T.C.
PAMUKKALE ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

BENZETİMLİ TAVLAMA YÖNTEMİNİN .NET FRAMEWORK İLE OPTİMAL PARALELLEŞTİRİLMESİ

YÜKSEK LİSANS TEZİ

KADİR YÜREKTÜRK

DENİZLİ, AĞUSTOS - 2019

T.C.
PAMUKKALE ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

BENZETİMLİ TAVLAMA YÖNTEMİNİN .NET FRAMEWORK İLE OPTİMAL PARALELLEŞTİRİLMESİ

YÜKSEK LİSANS TEZİ

KADİR YÜREKTÜRK

DENİZLİ, AĞUSTOS - 2019

KABUL VE ONAY SAYFASI

Kadir YÜREKTÜRK tarafından hazırlanan "BENZETİMLİ TAVLAMA YÖNTEMİNİN .NET FRAMEWORK İLE OPTİMAL PARALELLEŞTİRİLMESİ" adlı tez çalışmasının savunma sınavı 20.08.2019 tarihinde yapılmış olup aşağıda verilen jüri tarafından oy birliği ile Pamukkale Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği Anabilim Dalı Yüksek Lisans Tezi olarak kabul edilmiştir.

Jüri Üyeleri	İmza
Danışman Prof. Dr. Kadir KAVAKLIOĞLU Pamukkale Üniversitesi	.....
Üye Prof. Dr. Sezai TOKAT Pamukkale Üniversitesi	.....
Üye Dr. Öğr. Üyesi Mahmut SİNECEN Adnan Menderes Üniversitesi	.....

Pamukkale Üniversitesi Fen Bilimleri Enstitüsü Yönetim Kurulu'nun 28/08/2019 tarih ve 34/21 ..... sayılı kararıyla onaylanmıştır.

.....

Prof. Dr. Uğur YÜCEL

Fen Bilimleri Enstitüsü Müdürü V.

Bu tezin tasarımı, hazırlanması, yürütülmesi, araştırmalarının yapılması ve bulgularının analizlerinde bilimsel etiğe ve akademik kurallara özenle riayet edildiğini; bu çalışmanın doğrudan birincil ürünü olmayan bulguların, verilerin ve materyallerin bilimsel etiğe uygun olarak kaynak gösterildiğini ve alıntı yapılan çalışmalara atfedildiğine beyan ederim.

KADİR YÜREKTÜRK

ÖZET

BENZETİMLİ TAVLAMA YÖNTEMİNİN .NET FRAMEWORK İLE OPTİMAL PARALELLEŞTİRİLMESİ

YÜKSEK LİSANS TEZİ

KADİR YÜREKTÜRK

PAMUKKALE ÜNİVERSİTESİ FEN BİLİMLERİ ENSTİTÜSÜ

BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

(TEZ DANIŞMANI: PROF. DR. KADİR KAVAKLIOĞLU)

DENİZLİ, AĞUSTOS - 2019

Bir doğrusal olmayan optimizasyon probleminin çözümünde evrensel minimumu bulmaya yönelik yöntemlerden biri olan Benzetimli Tavlama (SA), deneme yanılma yoluyla rastlantısal olarak en iyi sonuca ulaştıran bir optimizasyon algoritmasıdır. Bu yöntemde metallerin tavlanması yani metallerin ısıtılması ve kontrollü bir şekilde soğutularak metal atomlarının en iyi düzene geçmesi örnek alınmıştır.

Günümüzde bilgisayarlarımızdan mobil cihazlarımıza kadar çoğu işlemci içeren cihazda birden fazla işlemci veya aynı işlemci içinde birden fazla çekirdek kullanılmaktadır. Akademik hesaplamalarda uzun zamandır kullanılan paralelleştirme konusu, son kullanıcıya kadar yaygınlaşan çok işlemcili cihazların kullanılması ile günümüzde üzerinde daha durulması gereken bir konu olmuştur. SA algoritmasının optimal paralelizasyonunu yapılan bu çalışmada .NET Framework u kullanarak hem seri hem de paralel olarak ortaya çıkan süreler üzerinden değerlendirmeler yapılmıştır.

ANAHTAR KELİMELER: Benzetimli tavlama, Paralel Programlama, Optimizasyon, .NET Framework

ABSTRACT

OPTIMAL PARALLELIZATION OF SIMULATED ANNEALING ALGORITHM WITH .NET FRAMEWORK
MSC THESIS
KADİR YÜREKTÜRK
PAMUKKALE UNIVERSITY INSTITUTE OF SCIENCE
COMPUTER ENGİNEERİNG

(SUPERVISOR: PROF. DR. KADİR KAVAKLIOĞLU)

DENİZLİ, AUGUST 2019

Simulated Annealing, which is one of the methods to find the global minimum in the solution of a nonlinear optimization problem, is an optimization algorithm which leads us to a solution by trial and error. In this method, the annealing of the metals, that is, the heating of the metals and the controlled cooling of the metal atoms in order to get the best order.

Nowadays, more than one processor is used in multiple processors or same processors in our device which includes many processors from our computers to mobile devices. The use of multi-processors, which have become widespread for ordinary users, has also become widespread in the academic environment. In this study, which performed the optimal parallelization of SA algorithm, evaluations were made on both the serial and parallel time periods using .NET Framework.

KEYWORDS: Simulated Annealing, Parallel Programming, Optimization, .NET Framework

İÇİNDEKİLER

Sayfa

ŞEKİL LİSTESİ

Sayfa

TABLO LİSTESİ

Sayfa

KISALTMALAR LİSTESİ

GPU	:	Grafik İşlemci Birimi
LINQ	:	Dil Tümleşik Sorgu (Language-Integrated Query)
PLINQ	:	Paralel Dil Tümleşik Sorgu (Parallel Language-Integrated Query)
PSA	:	Paralel Benzetimli tavlama (Parallel Simulated Annealing)
SA	:	Benzetimli Tavlama (Simulated Annealing)
TPL	:	Görev Paralel Kütüphanesi (Task Parallel Library)

ÖNSÖZ

Lisans eğitimim, yüksek lisans eğitimim ve tez çalışmam boyunca her aşamada ilgi ve desteğiyle yardımcı olan tez danışmanım Prof. Dr. Kadir Kavaklıoğlu'na teşekkür ederim.

Tüm eğitim hayatım boyunca yanımda olan ve beni destekleyen aileme, her aldığım kararda yanımda olan eşime teşekkür ederim.

1. GİRİŞ

1.1 Problemin Tanımı

Bilgisayar bilimlerinin gelişmesiyle ihtiyaçlar ve bu ihtiyaçların sürekli olarak artmasıyla pek çok problemle karşılaşılmaktadır. Bilgisayar teknolojileri bu problemlerin sıkça görüldüğü bir sektördür. Hem kurumsal hem de akademik çalışmalarda hesaplanmak istenen veri boyutlarının artması performans problemini doğurmaktadır. Hesaplamalarda doğruluk yanında en hızlı şekilde en iyi sonuca ulaşmak öncelikler arasına girmiştir. Hız, doğru sonuca en kısa sürede ulaşma mecburiyetinde olan hava tahminleme, mühendislik hesaplamaları gibi alanlarda hayati bir öneme sahiptir (Güneş, 2011).

Bu çalışmada kullanılan Benzetimli Tavlama (SA), sezgisel bir algoritmadır. Sezgisel algoritmalar problemin en iyi çözümüne ya da en iyi çözümün yakın bir noktasına hızlı şekilde ulaşmayı sağlamaktadırlar. Çoğu zaman en iyi çözüme dahi ulaşsalar o çözümün en iyi çözüm olup olmadığı kanıtlanamadığı için bu algoritmalara sezgisel denilmektedir (Sonuç, 2017).

Bu çalışmada SA yönteminin hem seri olarak hem de paralel olarak işlemcilerin birer birer işleme dahil edilmesi ile çözüm sürelerindeki farklar ve bu farklar üzerinden bazı hesaplamalarla karşılaştırmalar incelenmiştir.

1.2 Problemin Önemi

Bilgisayar bilimlerinde giderek yaygınlaşan çok çekirdekli ve çok işlemcili mimarilerin yaygınlaşması sonucu programların aynı oranda paralel hale getirilmediği görülmektedir. Algoritmaların incelenmesi ve paralel yapılacak kısımların doğru bir şekilde yazılmasıyla problem çözümlerinde hız kazanıp zamandan tasarruf edebileceğini göstermek için bu tez çalışması hazırlanmıştır.

Paralel programlamanın profesyonel geliştiriciler tarafından bilinmesi ve uygulanması daha verimli programlar yazmak için önem kazanmıştır. Fakat paralel programlama nadiren tartışılır ve teorikte ayrıntılı olarak öğretilmez. Program kullanıcıları arayüzün kilitlenmesi veya gereksiz verilerin hesaplanması gibi işlemlerde zaman harcamak istemezler. Günümüzde yazılım geliştirme, paralel programlamayı etkin bir şekilde kullanmayı böylelikle hızlanma sağlamalıdır. Modern sistemlerin avantajlarından yararlanmak için kodların mümkün olduğunca paralel olarak tasarlanması ve yazılması gerekir.

Yapılan bu çalışmada yer alan tablolar ve grafikler paralel programlamanın performansa olan etkisini açıkça ortaya koymaktadır. Problem ve algoritma uygunluğuna göre işletim sistemine bırakılmadan yazılan algoritmalar sayesinde ölçümlerin daha detaylı olarak yapılmasına olanak sağlamıştır.

1.3 Literatür Taraması

SA algoritması optimizasyon çözümlerinde bulunduğu günden beri kullanılmaktadır. Sonrasında bilgisayar bilimlerinde çok işlemcili veya birbirine bağlı bilgisayarların kullanılmasıyla paralel benzetimli tavlama (PSA) algoritması ortaya atılmış ve bu algoritma ile ilgili çalışmalar yapılmıştır.

Casotto ve diğ. (1987) çalışmalarında problemin büyüdükçe algoritmalarının daha iyi çalıştıklarını söylemiş ve böylelikle işlemcilerin daha efektif kullanılacağını söylemişlerdir. Çalışmalarında paralel programlama ile gelen maliyeti ve hızlanmayı karşılaştırmalı olarak deneyimlemişlerdir.

Darema ve Kirkpatrick (1987) çalışmalarında seri olarak SA' in paralel donanıma sahip bilgisayarlarda çalıştırılmasıyla sürelerdeki sapmaları ve PSA çözüm kalitesini ve paralel hesaplamalardaki etkinliğini çoklu işlemciye sahip bir bilgisayar olan IBM 3081'de sanal makinalar üzerinde incelemişler. SA'da bir sonraki seçilen komşu ile aradaki enerji farkını kullandıkları algoritma sayesinde paralel programlama sayesinde seri programa göre daha az bulmuşlardır. Böylelikle performansta %80'e yakın bir iyileşme sağlamışlardır.

Greening (1990) PSA metotlarında o güne kadar kullanılan teknikleri belli sorular altında incelemişlerdir.

- Kodlar işlemciler arasında nasıl bölünüyor?
- Paralel algoritma komşu seçimini aynı komşular üzerinde seçebiliyor?
- Bir işlemci tarafından yapılan işlemler diğer işlemcilerin maliyet olarak hesaplanmasında hata olarak çıkıyor mu?
- Bu hatalar kontrol edilebilir mi?
- Hızlanma nedir ?

Gibi sorular sormuştur ve sorulan bu sorulara cevaplar vermiştir. Paralel algoritmaların sınıflandırılmasına çalışmasında yer vermişlerdir.

Ram ve diğ. (1996) PSA algoritmaları adlı çalışmalarında SA algoritmalarını 3 ana sınıf altında incelemişlerdir. Bunlar; seri benzeri algoritmalar, düzenlenmiş değiştirilmiş algoritmalar ve asenkron algoritmalardır. Bu sınıflar için maliyet hesabının doğruluğu, paralellik, durum üretme ve genel giderler arasında bir denge kurduklarını söylemektedirler. İş atölyesi planlama ve gezgin satıcı problemlerinde uygulamışlar ve problemin alanı yani büyüklüğü arttıkça daha iyi sonuç verdiklerini gözlemlemişlerdir.

Onbaşoğlu ve Özdamar (2001) çalışmalarında geliştirdikleri 5 farklı PSA algoritmasını evrensel optimizasyon problemlerinde denemişlerdir. Çalışmalarını 106 farklı fonksiyonda çalıştırmış ve karşılaştırmalarını vermişlerdir. C dilinde ve Linux üzerinde birbirine bağlanmış 8 adet bilgisayar üzerinden testlerini yapmışlardır. Modifiye edilmiş yüksek bağlantılı senkron SA algoritmaları diğer algoritmalarına göre daha verimli sonuçlar vermiştir. En iyi sonucun bulunduğu algoritma, komşu algoritmasını paralel olarak farklı yönlere giderek yürüten ve ana işlem parçacığının işlemi yapan diğer iş parçacıklarıyla iletişimi sabit aralıklarla yapması ve iş yüklerini tekrar iş parçacıklarına modifiye edilmiş yüksek bağlantılı senkron iletişime göre dağıtmaktadır.

Debudaj-Grabysz ve Czech (2009) PSA'yı teorik ve pratik olarak incelemişler. Bu incelemede önce teorik olarak hızlanmanın ne olacağını belirtmişler ardından uygulamada bu değerlere yakın değerler üretmişlerdir. Çalışmalarında iletişimsiz

algoritmayı, periyodik iletişim algoritmayı ve melez iletişim algoritmalarını karşılaştırmışlardır.

Ogura ve diğ. (2002) PSA algoritmasında genetik çaprazlama kullanışlar ve bunun katkısını gözlemlenmiştir. Önerdikleri algoritmada sabit aralıklardaki çözümler bilgi alışverişinde bulunmak için genetik çaprazlama kullanmışlardır. Böylelikle işlem maliyetini düşürmüşler ve etkili bir şekilde arama yaptıklarını söylemişlerdir. Bir sonraki çözüm kümesi yani komşu seçiminde her bir işlemci için bulunan komşu değerleri birbiriyle genetik çaprazlama metodunu kullanarak değiştirmişlerdir. Önerdikleri bu algoritma ile diğer komşu seçim algoritmalarına (elit seçim, rulet seçim vb.) kıyasla daha etkili bir çözüm bulduklarını söylemişlerdir.

Akkaş (2016) yüksek lisans tezinde karesel atama problemini SA ve paralel programlama teknikleri kullanarak çözmüştür. SA paralelleştirmesini MATLAB ortamında farklı şekillerde yapmıştır. Bu işlemler sırasında işçiler arasında haberleşmenin olmadığı asenkron hesaplamalı SA ve periyodik olarak işçiler arasında haberleşme sağlanarak senkron hesaplamalı SA yönteminin seri SA metoduna göre daha iyi sonuçlar verdiğini gözlemlemiştir.

Sonuç (2017) doktora tezinde algoritmayı CPU yerine GPU yani ekran kartları üzerinde çalıştırmıştır. GPU üzerinde CUDA ile çalıştığı tezinde CPU ya oranla daha iyi sonuçlar gözlemiştir. Burada CPU üzerinde belli bir parçacığından öteye geçilememesi büyük bir etkendir. GPU ile daha fazla iş parçacığı üzerinde işlem yapılabilir olması daha avantajlı hale getirmektedir. Tezin sonunda hem süre hem de çözüm kalitesinde paraleleştirmenin katkı sağladığını göstermiştir.

1.4 Tezin Katkısı

Optimizasyon algoritmalarının paralel hale getirilmesi hem zamandan hem de verimlilikten kazanç sağlayacaktır. Paralelizasyonun önemini ve sağladığı kazançları somut olarak görmek için, SA algoritmasını çok çekirdekli bilgisayarlarda paralel çalışacak şekilde tasarlanmıştır.

Yazılım dünyasında yaygın olarak kullanılan .NET Framework ile SA algoritması önce seri olarak sonra paralel olarak yazılmış ve sonuçlar değerlendirilmiştir. Bu çalışma sonunda algoritmadaki kod parçası ile yapılan paralel optimizasyonun verimliliğinin etkisi gözlemlenmiştir. Bu sayede yazılımcılara ve program geliştirmek isteyenlere algoritmalarını inceleyerek paralelleştirmeye uygunsa bunu yapmalarının çözüm süresinde azalmaya gideceği için süreden tasarruf sağlandığı ve verimin arttığı gösterilmiştir.

Paralel programlamada dikkat edilmesi gereken bir diğer hususta verimliliği hesaplarken önemli olanın hız mı maliyet mi olduğuna karar verilmesidir. Çekirdek veya işlemci sayısındaki artış belli bir süre sonunda maliyet artışı oranında hız artışı sağlamamaktadır. Çalışmada yapılan testler ve sonrasında bulunan sonuçlar bu durumu göstermektedir.

Her bir işlemci katkısı hem verimlilik hem de yüzde performans iyileşmeleri üzerinden teker teker ele alınmış ve varılan bir noktadan sonra süre iyileştirmesinin düştüğü gözlemlenmiştir.

2. YÖNTEM

Çalışma içinde kullanılan yöntemlerin ve araçların detaylı açıklamaları bu bölüm altında incelenmiştir.

2.1 Benzetimli Tavlama

SA, çeliklerin tavlama işleminden örnek alınmış, deneme yanılmaya dayalı bir optimizasyon metodudur. Karmaşık non-lineer optimizasyon problemlerini çözmek için sıklıkla kullanılan sezgisel bir metottur. Tavlama katı bir maddenin sıcaklığının arttırılması ve sonra kristalleşme olana kadar yavaş bir şekilde soğutulmasıdır. Maddenin içindeki atomlar yüksek sıcaklık ile yüksek enerji seviyesine çıkarılır ve hareket halinde bulunmaları sağlanır. Sıcaklık düşürüldükçe enerji seviyeleri azalır ve madde kararlı hale getirilmeye çalışılır. En düşük enerji seviyesinde düzenli yapı elde edilir. Eğer soğutma işlemi hızlı bir şekilde gerçekleştirilirse burada düzensiz bir yapı ve kusurlu kristalleşmeler görülür. Sistem en düşük enerji seviyesine ulaşamadan ve polikristal halde sona erer (Pham ve diğ. 2000).

1953 yılında Nicolas Metropolis ve arkadaşları ayrı moleküllerin etkileşerek oluşan kabul edilebilir herhangi bir maddenin özelliklerini hesaplanması için bir Monte Carlo yöntemi geliştirdiler. Bu ilham ile Scott Kirkpatrick, C. Daniel Gelatt ve Mario P. Vecchi evrensel en iyileme için SA yöntemini geliştirdiler (Kirkpatrick ve diğ. 1983).

SA süreci, kristalleştirme sıcaklığından başlayarak sıcaklığın kademeli olarak azaltılması ve hem iyi hem de kötü sonuçlar arasında dolaşılması ile en iyi çözüme ulaşılmasıdır. Burada benzetimli tavlama metodunun kötü sonuçları da belli bir olasılığa göre değerlendirip kabul edilip edilmeyeceğine karar vermesi sistemi yerel çözümlerden kurtarmaya yarar.

SA metodunda, başlangıç sıcaklığı rasgele oluşturulur ve bu sıcaklıktan soğumaya doğru geçilir. Her komşu sıcaklıkta amaç fonksiyonundaki değerler bir önceki ile karşılaştırılır. Eğer enerji azalıyor ise direk kabul edilir, enerji artıyor ise de bir olasılık ve bu olasılığın rasgele bir sayı ile karşılaştırılmasına göre kabul edilir.

$$P\ (\Delta E) = e^{-\frac{\Delta E}{kT}} \quad (2.1)$$

Bu olasılık, Eşitlik 2.1 (Kirkpatrick ve diğ. 1983) gösterildiği gibi, bulunduğu enerji ile bir önceki enerji arasındaki farkın (ΔE), Boltzmann sabitiyle o anki sıcaklığa bölünmesi ile oluşur. Eğer ΔE, yani bulunulan sıcaklıktaki enerji ile bir sonraki aşamadaki enerji arasındaki fark eğer minimuma yaklaştırıyor ise kabul edilir. Eşitlik 2.2'de eğer kötü bir sonuç çıkıyor ve minimumdan uzaklaştırıyor ise $P\ (\Delta E)$ ile 0 ve 1 arasında rasgele oluşturulan bir sayı ile karşılaştırılır ve $P\ (\Delta E)$, rasgele bulunan bu sayıdan küçük çıkarsa sonuç kabul edilir (Kirkpatrick ve diğ. 1983).

$$r = random_u() \in [0,1) \quad (2.2)$$

Tablo 2.1'de açıklanan SA algoritmasının sözde kodunda (pseudocode) görüldüğü gibi işlemler soğuma işlemi tamamlanana veya yeterli görülen sonuca kadar devam eder.

Tablo 2.1: Benzetimli Tavlama Algoritması Sözde Kodu

Başla

1. *Başlangıç sıcaklığı belirle*: $t\ =\ init\_temp$
2. *Tekrarla*
   - 2.1. $iterasyon\ =\ 0$.
   - 2.2. *Tekrarla*
     - 2.2.1. *Amaç fonksiyonu hesapla* $(f(i))$
     - 2.2.2. *Yeni bir komşu hesapla* $(f(j))$
     - 2.2.3. *Eğer* ΔE *küçük* 0 *ise kabul, değilse min* $(1, e^{-\frac{\Delta E}{kT}})$ *değerine göre kabul veya reddet*
     - 2.2.4. *iterasyon arttır*
     - 2.2.5. *iterasyon bitene kadar*
3. $t\ =\ t * sıcaklık\_azalması$
4. *Her sıcaklıkta iterasyon sayısına göre sonuçlar değişir ve en iyi değeri kabul et*

Bitir.

Ayrıca SA metodunda başlangıç sıcaklığının hesaplanmasında, amaç fonksiyonun rasgele denemeler yapılması ve bu sonuçların ortalaması ile başlangıç

sıcaklığı bulunmuştur. Böylelikle amaç fonksiyonun başlangıç sıcaklığı probleme uygun bir şekle getirilmiştir. Sıcaklığın düşürülmesi ise başlangıç sıcaklığının doğrusal olarak yapılmıştır. Her bir sıcaklık bir önceki sıcaklığın 0,1 ile soğutulması ile hesaplanmıştır (Eşitlik 2.3).

$$T_{i+1} = T_i - T_i * 0{,}1 \tag{2.3}$$

Amaç fonksiyonunda daha iyi sonuçlar alabilmek için, hesaplamada bulunan en iyi sonuç ile Tablo 2.2'de sözde kodu tanımlanan SA tekrar uygulanmaktadır. Burada sıcaklık kümesi, bir önceki hesapta kullanılan sıcaklık değerlerinin 50'de 1'i kullanılarak alınmıştır.

Tablo 2.2: İnce Hesap için Benzetimli Tavlama Algoritması Sözde Kodu

Başla

1. *Başlangıç sıcaklığı belirle*: $t = t_{bulunulan\ sıcaklık}$
2. *Tekrarla*
   2.1. $iterasyon = 0$
   2.2. *Tekrarla*
      2.2.1. *Amaç fonksiyonu hesapla* $(f(i))$
      2.2.2. *Yeni bir komşu hesapla* $(f(j))$
      2.2.3. *Eğer* ΔE *küçük* 0 *ise kabul, değilse min* $(1, e^{-\frac{\Delta E}{kT}})$ *değerine göre kabul veya reddet*
      2.2.4. *iterasyon arttır*
      2.2.5. *iterasyon bitene kadar*
3. $t = t * sıcaklık\_azalması$
4. *Her sıcaklıkta iterasyon sayısına göre sonuçlar değişir ve en iyi değeri kabul et*

Bitir

2.2 Paralel Programlama

Gordon Moore tarafından, sürekli gelişmekte olan tümleşik devreler yani mikroişlemcilerin artışını bir öngörü haline getirmiştir. Moore Kanunu, tümleşik devre üzerindeki bileşen sayısında her 18 ayda 2 kat artış gözleneceğini, bir başka deyişle işlemci hızlarının 18 ayda bir iki katına ulaşacağını öngörmüştür (Moore, 1965).

Fakat günümüzde bu kanun başta güç problemi olmak üzere fiziksel olarak kondansatör kapasitesilerinin fiziksel sınırlara ulaşması gibi sebeplerden dolayı yavaşlamaya ve geçerliliğini yitirmeye başlamıştır. Donanım hızlarının sürekli artmasından dolayı yazılımcılar için algoritmalarını düzenlemelerine gerek kalmadan işlem süreler azalmaktadır. Fakat belli bir süre sonra işlemcilerin belli fiziksel sınırlara ulaşmasından dolayı bu işlemci hız artışı birden fazla işlemcinin kullanılması veya bir işlemcide birden fazla çekirdeğin kullanılmasıyla işlemcilerin gücünün artması sağlanmıştır. Bu durum yazılım tarafında paralel programlamanın önemini arttırmıştır (Ergün ve Sayar, 2014). Paralel programlama günümüzde son kullanıcıların dahi kolayca kullanılacakları bir seviyeye inmiştir. Çoğu programlar basit düzenlemelerle paralel programlama ile çalışabilir hale gelmektedir.

Paralel programlama kullanılmasındaki temel nedenler;

- Zaman ve maliyetten kazanılması
- Büyük veya karmaşık problemlerin çözülmesi
- Eşzamanlılık sağlaması
- Kaynaklardan verimli yararlanması
- Paralel donanıma sahip cihazları daha iyi kullanılmasıdır.

Paralel programlamanın kullanıldığı bazı alanlar;

- Atmosferin ve dünyanın incelenmesi
- Fizikte nükleer, uygulamalı alanlar, parçacık, füzyon gibi alanlar
- Biyoteknoloji, genetik bilimler
- Kimya ve moleküler bilimler
- Jeoloji
- Bilgisayar bilimleri, matematiksel işlemler

- Silah Endüstrisi

Yüksek güç gerektiren hesaplamalarda kullanılmak üzere yüksek performanslı bilgisayarlar meydana getirilmiştir. Bunlar son kullanıcılar için değil, daha çok araştırma enstitüleri ve firmalar için üretilmiştir. Bu bilgisayarların LINPACK kıstasına göre ilk 500 sırasını yayınlayan www.top500.org sitesinin güncel verilerine göre ülkeler ve ülkelerdeki sayıların dağılımları Tablo 2.3'de gösterilmiştir. (Top500, 2019)

LINPACK kıstası, J. Dongarra tarafından bilgisayarların kayan noktalı sayı testini yaparak işlemcinin gücünü ve hızını ölçmesi için 1979'da tasarlanmıştır (Dongarra ve diğ., 1979). Günümüzde halen süper bilgisayarların işlem güçleri ve sıralamaları için kullanılmaktadır. Yüksek performanslı hesaplamada, Rmax ve Rpeak, süper bilgisayarları LINPACK kıstası kullanarak performanslarına göre sıralamak için kullanılan puanlardır. Bir sistemin Rmax skoru, elde edilen maksimum performansı açıklar; Rpeak skoru teorik zirve performansını anlatır. Her iki skor için de değerler genellikle teraFLOPS veya petaFLOPS ile gösterilir.

Tablo 2.3: Ülkelerin Süper Bilgisayar Sayıları ve İşlem Güçleri

Ülke	Sayı	Yüzde	Rmax	Rpeak	Çekirdek Sayısı
Çin	227	45,4	438.228.339	806,368,243	26,632,672
Amerika	109	21,8	533.209.190	757,357,100	16,101,360
Japonya	31	6,2	109.436.242	170,880,045	5,710,372
İngiltere	20	4	41,729,303	52,509,525	1,625,892
Fransa	18	3,6	43,580,345	66,598,837	1,792,656
Almanya	17	3,4	60,502,637	86,333,952	1,575,350
İrlanda	12	2,4	19,789,320	25,436,160	691,2
Kanada	9	1,8	14,027,780	22,258,586	436,64
İtalya	6	1,2	31,110,650	49,243,746	814,864
Güney Kore	6	1,2	21,938,000	35,760,556	804,74
Hollanda	6	1,2	9,334,060	11,925,504	326,88
Avustralya	5	1	6,669,188	10,232,963	257,336
Hindistan	4	0,8	8,358,996	9,472,166	272,328
Polonya	4	0,8	4,604,365	6,216,160	153,128
İsviçre	4	0,8	4,653,054	6,565,116	139,408
Rusya	3	0,6	4,580,250	7,940,005	178,18
Suudi Arabistan	3	0,6	10,109,130	13,858,214	325,94
Singapur	3	0,6	4,308,220	5,525,299	146,112
İspanya	2	0,4	7,488,800	11,781,642	172,656
Tayvan	2	0,4	10,325,150	17,297,190	197,552
İsveç	2	0,4	23,126,750	29,347,305	453,14
Güney Afrika	2	0,4	2,152,470	2,779,930	71,256
Yeni Zelanda	1	0,2	908,892	1,425,408	18,56
Norveç	1	0,2	953,571	1,081,651	32,192
Brezilya	1	0,2	1,123,150	1,413,120	38,4
Finlandiya	1	0,2	1,250,000	1,689,293	40,608
Çek Cumhuriyeti	1	0,2	1,457,730	2,011,641	76,896

Bilgisayar mimarileri Flynn'a göre 4 ana sınıftan oluşur (Flynn 1972). Bu sınıflandırma işlemlerin nasıl işlemciye gittikleri ve hangi sırayla işlendiklerine göre belirlenmiştir. Bunlar;

- **Tek Komut Tek Veri Akışı (Single Instruction Single Datastream - SISD)**

  Birim zamanda tek komutun tek veri alınarak işlenmesidir. Seri bir şekilde bir komutun bitmesiyle diğer komutun çalışmasıdır.

- **Tek Komut Çok Veri Akışı (Single Instruction Multiple Datastream - SIMD)**

  Birim zamanda tek komutun birden fazla veriyle işlenmesidir. Komutun çalıştırılması farklı çekirdekler üzerinden paralel olarak gerçekleşir.

- **Çok Komut Tek Veri Akışı (Multiple Instruction Single Datastream - MISD)**

  Birim zamanda birden fazla komutun tek veriyle işlenmesidir. Komutların çalıştırılması farklı çekirdekler üzerinden paralel olarak gerçekleşir fakat aynı veri akışını kullanırlar.

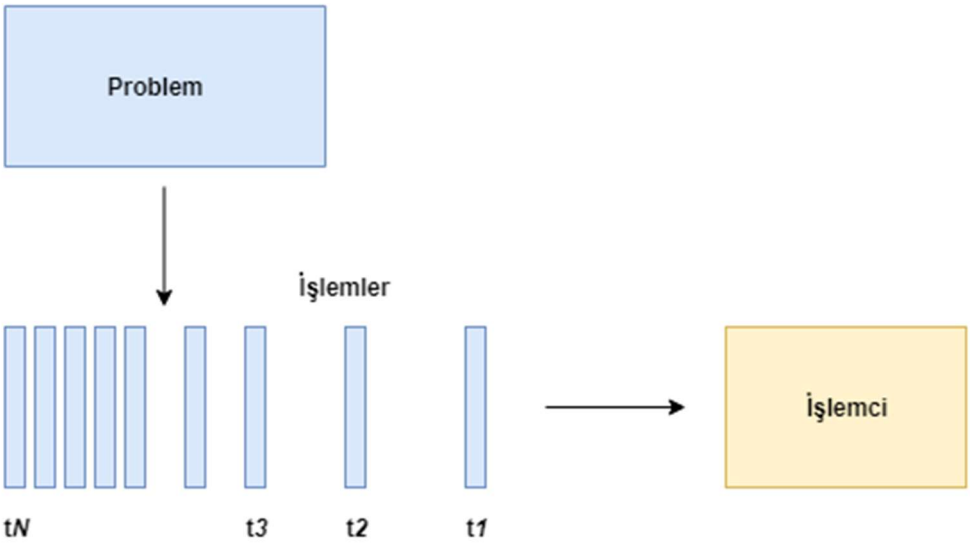
- **Çok Komut Çok Veri Akışı (Multiple Instruction Multiple Datastream - MIMD)**

  Birim zamanda birden fazla komutun birden fazla veriyle işlenmedir. Komutların çalıştırılması farklı çekirdekler üzerinden paralel olarak farklı veri akışları kullanılarak gerçekleştirilir.

Bu bilgisayar mimarileri işlemcilerin komutları ve verileri işleme şeklini ortaya koyduğu için programlamada seri ve paralel programlamanın nasıl bir yol izleyeceğini ortaya koymaktadır.

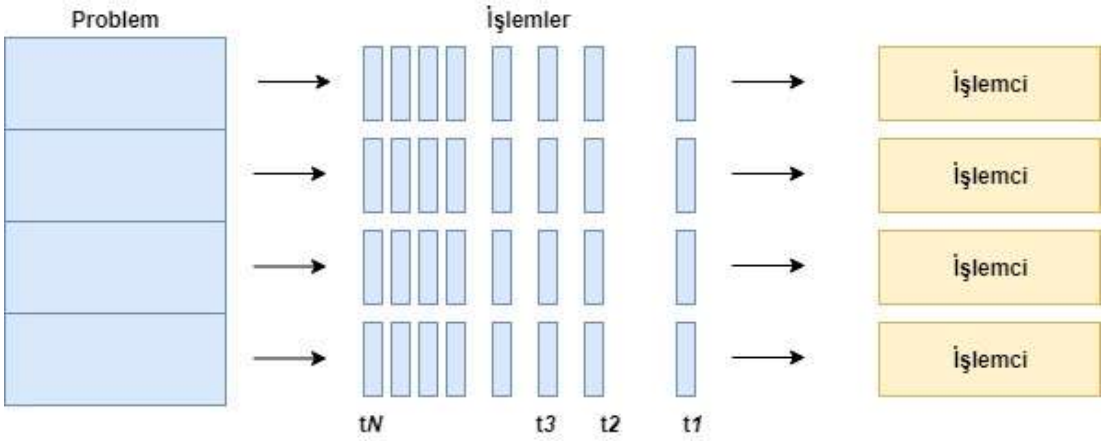
Seri programlama, yapılacak işlemlerin bir bilgisayar ve bir işlemci ile gerçekleştirilmesidir. Problem farklı komut serilerine bölünür ve her komut bir önceki komutun bitmesi ile işleme alınır. Herhangi bir anda sadece bir komut işlenir (Şekil 2.1). Genel itibariyle, yazılımlar seri hesaplama metoduyla yazılmıştır.

Şekil 2.1'de görüldüğü gibi seri programlamada işlemin süresi boyunca başka bir komut çalıştırılamamaktadır. Bu işlemlerin çözümünde hız, performans ve verimlilik gibi kıstaslar göz önünde bulundurulursa istenilen değerlerde çözüm üretememektedir.



Şekil 2.1: Seri Programlamada Görevlerin İşlemcilerdeki İşleyişi (SISD)

Paralel programlama ya da paralel hesaplama, birden fazla bilgisayar veya birden fazla işlemci ile işlemlerin gerçekleştirilmesidir. Problem farklı komutlara bölünür ve eşzamanlı olarak farklı işlemcilerle işleme alınır (Şekil 2.2).



Şekil 2.2: Paralel Programlamada Görevlerin İşlemcilerdeki İşleyişi (MIMD)

Paralel programlamada önemli konulardan biri de belleğin nasıl kullanılacağıdır. İşlemcilerin hangi veri yollarını ve hangi türde ortak bellek kullanımları işlemlerin işleyişini ve kontrolünde farklılığa yol açmaktadır. Kabul gören bellek türleri aşağıda belirtilmiştir.

- **Paylaşımlı Bellek**

  Çoklu işlemcili bilgisayarlarda işlemciler işlemlerini kendileri gerçekleştirir fakat aynı belleği kullanırlar. Bir işlemcinin bellekte yaptığı değişiklik bütün işlemciler tarafından görülür. Belleğe ulaşım zamanlarına göre iki ana sınıfa ayrılır:

  - Tek düzen bellek erişimi (Uniform Memory Access (UMA))

    Simetrik çoklu işlemcili bilgisayarlarda kullanılır. Belleğe erişim süreleri ve hızları eşittir. Kişisel olarak günümüzde kullandığımız çok çekirdekli bilgisayarlar buna örnektir.

  - Tek düzen olmayan bellek erişimi (Non-Uniform Memory Access (NUMA))

    Fiziksel olarak birbirine bağlı iki veya daha fazla çoklu işlemcili bilgisayardan oluşur. Bir bilgisayar diğer bilgisayarın belleğine erişebilir fakat aynı sürelerde erişim mümkün değildir.

- **Dağıtık Bellek**

  Bellekler arası iletişim ağı mevcuttur. Her işlemcinin kendi özel belleği vardır. Bir işlemcinin bellek üzerinde yaptığı değişiklik diğer işlemcileri ve onların belleklerini etkilemez. İşlemcilerin bellek verilerinin iletişimini programcı sağlar.

- **Karma Bellek**

  Günümüzde büyük ve hızlı bilgisayarlarda kullanılan bellek türüdür. Hem paylaşımlı bellek yapısını hem de dağıtık bellek yapısını kullanır. Paylaşımlı bellek bileşeni cihazdaki paylaşımlı bellek veya grafik

işlemci birimleri olabilir (GPU). Dağıtık bellek bileşeni, bir ağ üzerinden birden dağıtımlı bellek/CPU yapısı içerir. Sadece kendi bağlı olduğu bellek hakkında bilgiye sahiptir, başka bir cihaz üzerindeki belleğe erişemez. Bu nedenle ağ iletişimi verilerin bir cihazdan diğer cihaza taşınmasını gerektirir (Aad J. van der Steen, 2013).

2.2.1 Paralel Programlamada Temel Kanunlar

Paralel programlama, seri programlamaya göre algoritmanın yani uygulamanın belli bir oranda hızlanmasını sağlar. Paralel programlamanın uygulanmaya başladığı ilk zamanlardan beri performans ölçümleri ve öngörüleri yapılmaktadır. Bu konuda kesin bir ölçüm günümüzde halen yapılmasa da öncü hesaplamalar üstüne düzenlemeler yapılarak ölçümler iyileştirilmiştir. Problemin paralelleştirilemeyen bölümü, algoritmaların büyüklüğünün ölçeklendirilememesi, verilerin ve kaynakların (bellek, veri yolu vb.) kısıtlı ve ortak olması gibi kısıtlamalar ölçümlerin sağlıklı olmamasının sebebidir.

Paralel programlamada hızlanma hangi aşamada olmuştur, verimi nedir gibi sorulara cevap arayan bilim insanları birçok ölçüm kıstası ortaya koymuştur. Bu ölçümler zamanla ya önemini yitirmiştir ya da zor ölçeklendirilmeye başlanmıştır. Bu bölümde ölçümlerden en bilinenlerini anlatılmıştır.

2.2.1.1 Amdahl Yasası

Teorik olarak ne kadar paralel performans elde edebileceğini tanımlayan ve en çok bilinen kanunlardan biridir. Amdahl Yasasında hızlanma Eşitlik 2.4'te görüldüğü üzere, tek işlemci çalıştırılarak geçen süre $T(1)$ ile kullanılan işlemci sayısında n_p geçen sürenin $T(n_p)$ bölünmesi ile bulunmuştur (Amdahl, 1967).

$$S_p(n_p) = \frac{T(1)}{T(n_p)} \tag{2.4}$$

Bu hesaplama algoritmanın tamamının dağıtılması ile mümkün olacaktır. Fakat böyle bir mükemmel hızlanmayı çoğunlukla sağlayamayacağımızdan hızlanmayı ve

etkinliği ele alırken hesaba algoritmanın paralelleştirilen ve paralelleştirilemeyen kısımları farklı ele alınmalıdır. Eşitlik 2.5'te Amdahl Kanunu verilmiştir. Bu kanunda yer alan (f) paralelleştirilemeyen yani seri kısmı temsil eder (Amdahl, 1967).

$$S_p = \frac{1}{(f + \frac{(1-f)}{n_p})} \tag{2.5}$$

Hızlanma yüzdesi Eşitlik 2.6'da gösterilmiştir (El-Rewini ve Abd-El-Barr, 2004).

$$f = 1 - \left(1 - {}^{1}/_{S_p}\right)/(1 - {}^{1}/_{n_p}) \tag{2.6}$$

Amdahl kanunu tamamen yanlış olmamasıyla birlikte iki hatalı varsayımı mevcuttur. Birinci olarak çalışmaların yapıldığı tarihlerde algoritmaların paralelleştirilmesi için çaba harcanmaması, ikincisi de problemin büyüklüğüne bağlı olarak paralel ve seri kısımların eşit olarak işlem hızlarındaki büyüme orantısal olarak artmamaktadır (Ergün ve Sayar, 2014).

2.2.1.2 Gustafson-Barsis Yasası

Gustafson ve Barsis (1988), Amdahl yasasından farklı olarak paralel ölçeklenebilirliğin, problemin seri ve paralel kısmının yani boyutunun düzeltilmesinden ziyade işlemci sayısına göre ölçeklendirilmesi gerektiğini ortaya atmıştır (Gustafson, 1988). Gustafson, bilgisayarların gücü arttıkça problemin boyutunun artışına dikkat çekmiştir. Yasaya göre seri kısım az büyürken ya da sabit kalırken problemin boyutu *p* ile yani işlemci sayısı ile artarsa, işlemci sayısı arttıkça hızlanma artar. Gustafson-Barsis yasasında Eşitlik 2.7'de ifade edilmiştir. Formüle göre performansın üst sınır seri kısımla değil problemin büyüklüğü ile alakalıdır.

$$S_p = p - (p - 1)f \tag{2.7}$$

İki kanun da paralel optimizasyona yönlendirir. Amdahl Yasası, hızlanmayı çalışmanın seri kısmı yani paralelleştirilemeyen kısmı ile sınırlandırmış; Gustafson-Barsis işlemci sayılarının artmasıyla aynı zamanda daha büyük sorunların çözülmesi gerektiğini göstermektedir (McCool, Reinders ve Robison, 2012).

2.2.1.3 Karp-Flatt Ölçütü

Karp ve Flatt, Amdahl Yasası ve Gustafson Yasasının performans hesaplamalarında hangi işlemcinin ne kadar yük aldığının hesaplamasının yapılmadığından, yani bir işlemcinin paylaştırılan işi diğer bir işlemciyle tamamen aynı sürede yapamayacağından, seri kısmının kestirilmesi için başka bir formül geliştirmişlerdir. Bu ölçüt Eşitlik 2.8 de gösterilmiştir (Karp ve Flatt, 1990)..

$$f = \frac{\frac{1}{S_p} - \frac{1}{p}}{1 - \frac{1}{p}} \tag{2.8}$$

Problemin kaç kat hızlandığının veya verimliliğin tam olarak açıklama yapmadığını, Eşitlik 2.8'deki formül ile seri kısmının ölçümünün işlemci sayısıyla kıyaslanarak çalışmanın ne gibi kısıtlamalara maruz kaldığını göstermektedirler (Karp ve Flatt, 1990). f değerinin sabit kalması ideal paralel programlamaya ulaştığımızı yani kısıtlamaların artışı problemin büyümesiyle aynı orantıda olduğunu göstermektedir.

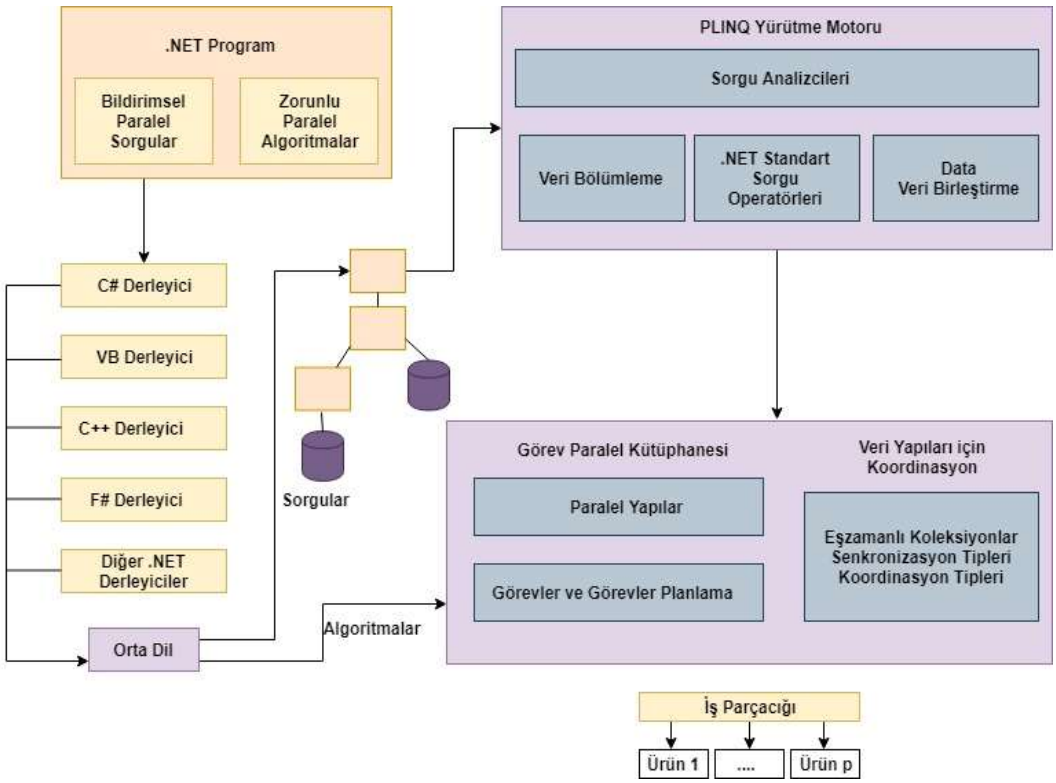
Bu çalışmanın sonuçlarının değerlendirilme kısmında Karp-Flatt ölçütü kullanılmıştır.

2.3 .NET Paralel Programlama

Visual Studio 2010 ve .NET Framework 4 yeni bir çalışma zamanı (runtime), yeni sınıf kütüphanesi türleri ve yeni tanımlama araçları sağlayarak paralel programlama desteği getirmiştir. Bu özellikler, doğrudan iş parçacıkları veya iş parçacığı havuzuyla çalışmak zorunda kalmadan doğal bir ifadede etkili, ayrıntılı ve ölçeklenebilir paralel kod yazmanız için paralel geliştirmeyi basitleştirir. Şekil 2.3'te .NET içindeki paralel programlama mimarisi için bir genel bakış sağlamaktadır.

.NET'in amaçlarından biri, geliştiriciler için çok çekirdekli makinelerde paralel programlama yazmalarını kolaylaştırmaktır. Bu amaca ulaşmak için .NET, geliştiricilerin bazı dağınık ayrıntılarla uğraşmak zorunda olduğu çeşitli paralel programlama ilkelerini tanıtmıştır. Uygun ilkenin .NET'te kullanılması kodu daha okunabilir, daha korunaklı, daha iyi performans alınabilir ve daha az hata eğilimli hale getirir (Ostrovsky 2010). Bu ilkeler şunlardır:

- Parallel.For ve Parallel.ForEach
- Parallel LINQ
- Eşzamanlı koleksiyonlar
- Koordinasyon ilkeleri
- Görev paralelleştirme



Şekil 2.3: .NET Paralel Programlama İş Akış Şeması (Microsoft, 2018)

2.3.1 Parallel.For ve Parallel.ForEach

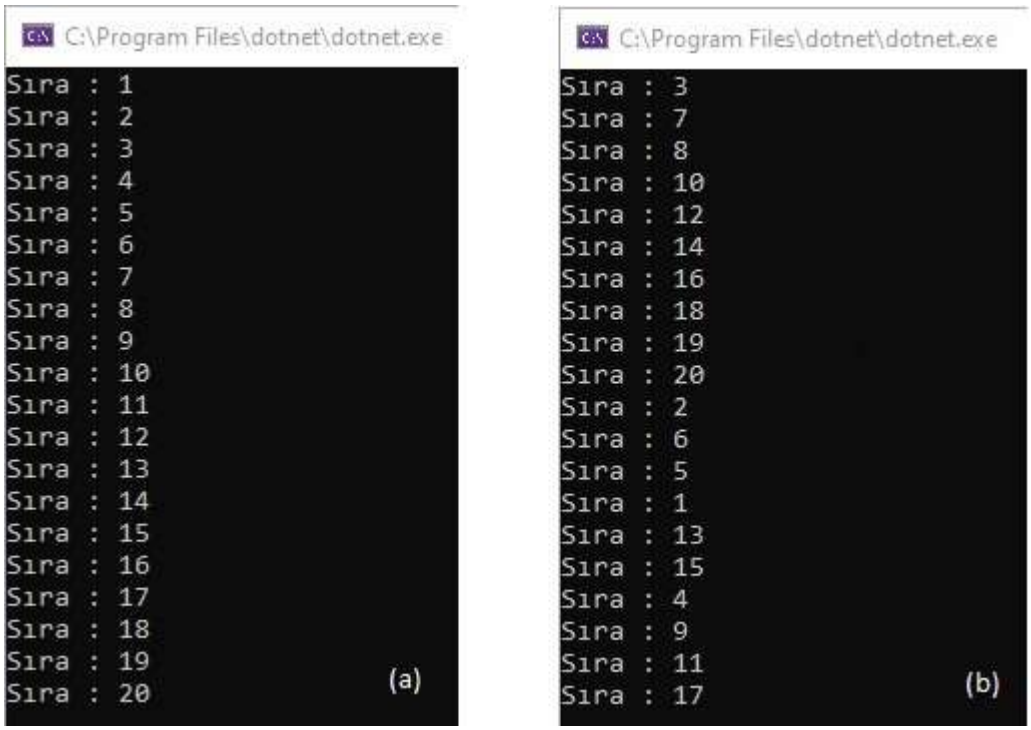
Paralel döngüler olan Parallel.For ve Parallel.ForEach kavramsal olarak for ve foreach döngüsü ile, döngü yapısının farklı adımlar yürütmek için birden çok iş parçacığı kullanması dışında benzerdirler. Paralel döngüler, genellikle bir dizi işlemin çözümünün hızlanmasında çok çekirdekli makinelerin avantajlarını kullanmak için kolay bir yoldur.

Paralel ve seri olan For ve Foreach döngülerinin farkını görmek için oluşturulan bir listenin hem normal hem de Paralel.For ile çağrılması için yazılan programın seri kod parçası Şekil 2.4'te ve program sonucu ortaya çıkan ekran görüntüsü Şekil 2.6(a)'da gösterilmiştir. Paralel kod parçası Şekil 2.5'te, program sonucu ortaya çıkan görüntüler Şekil 2.6(b)'de gösterilmiştir. Şekil 2.6(a)'da görüldüğü üzere işlem sırasına uygun bir şekilde sonuçlar gelirken, Şekil 2.6(b)'de sonuçlar herhangi bir sıralamaya bağlı kalmaksızın gelmektedir. Bunun sebebi paralel for ve foreach kullanımında işlemcilerin hangi sırayla çalışacağının verilmemesidir.

```
static void Main(string[] args)
{
    List<int> sayilar = new List<int> { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20 };

    for (int i = 0; i < sayilar.Count; i++)
    {
        Console.WriteLine("Sıra : {0}", sayilar[i]);
    }
    Console.Read();
}
}
```

Şekil 2.4: For Döngüsü Seri Kod Parçası



Şekil 2.6: For Döngüsü Ekran Çıktısı a) Seri Programlama b) Paralel Programlama

2.3.2 Paralel LINQ

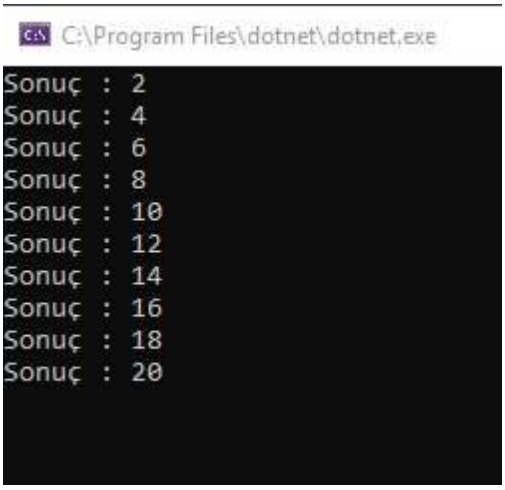
LINQ (Language Integrated Query) .NET Framework 3.5 ile kullanılmaya başlanan programlama diline entegre edilmiş sorgu teknolojisidir. Bu teknolojisi bize .NET ile program yazarken veritabanındaki gibi bir sorgulama yeteneği vermektedir.

Paralel LINQ (PLINQ), LINQ objelerindeki sıradan döngülerin paralel döngüler haline gelmesidir. PLINQ, LINQ sorgularını çalıştırır fakat bu sorguları birden fazla iş parçacığı üzerinde dağıtır. Paralleştirme konusunda getirdiği basitlik sayesinde kullanımı kolaydır.

Örnek çalışma olarak 1'den 20 'ye kadar olan bir listede çift sayıları bulan bir kod yazılmıştır. Seri olarak yazıldığı kod parçası Şekil 2.7'de, ortaya çıkan sonuçlar Şekil 2.8'de gösterilmiştir. Paralel olarak çağrılan kod parçası Şekil 2.9'da ekran görüntüsü Şekil 2.10'da verilmiştir.

```
static void Main(string[] args)
{
    List<int> sayilar = new List<int> { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20 };
    var seriSonuclar = from ite in sayilar
                       where ite % 2 == 0
                       select ite;
    foreach (var item in seriSonuclar)
    {
        Console.WriteLine("Sonuç : {0}", item);
    }
    Console.Read();
}
```

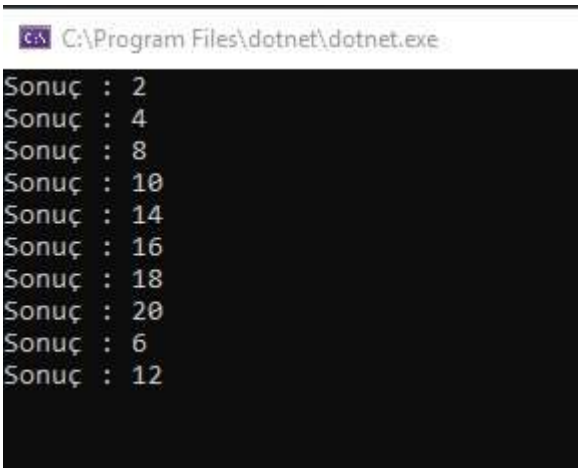
Şekil 2.7: LINQ Seri Programlama Kod Parçası



Şekil 2.8: LINQ Seri Programlama Ekran Çıktısı

```
static void Main(string[] args)
{
    List<int> sayilar = new List<int> { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20 };
    var paralelSonuclar = from ite in sayilar.AsParallel()
                          where ite % 2 == 0
                          select ite;
    foreach (var item in paralelSonuclar)
    {
        Console.WriteLine("Sonuç : {0}", item);
    }
    Console.Read();
}
```

Şekil 2.9: PLINQ Paralel Programlama Kod Parçası



Şekil 2.10: PLINQ Paralel Programlama Ekran Çıktısı

Şekil 2.9 ve Şekil 2.10'da görülmektedir ki çalıştırılan programın işlem sırası probleme göre önem taşımaktadır. Eğer sıralama önemli bir program yazılsaydı paralel programlama kötü sonuçlar verecekken seri programlama daha doğru sonuçlar verecektir. Ama programdan istenilen sıra değil de çözüm süresi olsaydı paralel programlama bu problem için iyi sonuçlar verecektir.

2.3.3 Eşzamanlı Koleksiyonlar

.NET, güvenli iş parçacığı ve birden çok iş parçacığının optimize şekilde eşzamanlı erişimini sağlamak için eşzamanlı koleksiyonlar tanıttı. Bunlar kuyruk, yığın ve sözlüklerin eşzamanlı versiyonları olan ConcurrentQueue, ConcurrentStack ve ConcurrentDictionary dir.

2.3.4 Koordinasyon İlkeleri

- Güvenli İş Parçacığı
- Kilitlenmeler
- Performans
- Ölçüm
- Önbellek ve Performans
- Performans Araçları

Her bir ilke yazılan program ve probleme göre farklılık göstermektedir. Burada dikkat edilmesi gereken konu, bu ilkelerin maliyet hesabı olarak ele alınması ve çözümde istenilen şeyin ne olduğunun bilinmesidir. Bu ilkelerden toplam bir maliyet hesabı çıkararak problem paralel programlama yapılarak çözülebilir.

2.3.5 Görev Paralelliştirme

Görevler (Task), .NET Framework 4 ve sonrasında geçerli olarak asenkron çalışma birimlerini temsil eder. Görevler daha önceki .NET Framework'lerde mevcut değildi ve geliştiriciler iş parçacığı havuzu (ThreadPool) çalışma öğelerini kullanmak zorunda kalıyordu. Ancak, taskların kullanışlı olduğunu destekleyen birtakım özellikler şunlardır:

- Görevler bekletilebilir
- Görevler iptal edebilir
- Görevlere zamanlama eklenebilir, bazı görevler bittikten sonra devam edilebilir.

Görevler istenilen her amaç için kullanabilir. Burada herhangi bir kısıtlama yoktur. Görev kodlanırken iş parçacığı güvenli ön planda tutulmalıdır. Görevler birden fazla çekirdekli bilgisayarlarda başarılı ile çalışabilecek şekilde tasarlanmıştır.

Görev Paralel Kütüphanesi (TPL), asenkron işlemleri temsil eden görev konsept kavramına dayanmaktadır. Bazı açılardan görev, bir iş parçacığına veya iş parçacığı havuzu çalışma öğesine benzese de daha yüksek bir ayırma sağlamaktadır. Görev paralelizasyonu bir veya daha çok eşzamanlı çalışan bağımsız görevler anlamına gelmektedir. Görevler iki temel kazanç sağlamaktadır:

- Sistem kaynaklarının daha verimli ve daha ölçeklenebilir kullanımı
- Bir iş parçacığından veya çalışma öğesinden daha kontrollü programlama kullanımı

Her iki sebepten dolayı .NET'te TPL çoklu iş parçacıklarıyla, asenkron olarak ve paralel kodlanacak programlarda tercih edilen ilkedir.

2.4 Paralel Benzetimli Tavlama

SA metodunun paralelizasyonunda;

- Asenkron olarak farklı başlangıç değerleri verilerek
- Senkron paralelleştirme amaç fonksiyonu değerlerini çalışanlar arası iletilerek
- Senkron paralelleştirme çözümleri arada çalışanlar arası ileterek
- Kümelenmiş paralelleştirme çözümlerin yoğun olarak alışverişini sağlayarak
- Oldukça bağlanmış senkronizasyonda her çalışanın sonucunu komşu çözümü için değerlendirilmesi

yöntemlerinden biri kullanılabilir (Onbaşoğlu ve diğ. 2001).

Problemin çözümünde senkron paralelleştirmede amaç fonksiyonu değerlerinin paralel işlemlerden arası karşılaştırılması kullanılmıştır. Önce tek işlemci ile başlandı, sonra kullanılan işlemci sayısı artırıldı. İşlemler sırasında işlemci sayıları, işlemci yükleri ve süreler kaydedilmiştir.

Tablo 2.4: Paralel Benzetimli Tavlama Algoritması Sözde Kodu

Başla

1. *Başlangıç sıcaklığı belirle*: $t = init\_temp$
2. *Tekrarla*
   2.1. *Görev Sayısı* = İ*zin Verilen* İ*şlemci Sayısı*
   2.2. *Görev* İ*terasyon Sayısı* = İ*terasyon Sayısı* / *Görev Sayısı*
   2.3. *Tüm görevlerde başla*
      2.3.1. *Görev* İ*terasyon* = 0
      2.3.2. *Tekrarla*
         2.3.2.1. *Amaç fonksiyonu hesapla* $(f(i))$
         2.3.2.2. *Yeni bir komşu hesapla* $(f(j))$

2.3.2.3. $Eğer\ \Delta E\ küçük\ 0\ ise\ kabul, değilse \min\left(1, e^{-\frac{\Delta E}{kT}}\right)$ $değerine\ \ göre\ kabul\ veya\ reddet$

2.3.2.4. $Görev$ İ$terasyon\ arttır$

2.3.2.5. $Görev$ İ$terasyon\ bitene\ kadar$

2.4. $Görevlerden\ gelen\ sonuçları\ karşılaştır\ ve\ en\ iyi$ çözümü al

3. $t = t * sıcaklık\_azalması$
4. $Her\ sıcaklıkta\ iterasyon\ sayısına\ göre\ sonuçlar\ değişir\ ve\ en\ iyi\ değeri\ kabul\ et$

$Bitir.$

2.5 Optimizasyon Test Fonksiyonları

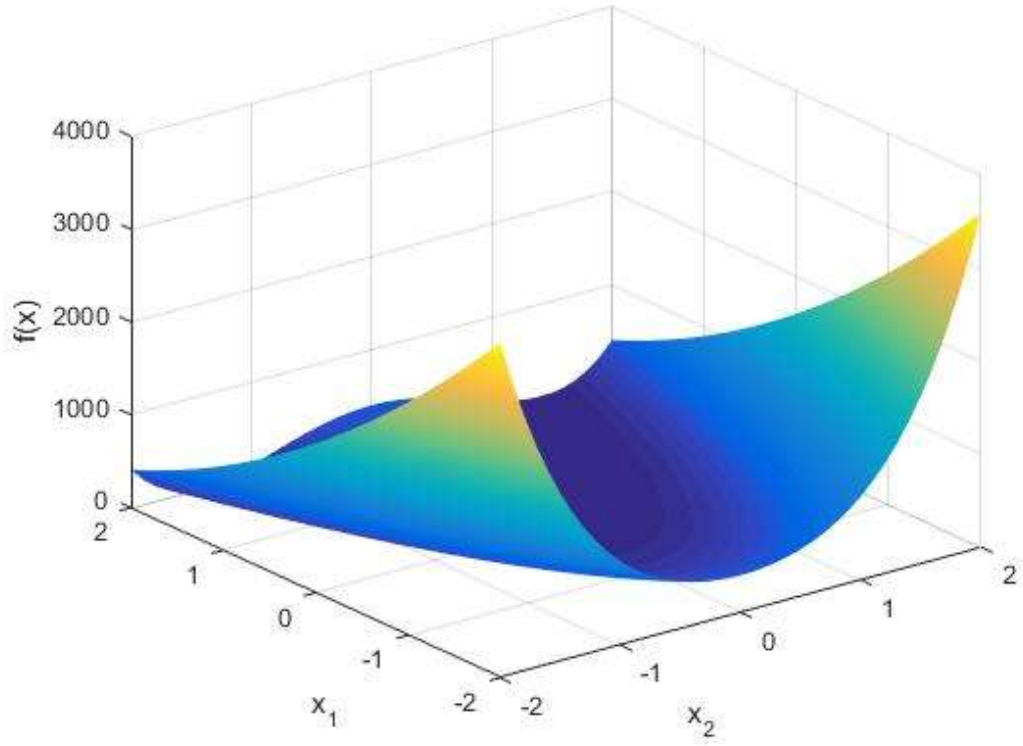
SA algoritmasının paralel optimizasyonunu test etmek için, optimizasyon testlerinde yaygın olarak kullanılan Rosenbrock ve Rastrigin fonksiyonları amaç fonksiyonları olarak seçilmiştir.

2.5.1 Rosenbrock Fonksiyonu

Rosenbrock fonksiyonu klasik optimizasyon performans test problemlerinden biridir. 1960 yılında Howard H. Rosenbrock tarafından bulunmuştur (Rosenbrock 1960). Evrensel minimum uzun, dar ve parabolik şekilli bir vadinin içindedir. Evrensel minimuma ulaşmak zor olduğundan, optimizasyon algoritmalarının performanslarını değerlendirilmesinde kullanılan bir fonksiyondur. Rosenbrock amaç fonksiyonu Eşitlik 2.9'da gösterilmiştir.

Fonksiyonda evrensel minimum dar ve parabolik bir vadide yer almaktadır. Ancak, vadi yapısından dolayı minimumu kolay olsa da evrensel minimuma yakınsaması zordur. Rosenbrock denklemi Şekil 2.11'de görüldüğü üzere vadi şeklinde olduğundan dolayı lineer programlama metotları ile bulacağı ilk minimumda kalacak ve evrensel minimuma ulaşamayacaktır. SA ile yerel minimumlardan kurtulacak ve evrensel minimuma yaklaşacaktır.

$$f(\boldsymbol{x}) = \sum_{i=1}^{n-1} 100\ (x_i^2 - \ x_{i+1})^2 + (x_i - 1)^2 \ \ -2.048 \leq x_i \leq 2.048 \qquad (2.9)$$



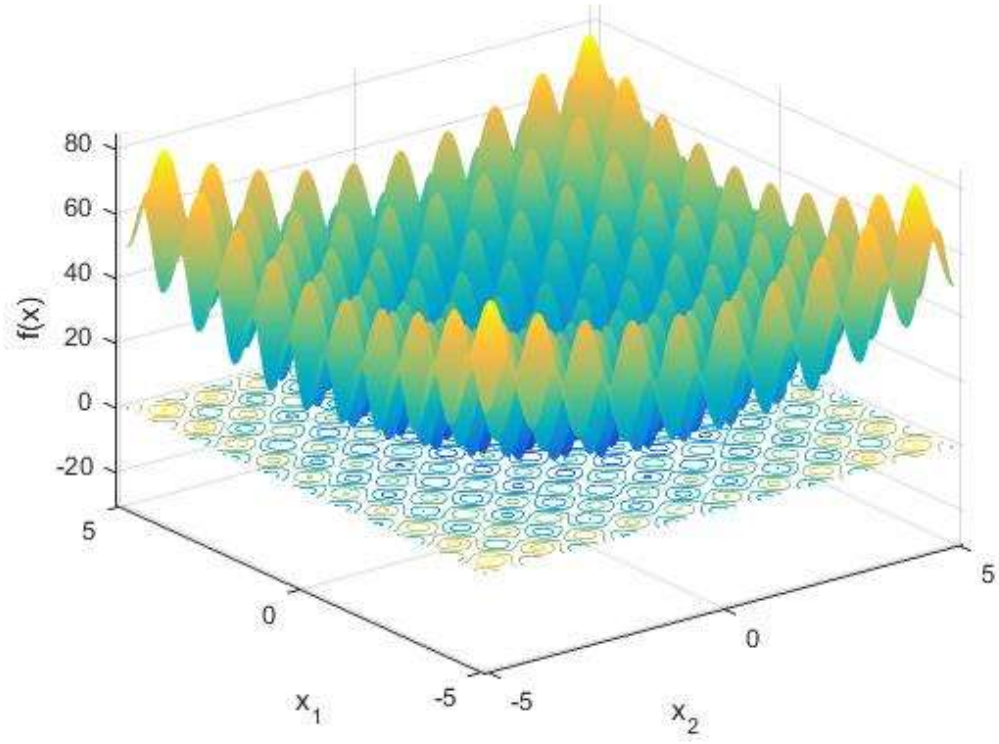
Şekil 2.11: Rosenbrock Fonksiyonu

2.5.2 Rastrigin Fonksiyonu

Rastrigin fonksiyonu optimizasyon problemlerinden biridir. Tek bir evrensel minimum ve kosinüs modülasyonu ilavesi ile çok fazla yerel minimuma sahiptir. Ve bu yerel minimumlar düzenli olarak dağıtılmıştır. Rastrigin amaç fonksiyonu Eşitlik 2.10'da gösterilmiştir.

Rastrigin fonksiyonunda Şekil 2.12'de görüldüğü gibi birçok noktada yerel minimum değeri bulunmaktadır. Yerel minimumların bu kadar çok olması bu fonksiyonu optimizasyon metotları için uygun bir amaç fonksiyonu haline getiriyor. SA algoritmamızı hem seri hem de paralel olarak bu fonksiyonu kullanarak test edilmiştir.

$$f(\boldsymbol{x}) = 10 \,.\, n + \sum_{i=1}^{n}(x_i^2 - 10 \,.\, \cos(2.\pi.x_i)) \quad -5.12 \leq x_i \leq 5.12 \quad (2.10)$$



Şekil 2.12: Rastrigin Fonksiyonu

2.6 Yazılım Tasarımı

Çalışmamız Visual Studio 2012 geliştirme ortamında .NET Framework 4.0 kütüphaneleri kullanılarak C# programlama dilinde yazılmıştır. Paralel programlama için System.Threading.Tasks kütüphanesi kullanılmıştır. Her sıcaklık aşamasında iterasyon sayısı, komşu hesaplaması için çap değeri sabit olarak verilmiştir.

Bilgisayarın içerdiği işlemci (iş parçacığı) sayısı bulunmuştur. Bunun için System.Enviroment kütüphanesinin ProcessorCount parametresi kullanılmıştır. Sonrasında sırayla işlemci-1'den sahip olduğu işlemci sayısına kadar arttırarak işlemler çalıştırıldı ve süreler toplanmıştır. System.Diagnostics.Process kütüphanesinin ProcessorAffinity parametresine yollanan değerlerle kullanılacak işlemci sayısı belirlenmiştir. Sıcaklıktaki iterasyon sayısı, işlemci (iş parçacığı)

sayısına bölünerek her bir işlemciye (iş parçacığına) bu bulunan yeni iterasyon sayısı kadar işlem yaptırılarak paralel olması sağlanmıştır.

Bir görev (task) dizisi oluşturularak bulunan yeni değerler ve diğer değerler parametre olarak hesaplama fonksiyonuna gönderilmiştir. Bu dizi her bir görev, her bir işlemcide çalıştırılan kod, sonrası ortaya çıkan sonuçları tutmaktadır. Bu dizide her bir işlemcinin farklı sürede çalışmayı bitirecek olmasından dolayı Görev kütüphanesinin ContinueWhenAll fonksiyonu kullanılmıştır. Bu fonksiyon içinde tüm görevlerden gelen sonuçlar karşılaştırılarak en iyi değer alınmıştır.

3. UYGULAMA SONUÇLARI

3.1 Test Ortamı

Bilgisayar-1: 2 adet 6 çekirdekli Intel Xeon Processor E5-2620 işlemcili 32 GB ram bulunan bir iş istasyonu

Bilgisayar-2: 1 adet 4 çekirdekli Intel I7-3770 işlemcili 4 GB ram bulunan bir bilgisayar

Kullanılan bilgisayarlardaki işlemciler Intel olduğu için işlemcilerde Hyper-Threading teknolojisi kullanılmaktadır. Hyper-Threading teknolojisinde, tek bir fiziksel işlemcide birden fazla komut zincirinin aynı anda işlenmesi ve böylelikle performansın artması sağlanmaktadır (Marr ve diğ., 2002). Hyper-Threading teknolojisiyle bir fiziksel işlemci, mantıksal olarak iki işlemciye bölünmektedir. Bu iki işlemci aynı fiziksel yonga üzerinde olmasına rağmen farklı komutları işleyebilir. Bu teknolojide mantıksal olarak oluşan işlemciler tek bir fiziksel işlemcinin kaynağını paylaşırlar. Bu sebeple bu teknoloji ile oluşturulan iki mantıksal işlemci işletim sistemi tarafından iki işlemci gibi görünse de iki fiziksel işlemci kadar performans sağlayamayacaklardır.

Bilgisayarların yazılımı test etmesi aşamasından bilgisayarların sanal çekirdeklerini kullanıma kapatarak ve açarak çekirdek sayıları arttırıldı. Böylelikle birinci bilgisayarda toplamda 12 çekirdek ve 24 sanal işlemci üzerinde, ikinci bilgisayarda toplamda 4 çekirdek ve 8 sanal işlemci kullanarak çekirdek sayılarının artışına göre yazılım sonuçları gözlemlenmiştir.

Test işlemleri sırasında bilgisayarların Hyper-Threading teknoloji kapatılarak önce çekirdeklerini sırayla devreye alarak, sonrasında ise Hyper-Threading teknolojisi açılarak tüm mantıksal işlemciler kullanılarak gerçekleştirilmiştir. İlk aşamada sadece bir çekirdek ile yazılım çalıştırılmış, daha sonra çekirdek sayısını birer arttırarak yazılım çalıştırılmasına devam edilmiştir ve tüm çekirdekler kullanıma açılmıştır. Çekirdek sayısı ve en iyi sonucun bulunduğu zamanların karşılaştırılması daha detaylı incelenecektir.

Uygulama esnasında problemin daha iyi bir çözüme ulaşması için 2 aşamalı işlem yapılmıştır. Birinci aşamada problem çözüme başlandığında daha geniş bir alanda çözüm aranması için başlangıç sıcaklığı ve çapı büyük tutulmuş ve en iyi çözüm bulunmuştur. Bu aşamaya İlk Hesaplama işlemi denilmektedir. İkinci aşamada çözümü daha iyi bir çözüme yaklaştırmak için ilk hesaplamaya göre daha düşük bir sıcaklık ve çap seçilmiştir. Bu aşamaya da Hassas Hesaplama denilmiştir. Her iki işlemde de aynı döngü sayıları ile işlem yapılmıştır. Her iki işlem için Boltzman sabiti 0,2 olarak alınmıştır.

Hassas Hesaplama, problemi ilk hesaplamada bulunan en iyi sonuç tekrar hesaplama yapılması ile bulunan sonuçları bulundurmaktadır. Hassas Hesaplamada sıcaklıkları ve komşu arama çaplarını ilk hesaplamadaki değerlerinin belli oranlarda azaltarak daha iyi sonuçlar alındığı gözlemlenmiştir.

SA algoritmasında performans, yöntemde kullanılan parametrelere büyük oranda bağlıdır. Bu sebeple parametrelerin uygun değerlerinin bulunması için uygulamanın ön çalışma kısmında bazı testler yapılmıştır.

Testler sonucunda, iki hesaplamalı uygulanılan algoritmada kaba hesaplama olarak bahsedilen ilk hesaplama için, başlangıç sıcaklığı için sabit bir değer yerine sonuç fonksiyonun rastgele 10000 sonuca verdiği ortalama değer alınmıştır. Böylelikle evrensel minimuma ulaşma şansının daha yüksek olduğu gözlemlenmiştir. Komşu seçiminde, amaç fonksiyonun sınırlarına göre bir çap belirlenmiş ve bu çap her sıcaklıkta 0,01 oranında azaltılarak, rastlantısal olarak seçilen komşuların yüksek sıcaklıklarda daha büyük bir alandan seçilmesi sağlanmıştır. Sıcaklık azaldıkça bu alan da azaldığı için evrensel minimuma yakın komşulardan seçimler yapılmaya başlanmıştır. İnce hesaplama olarak bahsedilen ikinci ve daha dar bir alanda çalıştırılan algoritma için sıcaklık kümesi, kaba hesaplamada elde edilen sıcaklık kümesinin *0,02* oranında azaltarak alınmıştır. İnce hesaplamada komşu seçiminde çap kümesi seçimini, yine kaba hesaplamada kullanılan komşu seçimindeki çap kümemi *0,25* oranında azaltarak elde edilmiştir.

Testlerde kullanılan işlemci sayısına göre çözüm süreleri karşılaştırılması yapılırken Karp-Flatt ölçütünden yararlanılmıştır.

Hızlanma katsayısı formülü,

$$S_p = \frac{T_1}{T_p} \tag{3.1}$$

Verimlilik formülü,

$$e = \frac{T_1}{p \times T_p} \tag{3.2}$$

Seri kısmının hesaplaması yani f formülü,

$$f = \frac{\frac{1}{S_p} - \frac{1}{p}}{1 - \frac{1}{p}} \tag{3.3}$$

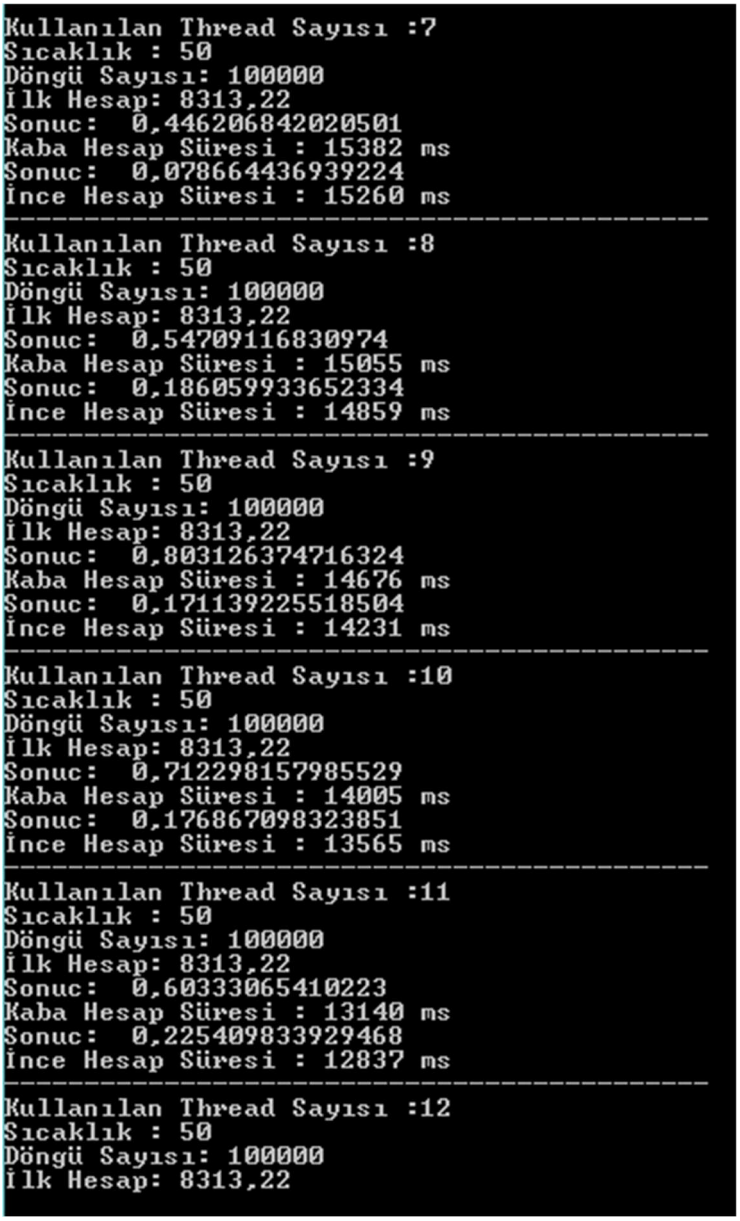
Ayrıca işlemci sayısı ve süre arasındaki kazanımı görmek amaçlı bir önceki işlemci sayısındaki çözüm süresi ile çözüm süresi işlemci sayısına bölünerek performans ölçümü yapılmıştır (Eşitlik 3.4). Bu sonuçla performans maliyet oranının hangi işlemci sayısında en üste çıktığı gözlemlenmiştir.

$$Yüzde\ Performans\ İyileşmesi = \frac{T_p - T_{p+1}}{T_p} \tag{3.4}$$

Şekil 3.13'te görülen Bilgisayar-1'den alınan ekran görüntüsüdür. Burada işlemci sayısı 12 olarak alınmış ve program başlatılmıştır. Program çalıştırılırken herhangi bir an Şekil 3.13'teki ekran görüntüsü alınıyor. Programın ürettiği değerler ve süreler şekilde görülmektedir. Program çıktısındaki bilgiler aşağıda açıklamıştır.

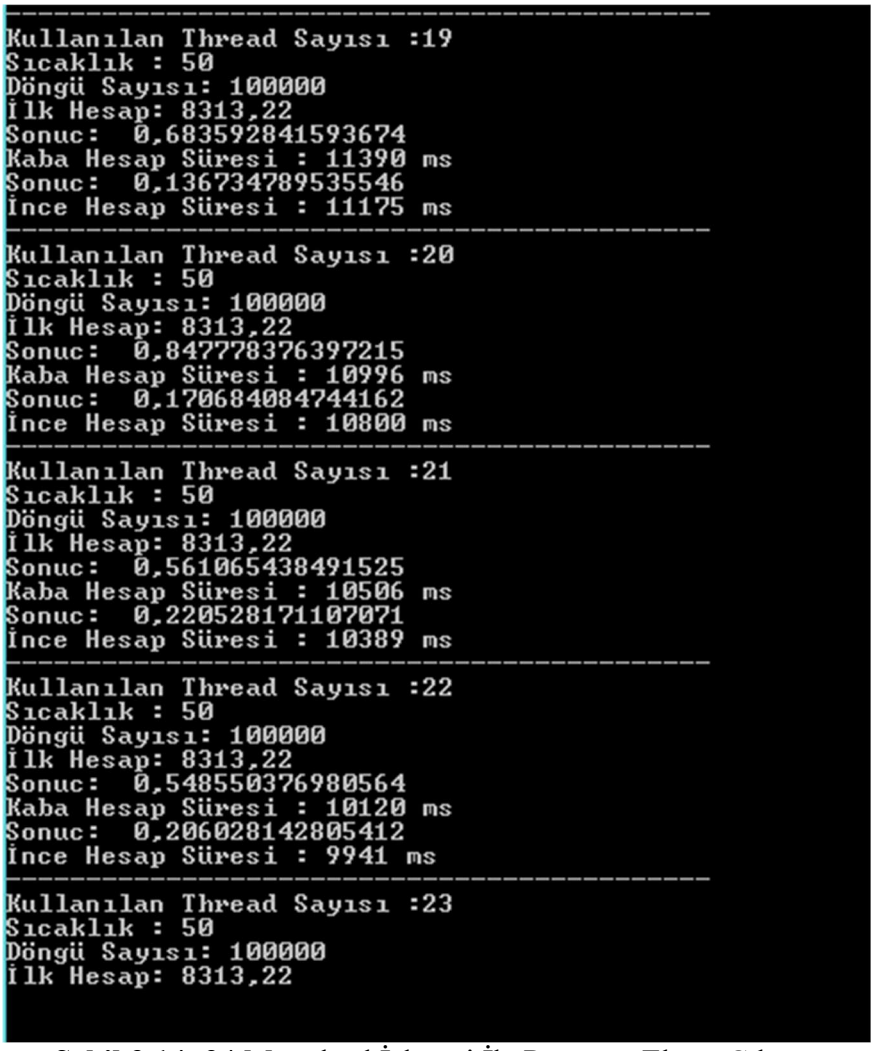
- **Kullanılan İş Parçacığı (Thread) Sayısı:** Hesap yapılırken kaç iş parçacığının işlem yapacağını göstermektedir.
- **Döngü Sayısı:** Kullanılan döngü sayısı göstermektedir.
- **İlk Hesap:** Amaç fonksiyonunun ilk sonucu göstermektedir.
- **Sonuç:** Amaç fonksiyonunda ulaşılan son sonucu göstermektedir.
- **Kaba Hesap Süresi:** İlk hesaplama süresinin kaç ms olduğunu göstermektedir.

- **İnce Hesap Süresi:** Hassas Hesaplama süresinin kaç ms olduğunu göstermektedir.



Şekil 3.13: 12 Çekirdek ile Program Ekran Çıktısı

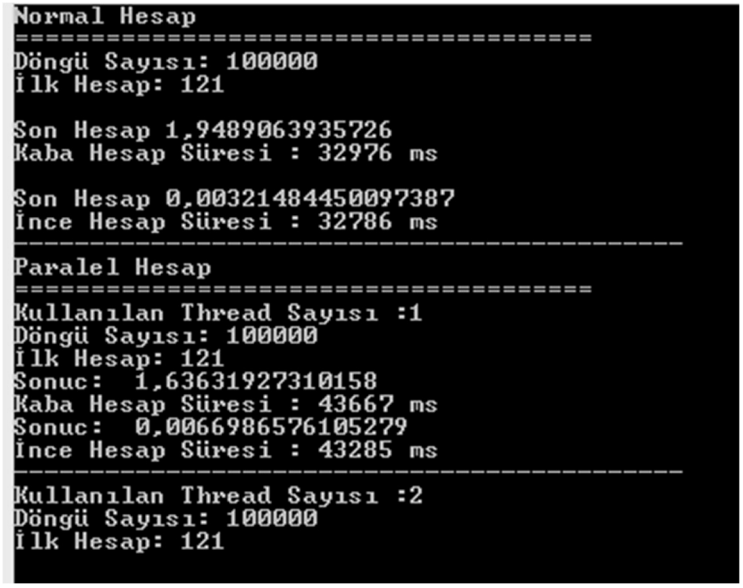
Şekil 3.14'te görülen Bilgisayar-1'den alınan ekran görüntüsüdür. Burada işlem iş parçacığı 24 olarak alınmış ve program başlatılmıştır.



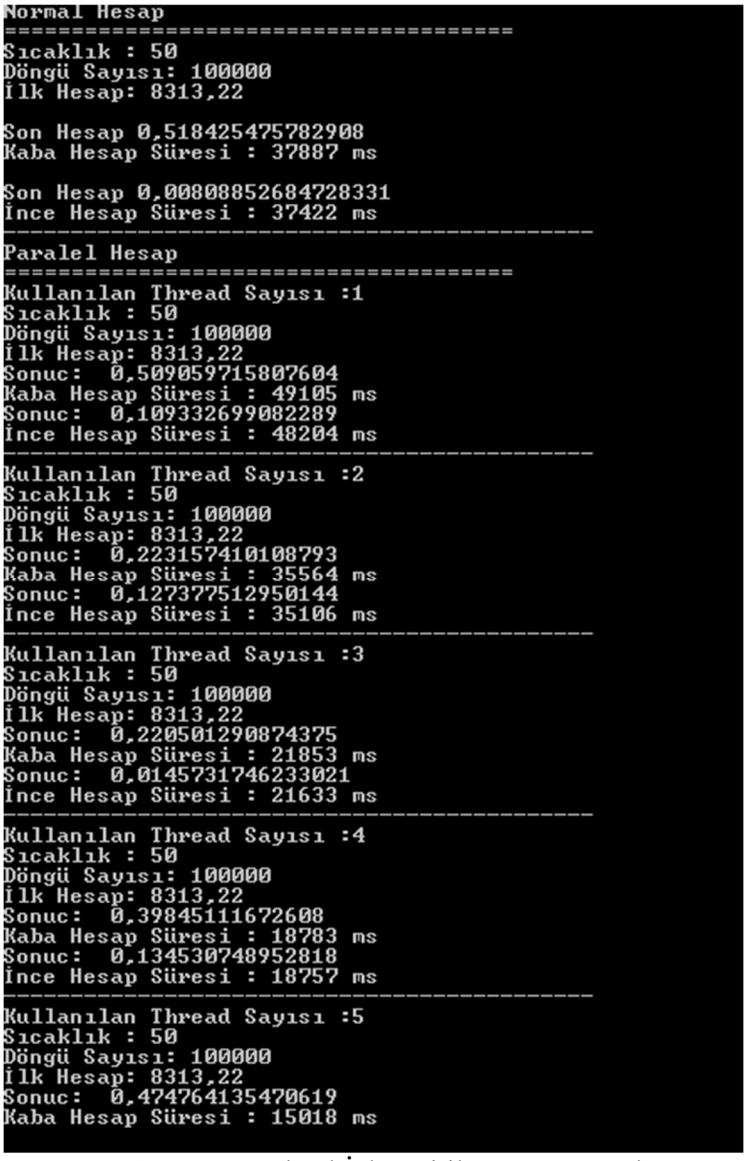
Şekil 3.14: 24 Mantıksal İşlemci İle Program Ekran Çıktısı

Şekil 3.15'te görülen Bilgisayar-2'den alınan ekran görüntüsüdür. Burada işlemci sayısı 4 olarak alınmış Hyper-Threading kapatılmış ve program başlatılmıştır.

Şekil 3.16'da görülen Bilgisayar-2'den alınan ekran görüntüsüdür. Burada Hyper-Threading açılmış ve iş parçacığı sayısı 8 olarak alınmıştır.



Şekil 3.15: 4 Çekirdek İle Ekran Çıktısı



Şekil 3.16: 8 Mantıksal İşlemci ile Program Çıktısı

3.2 Rosenbrock Fonksiyonu Çalışması

Rosenbrock fonksiyonunun amaç fonksiyonu olarak kullanıldığı, sırasıyla test bilgisayarlarında ilk olarak tek çekirdek ile hesaplama başlatılıyor. Sonrasında diğer çekirdeklere iş yükleri sırayla verilerek işlem süreleri gözlemleniyor. Veriler bu çalışmalar sonunda ortalama değerlere tekabül eden veri setlerinden alınmıştır.

3.2.1 Çalışma 1 : 4 Fiziksel Çekirdek

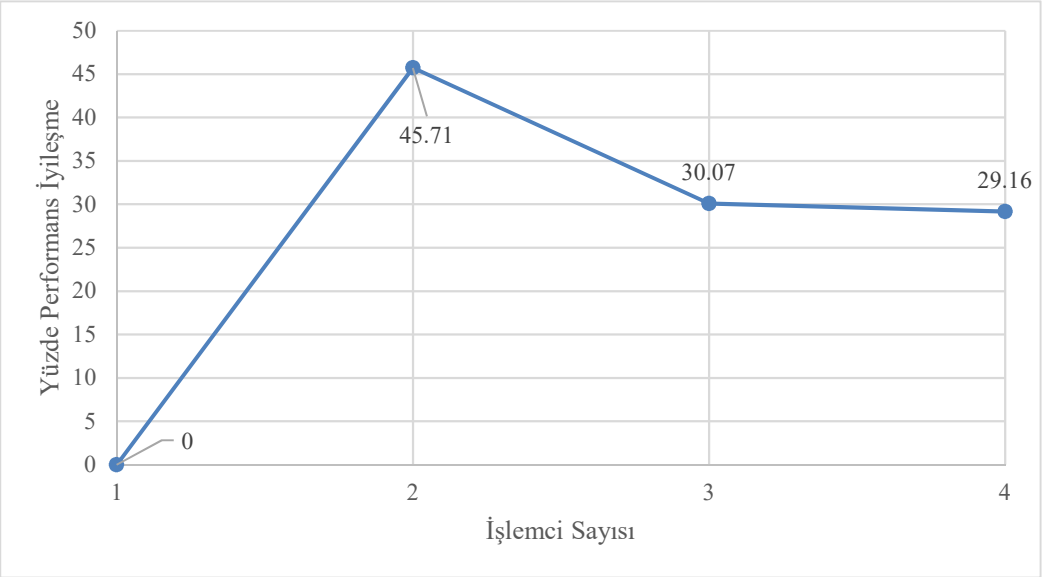
Çalışmanın bu uygulamasında test bilgisayarlarımızdan Bilgisayar-2 (4 fiziksel çekirdek 8 mantıksal işlemcili) kullanılmıştır. Bu test bilgisayarı ayarlarından Hyper-Threading (Mantıksal İşlemci) ayarları kapatılarak sadece 4 fiziksel işlemci kullanılarak uygulama çalıştırılmıştır. Tablo 3.5'te görüldüğü üzere hızlanma artmıştır.

Tablo 3.5: Rosenbrock Çalışma 1 Program Süreleri ve Değerler

İşlemci Sayısı	İlk Hesaplama (ms)	İnce Hesaplama (ms)	Hızlanma (kat)	Verimlilik (%)	f
1 işlemci	45669 ms	44874 ms	-	-	-
2 işlemci	24791 ms	24708 ms	1.84	0.92	0.08
3 işlemci	17334 ms	17768 ms	2.63	0.87	0.07
4 işlemci	12278 ms	12780 ms	3.72	0.93	0.02

Tablo 3.5'te görüldüğü üzere işlemci sayısı arttıkça işlem süresi azalmıştır. Verim olarak 3 işlemcinin kullanılması 0.87 verimlilik sağlarken 4 işlemcinin kullanılması 0.93 verimliliğe ulaştırmıştır. Seri kısmın etkisi de 4 işlemcinin çalıştırıldığı uygulamada gittikçe azalmıştır.

Şekil 3.17'de performans maliyet ölçümü yapıldığında en çok verimi ikinci işlemcinin devreye alındığında olduğu görülüyor. Fakat burada dikkat edilmesi gereken husus seri çalışmaya göre ilk paralel çalışmanın burada olduğu ve çözüm süresinde yaklaşık yarı yarıya düşüş burada gerçekleşmektedir. Bu sebeple yüzde iyileşme performansı ilk paraleleştirmenin olduğu ikinci işlemcinin devreye alındığı sırada %45,71 olarak görülmektedir.



Şekil 3.17: Rosenbrock Çalışma 1 Yüzde Performans İyileşmesi

3.2.2 Çalışma 2: 8 Mantıksal İşlemci

Çalışmanın bu uygulamasında test bilgisayarlarımızdan Bilgisayar-2 (4 fiziksel çekirdek 8 mantıksal işlemcili) kullanılmıştır. Bu test bilgisayarı ayarlarından Hyper-Threading (Mantıksal İşlemci) ayarları açılarak 8 mantıksal işlemci kullanılarak uygulama çalıştırılmıştır. Tablo 3.6'da görüldüğü üzere hızlanma artmıştır. Fakat burada hızlanmaya baktığımız zaman görüyoruz ki kullanılan işlemci sayısıyla arasındaki oran bir üstteki sadece fiziksel çekirdekler kullanılarak elde edilen sonuçlardaki kadar birbirine yakın değildir. Buradaki sebep mantıksal işlemci kullanımından kaynaklanmaktadır.

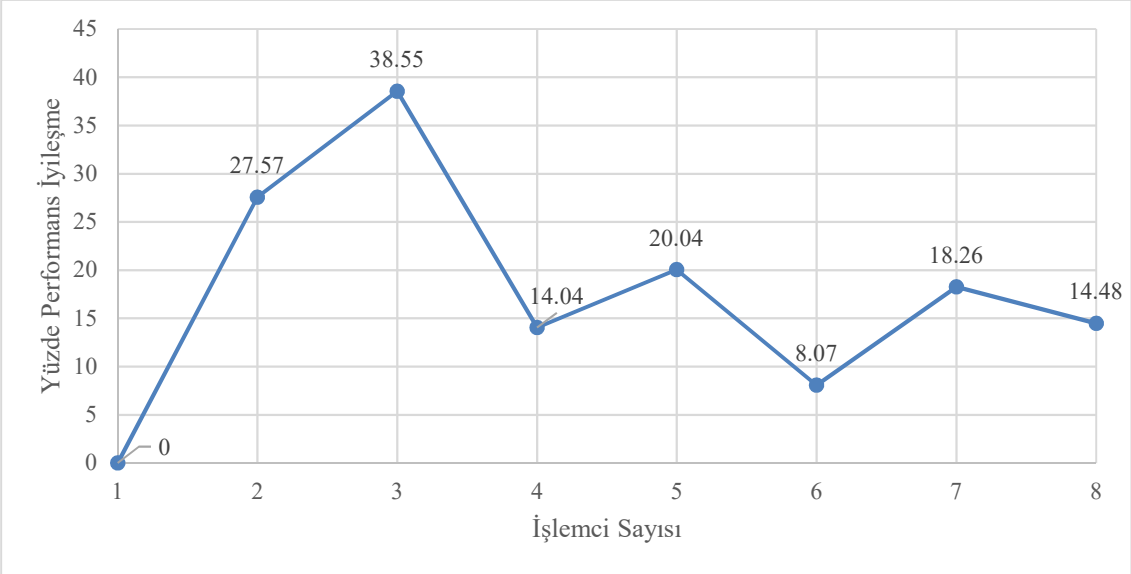
Tablo 3.5 ve Tablo 3.6 karşılaştırıldığında, işlemci sayısı arttıkça hızlanma oranı Tablo 3.1'de işlemci sayısıyla daha iyi bir orana sahiptir. Fakat Tablo 3.5 tüm kaynakların kullanıldığı yani 4 işlemci ile elde edilen süre ile Tablo 3.6'da tüm kaynakların kullanıldığı 8 işlemci ile elde edilen süre arasında fark olduğu

gözlenmektedir. Bu durum Hyper-Threading teknolojisinin fiziksel olarak çekirdek artırımı kadar olmasa da paralel işlemler için uygunluğunu göstermektedir.

Tablo 3.6: Rosenbrock Çalışma 2 Program Süreleri ve Değerler

İşlemci Sayısı	İlk Hesaplama (ms)	İnce Hesaplama (ms)	Hızlanma (kat)	Verimlilik (%)	f
1 İşlemci	49105	48204	-	-	-
2 İşlemci	35564	35106	1.38	0.69	0.44
3 İşlemci	21853	21633	2.24	0.74	0.16
4 İşlemci	18783	18757	2.61	0.65	0.17
5 İşlemci	15018	15075	3.26	0.65	0.13
6 İşlemci	13805	13566	3.55	0.59	0.13
7 İşlemci	11284	11081	4.35	0.62	0.10
8 İşlemci	9649	9870	5.08	0.63	0.08

Şekil 3.18'de görüldüğü gibi performans maliyet hesabıyla bakıldığında mantıksal işlemci kullanımı biraz karmaşıklık göstermektedir. Bu durum fiziksel olarak devreye giren bir sonraki işlemci sebebiyle olmaktadır. Aslında işlemci sayısı üçe çıktığında ikinci fiziksel çekirdeğinde devreye girmesinden performans maliyet oranı yükselmektedir.



Şekil 3.18: Rosenbrock Çalışma 2 Yüzde Performans İyileşmesi

3.2.3 Çalışma 3: 12 Fiziksel Çekirdek

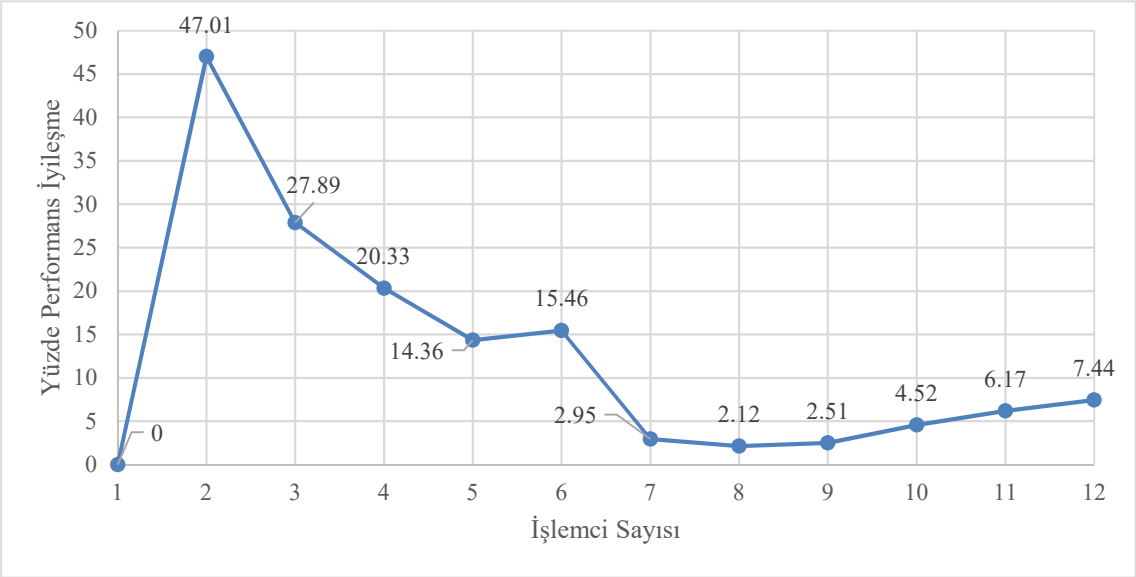
Çalışmanın bu uygulamasında test bilgisayarlarımızdan Bilgisayar-1 (12 fiziksel çekirdek 24 mantıksal işlemcili) kullanılmıştır. Bu test bilgisayarı ayarlarından Hyper-Threading (Mantıksal İşlemci) ayarları kapatılarak sadece 12 fiziksel çekirdek kullanılarak uygulama çalıştırılmıştır.

Bilgisayar-1 üzerindeki işlem süreleri ve hesaplamalar Tablo 3.7'de verilmiştir. Bu tabloda f yani paralelleştirilemeyen kod parçası oranı çok değişmediği için paralelleştirme başarılı bir şekilde gerçekleştirilmiştir. Verimlilik değeri en iyi sonucu ikinci işlemcinin devreye girmesiyle alınmıştır. Hızlanma oranına bakıldığında ise 12 işlemci sonucunda 5,91 kat artış olduğu görülmektedir. Bu da paraleleştirilemeyen kısımdan ve ortak kullanılan alanlardan dolayı on ikiye sadece teoride ulaşabilmektedir.

Tablo 3.7: Rosenbrock Çalışma 3 Program Süreleri ve Değerler

İşlemci Sayısı	İlk Hesaplama (ms)	İnce Hesaplama (ms)	Hızlanma (kat)	Verimlilik (%)	f
1 İşlemci	71910	70709	-	-	-
2 İşlemci	38104	37416	1.88	0.94	0.05
3 İşlemci	27475	26903	2.61	0.87	0.07
4 İşlemci	21893	21453	3.28	0.82	0.07
5 İşlemci	18749	18354	3.83	0.76	0.07
6 İşlemci	15850	15669	4.53	0.75	0.06
7 İşlemci	15382	15263	4.67	0.66	0.08
8 İşlemci	15055	14859	4.77	0.59	0.09
9 İşlemci	14676	14232	4.89	0.54	0.10
10 İşlemci	14005	13566	5.13	0.51	0.10
11 İşlemci	13140	12837	5.47	0.49	0.10
12 İşlemci	12162	12193	5.91	0.49	0.09

Şekil 3.19'da görüldüğü gibi performans maliyet formülüyle en iyi çözüm kıyaslaması ikinci işlemci için bulunmaktadır. Bu değer ilk paralelleştirme değeri olduğu için bunu haricinde gözlem yaptığımızda oranın giderek azaldığını gözlemlemekteyiz.



Şekil 3.19: Rosenbrock Çalışma 3 Yüzde Performans İyileşmesi

3.2.4 Çalışma 4: 24 Mantıksal İşlemci

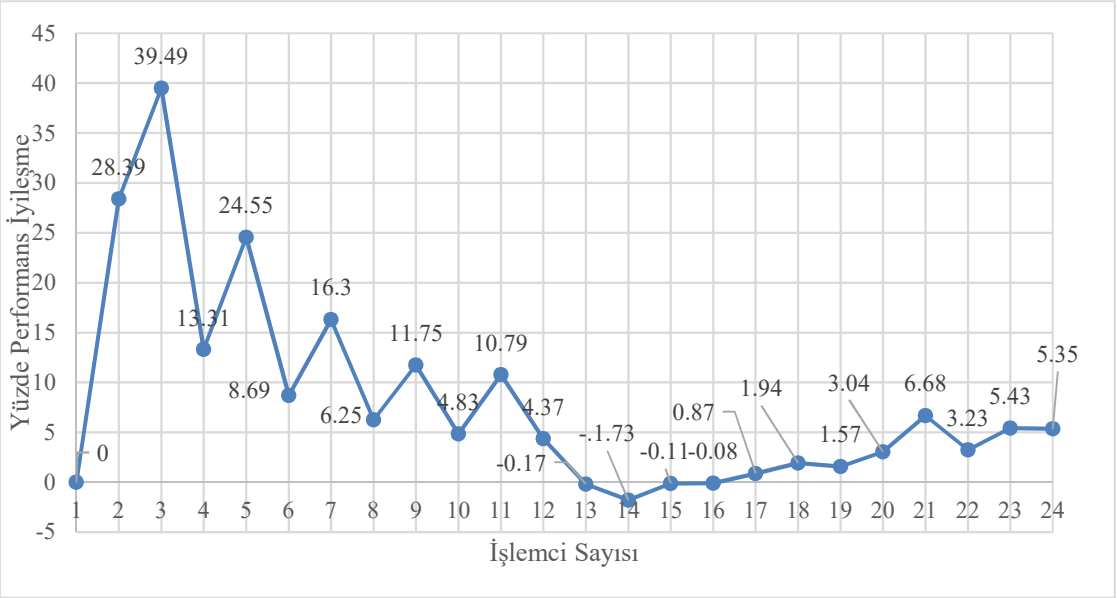
Çalışmanın bu uygulamasında test bilgisayarlarımızdan Bilgisayar-1 (12 fiziksel çekirdek 24 mantıksal işlemcili) kullanılmıştır. Bu test bilgisayarı ayarlarından Hyper-Threading (Mantıksal İşlemci) ayarları açılarak 24 mantıksal işlemci kullanılarak uygulama çalıştırılmıştır.

Tablo 3.8'te görüldüğü üzere seri kısmın etkisi üçüncü işlemci sonrasında giderek azalarak paralel olarak çalışan kısımlardaki etkisi azalmaktadır. Hızlanma Çalışma 3'e göre başlarda daha az olmaktadır. Bunun sebebi Hyper-Threading teknolojisidir. Tek sayılı işlemci sayıları aslında fiziksel olarak bir sonraki işlemcinin devreye girmesinden dolayı daha fazla verimlilik göstermektedir

Tablo 3.8: Rosenbrock Çalışma 4 Program Süreleri ve Değerler

İşlemci Sayısı	İlk Hesaplama (ms)	İnce Hesaplama (ms)	Hızlanma (kat)	Verimlilik (%)	f
1 işlemci	80320	78560	-	-	-
2 işlemci	57517	57045	1.39	0.69	0.43
3 işlemci	34799	34410	2.30	0.76	0.15
4 işlemci	30167	29824	2.66	0.66	0.17
5 işlemci	22759	22545	3.52	0.70	0.10
6 işlemci	20779	20638	3.86	0.64	0.11
7 işlemci	17391	17218	4.61	0.65	0.09
8 işlemci	16303	16086	4.92	0.61	0.09
9 işlemci	14387	14205	5.58	0.62	0.08
10 işlemci	13691	13655	5.86	0.58	0.08
11 işlemci	12213	12068	6.57	0.59	0.07
12 işlemci	11679	11569	6.87	0.57	0.07
13 işlemci	11700	11583	6.86	0.52	0.07
14 işlemci	11909	11802	6.74	0.48	0.08
15 işlemci	11923	11840	6.73	0.44	0.09
16 işlemci	11933	11790	6.73	0.42	0.09
17 işlemci	11829	11622	6.79	0.39	0.09
18 işlemci	11599	11438	6.92	0.38	0.09
19 işlemci	11416	11355	7.03	0.37	0.09
20 işlemci	11068	10835	7.25	0.36	0.09
21 işlemci	10328	10304	7.77	0.37	0.09
22 işlemci	9994	9868	8.03	0.36	0.08
23 işlemci	9451	9352	8.49	0.36	0.08
24 işlemci	8945	8886	8.97	0.37	0.07

Şekil 3.20'de görüldüğü gibi maliyet performans yüzdeleri belli değerlerde eksiye düşmektedir. On ikinci işlemci ile on beşinci işlemci arası bir önceki sürelere göre daha yavaş çalışması göstermektedir ki işlemci sayısı artışı her zaman bir artış sağlamamaktadır. Burada dikkat edilmesi gereken işlemci sayısıyla değerlendirme sürecinde maliyet hesabının yani maksimum verimliliğin elde edilmesi olmalıdır.



Şekil 3.20: Rosenbrock Çalışma 4 Yüzde Performans İyileşmesi

3.3 Rastrigin Fonksiyonu Çalışması

Rastrigin fonksiyonunun amaç fonksiyonu olarak kullanıldığı, sırasıyla test bilgisayarlarında ilk olarak tek çekirdek ile hesaplama başlatılıyor. Sonrasında diğer çekirdeklere iş yükleri sırayla verilerek işlem süreleri gözlemleniyor. Veriler bu çalışmalar sonunda ortalama değerlere karşılık gelen veri setlerinden alınmıştır.

3.3.1 Çalışma 1 : 4 Fiziksel Çekirdek

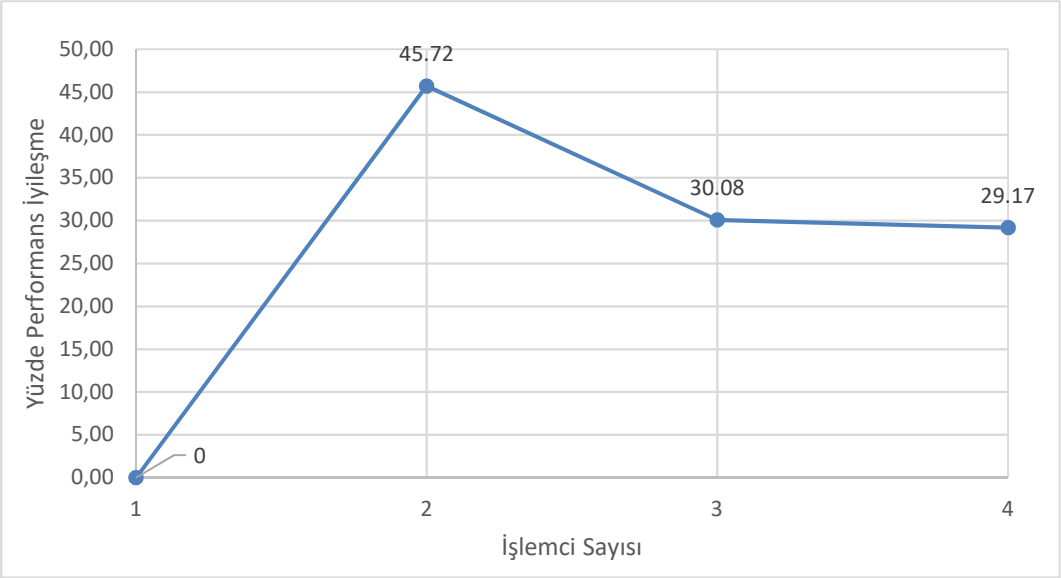
Çalışmanın bu uygulamasında test bilgisayarlarımızdan Bilgisayar-2 (4 fiziksel çekirdek 8 mantıksal işlemcili) kullanılmıştır. Bu test bilgisayarı ayarlarından Hyper-Threading (Mantıksal İşlemci) ayarları kapatılarak sadece 4 fiziksel işlemci kullanılarak uygulama çalıştırılmıştır.

Tablo 3.9'da görüldüğü gibi performansımız işlemci sayılarına göre artmakta ve paralel optimizasyonumuz çalışmaktadır. Hızlanma oranları ve verimlilik ele alındığında 4 işlemci ile maksimum verimliliğe ulaşıldığı gözlenmektedir.

Şekil 3.21'de görüldüğü üzere süreler üzerinden yapılan maliyet performans yüzdesi hesabımızda bir önceki çalışma süreleri değerlendirildiğinde 2 işlemciyi en yüksek performans olarak görmekteyiz. Bunun sebeplerinden biri seri kısmın yani paralelleştirilemeyen kısmın etkisinin burada daha az hesaba yansımasıdır.

Tablo 3.9: Rastrigin Çalışma 1 Program Süreleri ve Değerler

İşlemci Sayısı	**İlk Hesaplama (ms)**	**İnce Hesaplama (ms)**	**Hızlanma (kat)**	**Verimlilik (%)**	**f**
1 işlemci	43667	43285	-	-	-
2 işlemci	24091	23387	1.81	0.91	0.10
3 işlemci	17394	16701	2.51	0.84	0.10
4 işlemci	11991	11953	3.64	0.91	0.03



Şekil 3.21: Rastrigin Çalışma 1 Yüzde Performans İyileşmesi

3.3.2 Çalışma 2: 8 Mantıksal İşlemci

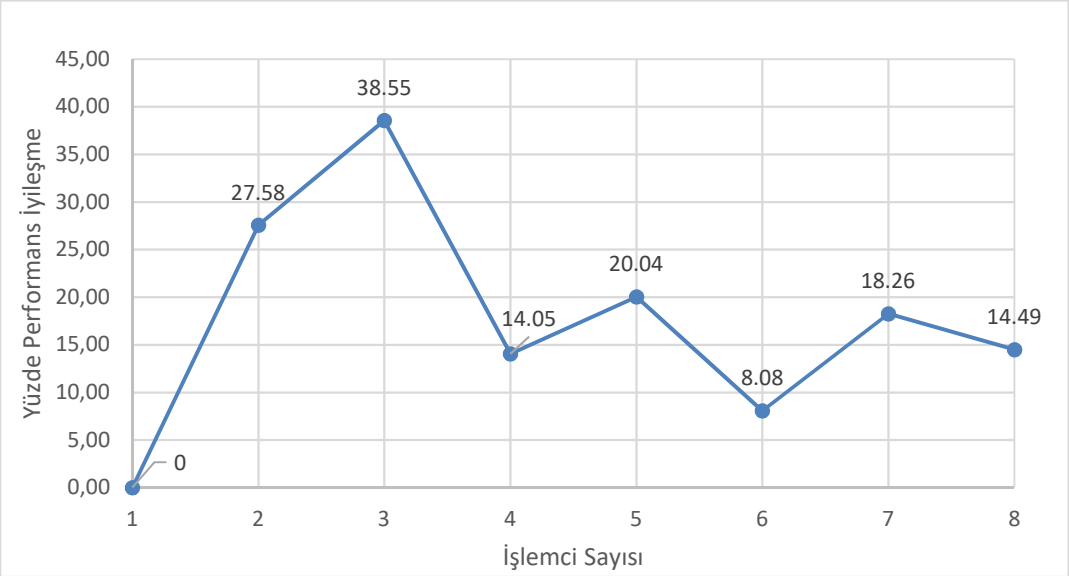
Çalışmanın bu uygulamasında test bilgisayarlarımızdan Bilgisayar-2 (4 fiziksel çekirdek 8 mantıksal işlemcili) kullanılmıştır. Bu test bilgisayarı ayarlarından Hyper-Threading (Mantıksal İşlemci) ayarları açılarak 8 mantıksal işlemci kullanılarak uygulama çalıştırılmıştır.

Tablo 3.10'da görüldüğü işlem sürelerimiz giderek azalmakta hızlanma sağlanmaktadır. Çalışma 1 ile kıyaslama yapılırsa tüm işlemcilerin çalıştırıldığındaki çözüm süresi 11991 ms den 8823 ms ye düşmektedir. Bu da Hyper-Threading teknolojisinin hızlanmaya olan katkısını bize göstermektedir.

Şekil 3.22'de görüldüğü gibi performans maliyet hesabıyla bakıldığında mantıksal işlemci kullanımı biraz karmaşıklık göstermektedir. Bu durum fiziksel olarak devreye giren bir sonraki işlemci sebebiyle olmaktadır. Aslında işlemci sayısı üçe çıktığında ikinci fiziksel çekirdeğinde devreye girmesinden performans maliyet oranı yükselmektedir.

Tablo 3.10: Rastrigin Çalışma 2 Program Süreleri ve Değerler

İşlemci Sayısı	İlk Hesaplama (ms)	İnce Hesaplama (ms)	Hızlanma (kat)	Verimlilik (%)	f
1 işlemci	43946	43728	-	-	-
2 işlemci	31657	31621	1.39	0.69	0.44
3 işlemci	19605	19543	2.24	0.75	0.17
4 işlemci	17137	17158	2.56	0.64	0.19
5 işlemci	13728	13650	3.20	0.64	0.14
6 işlemci	12231	12190	3.59	0.60	0.13
7 işlemci	9878	9875	4.45	0.64	0.10
8 işlemci	8823	8464	4.98	0.62	0.09



Şekil 3.22: Rastrigin Çalışma 2 Yüzde Performans İyileşmesi

3.3.3 Çalışma 3: 12 Fiziksel Çekirdek

Çalışmanın bu uygulamasında test bilgisayarlarımızdan Bilgisayar-1 (12 fiziksel çekirdek 24 mantıksal işlemcili) kullanılmıştır. Bu test bilgisayarı ayarlarından Hyper-Threading (Mantıksal İşlemci) ayarları kapatılarak sadece 12 fiziksel çekirdek kullanılarak uygulama çalıştırılmıştır.

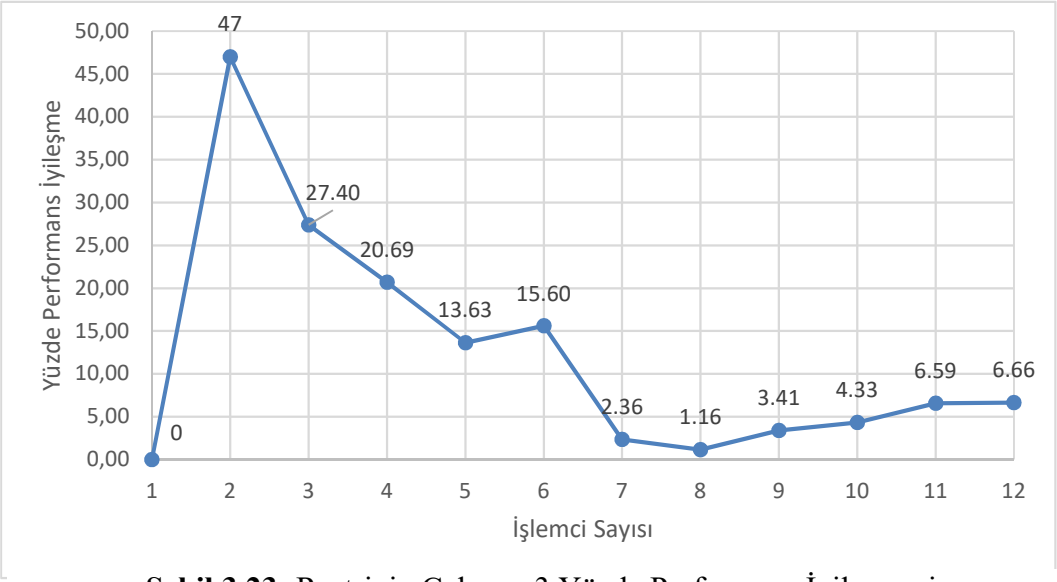
Bilgisayar-1 üzerindeki işlem süreleri ve hesaplamalar Tablo 3.11'de verilmiştir. Bu tabloda f yani paralelleştirilemeyen kod parçası oranı çok değişmediği

için paralelleştirme başarılı bir şekilde gerçekleştirilmiştir. Verimlilik değeri en iyi sonucu ikinci işlemcinin devreye girmesiyle alınmıştır. Hızlanma oranına bakıldığında ise on iki işlemci sonucunda 5,91 kat artış olduğu görülmektedir. Bu da paralelleştirilemeyen kısımdan ve ortak kullanılan alanlardan dolayı 12'ye sadece teoride ulaşabilmektedir.

Tablo 3.11: Rastrigin Çalışma 3 Program Süreleri ve Değerler

İşlemci Sayısı	İlk Hesaplama (ms)	İnce Hesaplama (ms)	Hızlanma (kat)	Verimlilik (%)	f
1 işlemci	67877	67453	-	-	-
2 işlemci	35977	35602	1.89	0.94	0.06
3 işlemci	26120	25664	2.60	0.87	0.08
4 işlemci	20717	20576	3.28	0.82	0.07
5 işlemci	17894	17587	3.79	0.76	0.08
6 işlemci	15103	15019	4.49	0.75	0.07
7 işlemci	14746	14705	4.60	0.66	0.09
8 işlemci	14575	14218	4.66	0.58	0.10
9 işlemci	14078	14080	4.82	0.54	0.11
10 işlemci	13468	13106	5.04	0.50	0.11
11 işlemci	12581	12430	5.40	0.49	0.10
12 işlemci	11743	12155	5.78	0.48	0.10

Şekil 3.23'te görüldüğü gibi performans maliyet formülüyle en iyi çözüm kıyaslaması ikinci işlemci için bulunmaktadır. Bu değer ilk paralelleştirme değeri olduğu için bunu haricinde gözlem yaptığımızda oranın giderek azaldığını gözlemlenmektedir.



Şekil 3.23: Rastrigin Çalışma 3 Yüzde Performans İyileşmesi

3.3.4 Çalışma 4: 24 Mantıksal İşlemci

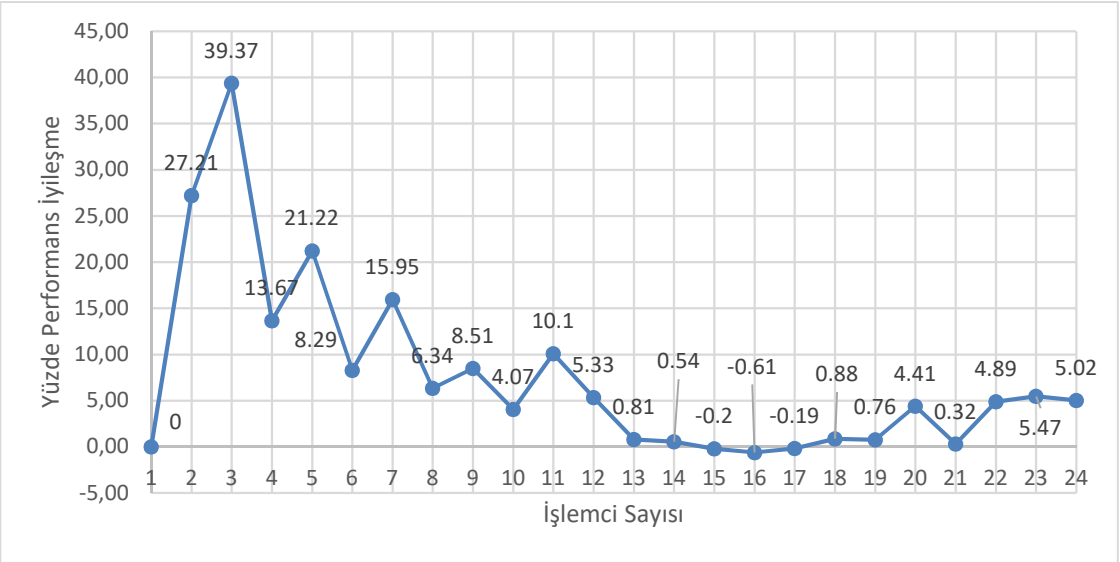
Çalışmanın bu uygulamasında test bilgisayarlarımızdan Bilgisayar-1 (12 fiziksel çekirdek 24 mantıksal işlemcili) kullanılmıştır. Bu test bilgisayarı ayarlarından Hyper-Threading (Mantıksal İşlemci) ayarları açılarak 24 mantıksal işlemci kullanılarak uygulama çalıştırılmıştır.

Tablo 3.12'de görüldüğü üzere seri kısmın etkisi Üçüncü işlemci sonrasında giderek azalarak paralel olarak çalışan kısımlardaki etkisi azalmaktadır. Hızlanma Çalışma 3'e göre başlarda daha az olmaktadır. Bunun sebebi Hyper-Threading teknolojisidir. Tek sayılı işlemci sayıları aslında fiziksel olarak bir sonraki işlemcinin devreye girmesinden dolayı daha fazla verimlilik göstermektedir.

Tablo 3.12: Rastrigin Çalışma 4 Program Süreleri ve Değerler

İşlemci Sayısı	İlk Hesaplama (ms)	İnce Hesaplama (ms)	Hızlanma (kat)	Verimlilik (%)	f
1 işlemci	68682	68184	-	-	-
2 işlemci	49996	49790	1.37	0.69	0.46
3 işlemci	30311	30111	2.27	0.76	0.16
4 işlemci	26168	26350	2.62	0.66	0.17
5 işlemci	20615	20642	3.33	0.67	0.13
6 işlemci	18907	18987	3.63	0.61	0.13
7 işlemci	15892	15961	4.32	0.62	0.10
8 işlemci	14885	14967	4.61	0.58	0.10
9 işlemci	13619	13517	5.04	0.56	0.10
10 işlemci	13065	12958	5.26	0.53	0.10
11 işlemci	11745	11522	5.85	0.53	0.09
12 işlemci	11119	11119	6.18	0.51	0.09
13 işlemci	11029	11023	6.23	0.48	0.09
14 işlemci	10969	10936	6.26	0.45	0.10
15 işlemci	10991	10941	6.25	0.42	0.10
16 işlemci	11058	10995	6.21	0.39	0.11
17 işlemci	11079	11034	6.20	0.36	0.11
18 işlemci	10982	10931	6.25	0.35	0.11
19 işlemci	10898	10726	6.30	0.33	0.11
20 işlemci	10417	10396	6.59	0.33	0.11
21 işlemci	10384	10170	6.61	0.31	0.11
22 işlemci	9876	9791	6.95	0.32	0.10
23 işlemci	9336	9311	7.36	0.32	0.10
24 işlemci	8867	8922	7.75	0.32	0.09

Şekil 3.24'te görüldüğü gibi maliyet performans yüzdeleri belli değerlerde eksiye düşmektedir. On ikinci ile on beşinci işlemci arası bir önceki sürelere göre daha yavaş çalışması göstermektedir ki işlemci sayısı artışı her zaman bir artış sağlamamaktadır. Burada dikkat edilmesi gereken işlemci sayısıyla değerlendirme sürecinde maliyet hesabının yani maksimum verimliliğin elde edilmesi olmalıdır.



Şekil 3.24: Rastrigin Çalışma 4 Yüzde Performans İyileşmesi

4. SONUÇ VE ÖNERİLER

Paralelleştirme yoluyla artan bilgisayar performansı ile ilgili bazı temel tanım ve teorileri bu tez çalışmasında uygulama ile çözüm süreleri ve işlemciler arası performans ile açıkladım. İki farklı test bilgisayarında sonuçlar karşılaştırılmış ve gösterilmiştir. İntel işlemcilerin kullandığı teknoloji Hyper-Threading teknolojisi de bu çalışma içinde incelenmiş ve belirli bir hızlanma sağlasa da çekirdeklerin arttırılması kadar bir hızlanma sağlamadığı gözlemlenmiştir.

Rosenbrock Çalışma 1'de çalıştırılan sonuçları Tablo 3.5'te verilen verimlilik 4 işlemci ile maksimum seviyeye ulaşmıştır. Yüzde performans iyileşmesi grafiği Şekil 3.17'de verilmiştir. Şekil 3.17'de verilen grafiğe göre en yüksek performans iyileşmesi ikinci işlemcinin çözümünde görülmektedir. f değeri incelendiğinde ise seri kısmın giderek azaldığı ve çözüme etkisinin azaldığı görülmektedir.

Rosenbrock Çalışma 2'de Hyper-Threading teknolojisi açılarak yani mantıksal işlemciler devreye alınarak elde edilen çözüm süreleri ve değerlendirmeler Tablo 3.6'da verilmiştir. Verimlilik değerinin sayısal olarak problemin değerlendirilmesinde bir hızlanma göstergesi olsa da tam olarak ifade edilememektedir. Yüzde performans iyileşmesi grafiği Şekil 3.18'de verilmiştir. Şekilde görülmektedir ki üç işlemcinin devreye alınmasıyla iyileşme %38,55'e ulaşmıştır. Bunun nedeni üçüncü işlemcinin aslında fiziksel olarak ikinci çekirdeğin devreye girmesiyle elde edilmesidir. Problemin paralelleştirilemeyen kısmının değer olarak ölçüldüğü Tablo 3.6'da görüldüğü gibi Hyper-Threading teknolojisinden dolayı ikinci işlemcinin değeri 0.44 gibi yüksek bir değerde çıkmaktadır. Rosenbrock Çalışma 1 ve Çalışma 2 karşılaştırılmasını son elde edilen süreler üzerinden yaparsak Çalışma 1'de tüm kaynakların kullanılması ile çözüm 12278 ms, Çalışma 2'de 9649 ms de ulaşılmıştır. Burada Hyper-Threading teknolojisinin belli bir oranda tüm kaynakların kullanılmasında hızlanma sağlamıştır. f değeri incelendiğinde ise giderek azalarak probleme seri kısmın katkısının azaldığı görülmektedir.

Rosenbrock Çalışma 3'te elde edilen değerler Tablo 3.7'de verilmiştir. Burada f değer incelendiğinde altıncı işlemci sonrasında arttığı görülmektedir. Bu durum işlemci sayısının artmasıyla beraber artan ek yük olduğunu göstermektedir. Yüzde performans iyileşmesi Şekil 3.19'da görüldüğü gibi altıncı işlemcinin devreye

alınması ile bir önceki veriye göre bir artarak daha iyi bir sonuç elde edildiği görülmektedir. Bu sonuç altıncı işlemcinin, beşinci işlemciye göre daha fazla kazanç getirdiğini göstermektedir. Yedinci işlemciye geçişte ise yüzde performans iyileştirmesinde büyük bir düşüş olduğu gözlemlenmektedir. Tekil olarak yedinci işlemcinin çözüm hızına katkısı bir önceki işlemcinin katkısına göre %2,95'tir. Sonraki işlemci artışlarında yüzde performans iyileşmesi %10'nun altında kalmaktadır. Burada varılan sonuç ile altıncı işlemci sonrasında hızlanma daha düşük gerçekleştiği için optimum işlemci artışı değerine ulaşıldığı söylenebilir.

Rosenbrock Çalışma 4'te Hyper-Threading teknolojisi açılarak program çalıştırılmıştır ve elde edilen değerler Tablo 3.8'te verilmiştir. On üçüncü işlemci itibariyle f değerinin arttığı aynı zamanda da hızlanmanın on ikinci işlemciye göre artmadığı hatta azaldığı Tablo 3.8 ve yüzde performans iyileşmesinin verildiği Şekil 3.20'de görülmektedir. f değerinin arttığı burada işlemcilerin artmasıyla ek yükün ortaya çıktığını göstermektedir. Çalışma 3 ve Çalışma 4 çalışmalarında tam performans (tüm işlemcilerin çalışması) ile elde edilen değerler kıyaslandığında 12162 ms ve 8945 ms elde edilmiştir. Bu durum Hyper-Threading teknolojisinin hızlanmadaki katkısını göstermektedir. Benzer sonuçlar ve değerler Rastrigin fonksiyonu üzerindeki sonuçlarda da elde edilmiştir.

Test sonuçlarıyla görülmüştür ki paralelleştirme işlem gecikmelerini yani sonucu hesaplamak için gereken toplam süreyi azaltabilir, hesaplanma hızını yani verimi arttırabilir ve hesaplamada kullanılan güç tüketim miktarını azaltabilir. Fakat şu unutulmamalıdır ki performans optimize edilirken, bir faktördeki hızlanma verim gibi ya da gelişme diğer bir faktörü olumsuz gecikme gibi etkileyebilir. Buradaki asıl amaç maksimum bir şekilde verimlilik elde etmektir. Teorik olarak hesaplama, işlemci sayılarıyla doğru orantıda hızlanmalıdır. Örnek olarak bir algoritmada iki işlemci kullanılıyorsa iki kat bir hızlanma beklenmektedir. Fakat maksimum verimlilik sağlanması zordur. Bunun sebebi iş yükünün işlemcilere eşit dağıtılması ve işlemlerin koordine edilmesi verilerin karşılaştırılması gibi durumlardır.

5. KAYNAKLAR

Aad J. van der Steen, "Overview of recent supercomputers, October 2013", Top500 Technical Report, (October), 1–77, (2013).

Akkaş, S., "Karesel Atama Probleminin Tavlama Benzetimi Ve Paralel Programlama Teknikleri Kullanarak Çözümü", Yüksek Lisans Tezi, Pamukkale Üniversitesi Fen Bilimleri Enstitüsü, Bilgisayar Mühendisliği Anabilim Dalı, Denizli, (2016).

Amdahl, G. M., "Validity of the single processor approach to achieving large scale computing capabilities", AFIPS spring joint computer conference 1967, 4, (1967).

Casotto, A., Romeo, F. ve Sangiovanni-Vincentelli, A. "A Parallel Simulated Annealing Algorithm for the Placement of Macro-Cells", IEEE Transactions Computer-Aided Design of Integrated Circuits and Systems. IEEE, 6(5), 838–847. doi: 10.1109/TCAD.1987.1270327, (1987).

Darema, F. ve Kirkpatrick, S.,"Parallel algorithms for chip placement by simulated annealing", IBM Journal of Research, 31(3). doi: 10.1147/rd.313.0391, (1987).

Debudaj-Grabysz, A. ve Czech, Z. J. (2009) "Theoretical and Practical Issues of Parallel Simulated Annealing", Parallel Processing and Applied Mathematics. Berlin, Heidelberg: Springer Berlin Heidelberg, 189–198. doi: 10.1007/978-3-540-68111-3_21, (1987).

Dongarra, J. J.,LINPACK User's Guide., Mathematics of Computation. Society for Industrial and Applied Mathematics. doi: 10.2307/2006212,(1979).

El-Rewini, H. ve Abd-El-Barr, M.,Advanced Computer Architecture and Parallel Processing. Hoboken, NJ, USA: John Wiley & Sons, Inc. doi:10.1002/0471478385,(2004).

Ergün, U. ve Sayar, A.,"Foksiyonel Programlama Dilleri ile Paralel Programlama", Niğde Üniversitesi Mühendislik Bilimleri Dergisi, 3(2), 1–17, (2014).

Flynn, M. J, "Some computer organization and their effectiveness IEEE Transactions on Computers, Vol", C-21, (1972).

Greening, D. R., "Parallel simulated annealing techniques", Physica D: Nonlinear Phenomena, 42(1–3), ss. 293–306. doi: 10.1016/0167-2789(90)90084-3, (1990).

Gustafson, J. L.,"Reevaluating Amdahl's law", Communications of the ACM, 31(5),

532–533. doi: 10.1145/42411.42415, (1988).

Güneş, A., "Paralel programlama ile el yazısı rakamlarının tanınması", Yüksek Lisans Tezi, Isparta Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği Anabilim Dalı, Isparta, (2011).

Karp, A. H. ve Flatt, H. P.,"Measuring parallel processor performance", Communications of the ACM, 33(5), ss. 539–543. doi: 10.1145/78607.78614, (1990).

Kirkpatrick, S., Gelatt, C. D. ve Vecchi, M. P.,"Optimization by Simulated Annealing", Science, 220(4598), ss. 671–680. doi: 10.1126/science.220.4598.671, (1983).

Marr, D. T.,"Hyper-Threading Technology Architecture and Microarchitecture", Intel Technology Journal, 6(1), ss. 1–12, (2002).

McCool, M., Reinders, J. ve Robison, A.,"Structured parallel programming patterns for efficient computation", doi: 10.1016/B978-008043924-2/50055-9,(2012).

Microsoft, "Parallel Programming in .NET",(04.05.2019),Web adresi: https://docs.microsoft.com/tr-tr/dotnet/standard/parallel-programming/,(2018).

Moore, G. E.,"Cramming more components onto integrated circuits. In: Electronics", Electronics, 38(8), 114, (1965).

Ogura, M.,"Parallel Simulated Annealing using Genetic Crossover", Transition, 1–6,(2002).

Onbaşoğlu, E. ve Özdamar, L.,"Parallel Simulated Annealing Algorithms in Global Optimization", Journal of Global Optimization, 19(1), 27–50. doi: 10.1023/A:1008350810199, (2001).

Ostrovsky, I., *Coding Guidelines*, http://download.microsoft.com/download/b/c/f/bcfd4868-1354-45e3-b71b-b851cd78733d/parallelprogramsinnet4_codingguidelines.pdf, (2010).

Pham, D. T. ve Karaboga, D,Intelligent Optimisation Techniques. London: Springer London. doi: 10.1007/978-1-4471-0721-7, (2000).

Ram, D. J., Sreenivas, T. H. ve Subramaniam, K. G,"Parallel Simulated Annealing Algorithms", Journal of Parallel and Distributed Computing, 212, 207–212, (1996).

Rosenbrock, H. H., "An Automatic Method for finding the Greatest or Least Value

of a Function", Computer J., ss. 175–184, (1960).

Sonuç, E., "Benzetilmiş Tavlama Algoritmasının Grafik İşlemci Kullanılarak Paralelleştirilmesi", Doktora Tezi, Karabük Üniversitesi Fen Bilimleri Enstitüsü, Bilgisayar Mühendisliği Anabilim Dalı, Karabük, (2017).

Top500, "TOP500 Supercomputer",(20.04.2019),Web adresi: https://www.top500.org/, (2019).

6. ÖZGEÇMİŞ

Kadir YÜREKTÜRK Almanya/Berlin'de 1987 yılında doğdu. İlköğretimini Gazi Mustafa Kemal İlköğretim Okulu'nda tamamladı. Denizli Anadolu Lisesinden 2006 yılında mezun oldu. Aynı yıl Pamukkale Üniversitesi Mühendislik Fakültesi Bilgisayar Mühendisliği bölümünü kazandı. Bölüm üçüncüsü ve Onur Öğrencisi olarak mezun oldu. Pamukkale Üniversitesi Bilgi İşlem Daire Başkanlığında akademik uzman olarak 2010 yılında başladığı görevine öğretim görevlisi olarak halen sürdürmektedir. Evli ve 1 kız babasıdır.

ADRES BİLGİLERİ

Elektronik posta :kadir@pau.edu.tr

İletişim Adresi :Pamukkale Üniversitesi

Kınıklı Kampüsü

Rektörlük Bilgi İşlem Daire Başkanlığı

Telefon :0 (532) 050 8598