İleri Teknoloji Bilimleri Dergisi
Cilt 2, Sayı 1, 76-83, 2013

Journal of Advanced Technology Sciences
Vol 2, No 1, 76-83, 2013

PARALEL GÖRÜNTÜ FİLTRELEME İÇİN ÇOK ÇEKİRDEKLİ BİLGİSAYAR ÜZERİNDE BAŞARIM ANALİZİ

Devrim AKGÜN*

Düzce Üniversitesi, Teknoloji Fakültesi, Bilgisayar Mühendisliği, 81620, Düzce, TÜRKİYE

Özet-Çok çekirdekli işlemci teknolojisinin gelişmesiyle birlikte yüksek işlem gücü gerektiren birçok algoritmanın hızlandırılması için paralel hesaplama yaklaşımının kullanımı yaygın hale gelmiştir. Sunulan çalışmada, Java iş parçacıkları ile gerçekleştirilen paralel görüntü filtresinin çok çekirdekli işlemci üzerinde başarım analizi gerçekleştirilmiştir. Deneysel sonuçlar, iki çekirdekli ve dört çekirdekli, Hyper Threading teknolojisini destekleyen işlemcilere sahip bilgisayarlar kullanılarak elde edilmiştir. Bu amaçla tek çekirdekli hesaplama sonuçları referans alınarak hızlandırma ve paralel verimlilik grafikleri karşılaştırmalı olarak elde edilmiştir. Sonuçlar yaklaşık olarak iki çekirdekli işlemcide iki iş parçacığı ile 1,9 kat dört iş parçacığı ile 2,9 kat, dört çekirdekli işlemcide dört iş parçacığı ile 3,6 kat, sekiz iş parçacığı ile 5,7 kata kadar hızlandırma elde edilmiştir.

Anahtar Kelimeler-Görüntü İşleme, Paralel Hesaplama, Java İş Parçacıkları, Çok Çekirdekli İşlemci.

PERFORMANCE ANALYSIS OF PARALLEL IMAGE FILTERING ON MULTI-CORE COMPUTER

Abstract-With the developing multi-core technology, utilization of parallel computing approach for the high performance algorithms become widespread. In this study, analysis of parallel image filter that was realized with Java threads on multicore computer has been presented. Experimental results were obtained using dual core and quad core processor with Hyper Threading technology. For this purpose comparative speed-up and parallel efficiency graphs have been plotted referencing the single core single core computation results. According to experiments, dual core processor using two threads produced about 1.9 fold speed-up while four threads produced about 2.9 fold speed-up and quad core processor using four threads produced about 3.6 fold speed-up while eight threads produced about 5.7 fold speed-up has been obtained.

Keywords-Image Processing, Parallel Computing, Java Threads, Multi-core Processor.

* devrimakgun@duzce.edu.tr

1. GİRİŞ (INTRODUCTION)

Kenar belirleme, gürültü yok etme, görüntü geliştirme gibi birçok görüntü işleme uygulamasında konvolüsyon işlemine başvurulur [1,2]. Uzunlukları resim ve maske boyutları ile değişen ve genelde büyük boyutlu döngülerin kullanıldığı bu algoritma yoğun işlem gücü gerektirir. Paralel hesaplama yaklaşımı böyle algoritmaların hızlandırılması için önemli bir potansiyel sunar. Konvolüsyon ile paralel görüntü filtreleme çeşitli formlarda yüksek başarımlı olarak gerçekleştirilmiştir [3,4,5]. Çok-çekirdekli donanımların sağladığı hesaplama potansiyelinin sıralı algoritmaların hızını artırmada kullanımı açısından paralel algoritmalar önemli alternatif oluşturur. Günümüzde çok-çekirdekli işlemci teknolojisinin son kullanıcı seviyesinde yaygın kullanımının artması, görüntü işleme uygulamaları gibi yüksek hesaplama gücü gerektiren algoritmaların hızlandırılması için paralel hesaplama yaklaşımını daha cazip hale getirmiştir [6,7]. Özellikle Java gibi nesne yönelimli programlama dilleri iş parçacıklarının yönetilmesi için sağlanan yerleşik kütüphaneler ile paralel uygulamaların kolaylıkla gerçekleştirilmesine imkân tanır [8,9]. İş parçacıkları aynı proses kaynaklarını kullanarak tanımlanmış programları, işletim sisteminin belirlediği zamanlarda öncelik ve giriş/çıkış işlemleri durumuna göre sırayla yürütülürler [10]. Tek çekirdeğe sahip bir işlemci üzerinde iş parçacıkları sahip oldukları önceliklere ve bulundukları duruma göre işletim sistemi tarafından diğerleri ile zaman paylaşımlı olarak yürütülürler. İşlemci yapısında birden fazla çekirdek olması durumunda birden fazla iş parçacığı aynı anda yürütülebilir. Bunun yanında işlemcinin Hyper-Threading teknolojisinin bulunması da paralel çalıştırılabilecek iş parçacığı sayısını da artırır. Bu sayede paralelleştirilebilen algoritmaların parçaları aynı anda işletilerek çalışma süresi sıralı çalışmaya göre kısalmaktadır. Java platformunun sahip olduğu java.lang.Thread sınıfı ya da java.lang.Runnable ara yüzü ile iş parçacıklarını tanımlamak ve yönetmek pratik bir şekilde gerçekleştirilebilir. Sunulan çalışmada, konvolüsyon tabanlı görüntü filtreleme algoritması günümüzde popüler ve birçok işletim sistemi ile uyumlu çalışabilen Java programlama dili ile paralel çalışacak şekilde kodlanmış ve günümüzde kolayca erişilebilen çok çekirdekli bilgisayar üzerinde başarım analizi gerçekleştirilmiştir. Bu amaçla konvolüsyon işlemi ile filtrelenecek görüntü paralel çalışacak iş parçacıkları arasında statik yük dengeleme yaklaşımı kullanılarak paylaştırılmıştır. Deneysel sonuçlar iki çekirdekli Intel i3 2100 ve dört çekirdekli Intel i7 3612QM model numaralı işlemcilere sahip bilgisayarda elde edilmiştir. Tek çekirdeğin kullanıldığı sıralı algoritma başarımı referans alınarak çekirdek sayısına bağlı hızlandırma ve verimlilik değerleri analiz edilmiştir. Paralel algoritmanın sağladığı başarım karşılaştırmalı grafiklerle ortaya konmuştur.

2. YÖNTEM (METHOD)

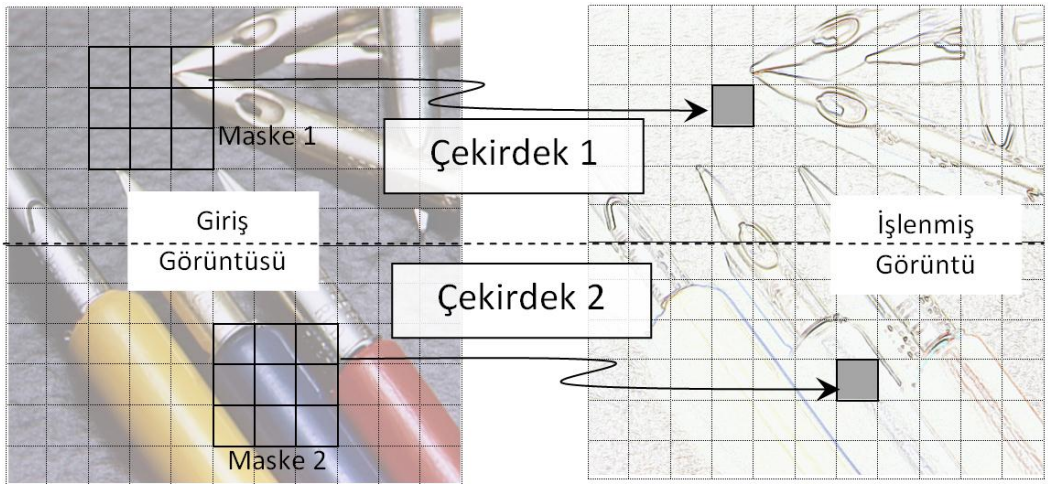
Giriş görüntüsü ile katsayıları belli karakteristiklere sağlayacak şekilde belirlenen filtre maskesinin konvolüsyonu, görüntü işlemede başvurulan oldukça faydalı temel matematiksel işlemlerden biridir. Doğrusal bir filtrenin giriş çıkışı ilişkisini tanımlayan konvolüsyon işlemi Ayrık domende Eş. 1'deki gibi tanımlanır [1].

$$y(m,n) = \sum_{p=-M}^{M} \sum_{q=-M}^{M} w_{k,l}\, x(m-p, n-q) \qquad (1)$$

Burada w filtre katsayılarını tanımlayan $M \times M$ boyutlu matrisi ve x görüntüyü tanımlayan iki boyutlu giriş işaretini tanımlar. Bu işlem temelde döndürülmüş filtre katsayı matrisinin görüntü üzerinde gezdirilmesi ile gerçekleştirilir. Matris elemanlarına karşılık gelen elemanların çarpılıp toplanması ile çıkış pikseli hesaplanır. Bu işlem çıkış görüntüsündeki tim pikseller için gerçekleştirilir.

2.1. Paralel Gerçekleme (Parallel Realization)

Prosesler içinde oluşturulabilen iş parçacıkları ile bir sıralı algoritmaya ait görevler parçalara bölünebilir ve aynı anda ayrı iş parçacıkları tarafından paralel işletilerek görevin daha kısa sürede tamamlanması sağlanabilir. Donanımın birden fazla iş parçacığının paralel çalışmasına destek sağlaması durumunda paralel gerçeklenebilen algoritmaların parçalarını aynı anda yürütülerek başarım kullanılan çekirdek sayısına bağlı olarak artırılır. Paralel hesaplamanın gerçekleştirilebilmesi için algoritmanın hesaplama yükünün iş parçacıklarına dağıtılması gerekir. Yükün iş parçacıklarına dağıtılması, statik dengeleme ile paralel hesaplamadan önce, dinamik dengeleme ile çalışma anında veya bu amaçla kullanılan benzer yöntemlerle gerçekleştirilebilir [11,12]. Bu çalışmada gerçekleştirilen deneysel ölçümlerde gerçeklemesi en kolay yöntemlerden biri olan statik yük dengeleme yaklaşımı kullanılmıştır. Şekil 1'de basitçe statik yük dağılımının kullanıldığı iki çekirdekli paralel görüntü işleme prensip ifade edilmiştir. Görüntü iki eşit parçaya bölünerek yük paylaşıldıktan sonra iki farklı görüntü gibi çekirdekler tarafından aynı anda işlenmektedir. Konvolüsyon ile görüntü işlemede her bir çıkış pikselinin hesabı birbirinden bağımsız olarak gerçekleştirilebildiği için işlenecek toplam piksel adedi paralel çalışacak iş parçacıkları arasında eşit paylaştırılmıştır.



Şekil 1. Paralel görüntü işleme için iki çekirdekli örnek uygulama (Two core implementation for parallel image processing)

Statik yük dengeleme yaklaşımına göre işlemci görüntünün işlenecek alanlarının başlangıçta iş parçacıklarında tanımlanması gerekir. Burada yük piksel bazında ele alındığı için piksel sayısı paralel çalışacak iş parçası sayısına bölünerek bulunabilir.

$$\left.\begin{aligned} minYük &= \text{floor}\left(\frac{M \times N}{nİşparcası}\right) \\ kalan &= \text{mod}(M \times N, nİşparcası) \end{aligned}\right\} \qquad (2)$$

İş parçası başına yük Eş. 2'de belirtildiği gibi görüntü satır ve sütun çarpımının iş parçası adedine bölümü sonucu elde edilir. Elde edilen sonuç tamsayı olmaması muhtemel olduğu için sonuç tabana yuvarlanmıştır. Kalan olması durumunda bazı iş parçacıkları diğerine göre bir piksel daha fazla işlem yapsa da bunun oluşturacağı dengesizlik ihmal edilebilir seviyelerdedir.

Tablo 1'de farklı sayılarda iş parçacıkları seçilmesi durumunda 100×100 boyutlu bir görüntü için oluşan örnek dağılımlar verilmiştir.

Tablo 1. 100×100 boyutlu bir görüntü için iş parçası adedine bağlı örnek dağılımlar (Example distributions versus the number of threads for the image size 100×100)

İş parçası adedi	Minimum yük	Kalan	İş parçası başına yük miktarları
2	5000	0	{5000, 5000}
3	3333	1	{3334,3333,3333}
4	2500	0	{2500,2500,2500,2500}
5	2000	0	{2000,2000,2000,2000,2000}
6	1666	4	{1667,1667,1667,1667,1666,1666}

2.2 Programlama yaklaşımı (Programming Approach)

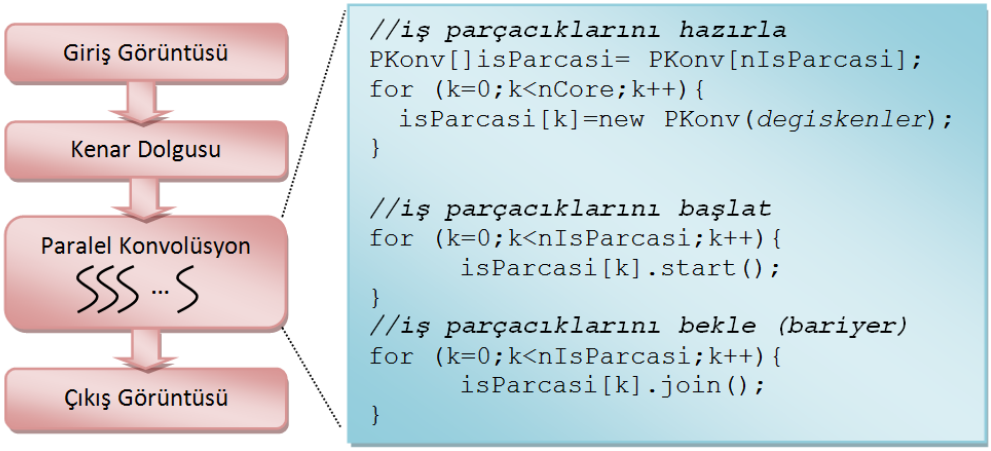
İş parçacıklarının oluşturulmasında kullanılacak sınıf, Thread sınıfı veya Runnable ara yüzü aracılığı ile tanımlanabilir. Bunlardan birinde iş parçacığı sınıfı Thread sınıfını miras alarak tanımlanırken diğerinde Runnable ara yüzünü gerçeklenir. Şekil 2.'de iş parçalarının tanımlandığı sınıfa ait basitleştirilmiş örnek kod görülmektedir. Thread sınıfının miras alınması sonucu iş parçacıkları ile paralel hesaplama için run() ile tanımlanan yöntem içinde paralel işletilecek kodların yazılması gerekir. Burada görüntü işleme için Denklem 1 ile verilen ifadeyi gerçekleştiren konvolüsyon algoritması tanımlanmıştır. Ayrıca giriş ve çıkışa ait görüntüler statik yapıda tanımlanarak iş parçacıları arasında paylaşılmaktadır. Böylece iş parçaları birbirinden bağımsız olarak aynı görüntünün yalnızca belirlenen koordinatlarında konvolüsyon işlemini gerçekleştirirler.

```
class PKonv extends Thread{
      //Sınıf ve nesne alanları
      public static BufferedImage girisGoruntu,cikisGoruntu
      private int xBaslangic,yBaslangic,pikselAdedi;
      ...
      run(){
            // Paralel işlemler
            Konvolusyon(xBaslangic,yBaslangic,pikselAdedi);
            ...
      }
}
```

Şekil 2. İş parçalarının üretilmesinde kullanılan konvolüsyon işleminin tanımlandığı sınıf (The class used for instantiating threads for convolution operation)

Paralel hesaplama öncesinde Şekil 3'de görüldüğü gibi görüntünün okunması ve kenar dolgu işleminin gerçekleştirilmesi gerekir. Görüntü okuma/yazma işlemleri Java'daki temel kütüphanelerden java.awt.image paketi içindeki BufferedImage sınıfı ile gerçekleştirilebilir. Deneylerde bu sınıftan oluşturulan bir nesnenin getRGB() yöntemi ile tüm pikseller bir diziye aktarılarak kullanılmıştır. Benzer şekilde işlenmiş dizi setRGB() yöntemi ile tekrar görüntü olarak kaydedilmiştir. Görüntü kenar dolgu ise işlenecek görüntünün kenarlarında, filtre maskenin komşu pikselleri kullanmasından dolayı maske boyutuna bağlı olarak ek satırların oluşturulması işlemidir. Paralel bölgede ise Şekil 3'de görüldüğü PKonv sınıfından nesneler üretilerek iş parçacıkları oluşturulur. Ardından üretilen her bir nesnenin start() yöntemi çağırılarak iş parçacıkları tanımlanan görevi gerçekleştirmesi için başlama komutu verilir. Görev tamamlanana kadar bariyer oluşturan join() yöntemi ile tüm iş parçacıklarının durumu kontrol edilir ve hesaplamanın paralel kısmı sonlandırılmış olur. Sıralı olarak yürütülmekte olan

bir programın içinde, paralel parçalar halinde işlenip tekrar sıralı hale dönülmesi genelde fork-join (ayrılma ve toplanma) olarak adlandırılır [9].



Şekil 3. Paralel konvolüsyon ile görüntü filtreleme (Parallel image filtering with convolution)

3. BULGULAR (FINDINGS)

Java dili ile kodlanan paralel görüntü işleme algoritmasın başarımı yaygın olarak elde kullanılan çok-çekirdekli bilgisayar üzerinde analiz edilmiştir. Deneysel sonuçlar, Windows 7 64 bit işletim sistemi üzerinde kurulu Java SE Development Kit 7 Update 11 kullanılarak elde edilmiştir. Deneylerde kullanılan donanımlara ait özellikler Tablo 1'de görülmektedir.

Tablo 2. Deneylerde kullanılan donanımlara ait özellikler (Specifications for the hardware used in the experiments)

CPU model	Çekirdek sayısı	Hyper Threading	Ön Bellek	RAM	İşletim Sistemi
Intel i3-2100 @3,10 GHz	2	var	3MB	4GB	Windows 7 64 bit
Intel i7-3612QM @2,10Ghz	4	var	6MB	4GB	Windows 7 64 bit

Elde edilen sonuçlar 100 kez tekrar edilerek en düşük olan değerler seçilmiştir. Çalışma süreleri iş parçacıklarının çalıştırıldığı start() yönteminden önce ve join() yöntemi ile sonlandırıldıktan sonra System.nanotime() ile ölçülen zaman değerlerinden hesaplanmıştır. Tablo 3 ve 4'de 600×600 boyutlu görüntüden 2400×2400 boyutlu görüntüye kadar dört farklı görüntü için iş parçacığı adedine bağlı ölçümler verilmiştir. İki çekirdekli işlemci ile alınan sonuçlardan görüldüğü gibi dört iş parçacığına kadar farklı boyutlardaki görüntüler için çalışma sürelerinde önemli miktarlarda düşme sağlanmıştır.

Tablo 3. Farklı boyutlardaki görüntüler için Intel i3-2100 işlemcisi ile iş parçacığı adedine bağlı çalışma süreleri (milisaniye) (Running times versus the number of threads for different sizes of images on the processor of Intel i3-2100 (milliseconds))

Boyutlar	1 iş parçası	2 iş parçası	3 iş parçası	4 iş parçası
600×600	22,34	12,88	10,22	7,80
1200×1200	89,20	50,45	37,74	30,42
1800×1800	200,37	103,59	83,88	68,13
2400×2400	355,43	178,77	144,37	120,95

Benzer başarım dört çekirdekli işlemci için de sağlanmıştır. Yalnızca 600×600 boyutlu görüntü de 6 iş parçacığı için sağlanan başarım 8 iş parçacığı için sağlanamamıştır. Başarımdaki bu düşme iş-parçacığı adedinin artırılmasıyla işletim sisteminin yükünün artarken ve iş parçası başına düşen hesaplama yükünün azalması sonucu ortaya çıkmıştır.

Tablo 4. Farklı boyutlardaki görüntüler için Intel i7-3612QM işlemcisi ile iş parçacığı adedine bağlı çalışma süreleri (milisaniye) (Running times versus the number of threads for different sizes of images on the processor of Intel i7-3612QM (milliseconds))

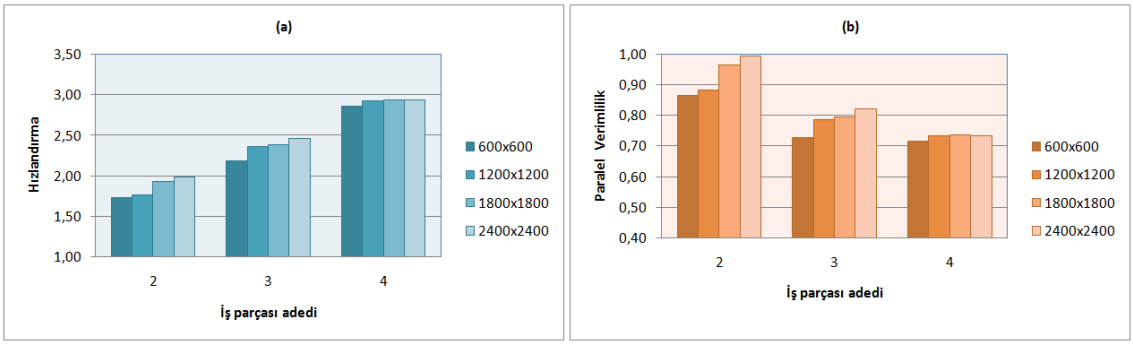
Boyutlar	1 iş parçası	2 iş parçası	3 iş parçası	4 iş parçası	6 iş parçası	8 iş parçası
600×600	22,38	11,66	8,31	6,41	5,41	6,46
1200×1200	89,23	46,48	32,97	24,84	20,60	15,74
1800×1800	200,42	103,78	73,86	55,44	45,50	34,94
2400×2400	355,42	183,88	130,84	98,43	78,83	61,65

Tablo 3 ve 4'de verilen sonuçların yorumlanması için hızlandırma ve paralel verimlilik değerleri pratikte kullanılan başarım ölçütleridir. Hızlandırma ve paralel verimlilik yöntemlerine ait formüller Eş. 3 ve 4'de verilmiştir[12]. Hızlandırma miktarı sıralı yani tek çekirdek üzerindeki çalışma zamanının paralel çekirdeklerle elde edilen çalışma zamanına oranından belirlenir. Paralel verimlilik ise hızlandırma miktarının çekirdek adedine oranından belirlenerek paralel hesaplamada kullanılan çekirdeklerin verimliliği hakkında bilgi verir.

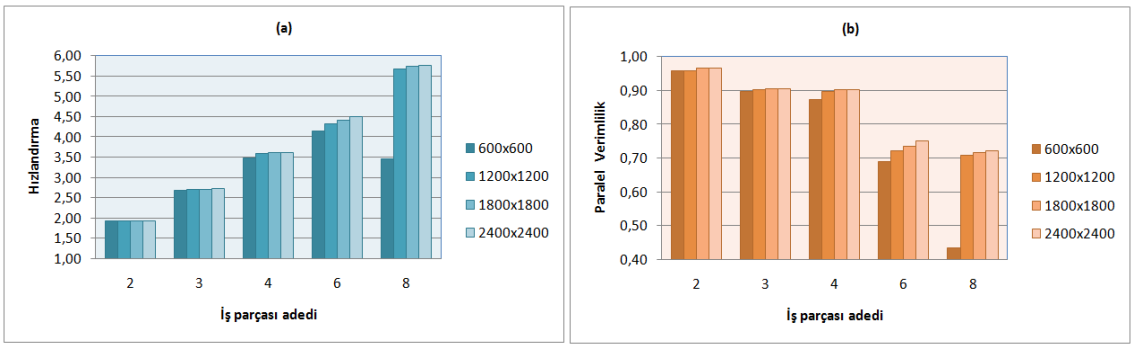
$$\text{Hızlanma} = \frac{t_{\text{sıralı}}}{t_{\text{paralel}}} \tag{3}$$

$$\text{Paralel Verimlilik} = \frac{\text{Hızlanma}}{\text{Çekirdek adedi}} \tag{4}$$

Tablo 3 ve 4'deki sonuçlardan Eş. 3 ve 4' göre sağlanan hızlandırma ve paralel verimlilik değerleri, Şekil 4 ve Şekil 5'de karşılaştırmalı olarak verilmiştir. İki çekirdekli işlemci için elde edilen hızlandırma sonuçları Şekil 4a'dan incelendiğinde, iki iş parçacığı için hızlandırma miktarları görüntü boyutuna bağlı olarak iki kata yaklaşmıştır. Bunun yanında Hyper-Threading özelliği de kullanılarak üç ve dört iş parçacığı için hızlanma değerlerindeki artış iki çekirdekteki kadar olmasa da önemli seviyelerde artmıştır. Bu durum Şekil 4b'de verilen paralel verimlilik grafiğinde de görülmektedir. Dört çekirdekli işlemcinin kullanıldığı analizlerde ise dört çekirdek için Şekil 5a'dan görüldüğü gibi 3,6 kata kadar hızlandırma elde edilmiştir. Verimlilik açısından bakıldığında ise Şekil 5b'den görüldüğü gibi dört çekirdekten sonra oldukça düşse de Hyper-Threading desteği ile sekiz iş parçacığı kullanılarak 5,7 kata kadar hızlandırma elde edilmiştir.



Şekil 4. İki çekirdekli işlemci için (Intel i3-2100) (a) hızlandırma (b) paralel verimlilik ((a)speed-up (b) parallel efficiency for two core processor (Intel i3-2100))



Şekil 5. Dört çekirdekli işlemci için (Intel i7-3612QM) (a) hızlandırma (b) paralel verimlilik ((a)speed-up (b) parallel efficiency for Quad core processor (Intel i7-3612QM))

4. SONUÇ VE TARTIŞMA (CONCLUSION AND DISCUSSION)

Birçok görüntü işleme uygulamasında ihtiyaç duyulan konvolüsyon ile görüntü işleme algoritmasının paralel hesaplama ile hızlandırılması, buna dayalı yürütülecek algoritmaların da performansını artırır. Sunulan makalede, paralel konvolüsyon algoritmasının Java'nın yerleşik kütüphanelerin sağladığı destek ile ürettiği paralel hesaplama başarımı, çok çekirdekli bilgisayar ortamında deneysel olarak analiz edilmiştir. Deneyler her ikisi de Hyper Threading teknolojisini destekleyen iki çekirdekli ve dört çekirdekli işlemcilere sahip bilgisayarlar üzerinde elde edilmiştir. Tek çekirdekli hesaplama sonuçları temel alınarak hızlandırma ve paralel verimlilik grafikleri karşılaştırmalı olarak elde edilmiştir. Deneysel sonuçlar ile yaygın kullanıma sahip Java platformu ve günümüz çok çekirdekli işlemci teknolojisi ile görüntü filtreleme algoritması için önemli seviyelerde paralel hesaplama başarımı sağlandığı görülmektedir. Sonuçlar farklı sayılarda paralel çalışan iş parçası için ve farklı boyutlarda görüntüler için karşılaştırılmıştır. İş parçası sayısının artması 600×600 gibi düşük boyutlu görüntülerde paralel verimliliği oldukça düşük seviyeye çekerken, genel olarak çekirdek adedi kadar iş parçası ile sağlanan hızlandırma oldukça iyi seviyelerde edilmiştir. Hyper Threading teknolojisini kullanmak için iş parçası adedinin iki kata kadar artırılması da hızlandırmaya önemli seviyelerde katkıda bulunduğu deneysel sonuçlardan görülmektedir. Ön bellek (cache) kullanımını dikkate alan farklı yöntemlerle paralel çalışan çekirdeklerin ve Hyper Threading teknolojisinin kullanımını daha da iyileştirilebilir.

5. KAYNAKLAR (REFERENCES)

[1]. R. C. Gonzalez, R. E. Woods,(2008). Digital Image Processing,Prentice Hall.
[2]. Babic, Z.V., (2003). An efficient noise removal and edge preserving convolution filter 6th International Conference on Telecommunications in Modern Satellite, Cable and Broadcasting Service, 2, 538 - 541 vol.2
[3]. S. Yu, M. Clement, Q. Snell, B. Morse, (1998) "Parallel algorithms for image convolution", Proceedings of the International Conference on Parallel and Distributed Techniques and Applications, Las Vegas, Nevada.
[4]. J. Kepner, "A Multi-Threaded Fast Convolver for Dynamically Parallel Image filtering", Journal of Parallel and Distributed Computing, Vol. 63, pp. 360–372, 2003
[5]. S. Nakariyakul, "Fast spatial averaging: an efficient algorithm for 2D mean filtering", The Journal of Supercomputing, DOI: 10.1007/s11227-011-0638-9, Online, 2011
[6]. P. Frost Gorder, (2007). Multicore Processors for Science and Engineering, IEEE Computing in Science & Engineering, 9, 3-7.

[7]. G. Andrews, (2000). Foundations of Multithreaded, Parallel, and Distributed Programming, Addison-Wesley
[8]. B. Sanden, (2004). Coping with Java threads, IEEE Computer, 37, 20-27
[9]. D. Lea, (1997). Concurrent Programming in Java: Design Principles and Patterns, Addison-Wesley.
[10]. Stallings (2005). Operating Systems, Internals and Design Principles. Pearson: Prentice Hall
[11]. K. K. Yue, D. J. Lilja, "Parallel Loop Scheduling for High-Performance Computers", High-Performance Parallel Computing Research Group Technical Report No. HPPC-94-13, 1994
[12]. Z. Fang, P. Tang, P.C. Yew, C. Q. Zhu, "Dynamic processor self-scheduling for general parallel nested loops", Vol. 39, pp. 919 – 929, 1990
[13]. Z. Juhasz, (1998). "An Analytical Method for Predicting the Performance of Parallel Image Processing Operations", The Journal of Supercomputing, Vol.12, pp.157-174