı İlişkisi (Baydar, 2014).

	Kelime Uzunluğu	**Tur Sayısı**
AES-128	4	10
AES-192	6	12
AES-256	8	14

3. PARALEL KRİPTOLOJİ

3.1 Kriptolojinin Paralel Yapılma Nedenleri ve Paralellik Çeşitleri

Paralel kripto, büyük ölçekli kriptografik problemlere paralel hesaplama uygulayarak gelişen bir araştırma alanıdır (Kaminsky, 2010). Kriptografik algoritmalar, internet üzerinden birbiriyle etkileşimde olan tarafların güvenlik düzeyini korumak için yaygın olarak kullanılmaktadırlar. Açık anahtar ve simetrik şifreleme standartları, insanların internete güvenmesini sağlamak amacıyla iş birliği yapmaktadırlar. Bazen e-Ticaret alanı gibi bütün bir iş internet üzerinden halledildiğinde güven konusu büyük önem kazanmaktadır. Bu sebeple, kullanıcı etkileşimi gerektirmeyecek şekilde kriptografik algoritmalarını kullanan SSL (Secure Socket Layer), TLS (Transport Layer Security) ve HTTPS (Hypertext Transfer Protocol Secure) gibi bazı protokoller tanımlanmıştır. Bu protokoller VPN (Virtual Private Network) gibi internet ortamında güvenli bir ağ kurabilen diğer teknolojileri görünür kılmaktadırlar. Böylelikle sadece izinli kişiler ağa katılabilmektedir. Dahası, ağa katılan tüm bilgisayarlar arasındaki trafik şifreli bir biçimdedir (Khalifa, 2011).

Sık kullanılan algoritmaların performansını arttırmak için başvurulan yollardan birisi de bu sorunu çözecek bir özel bir donanım parçası tasarlamaktır. Ancak aşağıda sıralanan bir dizi kısıtlamayla karşı karşıya kalınabilir.

(a) Bu yaklaşım çok pahalı olması nedeniyle büyük şirketler ve kuruluşlar tarafından kabul edilmesi zor olan bir yaklaşımdır. Fakat normal bilgisayar kullanıcıları tarafından benimsenebilir bir fikirdir.

(b) Genellikle özel tasarlanmış donanımların esneklik ve ölçeklenebilirlik konusunda bir sınırlandırması vardır yani bir süre sonra başka bir araç gerekmektedir (Khalifa, 2011).

Öte yandan, sorunu çözmek için paralel programlamadan faydalanan başka bir yaklaşım kullanılabilir. Bu çözüm, özel donanımın sağladığı performans gelişim seviyesine ulaşmayabilir ama özel donanıma göre daha ucuz bir alternatif sağlayarak ilk yaklaşım kısıtlamalarını çözebilir. Hatta normal kullanıcı makinelerinde bile uygulanabilir. Ayrıca bu yaklaşımda paralel algoritma güzel tasarlanmışsa ölçeklenebilirlik çok daha kolay sağlanmaktadır. Bundan başka, tipik bir bilgisayar sistemi bugünlerde sekiz ya da daha fazla çekirdeğe sahip çok çekirdekli işlemci barındırmaktadır. Bu sebepten, genel performansı arttırmak için tüm çekirdekleri kullanmak akıllıca bir karardır (Khalifa, 2011).

RSA algoritması çok güvenli olmakla çok ağır matematik hesaplamalarına dayalı olduğundan yavaş çalışmaktadır. Bu yüzden, algoritmayı paralelleştirmek hem performansı daha da hızlandıracak hem de algoritma tarafından sağlanan güvenlik düzeyi artan anahtar

uzunluğuyla daha da yükselecektir. Şu an RSA, yeterince güvenli kabul edilmeyen 1024 bit anahtar uzunluğu kullanmaktadır. Fakat ne kadar uzun anahtar kullanılırsa, bir o kadar güvenlik sağlanmakla beraber daha yavaş çalışan bir performans elde edilmektedir. Bu durum, performansı olumsuz yönde etkilemeden güvenliği arttırmaya yarayan paralel programlamaya ihtiyaç duymaya yol açmaktadır (Khalifa, 2011).

AES şifreleme ve deşifreleme işlemleri oldukça hızlıdır fakat kullanıcılar ile güvenli kanallar oluşturan internet sunucularının performansını geliştirmek gerekli olabilir. Performans iyileştirilmediği takdirde şifrelenen ve deşifrelenen mesajlar kanallarda bir tıkanıklık oluşturabilir (Khalifa, 2011).

Paralel kriptografi başka bir açıdan incelenecek olursa paralelleştirme yöntemleri, bilgisayar sistem paralelliğinin ne derece uygulandığı kıstasına göre sınıflara ayrılmaktadır. Burada bahsedilen ağ kriptografisidir fakat benzer örnekler tek kullanıcılı (yerel) kriptografide de var olmaktadır. Sınıflandırma düzeyleri, bağlantı başına (işlem başına), paket başına (dosya başına) ve paketler arası (veri düzeyi) paralellik şeklinde olmaktadır. Bağlantı başına paralellik yönteminde, sunucuya kullanıcı tarafından yapılan her bir bağlantı, sadece bir işlemci üzerinde çalışan bir iş parçacığı ya da işlem ile yürütülmektedir. Bu uygulama, paralelliğin en yaygın yöntemidir ve var olan algoritma için hiçbir değişiklik gerektirmez. Genellikle mevcut sunucu yazılımı için de bir değişikliğe ihtiyaç duyulmamaktadır. Sunucu işleminin birden fazla örneğini çalıştırarak bu paralellik seviyesi elde edilebilir. Bu metot, tek bir bağlantıyı hızlandırmada sınırlı kalmıştır. Bu paralellik seviyesi ayrıca yeni işlemcilerde çalışan eski algoritmaların sorunlarına çözüm bulamamaktadır. Yani, işlemci başına eski bir algoritma örneği çalıştırdığımızda bu yöntem, algoritmayı yeni işlemci üzerinde algoritmanın bir önceki mimaride çalışır versiyonuna göre daha etkin kılmak adına herhangi bir girişimde bulunmamaktadır. Bağlantı başına paralellik, kriptografik uygulamaları çalıştıran çok işlemcili makinelerin satılmasında büyük rol oynamaktadır. Bağlantı başına paralellik, modern mimarileri tamamen kullanmamaktadır (Seidel, 2003).

Paket başına paralellik, bağlantıların paket işleme yükünü birden çok işlemciye dağıttığı bir yöntemdir. Burada her bir paket ayrı ayrı işleme tabi tutulmaktadır. Bu yönteme örnek olarak tüm bağlantıları idare eden bir grup işlemcinin (ya da iş parçacığının) hazırlanan paketleri geçici bir belleğe koyması ve geçici bellekte yine bir grup işlemci tarafından işlenmesi (şifrelenmesi) verilebilir. Bu tasarımda tek bir işlemci çeşitli bağlantılardan paketleri üstlenebilir ya da tek bir bağlantı çeşitli işlemciler tarafından şifrelenen mesajları kullanabilir. Paket başına paralellik, uygulamaların kriptografi kısmına özel bir işlemci devredilen özel kriptografik donanım tasarımlarına benzemektedir (Seidel, 2003).

Paketler arası paralellik ağır algoritma tasarımlarına bağlı olduğundan paralelliğin en zor tipi olarak tanımlanabilir. Bu yönteme örnek olarak ECB (Electronic Code Book) modunda çalışan DES gibi bir blok şifreleyici verilebilir. ECB modunda, mesajın her bir bloğu diğerlerinden bağımsız olacak şekilde paralel olarak hesaplanabilir. Bu paralellik seviyesi kriptografik algoritmaların değiştirilmesine ihtiyaç duymaktadır. Üstelik bu paralellik, algoritmanın üzerinde çalıştığı işletim sistemi veya donanımın esnekliğine bağlı değildir (Seidel 2003).

3.2 Konuyla İlgili Yapılmış Çalışmalar

Konuyla ilgili geçmişte birçok çalışma yapılmıştır. Aşağıda bu çalışmaların yöntemleri ve sonuçları ve ayrıca en sonunda bizim çalışmamızın bu çalışmalardan farkı anlatılacaktır.

İnceleyeceğimiz ilk çalışmada RSA algoritmasının 3 farklı versiyonu OpenMP ile paralelleştirilerek Linux işletim sisteminde GCC yapısıyla iki çekirdeğe sahip bilgisayarlar çeşitli deneyler gerçekleştirilmiştir. Deneyler için, RSA uygulamasının 3 farklı versiyonu geliştirilmiştir. Öncelikle RSA algoritmasının seri uygulaması ve daha sonra modüler üs alma ve modüler indirgeme işlemlerinde kullanılan ve ünlü hafıza verim yöntemlerine dayanan 2 paralel uygulama yazılmıştır. Bu paralel uygulamalar, tekrarlanan kare ve çarpma ve sağdan sola ikili yöntemleridir. Bu yöntemler deneyler gerçekleştirildikten sonra önemli iyileştirmeler kaydedilmiştir. Tekrarlanan kare ve çarpma yöntemi, çok büyük e sayısı için performansı önemli ölçüde arttırmaktadır. Bu yöntemi uygulamak için başvurulan prensip, veri ayrıştırmasıdır. RSA algoritmasının modüler üs safhası da aynı mantıkla paralelleştirilmiştir. Şekil 3.1.'de yöntemin işleyiş biçimi gösterilmiştir (Saxena et al.,Kishore, Handa, Kapoor, 2013).

Eğer e bileşeni çift ise, sistemdeki çekirdeklerin durumuna bakılarak 2 ya da 4 parçaya ayrılmaktadır. Daha sonra üs hesaplamanın her bir parçası, çoklu iş parçacıklarına atanmakta ve sonunda toplam sonuca ulaşabilmek için her bir iş parçacığının bulduğu sonuç birbiriyle çarpılmaktadır. Eğer e bileşeni tek ise, aşağıdaki adımlar tatbik edilir:

1) g değeri bir A değişkeninde tutulur.
2) e değerini çift yapabilmek için 1 eksiği alınır.
3) e değeri, hedef makinedeki çekirdek sayısına bağlı olarak 2 ya da 4 parçaya ayrılır.
4) Her bir üs hesaplama, ayrı bir iş parçacığı ve işlemcide paylaştırılır.
5) Her bir iş parçacığından elde edilen sonuç, toplam sonucu hesaplamak için çarpılır.

Toplam sonuç ile A değişkeninin değeri çarpılır ve en son sonuç elde edilir (Saxena et al.,Kishore, Handa, Kapoor, 2013).

$$ModExp(g, e, m) = \begin{cases} 1, & \text{e = 0 ise} \\ (g \times ModExp(g, (e-1), m)) \bmod m, & \text{e tek ise} \\ ModExp(g, e/2, m)^2 \bmod m, & \text{e çift ise} \end{cases}$$

Şekil 3.1. Tekrarlanan Kare ve Çarpma Yöntemi (Saxena et al., Kishore, Handa, Kapoor, 2013).

İkinci yöntem sağdan sola ikili olarak adlandırılmıştır çünkü üssün ikili gösterimi sağdan sola doğru hesaplanmaktadır. İlk olarak, üs ikili gösterime çevrilmekte ve önce en sağdaki bit analiz edilmektedir. Algoritmadaki döngü, üssün ikili gösterimindeki bit sayısı kadar çalışmaktadır. Hesaplama Şekil 3.2'de gösterilen formül ile yapılmakta ve algoritma Şekil 3.3'den görülebilmektedir (Saxena et al**,** Kishore, Handa, Kapoor, 2013).

$$\prod_{i=0}^{n-1} \left(b^{2^i}\right)^{a_i} \pmod m$$

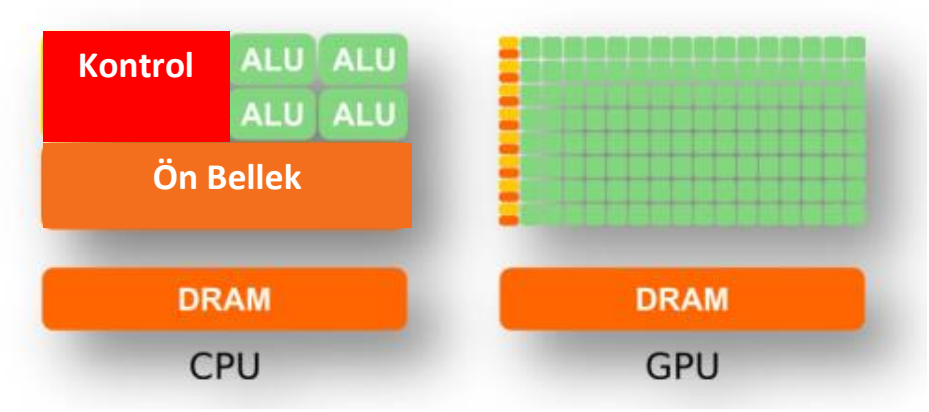
Şekil 3.2. Sağdan Sola İkili Yöntem Formülü (Saxena et al., Kishore, Handa, Kapoor, 2013).

```
Function power_binary(base, BinaryNumber, length,
modulus)
  Result =1
    FOR i = length to 0
     IF binaryNumber[i] % 2 == 1
        Result=(result * base)%modulus;
     END IF
     base = (base * base) % modulus;
    END FOR
  return Result
```

Şekil 3.3. Sağdan Sola İkili Yöntem Algoritması (Saxena et al., Kishore, Handa, Kapoor, 2013).

Programın seri versiyonu C dili ile geliştirilmiştir ve Linux platformunda GCC yapısı üzerinde çalıştırılmıştır. Paralel programlar geliştirmek için, C dilinin GCC yapısı üzerinde birleştirilmesiyle oluşan OpenMP API kullanılmıştır. Algoritmayı uygulamak için büyük tam sayılı hesaplamalara ihtiyaç duyulmuştur. Deney esnasında performans, zaman cinsinden analiz edilmiş ve performans kazanımı ölçülmüştür. Çeşitli versiyonların çalışma esnasında harcanan zaman, Linux'un zaman fonksiyonu kullanılarak ölçülmüştür. Tüm versiyonlar aynı mesaj seti ile tam olarak 25 kere çalıştırılmış ve her bir versiyon için sonuç değeri, ortalama zaman olarak alınmıştır. Tekrarlanan kare ve çarpma yöntemi, algoritmanın seri uygulamasına göre 22.42%'lik bir gelişim gösterirken sağdan sola ikili yönteminde 26.54%'lük bir gelişme söz konusudur (Saxena et al.**,**Kishore, Handa, Kapoor, 2013).

Yine RSA ile yapılan bir başka çalışmada ise GPU (Grafik İşleme Birimi)'ya yer verilmiştir. Sadece çok çekirdekli işlemciler değil, aynı zamanda güçlü grafik kartları da daha fazla kullanılabilir duruma gelmiştir. GPU'lar, esneklikleri ve makul ücretlerinden dolayı birçok yüksek hesaplama gerektiren uygulamalarda güçlü bir hızlandırıcı olarak artan bir şekilde kullanılmaktadır. CPU (Merkezi İşlem Birimi) ve GPU arasındaki temel fark, transistörlerin işlemci içinde nasıl yerleştirildiğinden gelmektedir. Şekil 3.4'den görüldüğü gibi, GPU'da mikrodevre alanı içerisinde ALU'lara (Aritmetik Mantık Birimi) daha fazla yer verirken, CPU'lar ön bellek için geniş bir alan kullanmaktadır. Bu çalışma, RSA algoritmasının çok çekirdekli CPU ve birden fazla çekirdeğe sahip GPU'lardan oluşan hibrit sistemde paralelleştirilmesini ele almaktadır (Fadhil and Younis, 2014).



(a) (b)

Şekil 3.4. a) Merkezi İşlem Birimi, b) Grafik İşleme Birimi (Fadhil and Younis, 2014).

RSA algoritması, key generation, şifreleme ve deşifreleme olmak üzere 3 ana parçaya ayrılmaktadır. 1024 bitlik gibi büyük sayıları gösterebilmek için BigInteger sınıfı kullanılmıştır. Şifreleme ve deşifreleme süreçleri aşağıda listelenen 4 farklı durumla gerçekleştirilir.

- ✓ RSA algoritmasını verimli bir şekilde çalıştıran ve standart olarak bilinen Crypto++ kütüphanesi kullanılmaktadır. Crypto++ kütüphanesi, 1024 bitlik standart bir anahtar uzunluğu kullanılarak yazılan bir kütüphanedir.
- ✓ RSA algoritmasının seri uygulaması CPU üzerinde Montgomery algoritmasına bağlı olarak çalışmaktadır.
- ✓ RSA algoritması çok çekirdekli CPU üzerinde paralel bir şekilde çalışmaktadır.
- ✓ RSA algoritması çok sayıda çekirdek barındıran GPU üzerinde çalışmaktadır (Fadhil and Younis, 2014).

Crypto++ kütüphanesinin aksine, önerilen farklı uygulamalar ihtiyaca göre değişken boyutta anahtarları desteklemektedir. RSA'in ana problemi, şifreleme sürecinin geniş bir veri

setinden oluşmasıdır. RSA algoritmasının paralel uygulamasını sağlayabilmek için veriler arasında bağımsızlıkların olmaması gerekmektedir. Veriler arasında bağımsızlığın olmaması, verinin küçük parçalara ayrılabilmesi anlamına gelmektedir ki, her bir iş parçacığı ayrı bir parçayı hesaplayabilir. Sonuç olarak; bu veri paralelleştirme yöntemi RSA'in hesaplama hızını arttırır (Fadhil and Younis, 2014).

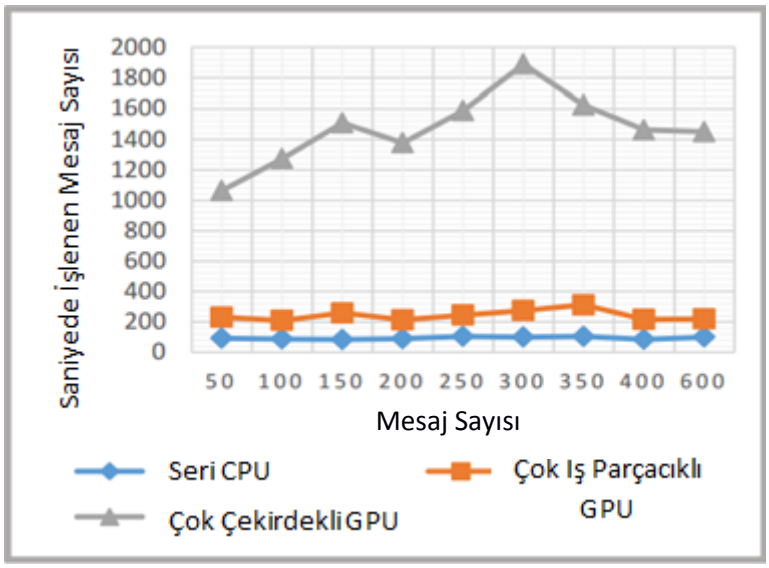
İş parçacığı seviyesinde, düz metin ya da şifrelenmiş metin aynı boyutta birçok parçaya ayrılmaktadır. Aynı şifreleme ve deşifreleme operasyonları her bir parçaya uygulanır. Yani şifreleme ve deşifreleme süreçleri çoklu iş parçacıkları ile yapılabilir. Her bir iş parçacığı sadece kendi atandığı veriden bir sonuç kazanmaya ihtiyaç duyar. Yani bir başka deyişle; her iş parçacığı farklı bir modüler üs görevi üstlenir. Çalışma için gereken deney grupları aşağıdaki gibi tanımlanmıştır:

Grup-1: 760 bitlik sabit uzunluktaki mesajın, 768 bitten 8192 bite kadar değişen uzunluktaki anahtar ile şifrelenmesi ve deşifrelenmesi.

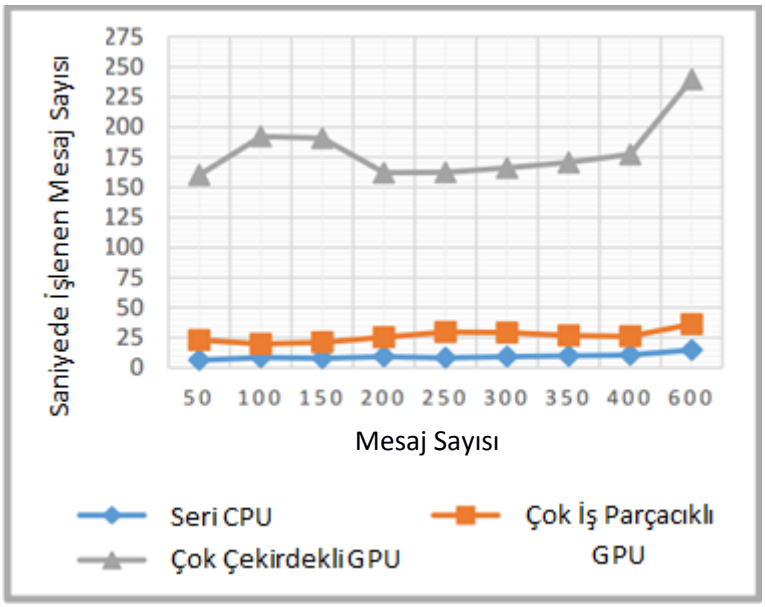
Grup-2: Girdi olarak kullanılan mesajın, şifreleme anahtarının uzunluğuna bağlı olarak artması (modülüs uzunluğundan 1 byte daha az olacak şekilde)

Grup-3: Blok genişliğinin, mesaj uzunluğunun katı olacak şekilde arttırılması. Bu bölümde, çıktıya göre değişen hız kazancı incelenmiştir (Fadhil and Younis, 2014).

GPU üzerinde çalıştırılan algoritma, CPU üzerinde seri uygulanan algoritmaya göre 23 kat daha hızlı çalışmaktadır. Yine CPU üzerinde gerçekleştirilen çoklu iş parçacıklı paralel uygulama, gecikme (latency) söz konusu olduğu için yine CPU üzerindeki seri algoritmaya göre sadece 6 kat daha hızlı çalışmaktadır. Veri bazında hızlanma incelendiğinde, 1024 bit anahtar için saniyede 1800 bit mesaj deşifrelenirken, 2048 bit anahtar için ise saniyede 250 bit mesaj deşifrelenmiştir. Deneyler, Intel Core I7-2670QM, 2.20 GHz CPU ve Nvidia GeForce GT630 GPU mekanizmalarına sahip bir laptop üzerinde yapılmıştır. Deney sonuçları GPU'nun RSA algoritmasını hızlandırmak için daha uygun olduğunu göstermiştir. Uygulamalar ise, C# programlama diliyle, GPU.net framework üzerinde gerçekleştirilmiştir. Şekil 3.5'de 1024 bit ve Şekil 3.6'da 2048 bit anahtarla deşifreleme işleminin verimliliği gösterilmiştir (Fadhil and Younis, 2014).



Şekil 3.5. 1024 Bit Anahtarla Deşifreleme İşleminin Verimlilik Grafiği (Fadhil and Younis, 2014).



Şekil 3.6. 2048 Bit Anahtarla Deşifreleme İşleminin Verimlilik Grafiği (Fadhil and Younis, 2014).

Bunun yanı sıra, bir başka yapılan çalışma AES algoritmasının paralelleştirilmesi üzerinedir. Bu çalışma, veri ve kontrol seviyesinde AES algoritmasının optimize edilmiş paralel bir mimarisini sunmaktadır. Ayrıca mimari, çok çekirdekli ortamla uyumlu bir şekilde çalıştırılmıştır. AES algoritması, C programlama dilinde ve OpenMP standartlarında paralelleştirilerek yazılmıştır. Uygulamanın performans analizi Intel VTune Amplifier XE 2013 kullanılarak yapılmıştır. AES algoritması paralelleşebilen ve paralelleşemeyen parçalara ayrılmıştır. OpenMP direktifleri veri bağımlılıkları ve senkronizasyon problemlerine karşı koyarak kullanılmıştır. Daha sonra fork-join model konsepti kodu tek bir parça haline getirmek için bu parçaları birleştirme işleminde kullanılmıştır. Kullanılan sistem Intel X64 mimarisidir ve 2.3 GHz CPU'ya sahiptir. Çizelge 3.1, farklı dosya uzunlukları için seri ve

paralel kodların çalışma sürelerini göstermiştir. Çizelgeden, dosya boyutu arttıkça paralel kodun daha iyi bir performans sergilediği sonucu çıkarılmaktadır. Tüm durumlarda, seri kodun çalışma süresi paralel koda göre daha fazla olmuştur. Dosya boyutu 5 KB'tan daha küçük olduğunda aralarındaki fark ayırt edilemez olmuştur (Navalgund et al.,Desai, Angalki, Yamanur, 2013).

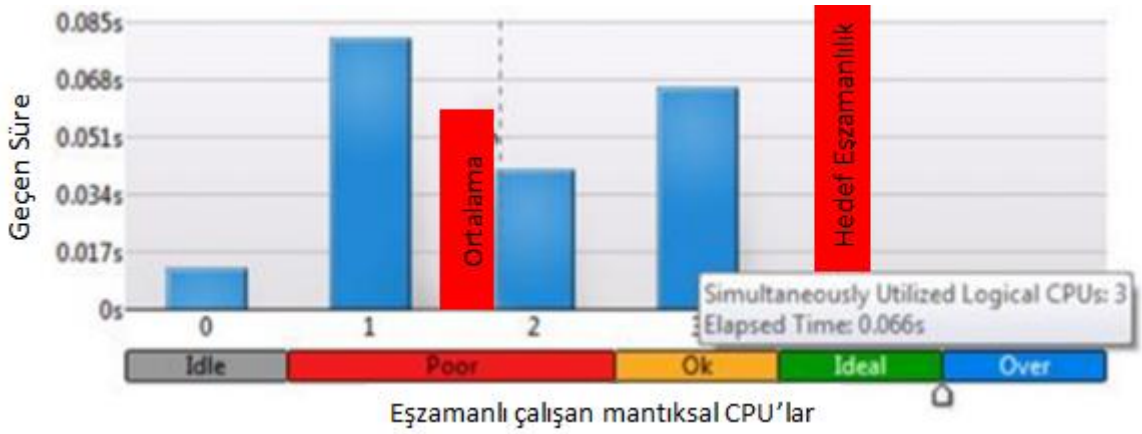
Diğer bir analiz, yine 5 KB boyutundaki aynı dosyanın şifrelenmesi için mevcut iş parçacıklarının sayısı değiştirilerek yapılmıştır. Bulgular Çizelge 3.2'de gösterilmiştir. VTuneTM Amplifier XE 2013 kullanılarak elde edilen sonuçların analizi ve optimizasyonu yapılmıştır. Şekil 3.7, geçen sürenin kırılımını gösteren CPU kullanım histogramını sunmaktadır. Bu histogram, aynı anda çalışan belirli sayıdaki CPU'ların toplam zamanın yüzde kaçını içerdiğini göstermektedir. Bu ölçüm, VTuneTM Amplifier XE 2013 kullanılarak karmaşık bölgelerin performans analizinin yapılmasıyla elde edilmiştir. Şifrelenen dosya büyüklüğü ise 50KB'dır. Sonuçlar gösteriyor ki; moden simetrik çoklu işlemci platformlarında OpenMP tabanlı yüksek seviye dil paralelleştirmesiyle ilgi çekici performans oranları elde edilmiştir (Navalgund et al.,Desai, Angalki, Yamanur, 2013).

Çizelge 3.1. Seri ve Paralel Kodun Zaman Bakımından Karşılaştırılması (Navalgund et al., Desai, Angalki, Yamanur, 2013).

Dosya Boyutu	**Çalışma Zamanı**	
	Seri Kod	**Paralel Kod**
5 KB	0.045 sec	0.025 sec
10 KB	0.072 sec	0.058 sec
20 KB	0.128 sec	0.104 sec
40 KB	0.245 sec	0.211 sec
50 KB	0.334 sec	0.279 sec
100 KB	0.770 sec	0.698 sec
200 KB	1.343 sec	1.002 sec
400 KB	2.461 sec	1.867 sec
800 KB	5.439 sec	4.238 sec
1 MB	6.799 sec	5.010 sec
2 MB	13.649 sec	11.483 sec
5 MB	32.397 sec	27.380 sec

Çizelge 3.2. İş Parçacığı Sayısına Bağlı Olarak Paralel Kodun Çalışma Süresi (Navalgund et al., Desai, Angalki, Yamanur, 2013).

5 KB Dosya Boyutu için Analiz Sonuçları	
İş Parçacığı Sayısı	**Çalışma Zamanı**
1	**0.045 sn**
2	**0.037 sn**
3	**0.032 sn**
4	**0.026 sn**



Şekil 3.7. İşlemci Kullanım Histogramı (Navalgund et al., Desai, Angalki, Yamanur, 2013).

Diğer bir çalışma RSA'in deşifreleme işleminin performansını arttırmaya yöneliktir. Günümüze kadar deşifreleme operasyonunu hızlandırmak için bazı yöntemler önerilmiştir. RSA'i hızlandırmak için uygulanan bu yöntemler, geliştirilmiş RSA algoritmaları, daha yüksek saat hızı, özel amaçlı donanımlar ve paralel bilgisayar ve algoritmalar şeklinde sıralanabilir. Bu çalışma, sıralanan seçeneklerin sonuncusuna odaklanmaktadır. Bu çalışmada, paralel makinelere çok uygun olan ve modüler üs işlemi için geliştirilen yeni bir algoritma ve onun uygulaması öne sürülmüştür. RSA algoritmasının yeni bir varyasyonu olan BS1PRSA (Batch RSA-S1 Multi-Power Improved RSA) adlı algoritma tasarlanmıştır. Bu varyasyon, Multi-Power RSA ile Batch RSA algoritmasına bağlı olan RSA-S1 sistemlerini etkin bir biçimde birleştirmektedir. BS1PRSA, Batch RSA gibi Setup, Percolate-Up, Exponentiation-Phase ve Percolate-Down safhalarından oluşmaktadır.

Setup: Güvenlik parametresi n ve dört ekstra k, c, b (girdi) ve b (batch genişliği) parametrelerinden oluşmaktadır.

- ✓ İki asal sayı olan p ve q'nun değerleri belirlenir. Güvenlik parametresi olan n [n/3] şeklinde üç parçaya ayrılır. Ayrıca $N = p^2xq$ ve N değerinin Phi fonksiyonu (p-1) x (q-1) şeklinde hesaplanır.
- ✓ Varsayalım ki; $e_1, e_2, \ldots\ldots, e_b$ şeklinde b sayıda farklı şifreleme üsleri olsun. Ayrıca, açık olan bu üsler hem birbirleri ile hem de N'in Phi fonksiyon sonucu ile

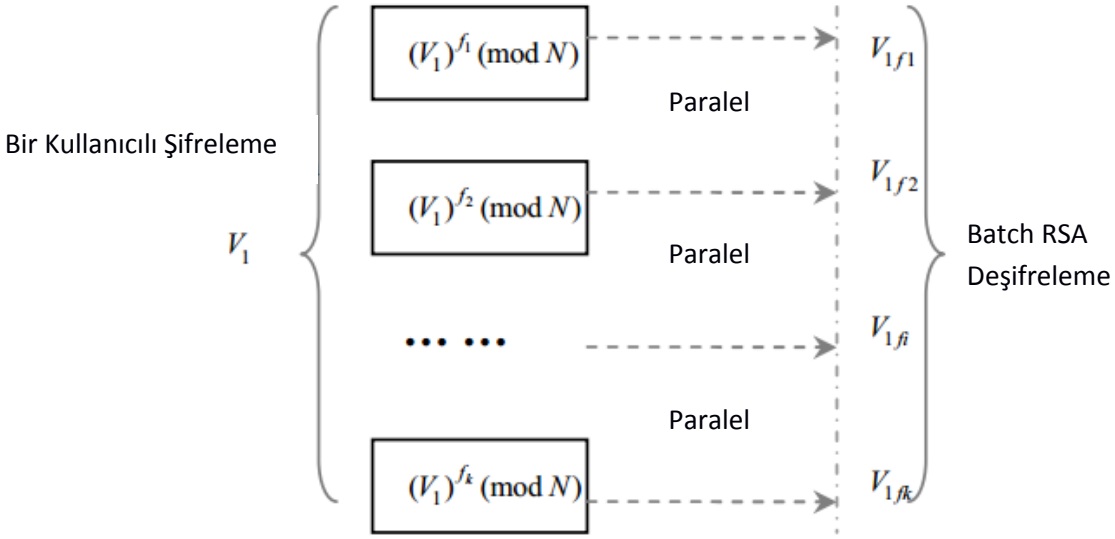
aralarında asal olmak zorundadır. Her bir e_i çok küçük olmalıdır çünkü fazladan aritmetik gerektiren bir işlem söz konusu olabilir. 1<= i <= b tamsayı aralığında, her e_i için $d_{ei} = e_i^{-1} \bmod \varphi(N)$, $E = \prod_{i=1}^{b} e_i \pmod N$ değerleri hesaplanır. Özel anahtar değeri $d = d_{e1} \times d_{e2} \times \ldots\ldots \times d_{eb} \bmod \varphi(N)$ şeklinde hesaplanır. Aynı zamanda; $e\_inv\_p = e^{-1} \bmod p$ ve $p^2_inv_q = (p^2)^{-1} \bmod q$olarak bulunur

Özel anahtar olan d, $d= f_1d_1 + \ldots..+ f_kd_k \bmod \varphi(N)$ şeklinde gösterilir. Her d_i ve f_i, 1<= i <= k aralığında ve c ve |n| sayıda bitin rasgele (random) vektör elemanı olarak ifade edilir. Makalede; k sayısı 2 olarak seçilir ve d sayısının gösterimi Exponentiation-Fazında kullanılmak üzere $d= f_1d_1 + f_2d_2 \bmod \varphi(N)$ eşitliğine dönüşür.

- ✓ Bu algoritmada 1<= i <= b aralığında her bir şifreleme için, özel anahtar <N, d_1, d_2> ve açık anahtar <N, e_i, f_1, f_2, e_inv_p, $p^2\_inv\_q$ > olarak belirlenir.
- ✓ Verilen $m_1,\ldots\ldots,m_b$ mesaj aralığı için $v_i = m_i \pmod N$ formülü kullanılarak 1<= i <= b aralığında v_i hesaplanır (Tan and Li, 2012).

Paralel bir algoritma tasarlamanın ilk adımı, problemi aynı anda çalışabilen daha küçük elemanlara ayrıştırmaktır. Bu ayrıştırmayı iki boyutlu olarak düşünebiliriz.

- ✓ Veri ayrıştırması, görevlerin ihtiyaç duyduğu veri ve verinin farklı kümelere nasıl ayrıştırılabileceği konuları üzerine yoğunlaşır. Veri kümelerine ilişkin hesaplama, eğer veri kümeleri birbirinden bağımsız bir şekilde çalışıyorsa ancak o zaman verimli olabilir. BS1PRSA ana verileri açık anahtar, özel anahtar, düz metin ve şifreli metindir. Standart RSA ile karşılaştırıldığında, açık anahtar <N, e_i, f_1, f_2, e_inv_p, $p^2\_inv\_q$ > ve özel anahtar <N, d_1, d_2> 'dir. Özel anahtar ve açık anahtar bir matrise ayrıştırılır. Matristeki her bir satır açık anahtar ve özel anahtar çiftinden oluşur. Bu çiftler, şifreleme ve deşifreleme operasyonları için girdi olarak alınır bu veri satırları birbirinden bağımsız olarak çalışabilir. Her veri satırı birbirinden bağımsız olarak çalıştığı için, veri satırlarını ayrı bir görevle ilişkilendirerek uygulamayı paralelleştirmek mümkündür.
- ✓ Görev ayrıştırması, problemin sırayla aynı anda çalışabilen görevlere bölünebilmesidir. Hesaplamanın verimli olabilmesi için, görevde yer alan işlemlerin diğer görevlerde çalışan işlemlerden tam anlamıyla bağımsız olması gerekir. BS1PRSA algoritması şifreleme ve deşifrelemeye odaklanmıştır. Her bir BS1PRSA şifrelemesi, k sayıda modüler üs işlemi gerçekleştirir. Şekil 3.8, tek bir paralel şifreleme işlemini ve Şekil 3.9 ise dört şifreleme kullanıcısının paralel şifreleme sürecini gösterir (Tan and Li, 2012).



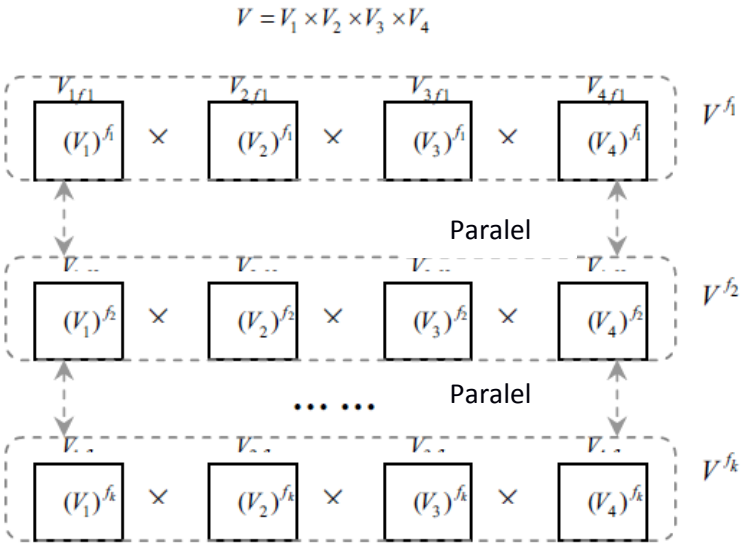
Şekil 3.8. Bir Kullanıcıya Ait Paralel Şifreleme İşlemi (Tan and Li, 2012).

Yukarıda gösterilen görevlerin birbirinden bağımsız bir şekilde çalışması, girdilerin birbirinden bağımsız olmasına bağlıdır. Şekil 3.10, BS1PRSA deşifrelemesinde birçok eş zamanlı olarak çalışabilecek görevleri göstermektedir. Böylece, veri ayrıştırması ve görev ayrıştırması analizine bağlı olarak, söz konusu algoritma verimli bir şekilde paralel olarak yazılabilir. Aynı zamanda Şekil 3.10, BS1PRSA deşifrelemesinin paralelleştirilebileceğini göstermektedir. Standart RSA büyük modüler üs işlemi, daha küçük modüler üslere ayrıştırılır. Söz konusu değişkenin şifreleme ve deşifreleme işlemleri aynı anda çalışan birden fazla görevlere ayrılabilir. Paralel uygulama için, paralel donanım platformlarından ziyade çok çekirdekli bilgisayarlar seçilebilir. Ayrıca yazılım geliştirme platformu olarak OpenSSL kriptografik kütüphanesi ve OpenMP kullanılabilir (Tan and Li, 2012).

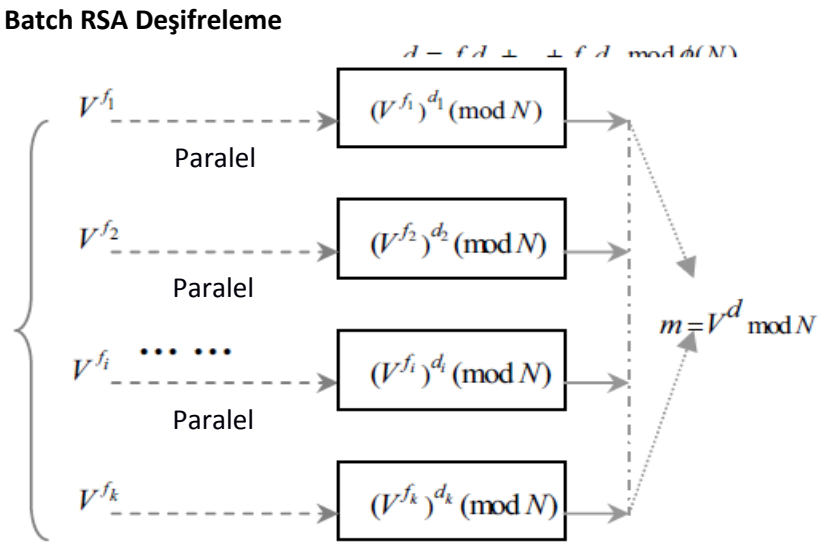
Gerçekleştirilen başka bir çalışmada AES şifreleme uygulaması yine OpenMP kullanılarak Visual Studio 2005, Intel C++ derleyici, Windows XP, 1GB RAM ve çift çekirdeğe sahip bir sistemde paralelleştirilmiştir. AES algoritmasının şifreleme ve deşifreleme süreçleri, çalışma süresinin kısaltılması için OpenMP direktifleri kullanılarak iki çekirdek arasında paylaştırılmıştır. Şifreleme ve deşifreleme için tutulan dosyadan bir zamanda n-blok veri okunmaktadır. Daha iyi bir sonuç almak için n sayısı 1000, 2000, 3000 gibi büyük sayılardan oluşmaktadır. İlk n/2 blok 0 nolu çekirdeğe atanırken, diğer n/2 blok 1 nolu çekirdeğe atanmıştır. Bu durumda blokların bir kısmının bir çekirdekte, bir kısmının da diğer çekirdekte olmasından dolayı eş zamanlı ve çoklu iş parçacığı konseptinden faydalanılmış olunmaktadır. Bu işlem dosya sonuna ulaşana kadar devam etmektedir. Son şifreleme ve deşifreleme adımlarından sonra tüm dosyaya ait şifreleme ve deşifrelemede harcanan zaman

hesaplanmaktadır. Performans kazancı aşağıdaki formül ile bulunmuştur (Nagendra and Sekhar, 2014).

$$\text{Zaman} = \frac{\text{(Seri Uygulama Zamanı-Paralel Uygulama Zamanı) x 100}}{\text{Seri Uygulama Zamanı}}$$

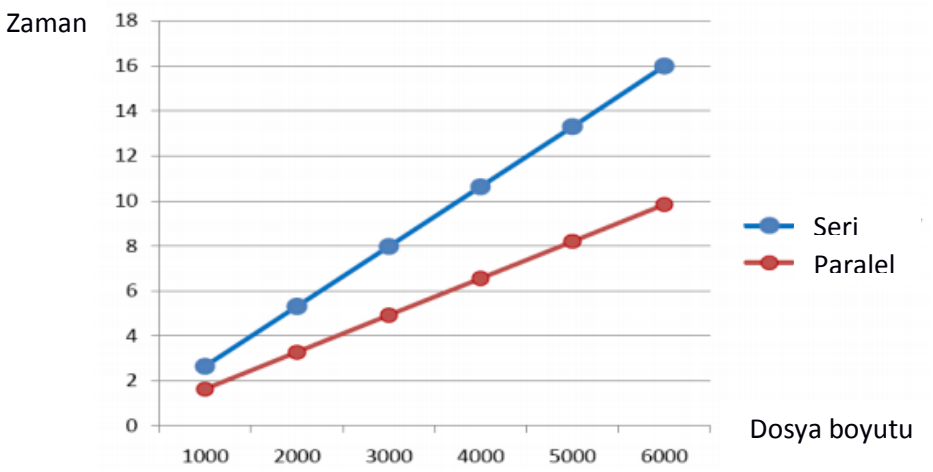


Şekil 3.9. Dört Kullanıcıya Ait Paralel Şifreleme İşlemi (Tan and Li, 2012).

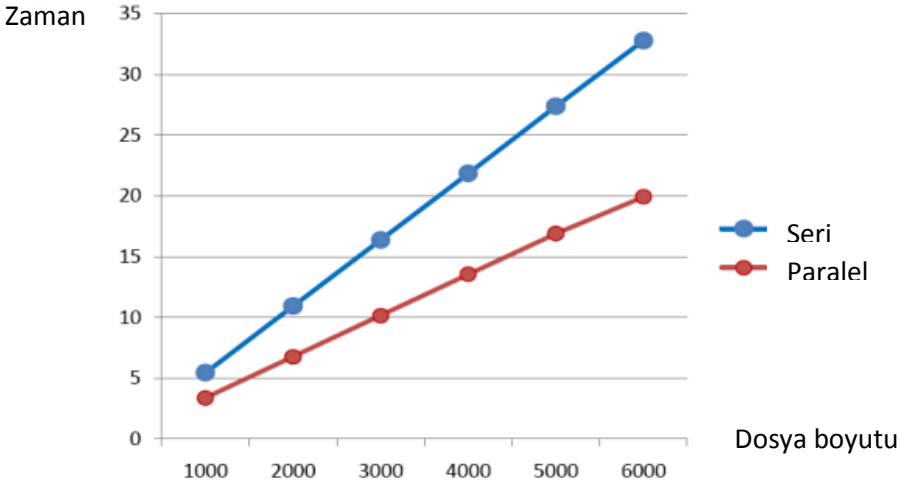


Şekil 3.10. Batch RSA Sunucusu İçin Deşifrelemenin Paralelleştirilmesi (Tan and Li, 2012).

Şekil 3.11'den görüldüğü gibi paralel uygulamada şifreleme sürecinde performans gelişimi vardır ve performans tüm dosya boyutları için sabit değildir. Küçük dosya boyutları için performans daha azken, dosya boyutu arttıkça performans da artmaktadır. Fakat bu artma belirli bir değere kadar devam etmekte ve bu değerden sonra sabit kalmaktadır. Şifrelemede, çok küçük dosyalar için performanstan kazanç 38% iken çok büyük dosyalar için bu kazanç maksimum 47%'ye kadar çıkmıştır. Bu sonuçtan ise paralelleştirme üzerinde veri boyutunun etkisi olduğu çıkarılmaktadır (Nagendra and Sekhar, 2014).



Şekil 3.11. Şifreleme için Harcanan Zaman (Nagendra and Sekhar, 2014).



Şekil 3.12. Deşifreleme için Harcanan Zaman (Nagendra and Sekhar, 2014).

Şekil 3.12'de deşifreleme sürecinin seri ve paralel performansların gösterilmiştir. Deşifrelemede ise yine paralel uygulamada performans gelişimi gözlemlenmiştir. Yine aynı şekilde çok küçük dosya boyutları için performans kazancı daha az ve dosya boyutu arttıkça deşifreleme performansı artmaktadır. Bu artış yine belli bir değere kadar çıkıp daha sonra sabit kalmaktadır. Deşifrelemede çok küçük dosyalar için performanstan kazanç 38% iken çok büyük dosyalar için bu kazanç maksimum 45%'e kadar çıkmıştır (Nagendra and Sekhar, 2014).

Konuyla ilgili yapılmış başka bir çalışma ise DES standardının paralelleştirilmesini ele almaktadır. Bu çalışmada algoritmayı paralelleştirmek için mevcut DES programında var olan döngülerin veri bağımlılığı analizleri yapılmıştır.Veri bağımlılıklarını ortaya çıkarmak için Petit yazılımı ve paralel algoritmayı sunmak için yine OpenMP standardı kullanılmıştır. Veri bağımlılıkları analiz edildikten sonra pek çok "for" döngüsünün paralelleştirmeye uygun olduğu görülmüştür. Bu döngülere Şekil 3.13'den görüldüğü gibi OpenMP direktifleri eklenmiştir. Klasik DES algoritması, paralelleşebilen ve paralelleşemeyen parçalara

ayrılmıştır. İki işlemcili makinede algoritmanın paralel parçalarıyla yapılan deneyler seri uygulamaya göre 1.95 kat daha hızlı çalışmıştır (Beletskyy and Burak, 2003).

```
#pragma omp parallel private (i,m)
#pragma omp for
 for ( i = 0; i < 16; i++) {
      if(edf == DE1 ) m = (15 - i) << 1;
      else m = i << 1;
      kn[m] = kn[m + 1] = 0L;
       #pragma omp parallel private(j)
       #pragma omp for
        for( j = 0; j < 28; j++ ) {
             if( (j + totrot[i]) < 28 ) pcr[j] = pc1m[j + totrot[i]];
             else pcr[j] = pc1m[(j + totrot[i]) - 28];
                  }
       #pragma omp parallel private(j)
       #pragma omp for
       for ( j = 28; j < 56; j++) {
             if( (j + totrot[i]) < 56 ) pcr[j] = pc1m[j + totrot[i]];
             else pcr[j] = pc1m[(j + totrot[i]) - 28];
             }
       #pragma omp parallel private( j )
       #pragma omp parallel for reduction( | : kn)
        for ( j = 0; j < 24; j++ ) {
             if( pcr[pc2[j]] ) kn[m] = kn[m] | bigbyte[j];
             if( pcr[pc2[j+24]] ) kn[m + 1] = kn[m + 1] | bigbyte[j];
             }
```

Şekil 3.13. Paralelleşebilen Döngülere Direktifler Ekleme (Beletskyy and Burak, 2003).

Yukarıda bahsedilen çalışmalardan farklı olarak bu çalışmada gerçekleştirilen güvenlik algoritmasının paralel uygulaması kriptografik MIRACL kütüphanesinin rutinlerini kullanmıştır ve deneyler 2 çekirdekli ve 4 çekirdekli donanım olmak üzere farklı iki platformda yapılmıştır. Farklı anahtar uzunluklarının performans analizine ek olarak, ideal çok çekirdekli işlemci tasarımı (çekirdek sayısı ve hyperthreading) konusu da incelenmiştir. Buna ek olarak, çalışmada yer alan güvenlik algoritması asimetrik anahtar algoritması ve simetrik anahtar algoritmasının bir arada kullanılmasından oluşmaktadır. Bu tür algoritmalara *hibrit güvenlik algoritmaları* denir. Bir sonraki bölümde bu algoritmalar tanıtılacaktır.

3.3 Hibrit Algoritmalar ve Çalışmada Kullanılan Güvenlik Algoritması

Bir bilgisayar ağı, ayrı hesaplama düğümlerinin birbirine bağlanmasıyla oluşturulmuş bir ortamdır. Bu ağda düğümler, protokol adı verilen düzgün tanımlanmış kurallar ve sözleşmeler bütününü kullanmaktadır. Başka bir deyişle, biri diğeriyle etkileşim içinde bulunmak istediğinde veya kontrol edilebilir bir şekilde kaynak paylaşımı yapılacağı zaman protokoller devreye girmektedir. Haberleşmenin, bugünün iş dünyası üzerinde büyük bir etkisi vardır. Veriyi daha güvenli bir şekilde iletmek artık bir ihtiyaç haline gelmiştir. İnternet teknolojisinin hızlı ilerlemesiyle internet saldırıları da çok yönlü olmaktadır. Bundan dolayı, geleneksel şifreleme algoritmaları (tekli veri şifreleme) bugünün bilgi güvenliğinde yetersiz kalmaktadır (Tianfu and Babu, 2012). Hibrit şifreleme ve deşifreleme yöntemleri yazılım ve

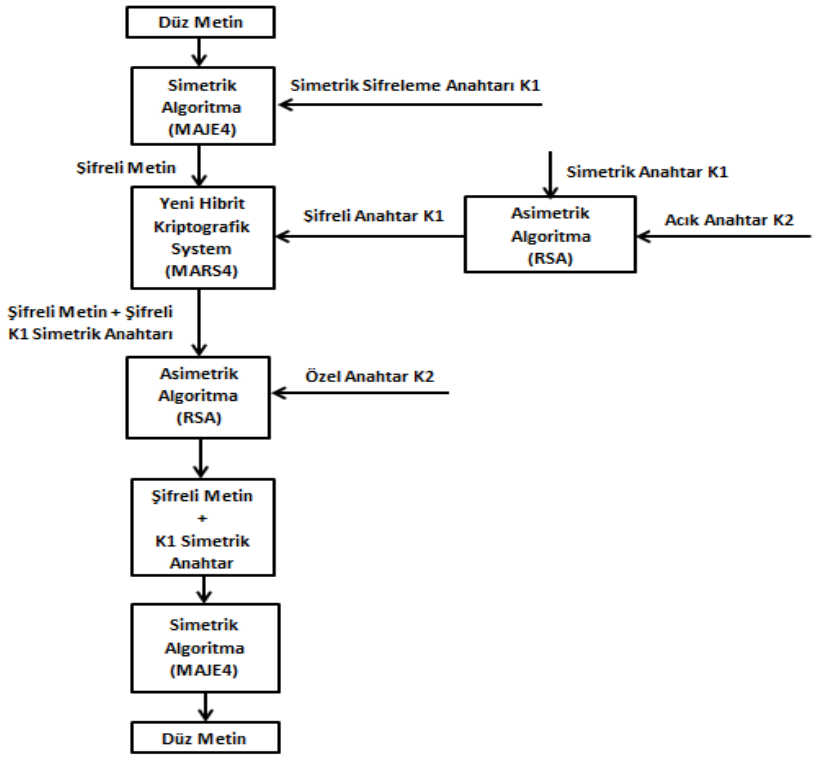
uygulamalarda yaygın olarak kullanılmaktadır. Mesajlar sadece simetrik ya da sadece asimetrik şifreleme kullanılarak gizli yapılabilir ama gizliliği geliştirmek ve daha güçlü yapmak için genelde karma sistemler önerilmektedir. Bu sistemlerden hibrit şifreleme yazılımlarına örnek verilecek olunursa GNU PRIVACY GUARD ve GPG PKI sıralanabilir. Mesajın güçlü derecede gizliliği, bilgi sızıntı riskini en aza indirmektedir (Irum et al.,Khan, Khıyal, 2011). Bizim çalışmamızda RSA algoritmasının AES güvenlik algoritmasıyla birlikte oluşturulan karma bir uygulaması geliştirilmiştir. Bunun nedeni ise sadece RSA ile çok büyük miktardaki verinin şifrelenememesi ve karma sistem ile güvenliğin en üst seviyede sağlanmaya çalışılmasıdır. RSA güvenlik algoritmasında şifrelenecek olan metin boyutunun önceden iki asal sayının çarpımı ile oluşturulmuş ortak anahtar olan "n" değerinden küçük olması gerekmektedir. Aksi takdirde algoritma düzgün çalışmamaktadır. RSA maksimum anahtar uzunluğu normalde kesin olarak belirtilmemiştir. Fakat anahtar uzunluğu arttıkça ve çok büyük bit değerlerine ulaştıkça günümüzde kullandığımız bilgisayarlarda algoritma çok yavaş çalışır hale gelmiştir. Çünkü anahtarları kullanarak bilgileri çözme işlemlerinde, işlemci zamanı çok fazla olmaktadır. Bu zaman ileti uzunluğu ile üssel olarak artmaktadır. Eğer yavaş çalışmasını önlemek amacıyla anahtar uzunluğu arttırılmaz ve sabit bırakılırsa bu sefer de seçtiğimiz veri boyutunda bir sınırlama olacaktır. Bununla beraber, verinin güvenliği de tehlikeye girecektir. Bu nedenle amacımız güvenliği sağlayarak çok büyük veriyle çalışma olduğundan sadece RSA algoritmasının kullanılması yetersiz kalmıştır. RSA algoritması ile birleştirilebilen AES algoritması ile bu sorun çözülmüştür. AES, çok uzun verilerin şifrelenmesi için oldukça elverişli bir uygulamadır.

Aynı zamanda AES ve RSA algoritmalarının verimlilikleri incelendiğinde çok geniş mesajları şifrelerken ve deşifrelerken AES'in RSA'den daha hızlı olduğu görülmektedir. RSA ise çok büyük sayıların çarpanlarına ayrılmasının zorluğuna dayandığından daha çok anahtar dağıtımı ve anahtar yönetimine uygun bir algoritmadır. RSA şifreleme anahtarını açık bir şekilde dağıtmakta ve deşifreleme anahtarını her zaman gizli tutmaktadır. AES yönteminde ise şifreleme anahtarı haberleşme öncesinde gizlice dağıtılmaktadır. Bunun yanı sıra AES algoritmasının özel anahtar kriptografiye ait DES, Üçlü DES, BlowFish algoritmalarına göre birkaç avantajı da bulunmaktadır. DES algoritması 1970'de tasarlanmış olup feistel ağını kullanan ve çok popüler olan bir uygulamadır. Buna rağmen bugünlerde birçok uygulama alanı için güvenli görülmemekte ve algoritmanın zayıf yönleri bulunmaktadır. Zayıflıkların ana nedeni ise algoritmada kullanılan anahtar boyutunun çok küçük bir değer olan 56 bit olmasıdır. DES'in bu zayıflığından yararlanarak yapılan birçok saldırı DES'i güvensiz bir blok şifrelemeye dönüştürmüştür. Üçlü DES algoritması, DES'i güçlendirme amaçlı

geliştirilmiştir. Özgün DES algoritması güvenliği arttırmak için üç kere uygulanmıştır (Rege et al.,Goenka, Bhutada, Mane, 2013).

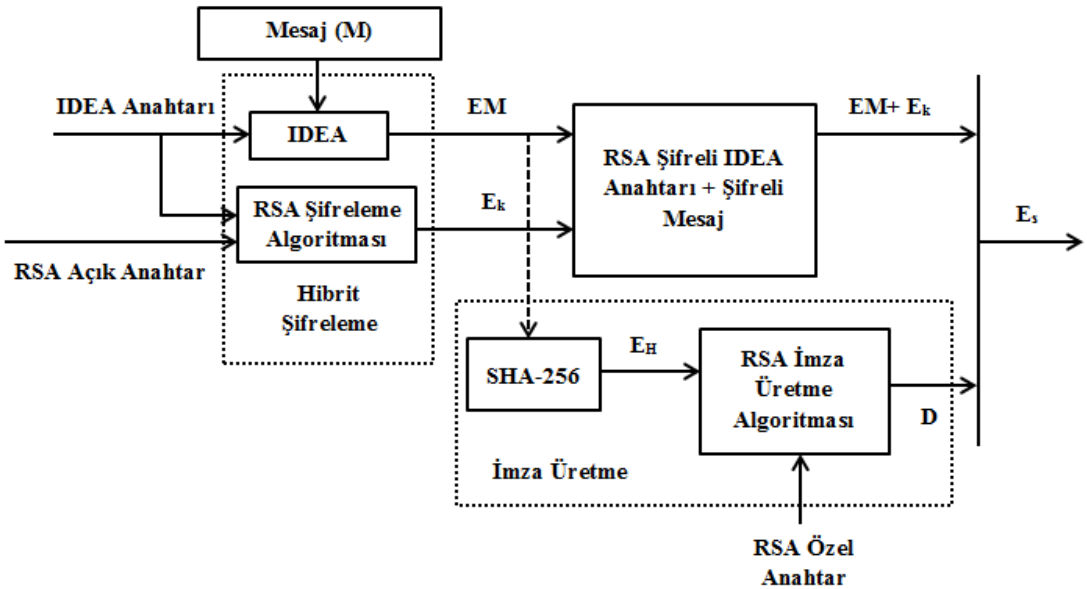
Tüm bu iyileştirmelere rağmen algoritmanın yavaş çalışması algoritmayı olumsuz yönde etkilemiştir. Genel ağda kullanılan Blowfish algoritması sadece yazılım uygulamaları için geliştirilmiştir. Yine zayıf anahtardan kaynaklanan problemler türemiştir. 128 bit anahtara sahip olan RC4 hızlı bir şifreleme algoritması olmasına rağmen genellikle pek çok saldırıya maruz kalmıştır. En fazla tercih edilen AES ise en iyi şifreleme standardı olarak bilinmektedir. Bugüne kadar AES'e karşı yapılması mümkün olan saldırı "brute force" olarak adlandırılmaktadır. Diğer algoritmaların zayıflıkları ele alındığında, AES diğer standartlara göre daha öncelikli kullanılmaktadır. Çalışmalar, işlem zamanı dikkate alındığında Blowfish algoritmasının en hızlı çalışan algoritma olmasına karşın güvenliğin sağlanmasında problemlerin ortaya çıktığını göstermiştir. Bu nedenle anahtar dağıtımı için RSA ile beraber kullanılan AES, iletilen verinin güvenliğini korumak amacıyla seçilmiş ve etkin bir yöntem önermektedir (Rege et al.,Goenka, Bhutada, Mane, 2013).

Geçmişte yapılmış çalışmalara bakıldığında, Khan ve Singh, hibrit şifreleme ve dijital imzayı birleştiren bir araştırma üzerine çalışmışlardır. Belgelerin güvenliğinde hibrit şifreleme ve kimlik denetiminde dijital imzayı kullanmışlardır. Hibrit şifreleme için IDEA ve RSA algoritmalarını kaynaştırmışlardır. Dijital imza için RSA dijital imza algoritmasını tercih etmişlerdir. Önerdikleri sistem 2.8 Mbps hızla çalışarak var olan uygulamalara göre daha hızlıdır. Yine aynı şekilde Mathew ve Jacob, kriptografik uygulamalar için hızlı bir teknik sunmuşlardır. Tasarlanıp geliştirilen çalışmada simetrik MAJE4 algoritması ve yine asimetrik RSA kullanılmıştır. Bu standartların her ikisinin de avantajlarından faydalanmak adına şifreleme yöntemlerini kombine ederek Şekil 3.14'de gösterilen MARS4 adında yeni bir hibrit sistem meydana getirmişlerdir. MAJE4 standardı dosyaların hem şifrelenmesi hem de deşifrelenmesi için kullanılmıştır çünkü çok daha hızlı ve bellekte RSA'e göre daha az yer işgal etmektedir. Yani daha düşük bellek gereksinimi sayesinde sınırlı belleğe sahip taşınabilir cihazlarda kullanılabilir. Ayrıca hızlı olduğundan Internet üzerinden gönderilen veri akışlarının şifrelenmesi ve çözülmesine ihtiyaç duyan uygulamalara için tercih edilmektedir. Mesajın kimlik doğrulaması RSA yardımıyla elde edilmektedir. Böylelikle, MARS4 gizliliğin yanı sıra kimlik denetimi koşulunu gerçekleştirerek göndericiden alıcıya mesajları transfer ettiği için çok sağlam bir teknik olduğunu kanıtlamıştır (Irum et al.,Khan, Khıyal, 2011). Şekil 3.15 ve Şekil 3.16'da birleşik hibrit imza şifreleme ve deşifreleme işlemleri gösterilmektedir.



Şekil 3.14. MAJE4 ve RSA Kullanılarak Tasarlanan Hibrit Kriptografik Sistem (Mathew and Jacob, 2006).

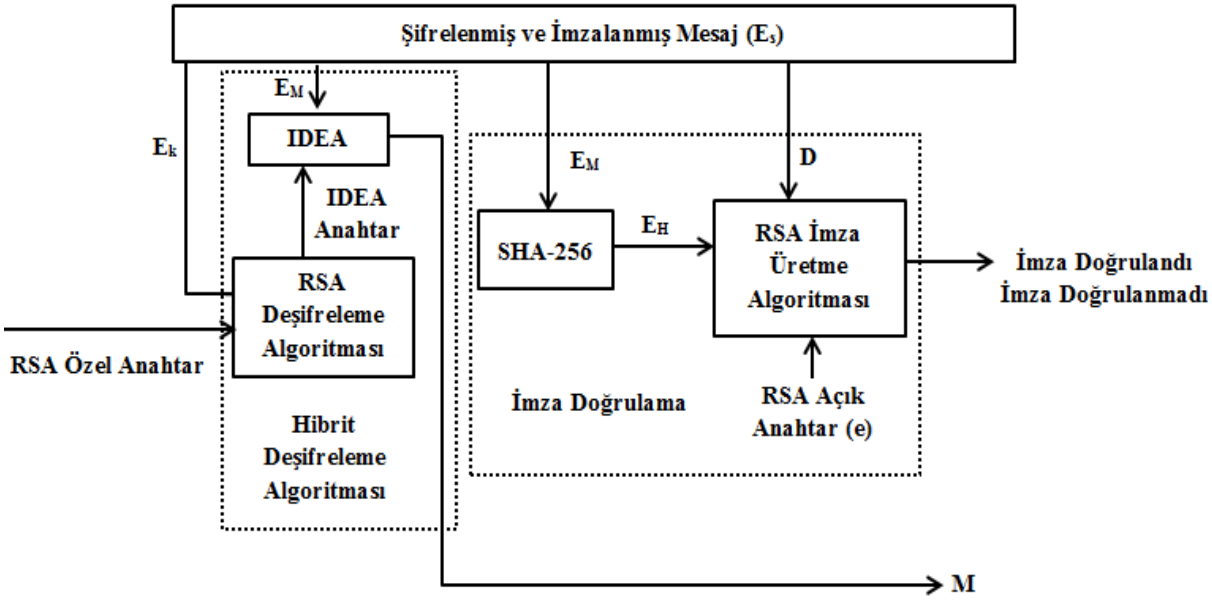
Şekil 3.15'de IDEA algoritmasının anahtarıyla hem orijinal mesaj IDEA algoritmasına göre hem de RSA açık anahtar RSA algoritmasına göre şifrelenmektedir. IDEA anahtarıyla şifrelenen metin ve RSA açık anahtar karşı tarafa yollanırken bu arada sayısal imzalı mesaj da alıcıya iletilmektedir.



Şekil 3.15. Birleştirilmiş Dijital İmza ve Hibrit Şifreleme Sistemi (Khan and Singh, 2005).

Şekil 3.16'da ise şifreli mesaj ve RSA açık anahtar çözülürken işlemlerin tersi gerçekleştirilmektedir. RSA açık anahtar çözülürken alıcının özel anahtarı kullanılmaktadır. Yine şifreli mesaja tekrar hashing işlemi ve sayısal imzaya alıcının açık anahtarı ile

doğrulama algoritması uygulanıp bu iki işlem sonucunda ortaya çıkan sonuçlar birbiriyle karşılaştırılmaktadır. Eğer sonuçlar birbirine eşitse mesaj doğru kişiden gelmiş demektir.



Şekil 3.16. Birleştirilmiş Dijital İmza ve Hibrit Deşifreleme Sistemi (Khan and Singh, 2005).

Çalışmada kullanılan hibrit algoritmanın güvenlik analizi yapılırsa, veri iletimi güvenliği AES ve RSA standartlarının her birinin güvenliğine bağlı olmaktadır. Uygulamanın çalışma verimi, AES'in hızına ve yüksek verimlilikteki şifreleme ve deşifreleme işlemlerine dayanmaktadır. Veri, AES'in özel tasarımı ve tüm anahtar uzunluklarının (128, 192, 256) gücü sayesinde güvenli kalmaktadır. Bilindiği üzere, RSA açık anahtar kriptografi için kullanılan tek popüler standarttır. Çok basamaklı sayıları makul bir zamanda çarpanlarına ayırmak zor olduğundan RSA güvenlik anlamında tatmin edici görülmektedir (Rege et al.,Goenka, Bhutada, Mane, 2013).

Ganesan ve Vivekanandan güvenlik sorunlarını aşmak için gizlilik, kimlik doğrulama üzerine bir araştırma yapmışlardır. Dijital zarf yöntemine bu sorunları en alt seviyeye indirmek için başvurulmuştur. Araştırma, javada dijital zarf geliştirerek simetrik ve asimetrik metodolojilerini harmanlamıştır. Kullanılan simetrik algoritma AES, asimetrik algoritma ise HECC'dir. MD5 özetleme fonksiyonu veri bütünlüğünü korumak için seçilmiştir. Algoritma test edilip sonuçlar incelendiğinde HECC bu çalışma için ECC ve RSA'den ziyade en iyi asimetrik anahtar tekniği olarak görülmüştür (Irum et al.,Khan, Khıyal, 2011).

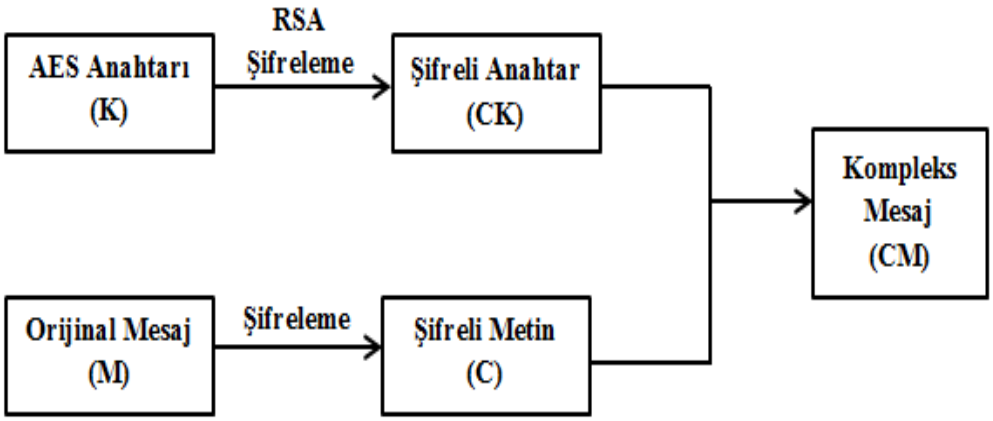
Başka bir çalışmada, RSA algoritması daha işlevsel ve AES şifreleme algoritması daha hızlı olduğu için bu iki algoritma birleştirilmiştir. Asimetrik kriptografi iletişimde görev alan tarafların kimlik doğrulamalarını yapmak ve simetrik kriptografi için kullanılan anahtarın şifrelenmesi için kullanılmaktadır. Büyük veri blokları yavaş çalışan RSA algoritmasından ziyade AES algoritması tarafından daha hızlı bir şekilde şifrelenmektedir. Ayrıca AES, diğer

simetrik algoritmalara göre güvenliği daha iyi bir derecede sağladığı ve daha az karmaşık bir yapıya sahip olduğu için bu dönemde var olan en güçlü ve verimli algoritma olarak gösterilmektedir. Bununla birlikte asimetrik kripto sistem olan RSA gizli anahtar dağıtma problemini çözmektedir. RSA algoritmasının önemli dezavantajı ise çok büyük uzunlukta anahtar kullanılmasıdır. AES geniş veri bloklarını şifreleyebildiğinden ve RSA anahtar yönetimi ve dijital imza uygulamaları için kullanıldığından bu iki algoritmanın hibrit olarak kullanılması en iyi çözümü vermektedir (Hasib and Haque, 2008).

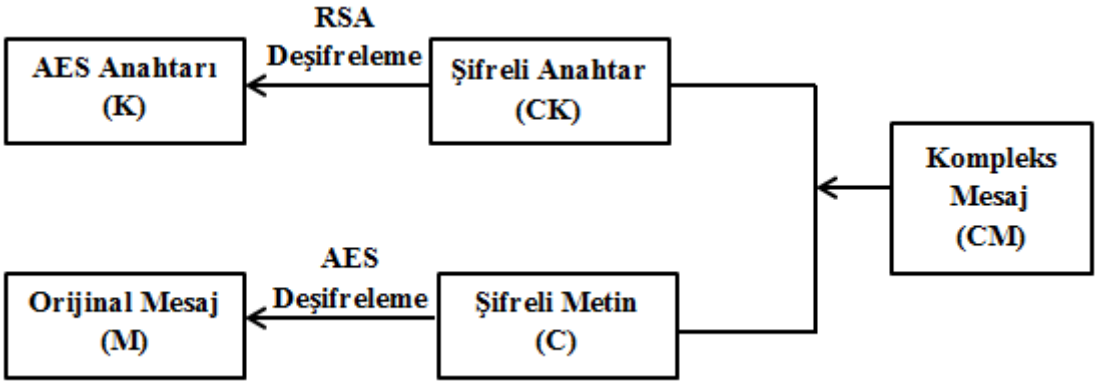
Çalışmamızda, RSA ve AES ile hibrit olarak yapılan şifreleme ve deşifreleme işlemleri aşağıdaki gibi gerçekleşmektedir. Karma sistemin çalışma mekanizmasına Şekil 3.17 ve Şekil 3.18'de yer verilmiştir.

1) 128, 192 ve 256 bitlik AES anahtarı öncelikle program tarafından MIRACL kütüphanesinin fonksiyonları kullanılarak rasgele bir biçimde üretilmektedir.
2) Aynı zamanda iki çok büyük, kendi aralarında asal olan ve p, q olarak adlandırılan değişkenler RSA genel anahtarı üretmek üzere 1024 bit seçilmişlerdir. Yine bu değişkenler programda rasgele üretilmektedirler. Bu nedenle RSA genel anahtar boyutu maksimum 2048 bite çıkmaktadır.
3) Daha sonra üretilen AES anahtarı ile 1 MB boyutuna sahip büyük veri dosyası şifrelenmektedir.
4) Veri şifrelendikten sonra 128, 192 ve 256 bitlik AES anahtarı, maksimum boyutu 2048 bitlik RSA genel anahtar ile şifrelenmektedir.
5) Gönderici, elindeki şifrelenmiş metni ve şifrelenmiş AES anahtarı alıcıya deşifreleme işlemleri için yollamaktadır.

Alıcı, öncelikle şifrelenmiş AES anahtarı RSA algoritmasının deşifreleme metodunu kullanarak açmaktadır. Burada deşifreleme için alıcı özel anahtarını kullanmaktadır. AES anahtarı ise yine MIRACL kütüphanesinin rutinleri kullanılarak Çin Kalan Teoremi ile bulunmaktadır. Alıcı, açılan AES anahtarı ile de şifrelenmiş metni AES algoritmasının deşifreleme metodunu kullanarak çözmektedir.



Şekil 3.17. Çalışmada Kullanılan Hibrit Şifreleme Süreci (Rege et al., Goenka, Bhutada, Mane, 2013).



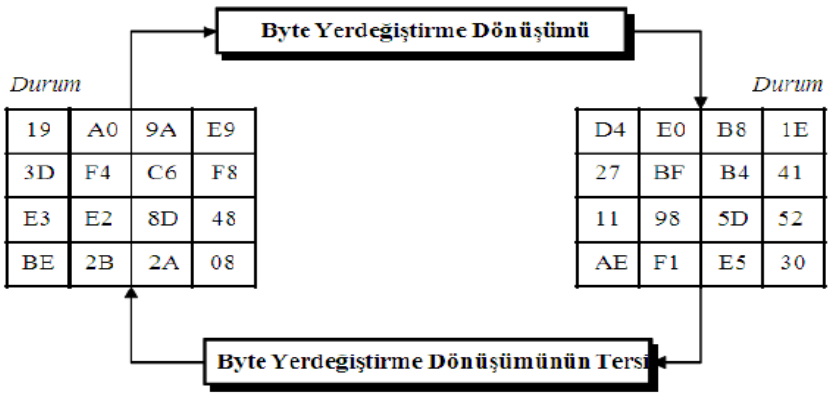
Şekil 3.18. Çalışmada Kullanılan Hibrit Deşifreleme Süreci. (Rege et al., Goenka, Bhutada, Mane, 2013).

Hibrit sistem kullanmanın başka nedenleri de vardır. Bu nedenler açık anahtar kriptografinin diğer dezavantajlarıyla açıklanabilmektedir. Öncelikle asimetrik algoritmalar düz metin saldırılarına karşı çok zayıftırlar. Şifreleme anahtarı açık bir şekilde yayınlandığı için saldırılar etkin bir şekilde yapılmaktadır Bir başka husus, asimetrik algoritmalara karşı yapılan algoritmik saldırılardır. Çok büyük n sayısının çarpanlarına ayrılmasının neredeyse imkânsız olduğu bilinmesine rağmen araştırmacılar özellikle teknolojinin gelişmesiyle ve bilgisayarların işlemci hızlarının artmasıyla bu saldırının mümkün olabileceğini göstermişlerdir. Ayrıca açık anahtar değerlerinin veri tabanlarında çok güvenli bir şekilde tutulması ve sadece bu değerlerin yetkili bir kişi tarafından değiştirilebilmesi, kötü niyetli üçüncü bir kişinin bu değerleri değiştirmesi riskini ortadan tamamen kaldırmamaktadır. Ortadaki Adam (Man in the Middle) saldırısı, açık anahtar kripto sistemlere karşı yapılan en ünlü saldırıdır. Bu sorunu çözmek için Interlock Protokolü öne sürülmüştür ancak bu protokolün etkinliği ve etkililiğine karşı hala büyük tartışmalar vardır. Sertifika Yetkilileri ortak anahtar dağıtımı için bir çözüm olarak kabul edilmekteydi. Fakat bu yöntem için hala sertifika, güven, otorite seçimi ve ne kadar bir süre için bu otoriterin güvenli olduğu gibi terimler kritik konuları oluşturmaktadır (Torkaman et al.,Kazazi, Rouddini, 2013).

Orijinal mesaj AES şifreleme anahtarı gizli kaldığı sürece güvencede olacaktır. Hatta hibrit algoritma kullanılarak iletilen veri izlenilse bile, üçüncü kişi AES şifreli anahtar ve şifreli metnin karmaşık mesajın hangi parçasında olduğunu anlayamaz. Üstelik alıcının özel anahtarı bu uygulamada bilinmediği için AES anahtarının şifreli hali ele geçirilse dahi çözülemeyeceğinden tüm metnin açılması da mümkün olmamaktadır. Böylece yüksek düzeyde bir güvenlik sağlanmış olup haberleşmeye aradaki üçüncü kişinin kulak misafiri olma riski minimuma inmiştir (Rege et al.,Goenka, Bhutada, Mane, 2013).

3.4 AES Algoritması Çalışma Mekanizması

AES algoritmasında tersi alınabilir döngüler kullanılmaktadır. Her döngüde şifreli metnin oluşumu, bayt değiştirme, satır kaydırma, sütun karıştırma, tur anahtarıyla toplama ve anahtar oluşumu gibi işlemler yürütülmektedir. Sadece son döngüde sütun karıştırma işlemi yapılmaz. Her döngüde ana anahtar kullanılarak oluşturulan farklı anahtarlarla şifreleme işlemi gerçekleştirilmektedir. Öncelikle şifreli metnin oluşturulabilmesi için orijinal metin 128 bitlik parçalara ayrılıp her bir parça 4x4'lük 16 bölmeden oluşan durum matrisine yerleştirilmektedir. En başta belirlenen 128 bitlik anahtar, durum matrisi olarak kabul edilip orijinal metnin yerleştirildiği durum matrisle toplandıktan sonra bahsedilen işlemler yapılmaya başlanılmaktadır. Şekil 3.19'da görüldüğü gibi bayt değiştirme işleminde durum matrisinde bulunan değerler yine Şekil 3.20'de gösterilen ve S-Tablosu adı verilen tablodaki değerler ile ayrı ayrı değiştirilir (Baydar, 2014).



Şekil 3.19. Byte Yerdeğiştirme İşlemi (Baydar, 2014).

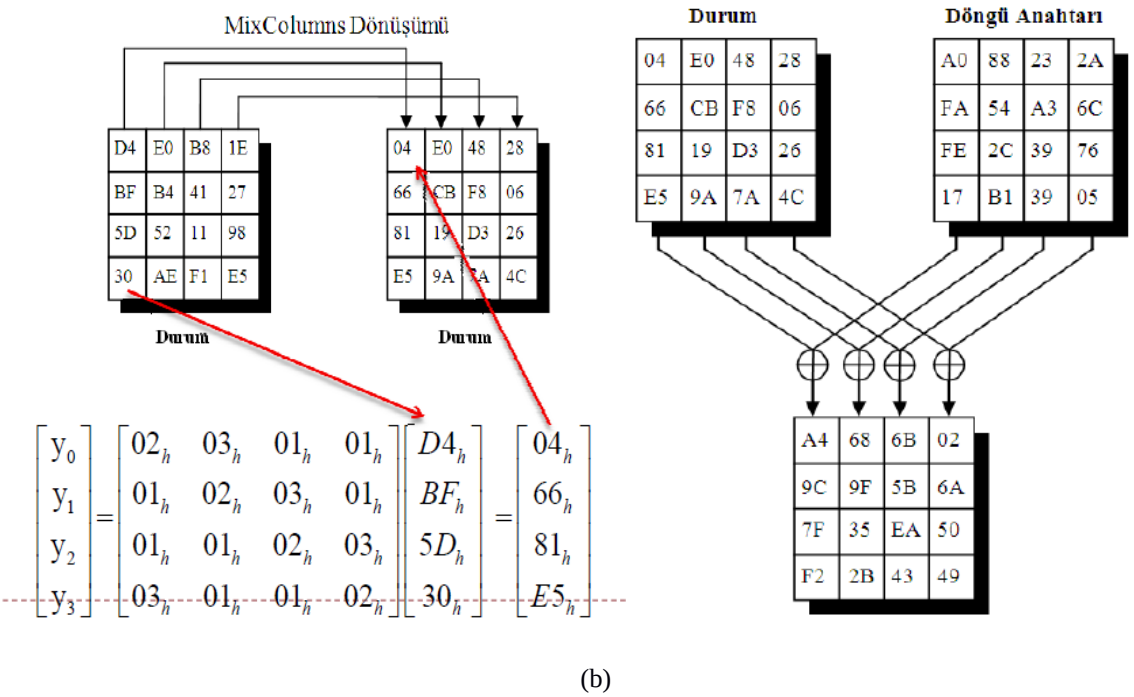
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Şekil 3.20. S-Tablosu (Baydar, 2014).

Bir sonraki aşama satır kaydırma işlemidir. Burada satırlar sıralı bir şekilde çevrimsel olarak kaydırılmaktadır. İlk satırda herhangi bir kaydırılma olmaz. İkinci satır sola 1, üçüncü satır sola 2 ve dördüncü satır sola 3 byte ötelenmektedir. Bölmeler matristen taştığı zaman kaydırmanın başına eklenmektedir. Bu işlem Şekil 3.21'den görülebilir (Baydar, 2014).

Şekil 3.21. Satır Öteleme Dönüşümü İşlemi (Baydar, 2014).

Daha sonra sütun karıştırma işlemine geçilir ve bu işlemde yeni sütun değerleri tek tek hesaplanarak elde edilmektedir. Şekil 3.22 (a)'dan görüldüğü üzere işlem yapılırken eski sütun elemanları kullanılmakta ve hesaplama çarpma ve toplama işlemlerinden oluşmaktadır (Baydar, 2014).



(b)

Şekil 3.22. a) Sütun Karıştırma İşlemi, b) Tur Anahtarıyla Toplama İşlemi (Baydar, 2014).

Son aşamalardan birisi de tur anahtarıyla toplama işlemidir. Her turda yapılan işlemler sonunda bir sonraki tur için yeni bir anahtar oluşturulmaktadır. Bu anahtar 128 bitlik sonuç durum matrisi ile yine 128 bitlik mevcut tura ait anahtarın toplama işlemine sokulmasıyla meydana gelmektedir. Toplama, bit bazında ve XOR işlemi ile yapılmaktadır. Bu operasyon Şekil 3.22 (b)'de gösterilmiştir. Anahtar oluşumu işlemi artık en son aşamadır. Yeni anahtar üretilebilmesi için önceki sütun ile dört önceki sütun toplanmaktadır (Baydar, 2014).

3.5 MIRACL Kriptografik Kütüphane

MIRACL, çok basamaklı tamsayı ve rasyonel aritmetiğine bağlı C programlama diliyle yazılmış bir kriptografik kütüphanedir. Shamus Yazılım'da çalışan mühendisler tarafından yazılmıştır. Bu kütüphane, seri yazılmıştır ve en yeni versiyonu dahil paralel bir teknik uygulanmamıştır. Fakat açık anahtar kriptografi standardı olan IEEE 1363 için çoklu iş parçacığını (multithreaded) destekleyen C++ sınıfına sahiptir (Ekici vd., Yücel, Kılınç, 2009). Bütün C ve C++ derleyicileri MIRACL kodlarını çalıştırabilir. MIRACL, kaynak kodunda derleme (assembly) kullandığı için farklı mimarilerde özel desteğe ihtiyaç duyar. Diğer birçok kriptografik kütüphane, bilgisayar kullanımına odaklıyken MIRACL aynı zamanda geliştiricilerin son derece kısıtlı ortamlar üzerine güvenliği kurmalarını sağlamaktadır. Bu sistemler gömülü sistemleri ve mobil uygulamaları içermektedir. MIRACL kütüphanesi çok basamaklı aritmetiğin tüm yönlerini kapsayacak şekilde 100'den fazla rutinden oluşmaktadır. Büyük tamsayılar için *big* ve büyük rasyonel sayılar için *flash* gibi iki yeni veri tipi tanımlanmıştır. Büyük tamsayı rutinleri Knuth algoritmalarına dayanmaktadır. Yuvarlatılmış

fraksiyonla çalışan Floating-slash aritmetiği D. Matula ve P. Kornerup tarafından önerilmiştir. Tüm rutinler standart ve taşınabilir kalarak hız ve verimlilik için iyice optimize edilmiştir. Belirli zamanlarda isteğe bağlı ve hızlı assembly dil alternatifleri için kritik rutinler ayrıca tanımlanmıştır. Bu tanımlanma özellikle 80x86 aralığındaki işlemciler için yapılmıştır (CertiVox, 2014).

3.6 OpenMP Kütüphanesi

Önceki bölümler, süreç ve iş parçacıkları yaratmayı ve bunlar arasında veri paylaşımını sağlayan düşük seviye yaklaşımları içermektedir. Neyse ki, pek çok yaklaşım geliştiricileri yüksek seviye uygulama tasarımlarına yöneltirken aynı zamanda iş parçacıklarını yönetme külfetinden de kurtarmaktadır. Bu yaklaşımlar veriyi derleyici ve çalışma zamanı (runtime) kütüphanelerine paylaştırmaktadır. İdeal durumda olması gereken, derleyicinin paralelliği destekleyen mekanizmaları sağlayarak kodun hem parçalarını tanımlaması hem de bu parçaların paralel çalışmasına izin vermesidir. Ancak günümüzdeki derleyici teknolojisi geliştiriciden yardım almadan uygulamadaki paralellikten nadiren yararlanabilmektedir. Bu durumu çözmek adına paralellik için en sık kullanılan dil uzantısı OpenMP API (Application Programming Interface)' dir. OpenMP geliştiricinin seri koduna direktifler eklemesine izin vererek derleyicinin aynı uygulamanın paralel şeklini üretmesini sağlamaktadır (Gove, 2011).

Paralelleştirmenin başka bir biçimi de otomatik paralelleştirmedir. Çoğu derleyici otomatik paralelleştirmeyi bir dereceye kadar gerçekleştirebilmektedir. Otomatik paralelleştirmeden istenilen sonuçlar alınsaydı, kendisine iyi bir derleyici optimizasyonu denilebilirdi. Fakat gerçeğe bakıldığında, elde edilen sonuçlar üzerinde önemli ölçüde sınırlamalar vardır. Bu sorun kuşkusuz zaman içinde gelişecek bir konudur fakat birçok durumda derleyicinin kodu paralelleştirmesi mümkündür. Oracle Solaris Studio ve Intel derleyicileri otomatik paralelleştirme işlemini yapabilmektedirler. Döngüler paralelleştirme için iyi bir hedeftir çünkü genellikle tekrarlanmakta olduğundan kod bloğunda büyük oranda zaman harcanmaktadır. Bundan dolayı bu derleyiciler sadece döngüleri otomatik olarak paralelleştirmektedir. Paralelleştirmenin getirdiği maliyetin üstesinden gelebilmek için paralel bölgenin büyük bir iş yapması gerekmektedir. Otomatik paralelleştirme ile kaynak kodda birtakım değişiklikler olmakta ve bu nedenle kodun anlaşılması zorlaşmaktadır. Alternatif olarak OpenMP, paralelliği ortaya çıkarmak için kodda en düşük seviyede değişiklik yapmaktadır. Birçok derleyici ile otomatik paralelleştirmeye ek olarak OpenMP kullanılabilir ve bu sayede daha fazla uygulama paralelleştirilebilir (Gove, 2011).

OpenMP, derleyici direktifleri, çalışma zamanı kütüphane rutinleri ve ortam değişkenleri olmak üzere üç ana bileşenden oluşmaktadır. Tüm derleyici direktifleri harfe duyarlıdır.

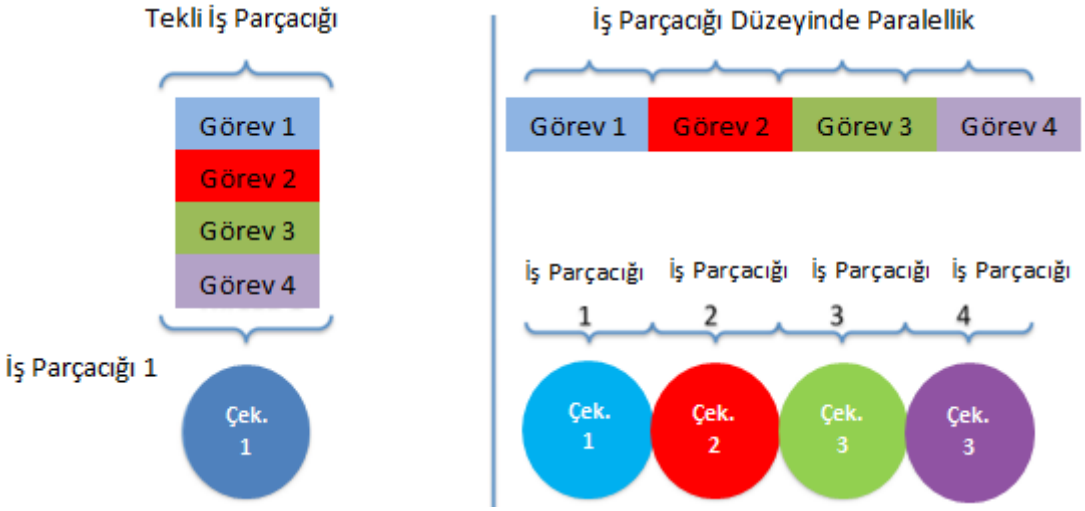
Paralel bölge, bölüm direktifi, tekli direktif, bariyer direktif ve ana direktif gibi yapılar bazı derleyici direktiflerine örnek olarak verilebilir. Aynı şekilde Omp_set_num_threads, Omp_get_num_threads, Omp_get_thread_num, Omp_get_num_procs veya Omp_get_max_threads gibi fonksiyonlar çalışma zamanı rutinleri olarak adlandırılmaktadır. OpenMP paralel kodun çalışmasını kontrol etmek amacıyla çeşitli ortam değişkenlerini desteklemektedir. OMP_NESTED, OMP_DYNAMIC, OMP_SCHEDULE, OMP_NUM_THREADS veya OMP_STACKSIZE gibi kullanımlar ortam değişkenlerine örnek olarak verilebilir (Nagendra and Sekhar, 2014).

3.7 OpenMP Paralelleştirme Şekilleri

3.7.1 Görev Düzeyinde Paralelleştirme

Paralelleştirmenin bir tipi de görev düzeyinde paralelleştirmedir. Görev paralelleştirme aynı zamanda fonksiyon paralelleştirme veya kontrol paralelleştirme olarak bilinmektedir. Görev düzeyinde paralelleştirme, merkezi işlem birimi için görevlendirilen birbirinden bağımsız iş parçacıklarının aynı anda çalıştırılmasını ele almaktadır. Görev düzeyinde paralelleştirme iş parçacıkları düzeyinde yapıldığından çok iş parçacıklı uygulamanın performansı donanım-spesifik, uygulama-spesifik gibi çeşitli faktörler tarafından etkilenmektedir. Her bir iş parçacığı bağımsız bir görev olarak düşünüldüğü için ki gerçekte programda veya işletim sisteminde bağımsız olmasa bile iş parçacığının kendine ait bir bellek yığıtı ve komutları bulunmaktadır. Görev düzeyinde paralelleştirme çoklu iş parçacığı tasarımına sahip işletim sistemleri ya da programlar tarafından kullanılabilir. Kavramsal olarak, görev düzeyi paralelleştirmenin performansı nasıl arttırdığını anlamak çok kolaydır. Her bir iş parçacığı görevler ile gerçek anlamda bağımsız çalışıyorsa, iş parçacıklarını işlemcideki çekirdekler arasında paylaştırmak çalışma zamanını azaltacaktır (Şekil 3.23). Performansı etkileyen diğer faktörler yük dengesi, bağımsız yürütmenin derecesi, iş parçacığı kilitleme mekanizmaları, zamanlama yöntemleri, gereken iş parçacığı belleği gibi konuları barındırmaktadır (Prinslow, 2011). İş parçacığı veya görev düzeyinde paralelleştirme derleyici ya da kullanıcı tarafından yaratılıp yine derleyici ya da donanım tarafından yönetilebilir. Uygulamada, haberleşme ve senkronizasyon yüklerinden ve algoritma karakteristiğinden kaynaklanan sınırlandırmalar söz konusudur (Nagarajan, 2014).

OpenMP, görev düzeyinde paralelleştirmeyi Şekil 3.24'deki gibi, birden fazla görevi çekirdekler arasında dağıtmayı sağlayan "pragma omp sections" yapısıyla gerçekleştirmektedir. Burada, her bir çekirdek koddan farklı bir parça çalıştırmaktadır (Texas Instruments, 2012).



Şekil 3.23. Görev Düzeyi Paralelleştirme (Prinslow, 2011).

```
#pragma omp parallel
#pragma omp sections
{
    #pragma omp section
        x_calculation();

    #pragma omp section
        y_calculation();

    #pragma omp section
        z_calculation();
}
```

Şekil 3.24. OpenMP ile Görev Düzeyi Paralelleştirme (Texas Instruments, 2012).

Ayrıca OpenMP iç içe geçmiş iş paylaşımı yapısını da desteklemektedir. Burada bir paralel bölge içinde başka bir paralel bölgeyle karşılaşılır. Bu yapı algoritmanın özyinelemeli olduğu durumlarda da yararlı olmaktadır. İç içe geçmiş paralelliğin kullanılabilmesi OpenMP'nin çevresel değişkenlerinden olan OMP_NESTED parametresinin "TRUE", ya da program içerisinde omp_set_nested rutininin parametresinin 0'dan farklı bir değer verilmesi gerekmektedir. Yine bir rutin olan omp_get_nested ile programın iç içe geçmiş paralelliği destekleyip desteklemediğini öğrenebiliriz. Şekil 3.25'den görüldüğü gibi birbirinden bağımsız paralel çalışan iki bölge "pragma omp section" direktifleriyle birbirinden ayrılmıştır. Hatta bu bölgelerin içinde de görev düzeyinde paralelleştirilmiş birer döngü bulunmaktadır. Bu kod parçacığı iç içe geçmiş iş paylaşımı yapısına örnek olarak verilebilir (Gove, 2011).

```
omp_set_nested( 1 );
#pragma omp parallel sections shared( array1, array2 )
{
  #pragma omp section
  {
    array1 = (double*)malloc( sizeof(double)*1024*1024 );
    #pragma omp parallel for shared(array1)
    for ( int i=0; i<1024*1024; i++ )
    {
      array1[i] = i;
    }
  }
  #pragma omp section
  {
    array2 = (double*)malloc( sizeof(double)*1024*512 );
    #pragma omp parallel for shared(array2)
    for ( int i=0; i<1024*512; i++ )
    {
      array2[i] = i;
    }
```

Şekil 3.25. OpenMP ile İç İçe Geçmiş İş Paylaşımı (Gove, 2011).

3.7.2 Veri Düzeyinde Paralelleştirme

Genellikle**,** büyük veri üzerinde koşan bir döngüyü paralelleştirme işlemidir. Veri düzeyi paralelleştirme, ortak verinin bellek tutarlılığı yoluyla çalışan süreçler arasında paylaştırılmasıdır. Bu sayede bellek yükleme ve belleğe ulaşma için gereken süre azaltılarak performans geliştirilmektedir. Çok çekirdekli bir merkezi işlem birimi açısından baktığımızda, ön bellekteki veri düzeyi paralelliğin çekirdekler tarafından paylaştırılması performans üzerinde önemli bir etkiye sahiptir (Prinslow, 2011).

Örnek olarak, tekli set komutlarını çalıştıran bir çoklu işlemci sisteminde, her bir işlemci asıl verinin dağıtılmış parçaları üzerinde aynı görevi uygulayabiliyorsa veri paralelliği sağlanabilir demektir. Örnek olarak; paralel bir platformda (CPU A ve CPU B) 2 işlemcili bir sisteme sahip olduğumuzu düşünelim. d verisi üzerinde bir işlem yapmak istiyoruz. Aynı anda CPU A'ya d verisinin bir kısmı üzerinde bir iş yaptırmak ve CPU B'ye kalan d verisi üzerinde yine aynı işi yaptırmak mümkündür (Wikipedia, 2013).

Çoklu çekirdekler üzerinde çalışan süreçler iş parçacığı olarak adlandırılmaktadır. Paylaşılan bellekten iş parçacıkları aynı veriyi okuduğunda performans kazanımlarının olması beklenmektedir. Bu senaryo verinin bir kopyasının birden çok iş parçacığı tarafından kullanılmasına izin vermektedir. Bu durum ise kopya sayısının ve çalışma zamanının azalmasını sağlamaktadır. İş parçacıkları ortak bir veri paylaşmadıklarında, her bir iş parçacığı kendine ait olan verinin kopyasını elinde tutmaktadır ve bu nedenle herhangi bir

kazanç elde edilemez. Bununla beraber, çoklu iş parçacığı talepleri belleğin bant genişliğini aşarsa bu durum belki de negatif performans etkileri doğuracaktır (Prinslow, 2011).

Bundan başka performans etkileri yazma işlemleri sırasında da oluşabilir. Birden fazla iş parçacığının aynı anda aynı bellek konumuna yazma girişiminde bulunması performansı olumsuz yönde etkilemektedir. Bilgisayar biliminde bu tür durumlarla en iyi şekilde başa çıkan mekanizmaya *spin locks* adı verilir (Prinslow, 2011). Spin lock, birden fazla komut gerektiren kritik bölgeler için gereklidir. Spin lock, o an müsait olmayan ve paylaşılan veriye ait kilidi elde etmeye çalışan bir iş parçacığını bekleterek o iş parçacığının söz konusu veriyi güncellemesini engellemektedir. İş parçacığı tekrarlamalı olarak kilidin boşta olup olmadığını kontrol etmektedir**.** İş parçacığı aktif olmasına rağmen faydalı bir iş yapmıyorsa bu kilidin kullanımı *busy waiting* adını almaktadır. Elde edilen kilit genelde bilinçli olarak serbest bırakılır. Eğer iş parçacığı gereksiz yere kilidi tutuyorsa otomatik olarak o iş parçacığı ya bloklanır ya da uykuya bırakılır. *Busy waiting*, işlemcinin devirlerini boşa harcasa bile iki durumda faydalıdır: yapılması gereken başka bir iş yoksa ya da kritik bölge küçükse yapılan bekleme bloklamanın maliyetinden daha düşük olacaktır (Anderson et al., 1990; Silberschatz and Galvin, 1994).

Performans etkileri, çatışmaların sıklığına dayanmaktadır. Genel olarak, iş parçacıklarının paylaşılan belleğin farklı alanlarına yazması ortaya çıkacak hataların olasılığını azaltmak için tercih edilen bir çözüm yoludur. NUMA, çekirdeklerin kullandığı veriyi bellekte fiziksel olarak çekirdeklerin daha yakınına koyarak bu çözümü desteklemektedir. Sınırlı ön bellek genişliği, sınırlı bant genişliği, çekirdekten ayrılmış ön belleğin gecikmesi gibi durumlar performans üzerinde olumsuz etkiye sahip olsa bile bazı durumlarda veri düzeyi paralellik performansı arttırabilir. Özetle, veri düzeyi paralellik ile ilgili gözlemler çok çekirdekli işlemcilerin performansını analiz etmede oldukça önemli bir yere sahiptir. Bunun nedeni, belleğin sınırlayıcı bir faktör olmasıdır (Prinslow, 2011).

Kısacası, paylaşılan belleğin avantajlarından bahsedilirse, bellek erişimi için kullanıcının global adres uzayını kullanması programlama bakımından daha faydalıdır. Ayrıca, veri paylaşımı işlemciler arasında daha hızlı yapılmaktadır. Dezavantajlarına bakılırsa, paylaşımlı belleği kullanan işlemci sayısının artması işlemci ile bellek arasında darboğaz oluşturmaktadır. Dahası, belleğin senkronizasyonu ve belleğe erişimin doğru yapılması kullanıcının denetimindedir (Dönmez, 2014). OpenMP ile veri düzeyinde paralelleştirme Şekil 3.26'daki gibi yapılmaktadır. "#pragma omp for" iş paylaşımı yapısı programcıların, for döngülerini çoklu iş parçacıklarına dağıtabilmelerini sağlamaktadır. Bu yapı birbirinden

bağımsız yinelemelere sahip “for” döngülerine uygulanmaktadır. Yinelemelerin çağırılma sıralarını değişebilir fakat sonuç değişmez (Texas Instruments, 2012).

```
void calc( double* array1, double * array2, int length )
{
  #pragma omp parallel for
  for ( int i=0; i<length; i++ )
  {
    array1[i] += array2[i];
  }
}
```

Şekil 3.26. OpenMP ile Veri Düzeyi Paralelleştirme-1.

Veri düzeyinde paralelleştirmeye Şekil 3.27’deki gibi bir örnek de verilebilir. Yine “#pragma omp for” iş paylaşımı yapısı programda girdi olarak kullanılan büyük boyuttaki veriyi çekirdekler arasında bölüştürerek veri düzey paralelleştirmeyi sağlamaktadır. Şekilden görüldüğü gibi 4 tane alt dizisi olan iki boyutlu bir dizi veri düzeyinde paralelleştirilerek her bir alt dizi üzerinde yapılan işlem farklı çekirdeklere dağıtılmaktadır.

```
#pragma omp parallel private(i)
 #pragma omp for
 for(i=0;i<4;i++)
 {
  calculate(array[i]);
 }
```

Şekil 3.27. OpenMP ile Veri Düzeyi Paralelleştirme-2.

OpenMP, veri düzeyinde paralelleştirilmiş bir fonksiyonun bir de kendi içinde görev veya veri düzeyinde paralelleştirilmesine ya da tersine izin verir. Bu şekilde iç içe bir paralelleştirme gerçekleştirilmektedir. Yine “calculate” fonksiyonu (Bkz. Şekil 3.27) kendi içinde Şekil 3.28’deki gibi veri düzeyinde paralelleştirildiğinde bu paralellik elde edilmektedir.

```
#pragma omp parallel private(i)
 #pragma omp for
for(i=0;i<4;i++)
{
 for(i=0;j<100000;i++)
 {
 result=result+array[i][j];
 }
}
```

Şekil 3.28. Fonksiyonu Veri Düzeyinde Paralelleştirme.

3.8 Çalışmada Kullanılan Programın Paralelleştirilmesi

Programın paralel parçalara ayrılmasında izlenilen adımlar aşağıda sıralanmıştır:

- ✓ Birbirinden bağımsız görevler tanımlanmış ve paylaşılan bellekteki senkronizasyon maliyetini minimum düzeyde tutmak amaçlanmıştır.
- ✓ Seri olarak yazılmış programın ne kadarlık bir kısmının paralelleştirileceğine karar verilmiştir.
- ✓ Seri kaynak kodda veri bağımlılığı analizi yapılmış ve daha sonra Par4All programı aracılığıyla çözümlenen veri bağımlılıkları kontrol edilmiştir.
- ✓ OpenMP geliştirme arayüzü ile paralelleştirilebilecek bölgelere direktifler eklenerek kriptografik algoritmanın paralel uygulaması yazılmıştır.
- ✓ Daha sonra yazılan paralel uygulama çeşitli deneylerle hem 2 çekirdekli hem de 4 çekirdekli platformda test edilip paralelliğin hangi bölgelerde iyi sonuçlar verdiği görülmüştür.
- ✓ Platformlara göre seri ve paralel performanslar birbiriyle karşılaştırılarak bir takım sonuçlara varılmıştır.

Bir sonraki bölümlerde veri bağımlılıklarının nasıl yapıldığına ve program parçalarının hangi paralellik türleriyle eş zamanlı olarak çalıştırıldığına dair bilgiler verilecektir.

3.8.1 Bağımlılıkların Bulunması

Programdaki bağımlılıkların bulunmasında Par4All programından yardım alınmıştır. Par4All, C ve Fortran seri programları için otomatik paralelleştirme yapan ve onları optimize eden bir derleyicidir. Ronan Keryell tarafından geliştirilmiştir. Bu kaynaktan kaynağa olan derleyicinin amacı mevcut uygulamaları çok çekirdekli sistemler, yüksek performanslı bilgisayarlar, grafik işleme birimleri ve bazı paralel gömülü heterojen sistemler gibi çeşitli donanımlara adapte etmektir. Yeni OpenMP, CUDA, OpenCL kaynak kodları yaratıp uygulamanın orijinal kaynak kodunun değişmeden kalmasını sağlayarak iyi biçimlenmiş programlar oluşturmaktadır (Par4All, 2014). Yani özetle, Par4All yazılımı sadece bağımlılık analizi yaparak paralelleştirme patternlerini saptamakta ve bu noktalara uygun direktifler yerleştirmektedir. Ancak bazı durumlarda bunu yaparken kodun bazı kısımlarını değiştirerek paralelleşmeye elverişli hale getirir. Ancak, her paralelleşebilir bloğun paralelleştirilmesi uygun değildir. Çok kısa sürede işletilen kod parçaları paralelleştirildiğinde sistem yükü kazançtan çok daha yüksek olabilir ve hızlanma yerine yavaşlama söz konusu olabilir. Bununla birlikte Par4All, çeşitli açık kaynak geliştirmelerini birleştiren bir açık kaynak projedir. Kaynaktan kaynağa olan yaklaşımların ana faydaları aşağıdaki şekilde sıralanabilir:

- ✓ Açık bir yazılım olarak, kullanıcı kaynak kodu değişikliklerden ücretsiz tutulmaktadır.
- ✓ Çeşitli donanım platformları için kaynakların paralel versiyonlarının üretilmesini ve satıcının optimize edilmiş araçlarına güvenmeyi sağlamaktadır.
- ✓ Üretilen kaynak kod manuel şekilde optimize edilebilmektedir (Par4All, 2014).

MIRACL kriptografik kütüphanesinde bulunan ve AES algoritmasının şifreleme ve deşifreleme işlemlerini barındıran "mraes.c" isimli kaynak kod dosyasının veri bağımlılıkları önce manuel olarak incelenmiş daha sonra kaynak kod Par4All programından geçirilerek daha sağlıklı bir duruma getirilmiştir. Şekil 3.29'da Par4All ile değişkenler arasında gözlemlenen veri bağımlıkları ve Şekil 3.30 ve Şekil 3.31'de kaynak kodun Par4All programından önceki ve sonraki hali gösterilmiştir.

```
toshiba@ubuntu:~/Desktop/par4all/examples/TEST$ cd MIRACL
toshiba@ubuntu:~/Desktop/par4all/examples/TEST/MIRACL$ p4a --openmp mraes.c
p4a: PIPS: user warning in get_main_entity: no main found, returning aes_decrypt
 instead
p4a: PIPS: user warning in unstructured_to_transformer:
p4a: PIPS: user warning in unstructured_to_flow_sensitive_postconditions:
p4a: PIPS: user warning in any_user_call_site_to_transformer: Signed/unsigned in
teger type conversion.
p4a: PIPS: user warning in any_expression_to_transformer: Implicit type coercion
 between int variable and overloaded expression may reduce semantic analysis acc
uracy.
p4a: PIPS: user warning in call_proper_reduction_p: The left hand side of assign
ment is not a reference, this is not handled and no reduction will be detected
p4a: PIPS: user warning in TestCoupleOfReferences: Dependence between differents
 variables: a and st
p4a: PIPS: user warning in TestCoupleOfReferences: Dependence between differents
 variables: buff and st
p4a: PIPS: user warning in TestCoupleOfReferences: Dependence between differents
 variables: st and a
```

Şekil 3.29. Par4All ile Veri Bağımlılıklarının Bulunması.

mraes.c ✖ mraes_par.c ✖

```
fell_off=0;
switch (a->mode)
{
case MR_ECB:
    aes_ecb_encrypt(a,(MR_BYTE *)buff);
    return 0;
case MR_CBC:
    for (j=0;j<4*NB;j++) buff[j]^=a->f[j];
    aes_ecb_encrypt(a,(MR_BYTE *)buff);
    for (j=0;j<4*NB;j++) a->f[j]=buff[j];
    return 0;

case MR_CFB1:
case MR_CFB2:
case MR_CFB4:
    bytes=a->mode-MR_CFB1+1;
    for (j=0;j<bytes;j++) fell_off=(fell_off<<8)|a->f[j];
    for (j=0;j<4*NB;j++) st[j]=a->f[j];
    for (j=bytes;j<4*NB;j++) a->f[j-bytes]=a->f[j];
```

Şekil 3.30. Par4All Uygulanmadan Önceki Kod Parçası.

mraes.c ✖ mraes_par.c ✖

```
   fell_off = 0;
   if (a->mode==0) goto _switch_1_case_0;
   if (a->mode==1) goto _switch_1_case_1;
   if (a->mode==2) goto _switch_1_case_2;
   if (a->mode==3) goto _switch_1_case_3;
   if (a->mode==5) goto _switch_1_case_5;
   if (a->mode==14) goto _switch_1_case_14;
   if (a->mode==15) goto _switch_1_case_15;
   if (a->mode==17) goto _switch_1_case_17;
   if (a->mode==21) goto _switch_1_case_21;
   if (a->mode==29) goto _switch_1_case_29;
   if (a->mode==10) goto _switch_1_case_10;
   if (a->mode==11) goto _switch_1_case_11;
   if (a->mode==13) {
   }
   else {
      goto _switch_1_default;
   }
_switch_1_case_13:   ;
   bytes = a->mode-10+1;
   for(j = 0; j <= bytes-1; j += 1)
      fell_off = fell_off<<8|(a->f)[j];
#pragma omp parallel for
   for(j = 0; j <= 15; j += 1)
      st[j] = (a->f)[j];
```

Şekil 3.31. Par4All Uygulanmadan Sonraki Kod Parçası.

Yukarıdaki şekillerden görüldüğü gibi Par4All veri bağımlılıklarını çözmek amacıyla, program kaynak kodunda yapısal olarak değişiklikler gerçekleştirerek kodu daha iyi paralel hale getirmektedir. Çalışmada kullanılan programda Switch-Case yapısı If-Goto yapısına dönüştürülmüştür.

3.8.2 Programın Paralel Uygulama için Parçalara Ayrılması

Programdaki veri bağımlılıkları çözümlendikten sonra bazı parçaları görev ve veri paralelleştirme ile yeniden düzenlenmiştir.

- ✓ RSA anahtarının üretilmesi ve AES anahtarının deşifrelenmesinde görev düzeyinde paralelleştirme kullanılmıştır. Görev düzeyi paralellik Şekil 3.32 ve Şekil 3.33'de gösterilmiştir.
- ✓ Büyük verinin şifrelenip deşifrelenmesinde veri düzeyinde paralelleştirme kullanılmıştır. Veri düzeyi paralellik Şekil 3.34 ve Şekil 3.35'den görülebilir.
- ✓ Ayrıca MIRACL kütüphanesine ait mraes.c kaynak kodun içindeki fonksiyonlar veri düzeyinde paralelleştirilmiştir (Şekil 3.36 ve Şekil 3.37).

```
for(i=0; xgcd(e,phi,d,d,t)!=1; i++)
{
   #pragma omp parallel sections
   {
    #pragma omp section
    {
     bigbits(1024,p);
     if (subdivisible(p,2))
     {
      incr(p,1,p);
     }
    for(j=0; !isprime(p); j++)
    {
     incr(p,2,p);
    }
   }//section end
```

Şekil 3.32. RSA Anahtarının Üretilmesinde Görev Seviyesi Paralellik.

```
#pragma omp parallel sections
{
 #pragma omp section
 {
  xgcd(p,q,inv,inv,inv);    /* 1/p mod q */
 }
 #pragma omp section
 {
  copy(d,dp);
 }
 #pragma omp section
 {
  copy(d,dq);
 }
}
```

Şekil 3.33. AES Anahtarının Deşifrelenmesinde Görev Seviyesi Paralellik.

```
#pragma omp parallel private(i)
#pragma omp for
for(i=0;i<16;i++)
{
  aes_encrypt(&a,buffer[i]);
}
```

Şekil 3.34. Verinin Şifrelenmesinde Veri Seviyesi Paralellik.

```
#pragma omp parallel private(i)
#pragma omp for
for(i=0;i<4;i++)
{
  aes_decrypt(&a,buffer[i]);
}
```

Şekil 3.35. Verinin Deşifrelenmesinde Veri Seviyesi Paralellik.

```
#pragma omp parallel for
   for(j = 0; j <= 15; j += 1)
      st[j] = (a->f)[j];
   for(j = bytes; j <= 15; j += 1)
      (a->f)[j-bytes] = (a->f)[j];
   aes_ecb_encrypt(a, (unsigned char *) st);
#pragma omp parallel for
   for(j = 0; j <= bytes-1; j += 1) {
      buff[j] ^= st[j];
      (a->f)[16-bytes+j] = buff[j];
   }
   return fell_off;
```

Şekil 3.36. AES Şifreleme Fonksiyonunda Veri Seviyesi Paralellik.

```
#pragma omp parallel for
   for(j = 0; j <= 15; j += 1) {
      buff[j] ^= st[j];
      st[j] = 0;
   }
   return 0;
_switch_1_case_0:    ;
   aes_ecb_decrypt(a, (unsigned char *) buff);
   return 0;
```

Şekil 3.37. AES Deşifreleme Fonksiyonunda Veri Seviyesi Paralellik.

3.9. Performans Analizi

Hız faktörü, hesaplamalı problemleri paralel bir şekilde çözmenin yararını ölçü bakımından bize gösterir. P tane işlemci üzerinde paralel programlamaya dayanan hız faktörü aşağıdaki oran ile ifade edilmektedir. T_s ifadesi, hesaplamanın tek bir işlemcide çalıştırıldığında elde edilen program süresini temsil etmektedir. T_p ifadesi ise, aynı programlamanın p tane işlemcide çalıştırılmasıyla elde edilen program süresini simgelemektedir.

$$\mathbf{S_p = T_s / T_p}$$

Sonuç ve Değerlendirme bölümünde öncelikle çalışmada kullanılan platformlar karşılaştırılacak ve paralel olarak yazılmış program ile seri olarak yazılmış program bu platformların özelliklerine göre performans ve süre bakımından karşılaştırılacaktır.

4. SONUÇLAR VE DEĞERLENDİRME

Çalışmada kullanılan iki platform Şekil 3.38'den görülmektedir. Bu iki platform genel özellikler yönünden karşılaştırılırsa M4 daha çok çekirdeğe, ön belleğe ve daha kompleks bir GPU'ya sahip olduğu için M2'ye göre çok daha karmaşıktır. Buna bağlı olarak kontrolör de donanımsal olarak daha kompleks bir yapıdadır. Ayrıca ana belleğe yüklü kontrolör sürücü yazılımı da çok daha karmaşıktır. 2 işlemci de 1600 MHz'de 2 kanallı bir RAM kullanmaktadır. Bu şekilde işlemcilerin dış dünya ile veri iletişim hızını aynı bellek kısıtı belirlemektedir. Bu kısıt 2 işlemcinin de hızını dizginlemekte ve M4'ü daha çok

yavaşlatmaktadır. Bunun nedeni ise M2'nin frekansının belleğin frekansına daha yakın olmasıdır. O yüzden M2 bellekle daha çok uyumludur.

M2'de 4736 KB (4.75 MB), M4'de 7427 KB (7.25 MB) büyüklüğünde önbellek bulunmaktadır. Hiç uygulama olmadığında ana bellekte 510-530 MB'lık bir kısmın kullanımda olduğu görülmektedir. Sadece OS'a ayrılan bu alanın büyüklüğü önbellekten çok daha fazladır. Dolayısıyla önbellek OS verileri tarafından doldurulmaktadır. Uygulamalar çalışmaya başladığında bellek kullanımı 2-6 MB arası artmakta ve çalışırken muhakkak önbellek kayıpları (cache miss) olmaktadır. Kontrolör sürücü yazılımı hep kullanımda olduğu için onun da sık sık önbelleğe gelmesi gerekmektedir. Bu yüzden uygulama sırasında 2 CPU'da da ana bellek trafiği yoğun olmaktadır. M4'ün OS'a yüklü kontrolör sürücü yazılımı daha kompleks ve büyük olduğu için daha büyük önbellek ile rahatlatmaya çalışmışlar. Ancak gerek OS, gerek uygulama iş parçacıklarına ait verilerin gidip gelmesiyle önbellek kaybı yaşanmaktadır. Kontrolör ve ekran kartı sürücülerinin daha kompleks olması sebebiyle muhtemelen M4'de daha çok önbellek kaybı yaşanmaktadır.

M2	**M4**
ASUS K551L: • Intel Core i7 4500U cpu (4th G.) o 2/4 cores o 1.8 GHZ clock frequency o 1300 million transistors o Max. Power Cons: 15 Watt o Die size:118 mm2 o Ivy Bridge microarchitecture o GPU: Intel® HD Graphics 4400 (200 Mhz) ▪ Perf: 535 • Memory: o DDR3L 1600 MHz SDRAM, 4GB o 2 memory channels o Level 1 cache: 128 KB (per core) o Level 2 cache: 512 KB (per core) o Level 3 cache: 4096 KB (shared) • Graphics card: o NVIDIA® GeForce® GT 840M with 2GB DDR3 VRAM ▪ 1575 million transistors	ASUS N750JK • Intel Core i7 4700 HQ cpu (4th G.) o 4/8 cores o 2.4 GHz Clock frequency o 1400 million transistors o Max. Power. Cons.: 47 watt o Die size: 177 mm2 o Ivy Bridge microarchitecture o GPU: Intel® HD Graphics 4600 (400 MHz) ▪ Perf: 730 • Memory: o DDR3L 1600 MHz SDRAM, 2 GB o 2 memory channels o Level 1 cache: 256 KB (per core) o Level 2 cache: 1024 KB (per core) o Level 3 cache: 6144 KB (shared) • Graphics card: o NVIDIA® GeForce® GTX850M bellekli 2GB DDR3 VRAM ▪ 1870 million transistors

Şekil 3.38. Platformların Karşılaştırılması.

Problemimizde MIRACL ile AES'in şifreleyeceği veri miktarı çok düştüğü ve anahtar 256 bit olduğu için bellek üzerinde çok büyük bir baskı mevcut değildir. Ancak RSA ile anahtar şifreleme sırasında büyük bir işlem yükü ortaya çıkmaktadır. İş yükü fazla olan bu senaryoda kontrolör performansı önem kazanmaktadır.

htop ile işlemci yüklerini incelediğimizde sistemde 70-80 arası OS süreci ve bunlara bağlı 65-75 iş parçacığı görülmektedir. Bizim uygulamalarımız, çok sayıda iş parçacığı üretilmesine rağmen süreç ve iş parçacığı sayısını 1 arttırmaktadır. Bunun sebebi OpenMP iş parçacıkları işlemci kontrolörü tarafından yaratılmaktadır. Yani OS iş parçacıkları yaratılmasında bir rol oynamamaktadır. Bu nedenle uygulamamız OS tarafından 1 görev ve 1 iş parçacığı olarak görülmektedir. OS iş dağıtıcısı, uygulamayı 1 bütün halinde işlemciye göndermekte ve işlemci kontrolörü uygulama yeni geldiğinde onu 1 bütün halinde 1 çekirdeğe atmaktadır. İşlemci kontrolörü uygulamada #omp …direktifi ile karşılaştıkça bir iş parçacığı yaratmakta fakat yaratılan iş parçacığı boş bir çekirdeğe atanmamaktadır. Yani birçok uygulama iş parçacığı 1 çekirdek üzerinde çalışırken ortamda boş çekirdek bulunmaktadır. Demek ki atama yapılırken çekirdek doluluğu pek kontrol edilmemekte ve kontrolör işlerini basitleştirmek için rasgele bir çekirdek seçmektedir. Kontrolör, yaratılan iş parçacıklarının çalışacağı çekirdeği belirlerken bir Rasgele Sayı Üreticisi kullanmaktadır. Çok kısa aralıklarla çok sayıda iş parçacığı yaratıldığı için bu RNG birçok RNG'ler gibi işlemcinin saatini (timer) kullanmakta ve zaman değerleri çok yakın olduğu için hep aynı çekirdek numarasını üretmektedir. Nadiren de olsa RNG fonksiyonu başka bir çekirdek numarası üretmekte ve o andan itibaren yaratılan tüm iş parçacıkları bu yeni çekirdeğe atanmaktadır. Çünkü RNG uzun bir süre aynı random sayıyı türetmektedir. Ayrıca, OS iş parçacıkları periyodiktir ve her bir periyotta az iş yapılmaktadır. Bazen aktif OS iş parçacığı bulunmamakta ve genelde çalışır durumda bulunun 65-75 OS iş parçacığının en fazla 2-3 tanesi aktif durumda olmaktadır. Daha az sıklıkla en fazla 5-6 tanesi aynı anda aktifleşebilmektedir.

Deneylerimizi Kubuntu 14.04 Linux işletim sistemi yüklü ve yukarıdaki özelliklere sahip iki dizüstü bilgisayarda gerçekleştirdik (Bkz. Şekil 3.38). Dış koşullardan etkilenme düzeyini minimuma indirgemek için kablosuz ağ arayüzünü devre dışı bıraktık ve bilgisayarın gücünü sürekli olarak elektrikten aldık. Her bir anahtar uzunluğunda (128, 192 ve 256) ve işletim modunda 100 deney yaptık. Her bir deneyde 1 MB büyüklüğünde bir dosyayı işlemden geçirdik. Çizelge 3.3, 3.4 ve 3.5'de 4 çekirdekli işlemci için yaptığımız deneyler sonucunda elde ettiğimiz sayısal performans sonuçları listelenmiştir. Bu sonuçlara göre aşağıdaki gözlemleri elde ettik:

Her 100 deney grubunda çok farklı sonuçlar aldık. Çizelge 3.3'de şifreleme ve açma için geçen toplam sürelerin ortalamaları ve standart sapmaları görülmektedir. Özellikle şifreleme işlemesinde bu değişkenlik çok yüksekti. Açma işlemi ise MIRACL kütüphanesinin üstün yetenekleri dolayısı ile çok daha kısa sürede ve daha kararlı yerine getiriliyordu.

Çizelge 3.3. Ortalama İşletim Süresi İstatistikleri.

Anahtar Uzunluğu	Çalışma Modu	Şifreleme		Açma	
		ort	*std*	*ort*	*std*
256	Seri işletim	1765 ms	1162 ms	56 ms	10 ms
	Veri paralelleştirme	1565 ms	1097 ms	55 ms	10 ms
	AES içi veri paralelliği	1867 ms	1040 ms	55 ms	9 ms
192	Seri işletim	1604 ms	1021 ms	55 ms	8 ms
	Veri paralelleştirme	1651 ms	1084 ms	55 ms	9 ms
	AES içi veri paralelliği	1720 ms	1168 ms	55 ms	7 ms
128	Seri işletim	2010 ms	1166 ms	55 ms	9 ms
	Veri paralelleştirme	1790 ms	1108 ms	55 ms	9 ms
	AES içi veri paralelliği	1689 ms	1169 ms	55 ms	8 ms

Süre ölçümlerinde gözlenen çeşitliliğin analizi için Kubuntu Linux'ün çekirdeklerin ve belleğin kullanımını gösteren *htop* yardımcı programını kullandık. *htop* arayüzünden işletim sistemi ve kullanıcı süreçlerinin tüm çekirdeklere karışık olarak dağıtıldığını gördük. Ayrıca, sistem yükü düşükken 8 mantıksal işlemcide boş çekirdekler varken bir çekirdeğe 4-5 iş parçacığının atandığı da dikkatimizi çekti. Kendi programlarımızı çalıştırdığımızda ise işlem yükü çok arttı ve işlemci tüm işletim sistemini ve uygulama iş parçacıklarını 1 mantıksal çekirdek üzerinde çalıştırmaya başladı. Böylece, kendi yazılımlarımızın sürelerinde ortaya çıkan büyük varyasyonun, onlarca iş parçacığının sadece tek bir çekirdeği zaman paylaşımlı olarak kullanmasından kaynaklandığını anladık.

Daha sonra anahtar ve veri işlemleri (şifreleme ve deşifreleme) için ayrı analizler yapmaya karar verdik. Şifreleme ve deşifreleme süreçlerinde geçen deneylerimiz, her bir anahtar uzunluğu için aşağıda listelenen 6 parçadan oluşmaktadır:

- ✓ Anahtar seri, veri seri, mraes dosyası seri
- ✓ Anahtar paralel, veri seri, mraes dosyası seri,

Yukarıda hangi konfigürasyon iyi sonuç verdiyse, aşağıdaki iki adım o konfigürasyon ile devam ettirilmiştir:

- ✓ Anahtar seri/paralel, veri seri, mraes dosyası seri
- ✓ Anahtar seri/paralel, veri paralel, mraes dosyası seri

Yukarıda hangi konfigürasyon iyi sonuç verdiyse, aşağıdaki iki adım o konfigürasyon ile devam ettirilmiştir:

- ✓ Anahtar seri/paralel, veri seri/paralel, mraes dosyası seri
- ✓ Anahtar seri/paralel, veri seri/paralel, mraes dosyası paralel

En iyi işletim sürelerini ölçebilmek için 100 deney sonucunu küçükten büyüğe sıraladık ve ilk 20 sonucun ortalamalarını aldık. Anahtar şifreleme/açma için yapılan deneylere ait

sonuçlar Çizelge 3.4'de verilmektedir. Anahtar şifreleme için tüm anahtar uzunluklarında seri işletim ve görev paralelliği sonuçları birbirine çok yakındı. Sadece 128 bitlik anahtar için görev paralelliği 4 ms daha az zaman alıyordu ki, bu da çok önemsiz bir farktır. Anahtar açma ise tüm anahtar uzunlukları ve işletim modlarında çok yakın zamanlarda gerçekleşti. Dolayısı ile anahtarla ilgili işlemlerde yazılımcıyı paralellik zahmetine sokmaya değecek bir sonuç olmadığına karar verdik.

Çizelge 3.4. Anahtar İşlemleri İstatistikleri.

Anahtar Uzunluğu	Çalışma Modu	Anahtar Şifreleme	Anahtar Açma
256	Seri işletim	332 ms	54 ms
	Görev paralellik	346 ms	54 ms
192	Seri işletim	337 ms	54 ms
	Görev paralellik	374 ms	54 ms
128	Seri işletim	458 ms	54 ms
	Görev paralellik	454 ms	54 ms

Çizelge 3.5. Veri İşlemleri İstatistikleri.

Anahtar Uzunluğu	Çalışma Modu	AES veri şifreleme	AES veri açma
256	Seri işletim	6860 ns	6540 ns
	Veri paralelleştirme	7283 ns	6681 ns
	AES içi veri paralelliği	7142 ns	6553 ns
192	Seri işletim	6284 ns	5875 ns
	Veri paralelleştirme	6105 ns	5862 ns
	AES içi veri paralelliği	6310 ns	5990 ns
128	Seri işletim	5785 ns	5478 ns
	Veri paralelleştirme	5862 ns	5376 ns
	AES içi veri paralelliği	6028 ns	5529 ns

Farklı anahtar uzunlukları için veri şifreleme/açma işlemlerinde ve işletim modlarında aldığımız sonuçları da Çizelge 3.5'de veriyoruz. Çizelge 3.4 ve 3.5'i incelediğimizde AES ile veri şifreleme/açma işlemlerinin RSA ile anahtar şifreleme/açma işlemlerine göre ortalama olarak 50-60 kat kısa sürdüğünü gördük. Dolayısı ile anahtar işlemlerinin yükü çok daha fazla idi ve Çizelge 3.4'deki verilere göre bu işlemlerin seri modda yapılması yeterli idi. Veri şifreleme açma işlemlerinde elde edilen sonuçlar da birbirine yakındı. Sadece 192 bitlik anahtar kullanıldığında şifreleme ve açma işlemlerinde veri paralelliği biraz öne çıktı ve 128 bitlik anahtar kullanıldığında olduğunda veri açmada veri paralelliği biraz daha iyi sonuç verdi. Hem farkların düşük olmasından dolayı, hem de veri işlemlerinin anahtar işlemlerine göre çok daha az yük içermesinden dolayı veri şifreleme/açma işlemlerinde de yazılımcıyı yüke sokmamak için seri işletimin uygun olduğuna karar verdik.

2 çekirdekli işlemci üzerinde yaptığımız deneyler sonucunda elde ettiğimiz veriler ise Çizelge 3.6 ve Çizelge 3.7'de gösterilmiştir. 2 çekirdekli işlemcide şifreleme ve açma için geçen toplam sürelerin ortalamaları ve standart sapmaları, 4 çekirdekli işlemciden elde ettiğimiz Çizelge 3.3'deki değerler ile hemen hemen aynı bulunmuştur. Anahtar şifreleme ve açma ile ilgili yapılan deneylere ait sonuçlar Çizelge 3.6'da verilmektedir. 256 ve 192 bitlik anahtarlar için şifrelemede seri işleme daha performanslı görülmüştür. 128 bitlik anahtarda ise görev paralelliği daha iyi sonuç vermiştir. Bunun sebebi, 128 bitlik anahtarda işlem yükünün daha az olmasından kaynaklanmaktadır. *htop*, her anahtar uzunluğunda bize seri moda geçildiğini göstermiştir. Demek ki işlemciye binen yük 128 bitlik anahtarda daha az ve daha az komut işlemcinin komut tamponunda beklemektedir. Komutlar 128 bitlik anahtarla daha hızlı işlendiği için daha çabuk tüketiliyor. Anahtar şifreleme sonuçları, 2 ve 4 çekirdekli işlemcilerde genelde birbirine yakın bulunmuş ve şifreleme açmaya göre çok daha uzun sürmüştür. AES anahtarının deşifrelenmesi ise yine her iki işlemcide çok kısa süren bir işlemdir.

Çizelge 3.6. 2 Çekirdekte Anahtar İşlemleri İstatistikleri.

Anahtar Uzunluğu	Çalışma Modu	Anahtar Şifreleme	Anahtar Açma
256	Seri işletim	329 ms	62 ms
	Görev paralellik	365 ms	62 ms
192	Seri işletim	361 ms	62 ms
	Görev paralellik	408 ms	62 ms
128	Seri işletim	361 ms	62 ms
	Görev paralellik	311 ms	62 ms

Çizelge 3.7. 2 Çekirdekte Veri İşlemleri İstatistikleri.

Anahtar Uzunluğu	Çalışma Modu	AES veri şifreleme	AES veri açma
256	Seri işletim	8755 ns	7680 ns
	Veri paralelleştirme	8947 ns	7692 ns
	AES içi veri paralelliği	8896 ns	7667 ns
192	Seri işletim	8193 ns	7117 ns
	Veri paralelleştirme	8409 ns	7155 ns
	AES içi veri paralelliği	8345 ns	7155 ns
128	Seri işletim	7616 ns	6643 ns
	Veri paralelleştirme	7731 ns	6656 ns
	AES içi veri paralelliği	7859 ns	6630 ns

Veri şifreleme ve açma nsn cinsinden msn'ye dönüştürüldüğünde tek haneli rakamlar elde edilmektedir. Çizelge 3.7'den görüldüğü üzere, paralel ve seri işletimler arasında çok büyük

değer farkı yoktur ve ağırlık anahtar işlemlerinde olduğu için veri işlemlerini paralelleştirmek gereksizdir. Yazılımcı yükü artmış olur. Sonuç olarak, 2 ve 4 çekirdek için sonuçlarda anahtarlı işlemler msn cinsinden, veri işlemleri ise nsn cinsinden gösterildiği için performans anahtar işlemlerine bağlı kalmaktadır.

Çekirdek sayıları bakımından karşılaştırma yapılırsa 2 çekirdekli işlemcide anahtar açma işlemi daha uzun sürmektedir. Yavaşlık oranı, 62 / 54 = 1.2 eşitliğinden de bulunabilir. Daha uzun süre alan anahtar şifreleme işlemine bakacak olursak anahtar 256 ve 128 bit olduğunda 4 çekirdekli makina daha yavaş çalışmıştır. Normalde teorik beklentilere göre 4 çekirdeklinin 2 katı hızla çalışması beklenir. Ancak çekirdeklerin OS iş parçacıklarının ve uygulama iş parçacıklarının zamanlamaları birlikte yapılmaktadır. Çizelge 3.3'de gösterilen düşük tahminlenebilirlik düzeyi de bu durumun normal olduğunu göstermektedir. Demek ki, çok sayıda çekirdek ile çalışmak, istenen performansı sağlayamamaktadır.

Bu deneylerle ilgili araştırmamızın başında paralel programlama ile yüksek miktarda veri şifreleme/açma işlemlerinde çok çekirdekli işlemcinin çekirdek sayısı ile doğru orantılı olacak şekilde yüksek performans alacağımızı umuyorduk. Ancak, htop yardımcı yazılımı çıktısının da gösterdiği gibi şu nedenden dolayı beklentilerimize uyumlu sonuç alamadık: Yukarıda platformlar karşılaştırılmasında anlatıldığı gibi işlemci kontrolcüsü sadece 1 çekirdeği kullanmaya başlıyor. Ayrıca, hiçbir uygulama programı çalışmıyorken işletim sistemi iş dağıtıcısı, çekirdeklerin hangilerinin boş olduğunu kontrol etmeden mevcut iş parçacıklarını rasgele seçtiği bir çekirdeğe yönlendiriyor. Rasgele sayı üreticisi zaman sayacını kullandığı için aynı sayıyı üretiyor.

Yukarıdaki deneyler farklı anahtar uzunluklarında ve her iki platformda gerçekleştirilerek sonuçlara ulaşılmıştır. Bu deneylerin amacı, farklı anahtar uzunluklarının performanslarını görmektir. Bundan sonra aşağıda anlatılacak olan deneylerin yapılma amacı ise çok çekirdekli işlemciler için ideal konfigürasyon kaç çekirdek olmalı ve hyperthreading olmalı mı sorularına cevap bulmaktır. Bu deneylerde anahtar uzunluğunu sadece 256 bit olarak ayarladık. Öncelikle 2 fiziksel çekirdekli platformda anahtar ve veriye ait şifreleme ve açma işlemlerini hyperthreading devredeyken (2 fiziksel + 2 mantıksal çekirdek), hyperthreading devrede değilken (sadece 2 fiziksel çekirdek) ve sadece 1 fiziksel çekirdek varken gerçekleştirdik. Bu deneylere ait sonuçlar Çizelge 3.8'den görülmektedir.

Çizelge 3.8. a) Anahtar Şifreleme ve Açma Sonuçları, b) Veri Şifreleme ve Açma Sonuçları

(a)

		Anahtar Şifreleme	Anahtar Deşifreleme
4 çekirdek (2 çekirdek + hyp.)	Seri	329 ms	62 ms
	Paralel	365 ms	62 ms
2 çekirdek	Seri	438 ms	61 ms
	Paralel	362 ms	62 ms
1 çekirdek	Seri	559 ms	65 ms

(b)

		Veri Şifreleme	Veri Deşifreleme
4 çekirdek (2 çekirdek + hyp.)	Seri	8755 ns	7680 ns
	Paralel	8947 ns	7692 ns
2 çekirdek	Seri	8640 ns	7642 ns
	Paralel	8512 ns	7654 ns
1 çekirdek	Seri	9075 ns	8166 ns

Hibrit olarak geliştirilen algoritmanın bellek ihtiyacı azdır. Şifrelemede ise RSA'nın iş yükü fazladır. AES'in iş yükü ve veri ihtiyacı MIRACL nedeniyle düşük olduğu için sadece RSA'yı dikkate alarak analiz yaptık. Sadece 1 fiziksel çekirdeğin kullanıldığı durumda OS ve uygulama iş parçacıkları aynı çekirdekte çalıştığı için en yüksek işletim süresi elde edilmiştir. Sadece 2 fiziksel çekirdeğin kullanıldığı durumda seride OS iş parçacıkları başka çekirdekte olduğu için süre düşmüştür. 2 çekirdek paralelde ise arada uygulama iş parçacıklarının yükü diğer çekirdekle paylaşıldığı için süre düşmüştür. 4 çekirdek olduğunda hyperthreading, işlemci için performans artışı sağlamıştır. Seri işletimde ise en düşük sonucu elde ettik. Sebebi genel olarak uygulama iş parçacıkları ve OS iş parçacıklarının ayrı çekirdeklerde işletilmesi ve içerik değiştirmelerin (context switch) buna bağlı olarak azalmasıdır. Deşifreleme ise MIRACL sayesinde RSA algoritmasının düşük deşifreleme iş yükü nedeniyle çok kısa sürmüştür ve her durumda eşittir.

SONUÇ: Bu platformda 2 çekirdek + hyperthreading ile en iyi süreyi elde ettik. Bu senaryoda da OS iş parçacıkları uygulama iş parçacıklarından ayrıldığı zaman daha kazançlı oluyoruz.

2 fiziksel çekirdekli platformdan sonra aynı deneylerimizi 4 fiziksel çekirdekli platformda da gerçekleştirdik. Yine öncelikle 4 fiziksel çekirdekli platformda anahtar ve veriye ait şifreleme ve açma işlemlerini hyperthreading devredeyken (4 fiziksel + 4 mantıksal çekirdek), hyperthreading devrede değilken (sadece 4 fiziksel çekirdek), çekirdek sayısını 2'ye ve daha sonra 1'e düşürerek gerçekleştirdik. Bu deneylere ait sonuçlar Çizelge 3.9'dan görülmektedir. Bu işlemcide frekans farklılığına ek olarak RSA iş yükü de çekirdeği zorlamıştır. Hibrit uygulama için M4'de ağırlıklı olarak kontrolör yükü vardır çünkü daha karmaşıktır. Seri işlemede 1 çekirdekten 2 çekirdeğe, 2 çekirdekten de 4 çekirdeğe çıkıldığında sürede artış var. Bunun sebebi çekirdek sayısı arttıkça kontrolör yükü de artmaktadır. Hyperthreading eklenince seri işletim süresinde çok büyük düşüş yaşanmış ve en iyi performans elde edilmiştir. Bunun

tek bir sebebi olabilir: sürücü kodunun ve kontrolörün donanımsal olarak 8 çekirdeğe göre yazılmış olmalarıdır. Paralel performansta ise 2 ve 4 çekirdekte yakın sonuçlar alınmıştır. Bu durumda 2 ve 4 çekirdek eşdeğerdir diyebiliriz. Kontrolör donanım ve yazılımın maksimum konfigürasyona göre tasarlanması ve hyperthreading eklenmesi 8 çekirdekte süreleri daha da düşürmüştür.

SONUÇ: 2 durumda da 2 çekirdek ve 4 çekirdek eşdeğer performans vermiştir. 2 çekirdeği tercih etmek daha uygundur. Hyperthreading de eklenince en iyi performans elde edilmiştir. Dolayısı ile çekirdek sayısı 2'ye düşürülürse, frekans uyumlu hale gelirse ve kontrolör donanım ve yazılımı esnekleştirilirse çok büyük bir iyileştirme olacaktır.

Çizelge 3.9. a) 4 Çekirdekte Anahtar Şifreleme ve Açma Sonuçları, b) 4 Çekirdekte Veri Şifreleme ve Açma Sonuçları

(a)

		Anahtar Şifreleme	Anahtar Deşifreleme
8 çekirdek (4 çekirdek + hyp.)	Seri	332 ms	54 ms
	Paralel	346 ms	54 ms
4 çekirdek	Seri	596 ms	58 ms
	Paralel	360 ms	58 ms
2 çekirdek	Seri	425 ms	58 ms
	Paralel	367 ms	58 ms
1 çekirdek	Seri	364 ms	58 ms

(b)

		Veri Şifreleme	Veri Deşifreleme
8 çekirdek (4 çekirdek + hyp.)	Seri	6860 ns	6540 ns
	Paralel	7283 ns	6681 ns
4 çekirdek	Seri	7258 ns	6912 ns
	Paralel	7053 ns	6822 ns
2 çekirdek	Seri	7053 ns	6835 ns
	Paralel	7155 ns	6848 ns
1 çekirdek	Seri	7219 ns	6835 ns

5. ÖNERİLER

Çok çekirdekli işlemcilerden beklenilen performans artışının elde edilebilmesi için şu çözümlerin uygun olduğu görüşündeyiz:

1. İşlemci ve bellek frekansları yakın olmalıdır.
2. Donanımsal yükün hafifletilmesi için çekirdek sayısı 2'ye indirilmelidir: Tüm Intel işlemcilerde 2 kanallı bellek kullanılıyor. 4 çekirdek olduğunda yine 2 kanal kullanılması tıkanıklığına neden oluyor ve kazanç elde edilmiyor. Bu şekilde kontrolörün donanımsal tasarımı ve sürücü kodu basitleşir ve daha az yer kaplar. Sistem yükü çok hafifler.
3. İşletim sistemi yükünün düşük ve tahminlenebilirliğin yüksek olması için OS iş parçacıkları ve uygulama iş parçacıkları farklı çekirdeklerde çalışmalıdır.
4. Hyperthreading işlemci içi performansı arttırmıştır. Ancak kontrolcü yükünü arttırdığı da bir gerçektir. Biri HT desteği olan diğer HT desteği olmayan eşdeğer 2 işlemci üretilip performansları karşılaştırılırsa HT'nin gerçekten gerekli olup olmadığı anlaşılabilir.

KAYNAKLAR DİZİNİ

Akgün, D., Şahin, İ., Yücedağ, İ. ve Bayıroğlu, H., 2012, Paralel Matris Çarpma Algoritmasının Çok Çekirdekli Bilgisayar Üzerinde Java İş Parçacıkları ile Başarım Analizi, e-Journal of New World Sciences Academy, 12 (7): 718s.

Anderson, R.,2008, Security Engineering, John Wiley & Sons, United States of America, 1040p.

Anderson, T. E.,1990, The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors, 11 (1): 6 pp.

Baydar, V.,2014**,** "AES (Advanced Encryption Standart)",**http://slideplayer.biz.tr/slide/1964762/** (Erişim tarihi: 08 Ağustos 2014)

Beletskyy, V., Burak, D., 2003,Parallelization of the Data Encryption Standard (DES) algorithm, 15: 1 pp.

CertiVox, 2014,"MIRACL Crypto SDK", **http://www.certivox.com/miracl/miracl-download/** (Erişim tarihi: 11 Ağustos 2014)

Cleveland Institute of Electronics, 2011**, "**Microprocessor/Microcontroller**",** **http://cie-wc.edu/Microprocessor-7-19-2011.pdf** (Erişim tarihi:22 Temmuz 2014)

Dönmez, F.,2014**, "**Paralel Programlama**",** **http://www.fahridonmez.com/wp-content/uploads/bilgisayar_programlama_2.pdf** (Erişim tarihi: 15Ağustos 2014)

Ekici, B., Yücel,Ç., Kılınç, G., 2009, Parallelization of Mathematical Functions of a Cryptographic and Multiprecision Library,BSc Thesis, İzmir Institute of Technology (unpublished).

Fadhil, M. H., Younis, M. I., 2014, Parallelizing RSA Algorithm on Multicore CPU and GPU, International Journal of Computer Applications, 8 (87): 15-21 pp.

Fedkiw, R.,2014**, "**Parallel Programming with Pthreads**",** **http://web.stanford.edu/class/cs248/pdf/class_04_pthreads.pdf** (Erişim tarihi:05 Ağustos 2014)

Gove, D., 2011, Multicore Application Programming, Pearson Prentice Hall, United States of America, 441p.

IBM, 2006**, "**Shared Memory Programming: pThreads and OpenMP**",** **http://www.spscicomp.org/ScicomP12/Presentations/IBM/Tutorial_5.OpenMP.pdf** (Erişim tarihi:04 Ağustos 2014)

KAYNAKLAR DİZİNİ (devam)

Hasib, A. A., Haque, A. A. M. M., 2008, A Comparative Study of the Performance and Security Issues of AES and RSA Cryptography,Third International Conference on Convergence and Hybrid Information Technology (ICCIT), 6: 509-510 pp.

Irum, M., Khan, A., Khıyal, M. S. H., 2011, Confidentiality of Messages in a Cardless Electronic Payment System, 4 (6): 29-30 pp.

Jose, JJR, Raj, EGP, 2012, A Survey on the Performance of Parallelized Symmetric Cryptografic Algorithms, International Journal of Research and Reviews in Computer Science (IJRRCS).

Kaminsky, A., 2010**,** "Parallel Crypto: Applications of Parallel Computing in Cryptography",**http://www.cs.rit.edu/~ark/parallelcrypto/pp10/parallelcryptopp10.pdf**

(Erişim Tarihi: 09 Ağustos 2014)

Khalifa, O., 2011, The Performance of Cryptographic Algorithms in the Age of Parallel Computing, MSc Thesis, Heriot Watt University, 94p (unpublished).

Khan, A. M., Singh, Y. P., 2005, On the security of Joint Signature and Hybrid Encryption, 4: 111 pp.

Korkusuz, E., 2010**, "**Cpu Nasıl Çalışır?**",** **http://www.nef.balikesir.edu.tr/~emin/donanim/CPU%20NASIL%20%c7ALI%deIR.pdf** (Erişim tarihi:02 Ağustos 2014)

Majo, Z., Gross, T. R., 2011, Memory System Performance in a NUMA Multicore Multiprocessor, 10: 602 pp.

Mathew, S., Jacob, P. K., 2006, A Novel Fast Hybrid Cryptographic System: MARS4, 5: 1 pp.

Nagarajan, V., 2014**,** "Types of Parallelism",**http://www.inf.ed.ac.uk/teaching/courses/pa/Notes/lecture02-types.pdf**

(Erişim tarihi: 12 Ağustos 2014)

Nagendra, M., Sekhar, M.C., 2014,Performance Improvement of Advanced Encryption Algorithm using Parallel Computation,International Journal of Software Engineering and Its Applications, 10 (8): 287-294 pp.

Navalgund, S. S., Desai, A., Angalki, K. and Yamanur, H., 2013, Parallelization of AES Algorithm Using OpenMP, 4 (1): 144-146 pp.

Örnek, O., 2013**,** "Mikroişlemci Mimarileri",**http://akademikpersonel.kocaeli.edu.tr/oyaornek/ders/oyaornek25.02.2013_04.19.21ders.pdf** (Erişim tarihi: 03 Ağustos 2014)

KAYNAKLAR DİZİNİ (devam)

Par4All, 2014,"Par4All",**http://www.par4all.org/** (Erişim tarihi:15 Ağustos 2014)

Prinslow, G., 2011, Overview of Performance Measurement and Analytical Modeling Techniques for Multi-core Processors, 16: 2-6 pp.

Rege, K., Goenka, N., Bhutada, P., Mane, S., 2013, Bluetooth Communication using Hybrid Encryption Algorithm based on AES and RSA, 4 (87): 12-13 pp.

Saxena, S., Kishore, N., Handa, D. and Kapoor, B., 2013, Comparative Analysis of Sequential and Parallel Implementations of RSA, 4 (4): 2100-2102 pp.

Seidel, E. C., 2003, Preparing Tomorrow's Cryptography: Parallel Computation via Multiple Processors, Vector Processing, and Multi-Cored Chips, 154: 58-59 pp.

Schauer, B., 2008, Multicore Processors – A Necessity ProQuest Discovery Guides, 14: 1-5 pp.

Silberschatz, A., Galvin, P. B., 1994, Operating System Concepts, Addison-Wesley, United States of America, 780p.

Silberschatz A., Galvin, P. B.,Gagne, G., 2005, Operating System Concepts, John Wiley & Sons, United States of America, 886p.

Stallings, W., 2010, Computer Organization and Architecture Designing for Performance, Pearson Prentice Hall, United States of America, 774p.

Tan, X., Li, Y., 2012, Parallel Analysis of an Improved RSA Algorithm, 3: 144-146 pp.

Tanenbaum, A. and Austin, T., 2013, Structured Computer Organization, Pearson Prentice Hall, United States of America, 769p.

Texas Instruments, 2012**, "**Multicore Programming Guide**",** **http://www.ti.com/lit/an/sprab27b/sprab27b.pdf** (Erişim tarihi:11 Ağustos 2014)

Tektaş, M., 2014**,** "Şifreleme Algoritmalarının Sınıflandırılması ve Algoritmalara Saldırı Teknikleri"**, http://tektasi.net/wp-content/uploads/2014/01/sifreleme-algoritmalarinin-siniflandirilmasi.pdf** (Erişim tarihi:08 Ağustos 2014)

Tianfu, W., Babu, K. R., 2012, Design of a Hybrid Cryptographic Algorithm, 8 (2): 277 pp.

Torkaman, M. R. N., Kazazi, N. S., Rouddini, A., 2012,Innovative Approach to Improve Hybrid Cryptography by Using DNA Steganography, 12: 227 pp.

Tudor, M. B., Teo, Y. M. and See, S., 2011, Understanding Off-chip Memory Contention of Parallel Programs in Multicore Systems, 10: 602 pp.

Venu, B., 2011, Multi-core processors - An overview, 6: 1-5 pp.

KAYNAKLAR DİZİNİ (devam)

Vikipedi, 2013**,** "Moore Yasası",**http://tr.wikipedia.org/wiki/Moore_yasası** (Erişim tarihi: 30 Temmuz 2014)

Vikipedi, 2014**,** "SIMD",**http://en.wikipedia.org/wiki/SIMD** (Erişim tarihi: 03 Ağustos 2014)

Wikipedia, 2013**,** "Data Parallelism",**http://en.wikipedia.org/wiki/Data_parallelism** (Erişim tarihi: 14 Ağustos 2014)

Wikipedia, 2014**,** "Cryptography",**http://en.wikipedia.org/wiki/Cryptography** (Erişim tarihi: 07 Ağustos 2014)

Yerlikaya, T., Buluş, E., Buluş, N., 2006, Kripto Algoritmalarının Gelişimi ve Önemi, 5: 280 s.

Zhang, H., Zhang, D. and Bi, X., 2012, Comparison and Analysis of GPGPU and Parallel Computing on Multi-Core CPU, 3 (2): 185 pp.

ÖZGEÇMİŞ

Ad Soyad : Ecem İREN

Doğum tarihi : 15.07.1990

Doğum yeri : Trabzon (TR)

Adres :Emlak Bankası Konutları 668. Sok. Ata-2 Sitesi No:2 3.Etap Evleri K.5 D.12, Gaziemir/İzmir

Telefon :(+90) 554 545 14 72

E-mail : ecem.iren@gediz.edu.tr

Eğitim :

- 2008–2012 Yaşar Üniversitesi Bilgisayar Mühendisliği Bölümü 3,62/4
- 2004–2008 Konak Anadolu Lisesi 86,10/100

İş Tecrübeleri :

- 07/2012–10/2012 Yazılım Destek Merkezi Uzmanı (NETSİS)
- 11/2012-halen Araştırma Görevlisi (GEDİZ ÜNİVERSİTESİ)