GAZİOSMANPAŞA BİLİMSEL ARAŞTIRMA DERGİSİ (GBAD)
Gaziosmanpasa Journal of Scientific Research
ISSN: 2146-8168
http://dergipark.gov.tr/gbad
*Araştırma Makalesi (Research Article)*

**Cilt/Volume : 11**
**Sayı/Number: 3**
**Yıl/Year: 2022**
**Sayfa/Pages: 69-83**

***Alınış tarihi (Received)*:** 18.09.2022
***Kabul tarihi (Accepted)*:** 16.11.2022

# Çok Çekirdekli İşlemciler İçin Yük Dengelemeli Melez Bir Paralel Sıralama Metodu

**Cengiz GÜNGÖR** [1*]

[1] *Tokat Gaziosmanpaşa Üniversitesi, Mühendislik ve Mimarlık Fakültesi, Bilgisayar Mühendisliği, Tokat*
[*] *Sorumlu yazar: cengiz.gungor@gop.edu.tr*

**ÖZET:** Sıralama işlemi, bilgisayar bilimlerinin en temel problemlerindendir. Karmaşık verilerin, sıralama işlemleriyle düzenli hale getirilmesi birçok yararlar sağlamaktadır. Sıralama gereksinimi de aslında düzensiz verilerin işlenmesinde yaşanan sıkıntılardan kaynaklanır. Metinlerde alfabetik sıralama yapılması veya sayısal verilerin büyükten küçüğe (veya tersine) sıralanması buna örnek olarak verilebilir. Bu çalışmada bilinen sayısal sırama işleminin günümüz ev bilgisayarlarında dahi mevcut olan çok çekirdekli işlemcilerin kullanımı ile ne kadar hızlandırılabileceğini gösterilmektedir. Eğer basitçe ve hızlıca sayı sıralanması istenirse, quicksort veya merge-sort kullanılabilir, en çok bilinen sıralama algoritmaları da bunlardır. Bu algoritmaların işlem karmaşıklığı literatürde $O(n \lg n)$ olarak verilir. Özel şartlara sahip sayılarla yapılan sıralamalarda ise $O(n)$ karmaşıklığa kadar inilebilir. Ancak bu değerlerle sıralama işleminin çalışma süresi kesin olarak bilinemez, sadece tahmin edilebilir. Öyle ki, işlem karmaşıklığı aynı kalsa da algoritmalar hızlandırılabilir. Bunun yöntemi paralel hesaplama tekniklerdir. Bu çalışmada standart bir i5 işlemcili bir bilgisayarda OpenMP kütüphanesi ile 8 çekirdek üzerinde çalışılmış, 45 kata kadar önemli hız değerleri elde edilmiştir.

***Anahtar Kelimeler*** *– Sıralama algoritmaları, Paralel hesaplama, Çekirdek programlama, OpenMP*

# A Hybrid Parallel Sorting Method With Load Balancing For Multi-Core CPU's

**ABSTRACT:** Sorting is one of the most fundamental problems of computer science. If the data comes in a random order, the data is processed by sorting and put into a sorted form. This process provides many benefits. The requirement of sorting jobs is actually caused by difficulties in processing unsorted data. For example, alphabetical sorting of texts, or sorting numerical data in ascending or descending order. This study is focused to how many times can be accelerated for the numerical sorting process by using processors and its cores of today's home computers. If we want to simply and quickly sort of some numbers, the well-known sort algorithms are quicksort or merge-sort. The processing complexity of these algorithms is given in the literature as $O(n \lg n)$. If data has some specific numbers, algorithm complexity can be reduced until $O(n)$. However, this formula certainly does not give an idea of the working time of the sorting process. Although the complexity remains the same, the algorithms can be accelerated. The suggested method in this work uses parallel computing techniques for speed-up. In this study, we used OpenMP library on a standard i5 computer with 8 cores and speed-up values is reached to 45 times.

***Keywords*** *– Sorted algorithms, Parallel computing, Core programming, OpenMP*

## 1. Giriş

Sıralama işlemi düzensiz duran nesnelerin bir düzene sokulmasıdır. Bu işlem günlük hayatta çeşitli alanlarda uygulanmaktadır. Bilgisayar alanında ise, sıralama en temel problemlerden birisidir ve sıralama işlemini yapan çeşitli algoritmalar bulunmaktadır. Örneğin veri tabanlarında indeksleme bu amaçla kullanılmaktadır. İncelenen tabloda bir indeksleme yoksa tüm tabloya bakmak bir zorunluluktur, fakat tabloda indeks varsa sorgulama işlemi belli bir aralıkta sıralı okuma ile verileri işlemden geçirir ve sonlanır. Aslında sıralamadaki en büyük

amaç da veri düzenlemedir. Sıralama yapılırken alfabetik alanlarda kelimeler harf sırasına göre dizilirken, sayısal alanlar ise niceliklerine göre küçükten büyüğe (veya tam tersi) dizilirler. Sıralama algoritmaları için pek çok yöntem geliştirilmiştir.

Bu çalışmada sayısal alanlardaki sıralamaları hızlandırmak hedeflenmiştir. Eğer sıralanacak sayılar tamsayı ve belli aralıkta iseler, yani özel bir formda iseler bunları sıralamak çok hızlı yapılabilmektedir. Örneğin sayı tabanı sıralaması (radix sort) böyle bir sıralamadır (Wikipedia-RadixSort, 2022). Ancak böyle özel bir durumla nadiren karşılaşılır, sıralanacak sayıların değerlerinin bir limiti veya özelliği yoktur. Bu tür durumlarda da en çok bilinen algoritmalar hızlı sıralayıcı (quicksort) (Sedgewick ve Wayne, 2011 (1)) ve birleştirmeli sıralayıcıdır (merge-sort) (Sedgewick ve Wayne, 2011 (2)).

Sıralama algoritmaları temelde veri karşılaştırma ve yer değiştirme yapmaktadırlar. Eğer incelenen sayı karşılaştırılandan küçükse (veya büyükse) yer değiştirme yapılır. Quicksort ve merge-sort ele aldıkları problemi sürekli iki parçaya bölerek, öz-yinelemeli çalıştıkları için maksimum lg( $n$ ) aşama kullanırlar ve her aşamada $n$ adet sayıyı incelerler. Burada lg() fonksiyonu 2 tabanında logaritmayı göstermektedir. Toplamda yapılan karşılaştırma-yer değiştirme işlemlerinin karmaşıklığı ise O( $n$ lg $n$ ) notasyonuyla verilir (Horowitz ve ark., 1998). Ancak bu notasyon toplam işlem süresini vermez, buradan hareketle bir süre tahmini yapılamaz. Süreye etki eden faktörler, öncelikle problemin boyutu, daha sonra işlemci sayısı, çekirdek sayısı, önbellek ve ana bellek boyutları, kod derleyicinin kalitesi, hatta disk hızı dahi olabilir. Toplam süreyi kısaltmak için, kullanılan algoritmanın hızlandırılması veya tek işlemci ile yapılan iş yükünün, mevcut işlemciler ve çekirdeklere dağıtılarak yapılması gerekmektedir.

Günümüzde işlemcilerin 4-6-8 gibi sayılarda çekirdek adetleri bulunmaktadır. Daha kısa sürede sonuç almak için ve işlemcilere düşen iş yükünü azaltmak için paralel programlama teknikleri geliştirilmiştir. Burada en önemli soru şudur: eğer problem $k$ adet aynı boyutta parçalara bölünerek, çekirdeklere eşit olarak dağıtılırsa toplam süre ne kadar kısaltılabilir? Çekirdeklerdeki işlem karmaşıklığı değişmemektedir, ancak parçaların boyutları $m = n / k$ adet olacağı için süre kazancı elde edilir ve işlemin hızlanma değeri Formül -1'deki gibi hesaplanmaktadır:

Hızlanma (speed-up): $S = t_{seri} / t_{paralel}$ (Formül-1)

Burada:

- $t_{seri}$ : $n$ adetlik verinin tek işlemcide sıralanma süresi,
- $t_{paralel}$ : $m$ adetlik veri parçalarının $k$ işlemcide sıralanma süresi olmaktadır.

O halde $O( n \lg n )$ karmaşıklığa sahip bir algoritma ile çalışılırken, iş yükü çekirdeklere eşit olarak dağıtılıp, paralel olarak hepsi birden çalıştırılırsa ve hepsi aynı anda sonlanırsa sıralama işlemi karmaşıklığı $O( m \lg m )$ olmaktadır. Görüleceği gibi karmaşıklık notasyonu değişmez, ama parça boyutları küçük olduğundan süre kısalır. Eğer $n$ çok büyük bir sayı ise $\lg( n ) \approx \lg( m )$ (yaklaşık eşit) olur. Dolayısı ile beklenen hızlanma da $S \approx n / m = k$ yani işlemci sayısı kadar olmaktadır. Eğer sıralama algoritması olarak $O( n^2 )$ işlem karmaşıklığı olan "Seçerek Sıralama" (selection sort) (Sedgewick ve Wayne, 2011 (3)) veya aynı yavaşlıkta bir algoritma kullanılırsa, dağıtılmış sıralama işleminin karmaşıklığı tam olarak $O( m^2 )$ olur, hızlanma da $S = k^2$ olmaktadır. Özetle sıralama işlemleri teorik olarak $k$ adet çekirdekli işlemcide $k$ ile $k^2$ aralığında hızlanma beklenmelidir. Ancak algoritmalarda seri

kısımlar da olduğundan hiçbir zaman tümüyle paralel yapılamazlar, dolayısıyla bu hızlara ulaşılamamaktadır.

Temel sıralama problemi işlemci ve çekirdeklere dağıtılınca, her çekirdekte ayrı ayrı yapılan sıralamanın sonucu çekirdeklerin kendi özelinde, yani yerelde sıralanmış parçalar halinde olmaktadır. Bunların bir merkezde sonuç veri alanında tekrar sıralanarak birleştirilmesi (İngilizce merge işlemi) gereklidir. Birleştirme algoritması aslında okuma ve karşılaştırma yaparak sonuç alana veri yerleştirme işlemi olduğundan problem boyutu ile doğru orantılıdır, yani $O(n)$ notasyonuna sahiptir. $O(n)$ karmaşıklığı da $O(n \lg n)$ veya $O(n^2)$ karmaşıklıklarına göre çok daha kolay yapılan bir işlemi göstermektedir.

Sıralama algoritmalarının iş yükünün işlemci çekirdeklerine eşit olarak paylaştırılmasına yük dengelemeli paralel sıralama algoritmaları denilmektedir. Bu alanda pek çok çalışma yapılmıştır. En yeni derleme yayınlarından birisi Yu ve Li tarafından 20 civarında yayın incelenerek yapılmıştır (Yu ve Li, 2022). Bu ve benzeri yayınlardan hareketle literatür incelendiğinde mevcut algoritmaların iş yükünü çekirdeklere dağıtınca nasıl farklılaştıkları, işlemci ve bellek kullanımının etkinleştirildiği, birkaç tekniğin birlikte, melezleme (hibrit) kullandığı görülmektedir. Örneğin belli bir seviyeye kadar quicksort ile çalışılıp, sayılar azaldığında selection-sort yapmak gibi. Bunun en önemli sebebi yavaş algoritmaların bile problem boyutu küçülünce diğerlerinden daha hızlı olabilmesidir. Algoritmaları kıyaslarken, hızlanma oranı, toplam zamanı ne kadar kısalttığı, saniyedeki sıralama oranı, iş yükü dengesi, ön belleği ve ana belleği ne kadar etkin kullandığı, çakışma azlığı gibi ölçütler kullanılmaktadır (Yu ve Li, 2022). Benzer şekilde Lumunge, 2021 tarihli web sitesi paylaşımında detaylı açıklamalar ve algoritmalar vermiştir.

Bu çalışmada birkaç sıralama algoritması dengeli yüklenerek, paralel olarak yürütülmüş ve parçaların geri toplanması için melez bir yöntem önerilerek, sonuçlar değerlendirilmiştir.

## 2. İlgili Çalışmalar

Daha önce yapılan çalışmalar incelendiğinde, araştırmacıların paralel işlem gücü elde etmek için çok çekirdekli işlemciler, ekran kartları veya diğer dağıtık hesaplama ortamları ile çalıştıkları görülmektedir. Algoritma olarak da birleştirmeli (merge-sort), taban sıralama (radix-sort), baloncuk (bubble) ve quicksort gibi temel algoritmalar tercih edilmektedir.

İlk olarak Richard Cole tarafından önerilen paralel merge-sort yöntemi, işlemci üzerinde, yük dengeleme, bellek kullanımı ve tek-çift fazlı merge olarak çalışılmıştır (Cole, 1988; Akl ve Santoro, 1987; Jeon ve Kim, 2003; Herruzo ve ark, 2007; Oded ve ark, 2012). Merge-sort ekran kartında (GPU) da oldukça hızlı çalışmaktadır. GPU ile yapılan çalışmalarda; GPU'larda etkin sıralama algoritmaları ve merge-sort (Satish ve ark, 2009), OpenCL kullanarak GPU ile tek-çift merge-sort'un hızlandırılması (Zhang ve ark, 2011), merge-sort birleştirme rotası (Green ve ark, 2014), GPU ve matris sıralama tekniği ile merge-sort (Panwar ve ark, 2014), GPU üzerinde çalıştırılan ikili eşleşmeli merge-sort'un en kötü durum girdi optimizasyonu (Berney ve Sitchinava, 2020) çalışmaları öne çıkmaktadır. Ayrıca Cole 2017'de işlemci, ekran kartı ve diğer donanımlarla paralel birleştirme (merge) ve diğer sıralama algoritmalarını çalışarak çok önemli başarımlar elde etmiştir.

Merge-sort sıralama yapabilmek için ek bellek kullanmaktadır, ek bellek ihtiyacı olmayan (in-place) paralel algoritmalar; baloncuk sıralama, quick-sort, radix sort olarak öne çıkmaktadır. İşlemciler üzerinde paralel quicksort ile Langr ve Schovánková (2021), radix-

sort üzerine işlemcilerle Cho ve arkadaşları (2015), ekran kartı ile de Xiao ve arkadaşları (2020) çalışmışlardır.

Bu çalışmada da kullanılan OpenMP kütüphanesinin merge-sort, quicksort, counting-sort, bubble-sort sıralama algoritmaları üzerindeki performansı Takayuki ve Shuhei tarafından çalışılmıştır (2015). Sadece baloncuk sıralama (buble-sort) üzerine (Alyasser, 2014) ve x86 tabanlı işlemcilerle paralel sıralama için otomatik vektörler oluşturan çalışmalar da (Hou ve ark, 2018) bulunmaktadır.

Ekran kartı kullanan hızlı melez algoritmalar çok daha önceden çalışılmıştır (Sintorn ve Assarsson, 2008). Yazarların çalışması bu çalışmamızda kullanılan yönteme benzemektedir, ancak quick-sort ile çok küçük boyutta parçacıklara kadar inip, ardından ekran kartında çalışan paralel merge-sort çalıştırmayı tercih etmişlerdir. Yine ekran kartı ile daha güncel CUDA platformunda çalışılmış yayınlar da mevcuttur (Valerievich ve ark, 2017). Çok çekirdekli, işlemci ağları yapılarında farklı sıralamaların denendiği yayınlar da bulunmaktadır (Chowdhury ve ark, 2013).

## 3. Materyal ve Yöntem

Sıralama algoritmaları değişik özelliklerine göre birkaç gruba ayrılmaktadır. Öncelikle çalışma ortamı farklılıkları önemlidir, yani algoritmanın nasıl bir cihazda kullanıldığı, işlemcide mi, ekran kartında mı çalıştırıldığı şeklinde farklılıklar vardır. Bazı algoritmalar da bellek kullanımı açısından farklıdır. Veriyi kendi içinde, ek bellek kullanmadan (in-place) sıralayanlar ve ek belleğe ihtiyacı olanlar bulunmaktadır. Bu çalışmada ortam olarak, işlemci çekirdekleri ve ana bellek kullanılmıştır. Orijinal isimleri quicksort ve selection sort olan algoritmalar ek bellek kullanmamakta, ancak parçalı sıralanmış verileri birleştiren (orijinal ismi merge olan) algoritma ise ek bellek gerektirmektedir.

**- Yer değiştirme fonksiyonu (Swap)**

Sıralama işleminin en önemli parçası yer değiştirmedir. Kaç adet yer değiştirme yapıldığı da bir ölçüt olabilmektedir.

**Algoritma-1:**

```
Swap(A, i, j) // i ve j pozisyonlarını yer değiştir
   temp = A[i], A[i] = A[j], A[j] = temp
```

Algoritma-1, dizi boyutu ile ilgili bir işlem yapmaz. Dolayısıyla bu tür basit fonksiyonların karmaşıklığı $O(1)$ olmaktadır.

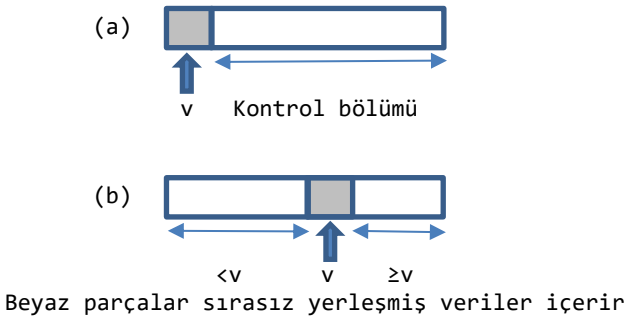
**- Hızlı sıralayıcı (Quicksort)**

Algoritma Partition ve Quicksort olarak iki parçalıdır. Algoritma-2, diziyi verilen aralıkta incelemekte, `Partition` (bölütleme) kısmı görevini bitirdiğinde, referans sayı (algoritmada değeri *v,* konumu *k* olmaktadır) sıralanmış dizide tam yerine yerleşmektedir. İşlemin başlangıcı Şekil 1 (a) ve sonucu Şekil 1 (b)'de verilmiştir. Şekil 1 (b)'de referans değer olması gereken yerde, kendinden küçükler solunda, kendinden büyük ve eşitler sağında, tüm sayılar sırasız olarak yerleşirler. Yani işlem sadece referansın yerini tam olarak bulmaktadır.

**Algoritma-2:**

```
Partition(A, min, max)
   i = min, j = max+1 // sol ve sağ işaretçiler
   v = A[min]         // referans
   while (true)
      while (A[++i] < v) if (i == max) break
      while (v < A[--j]) if (j == min) break
      if (i >= j) break
      Swap(A, i, j) // i ve j pozisyonlarını yer değiştir
   Swap(A, p, j)   // referansın yeri j'dir ve gerçek yeridir
   return j        // A[p..j-1] <= A[j] <= A[j+1..q]

Quicksort(A, p, q)
   if (p > q) return
   k = Partition(A, p, q)
   Quicksort(A, p, k-1)
   Quicksort(A, k+1, q)
   return
```




**Şekil-1.** (a) Üstte: Bölütleme öncesi, (b) Altta: Bölütleme sonrası durumlar.

***Figure-1.*** *(a) Top: Before Partition, (b) Below: After Partition cases.*

Dolayısıyla bulunan referans değerinin pozisyonu olan *k* noktasına dokunmadan quicksort soldaki ve sağdaki parçalar için tekrar çağrılmaktadır. Öz-yinelemeli olduğu için bu işlem dizinin boyutunun 2 tabanında logaritması (yani lg( *n* )) kadar çağrımla bitmektedir. İşlem bittiğinde dizideki tüm sayılar bir bölütlemeden geçmiş, geçerken de her biri mutlaka bir yerde referans değer olmuştur, dolayısıyla yerine yerleştirilmiştir. lg( *n* ) adımın her aşamasında *n* adet sayı işlendiği için de toplam karmaşıklık *O*( *n* lg *n* ) olmaktadır (Horowitz ve ark., 1998).

**- Seçerek sıralayıcı (Selection Sort)**

Algoritma iç içe iki döngü kullanır.

**Algoritma-3:**

```
SelectionSort(A, p, q)
   for (i=p to q-1)
      min = i // Minimum sayı bu pozisyondadır
      for (j=i+1 to q)
         if (A[j] < A[min])    // Daha küçük bir değer var mı?
            min = j            // Varsa yeni minimum pozisyonu atanır
      Swap(A, i, min)          // Bulunan minimum sayı başa alınır
```

Algoritma-3 diziyi verilen aralıkta incelemekte, ilk seferinde bulunan minimum sayıyı en başa kaydetmekte, sağda kalan *n*-1 sayı için aynı işlemi tekrar yapmaktadır. Her seferinde bir eksik işlem yapılmaktadır. Dolayısıyla (*n* - 1) + (*n* - 2) + .. + 2 + 1 kadar işlem, yani toplamda *n*(n-1)/2 işlem yapılmaktadır. Toplam karmaşıklık $O(n^2)$ olmaktadır (Horowitz ve ark., 1998).

**- Birleştirme (Merge)**

Bu yayına özel olarak MergeSort algoritmasına ait merge kısmından paralel programlama tekniği ile geliştirilmiştir, parça parça sıralanmış yerel veribloklarını bir sonuç dizisinde birleştirmektedir.

**Algoritma-4:**

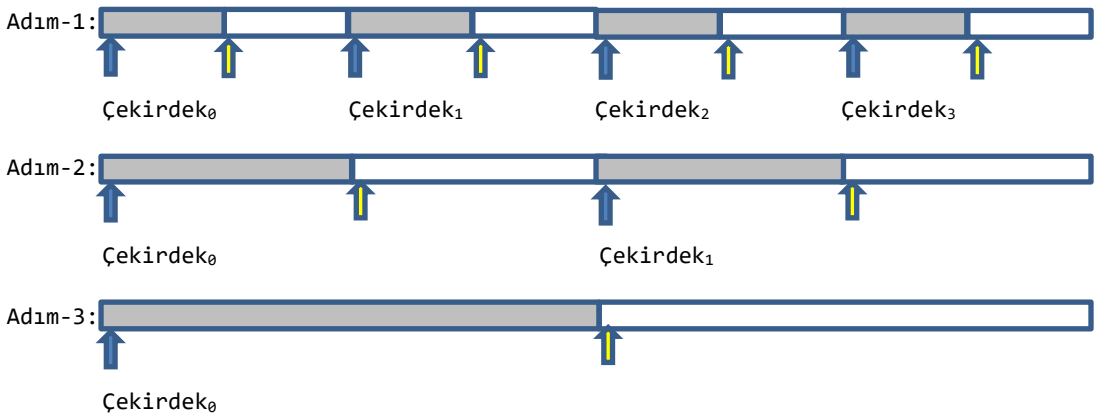
```
Merge(A, B, min, max) {
   msize = (max-min) / 2
   // Alt listelerin ilk ve son elemanları
   first1 = min
   first2 = first1 + msize
   last1 = first2
   last2 = first2 + msize
   if ((last2+msize) > N) max = last2 = N; // Son parça düzeltmesi

   // işaretçiler
   p1 = first1
   p2 = first2

   for(k = min to max-1)
      if (p1 < last1)
         if (p2 < last2)
            if (A[p1] < A[p2])
               B[k] = A[p1++]
            else
               B[k] = A[p2++]
         else
            B[k] = A[p1++]
      else
         B[k] = A[p2++]
   // Sıralanmış veri asıl diziye kopyalanır
   for (k = min to max)
      A[k] = B[k]
```

Algoritma-4 dizi üzerinde Şekil-2'deki gibi komşu parçaların elemanlarını sıralayarak birleştirme işlemi yapmaktadır. Bu aşamada da her **`Merge`** ayrı çekirdekte paralel çalıştırlır, *p* çekirdek sayısı iken, lg( *p* ) kez n sayı üzerinde, iki döngü şeklinde tekrarlanan toplam *n* x 2lg( *p* ) işlemin toplam karmaşıklığı 2lg( *p* ) çok küçük olduğundan *O*( *n* ) olmaktadır.

Algoritmanın çalışması şu şekildedir: İşi alan çekirdek kendi parçasının başına *p*1 işaretçisini (koyu renkli oklar), sağ yan komşu parçanın başına da *p*2 işaretçisini koymaktadır (açık renkli oklar). İlk aşamada çekirdeklerin yarısı çalışmaktadır. Her aşamada çekirdek sayısı yarıya düşmektedir. *p*1 ve *p*2 işaretçileri ilerlerken gösterdikleri pozisyonlardan değer olarak hangisi küçük ise o sayı sonuca yazılıp, işaretçisi bir ilerletilmektedir, *p*1 veya *p*2'den herhangi birisi kendi sınırına geldiğinde diğer parçada kalan veriler olduğu gibi kopyalanmaktadır.




**Şekil-2.** 8 çekirdekli bir sistemde, kendi içinde sıralı her veri parçası bir işlemci çekirdeği tarafından sağındaki parça ile birleştirilir. Aşamalar ilerledikçe çalışan çekirdek sayısı azalır.

**Figure-2.** In the eight cores system, each partially sorted data part merged with right part by a CPU-core. Number of busy cores is decreasing every steps.

## - OpenMP kütüphanesi ve ParalelQuickSort önerileri

OpenMP **(open multiprocessing)** kâr amacı gütmeyen bir birliktir ve geliştirilen kütüphane paralel uygulamalar yazmak için programcılara basit ve uygun bir arayüz veren ölçeklenebilir ve taşınabilir bir model kullanır (Wikipedia-OpenMP, 2022; OpenMP, 2022). Sistem olarak standart masaüstü bilgisayarlardan, süper bilgisayarlara varana kadar geniş bir kullanımı vardır.

OpenMP C++ ve Fortran ile çalışmaktadır. OpenMP programlama tekniğinde derleyici direktifleri ile kodun içerisindeki paralel yapılmak istenen kısımlar belirtilir. Örneğin: **`#pragma omp parallel for`** denildiğinde ardından gelen for döngüsü işlemci çekirdeklerince paylaşılarak birlikte çalıştırılır.

Sıralama algoritmaları OpenMP'de kullanılmak üzere düzenlenebilir. Algoritma-5'te OpenMP kütüphanesinin sürüm 3.0 ve üzerine uygun ParallelQuickSort algoritması verilmektedir (OpenMP-Sort, 2016). Bu çalışmada ise işlemci çekirdekleri sadece kendi alanlarındaki sayıları sıraladıkları için bu türden karmaşık yollara gerek olmamıştır.

Algoritmada geçen 'task' görev demektir. Bu teknikte OpenMP görev paylaşımını kendisi yönetmektedir. Görevler geldikçe boşta olan süreçler görevleri almaktadır. Ancak bu görev yönetim tarzı karmaşık bir işlem olduğundan biraz yavaşlama yaşatmaktadır.

**Algoritma-5:**

```
ParallelQuickSort(A, min, max) {
  if (min < max)
    k = Partition(A, min, max) // Algoritma-2'deki ile aynı

    #pragma omp task default(none) firstprivate(min, k)
    {
      ParallelQuickSort(A, min, k - 1)
    }
    #pragma omp task default(none) firstprivate(max, k)
    {
      ParallelQuickSort(A, k + 1, max)
    }
```

Ayrıca Lumunge'nin yayınında verilen Optimized Quick Sort yayın için denenmiştir, ancak çok yavaş olduğu görülmüştür (Lumunge, 2021). Bu algoritma Algoritma-6'da verilmiştir. Lumunge ayrıca, bu çalışmamızda önerilen yönteme benzer (3 nolu yaklaşımı) bir yöntemden de bahsetmiştir, ancak bu yayında önerilen yöntem çok daha net ve basittir. Lumunge'nin yayınında olduğu gibi işlemcilerin veri paylaşımı yapmasına gerek olmamıştır.

**Algoritma-6:**

```
OptimizedParallelQuickSort(A, min, max) {
  if (min < max)
    k = Partition(A, min, max) // Algoritma-2'deki ile aynı

    #pragma omp parallel
    #pragma omp sections nowait
    {
      #pragma omp section
      {
        OptimizedParallelQuickSort(A, min, k - 1)
      }
      #pragma omp section
      {
        OptimizedParallelQuickSort(A, k + 1, max)
      }
    }
```

Algoritmanın mantığı her quicksort çağrımını ayrı alt bölümler (section) tanımlayarak uygun süreçlere yüklemektir, ancak bu işlem çok yavaş kalmaktadır. Hatta seri halinden bile yavaş olmaktadır. Bu algoritma da denenmiş, ancak karşılaştırmalarda kullanılmamıştır.

## 4. Bulgular ve Tartışma

Bu çalışmada C++ dili ve OpenMP kütüphanesi kullanılmıştır, derleme amacıyla 2015'te çıkan OpenMP 4.5 sürümünü desteklediği için MinGW kurulmuştur. Visual Studio'da da OpenMP desteği vardır, ancak eski bir sürümü olan 2.5 desteklenir ve task gibi yeni direktifler yazılamaz. Kullanılan makine 4 gerçek + 4 sanal çekirdekli, 8 GB bellekli, Intel i5 işlemcili bir dizüstü bilgisayardır.

**Çizelge-1.** Test edilen algoritmalar ve kullanılan kısaltmaları.

***Table-1.*** *Tested algorithms and its abbreviations.*

| **Test edilen algoritmalar ve kısaltmaları** | | |
|---|---|---|
| **1** | **SSort-Tekli** | Seçmeli sıralama (Selection Sort) |
| **2** | **QSort-Tekli** | Tek işlemcili Quicksort |
| **3** | **QSort-Paralel** | OpenMP task kullanan paralel Quicksort |
| **4** | **SSort-Oneri** | Çalışmada önerilen melez seçmeli sıralama |
| **5** | **QSort-Oneri** | Çalışmada önerilen melez Quicksort |

Bu testler için aşağıdaki algoritmaların süre ve hızlanma ölçümleri yapılmıştır. Önerilen melez sıralamalarda Merge fonksiyonu sonucu toparlamakta kullanılmaktadır.

**1. SSort-Tekli** ölçümü için Algortima-3'de verilen fonksiyonun `SelectionSort(A, 0, N-1)` şeklinde çağrımı yapılır ve süre ölçülür.

**2. QSort-Tekli** ölçümü için Algortima-2'de verilen fonksiyonun `QuickSort(A, 0, N-1)` şeklinde çağrımı yapılır ve süre ölçülür.

**3. QSort-Task** ölçümü için Algoritma-5'deki OpenMP task kullanan `ParalelQuicksort` algoritması OpenMP sitesinden bakılarak uyarlanmış ve süre ölçümü yapılmıştır. Bu algoritmanın, programın main kısmından çağrılması için aşağıdaki kod yazılmalıdır.

```
#pragma omp parallel default(none)
{
   #pragma omp single nowait
   {
      ParallelQuickSort(0, N - 1)
   }
}
```

**4. SSort-Oneri** ve **5. QSort-Oneri** için işlemci çekirdeklerini kullandıran OpenMP direktifleri yazılmıştır. Bu ölçümler için Algortima-2, 3 ve 4'deki `QuickSort(...)`, `SelectionSort(...)` ve `Merge(...)` çağrımları aşağıdaki gibi melezlenip süre ölçümleri alınmıştır.

```
size = omp_get_num_procs()  // Toplam çekirdek sayısı
psize = N / size            // Verilerin bir çekirdeğe düşen boyutu

#pragma omp parallel private(myid)
{
   myid = omp_get_thread_num()
   // Çekirdek görevi için bölge sınırları min ve max hesaplanır
   min = myid * psize
   max = min + psize – 1
   if (max+psize > N) max = N-1
   SortAlgorithm(A, min, max) // SelectionSort veya QuickSort
}

while(size > 1)
   size = size / 2
   msize = N / size  // Verilerin bir çekirdeğe düşen boyutu

   #pragma omp parallel private(myid)
   {
      myid = omp_get_thread_num() // Kaç numaralı çekirdek olduğunu öğrenmek
      if (myid < size)
         min = myid * msize
         max = min + msize – 1
         if (max+msize > N) max = N-1
         Merge(A, B, min, max)
   }
```

OpenMP mevcut tüm işlemcileri ve çekirdekleri görmektedir. `size = omp_get_num_procs()` ile kaç çekirdek sahibi olunduğu öğrenilmektedir. Çalışma anında aksi belirtilmedikçe, her çekirdeğe bir iş parçacığı (thread) atanmaktadır. OpenMP'de işlemci çekirdekleri sıfırdan başlayarak numara alırlar ve `myid = omp_get_thread_num()` ile işlemci çekirdeği numarası öğrenilmektedir. Buna göre çalışılacak alanın sınırları belirlenir. Kodda da görüleceği gibi herhangi bir sıralama algoritması `SortAlgorithm` yazılı kısma konularak test edilebilmektedir. Bu çalışmanın katkısı, işlemci çekirdeklerine verilen görevleri net belirlemek, parçalı sıralı sonucu paralel işlem gücünü kullanarak, birleştirme (`Merge`)

fonksiyonu ile çekirdeklere yaptırmaktır. Önerideki melez yöntemle, çekirdeklere dengeli dağıtılan iş yükü hız getirmektedir.

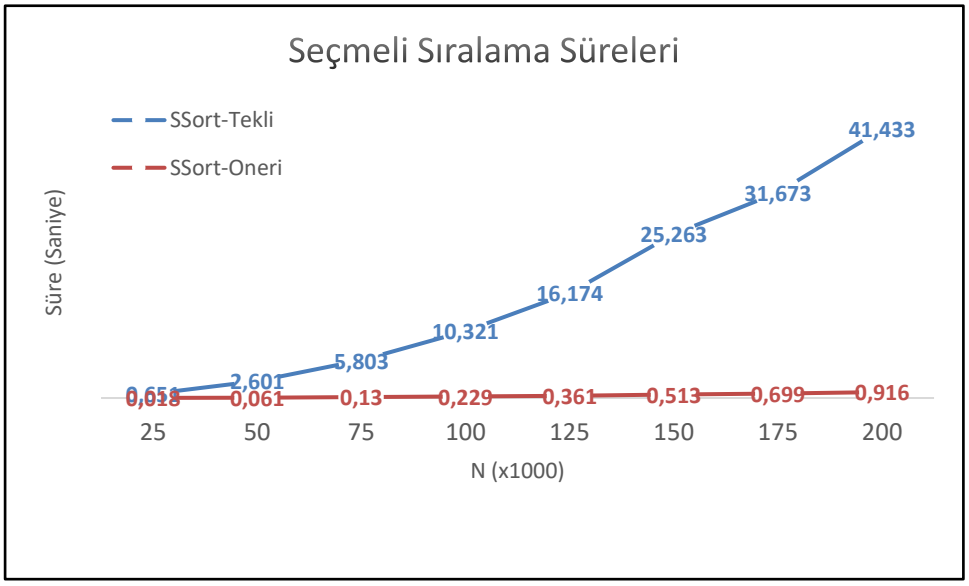
**Seçmeli sıralama üzerine testler:**

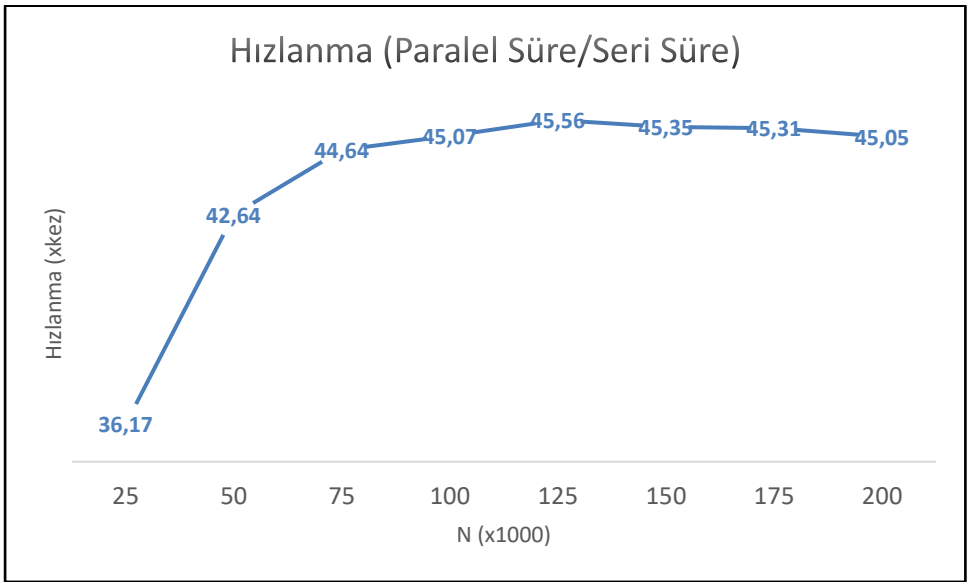
Bu sıralamayı ayrı test etmek gerekmektedir çünkü $O(n^2)$ karmaşıklıkta bir algoritmaya milyon sayı sıralatmak istenirse saatlerce sürmektedir.

**Çizelge-2.** Seçmeli sıralama testleri.

***Table-2.*** *Tests of selection sorts.*

| Seçmeli Sıralama süreleri ve hızlanma | | | | |
|---|---|---|---|---|
| ***N* (bin)** | **SSort-Tekli (saniye)** | **SSort-Oneri (saniye)** | **Merge Oranı (%)** | **Hızlanma** |
| **25** | 0,651 | 0,018 | 0,0 | 36,17 |
| **50** | 2,601 | 0,061 | 0,0 | 42,64 |
| **75** | 5,803 | 0,130 | 0,77 | 44,64 |
| **100** | 10,321 | 0,229 | 0,44 | 45,07 |
| **125** | 16,174 | 0,361 | 0,28 | 45,56 |
| **150** | 25,263 | 0,513 | 0,78 | 45,35 |
| **175** | 31,673 | 0,699 | 0,29 | 45,31 |
| **200** | 41,433 | 0,916 | 0,63 | 45,05 |






**Şekil-3.** Seçmeli sıralama algoritması için 25 binlik dilimlerde (üstte) tek ve 8 çekirdekte saniye olarak süreler, (altta) sürelerin oranı (hızlanma)

**Figure-3.** *For the selection sort algorithm, in 25 thousand slices (top) times in seconds for using single and 8 cores, (below) rate of times (speed-up)*

Çizelge 2 ve Şekil 3'teki sürelere bakıldığında $O(n^2)$ algoritmalardaki süre artışının ne kadar dramatik olduğu görülmektedir. Bunun nedeni bu tür algoritmaların basit ama büyük sayıda verileri sıralamaya uygun olmamasıdır. Yine de çok küçük verilerle çalışılırken tercih edilirler. Ancak önerilen paralel seçmeli sıralama algoritması ve birleştiren algoritmanın melezlenmiş kullanımının 45 kat hızlı olduğu gözlenmiştir. Buna göre 200 binlik verilerle 40 saniyeyi aşan sıralama yapmak yerine, çalışmada önerilen yöntemle 1 saniyenin altında kalınır. Veya önerilen yöntemle çok daha büyük verilerle çalışılabilir. Örneğin 1 milyonluk veri ile (her çekirdekte 125 binlik veri) önerilen algoritma 27,86 saniye sürmüştür. Bunun da hızlandırılması için veri büyüdükçe daha fazla parçalarla çalışmak çözüm olabilir. Birleştirme (merge) işleminin grafiği çizilmemiştir çünkü sağlıklı veriler elde edilememiştir. Birleştirme işlemi toplam sürenin %1'inden az çıkmaktadır. Veri büyüdükçe birleştirme oranı çok daha düşmektedir.

## Hızlı sıralama (quicksort) üzerine testler:

Bu sıralama $O(n \lg n)$ karmaşıklıkla milyonlarca sayıyı sıralanmak için uygundur. Çizelge 3 ve Şekil 4 sonuçlarına bakıldığında $O(n \lg n)$ algoritmalardaki süre artışının seçerek sıralayan yönteme göre daha mantıklı arttığı görülmektedir. Veri boyutu ne olursa olsun, quicksort varyasyonları iyi olmaktadır. Ancak tüm iş yükü çekirdeklere dağıtıldığında gözlemlenen; OpenMP'nin task kullanan yöntemi bu çalışmadaki önerilen yöntemden yavaş kalmaktadır. Sebebi görev dağıtımının uğraşılması gereken bir işlem olması olabilir. Bu çalışmada önerilen teknikte ise temiz bir şekilde çalışma alanını belirleme ve birleştirme yapılmaktadır.

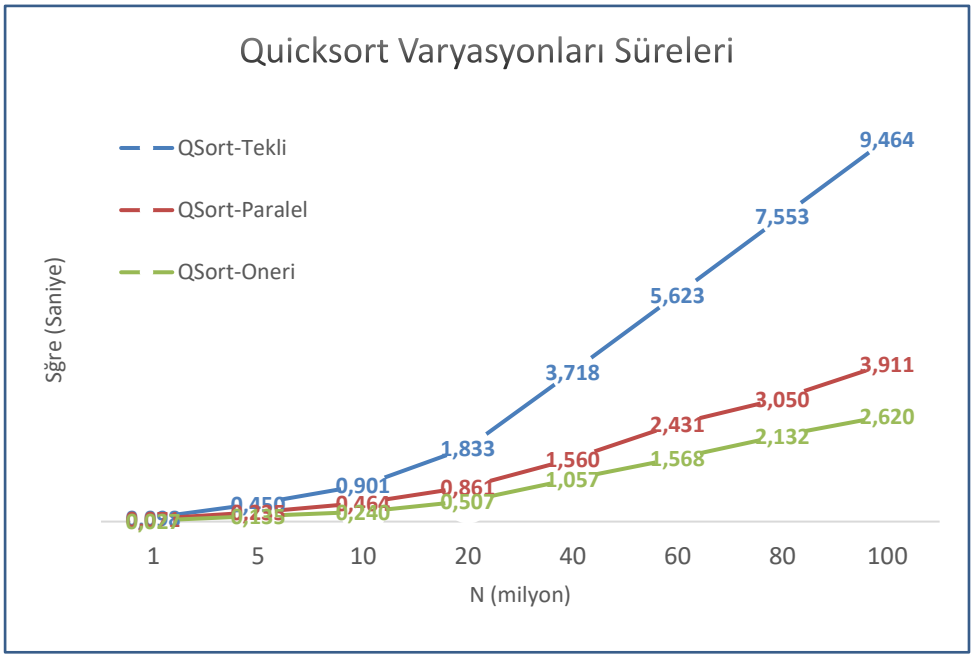
Çizelge 3 ve Şekil 4 incelendiğinde; hızlanmalar yaklaşık olarak eşit seviyede devam etmektedir. Bu çalışmadaki önerideki hızlanmalar çok daha iyi çıkmaktadır, sonuçlardan hareketle sabit 3,5 kat hızlanılır denilebilir. Çekirdek sayısı 4 gerçek ve 4 sanal olduğu için 4 veya üstü beklenebilir. Hızlanmanın biraz düşük olmasının nedeni birleştirme işleminin de süre açısından bir yük getirmesidir. Veri arttıkça bu oran düşmekte, ancak bu düşüş seçmeli

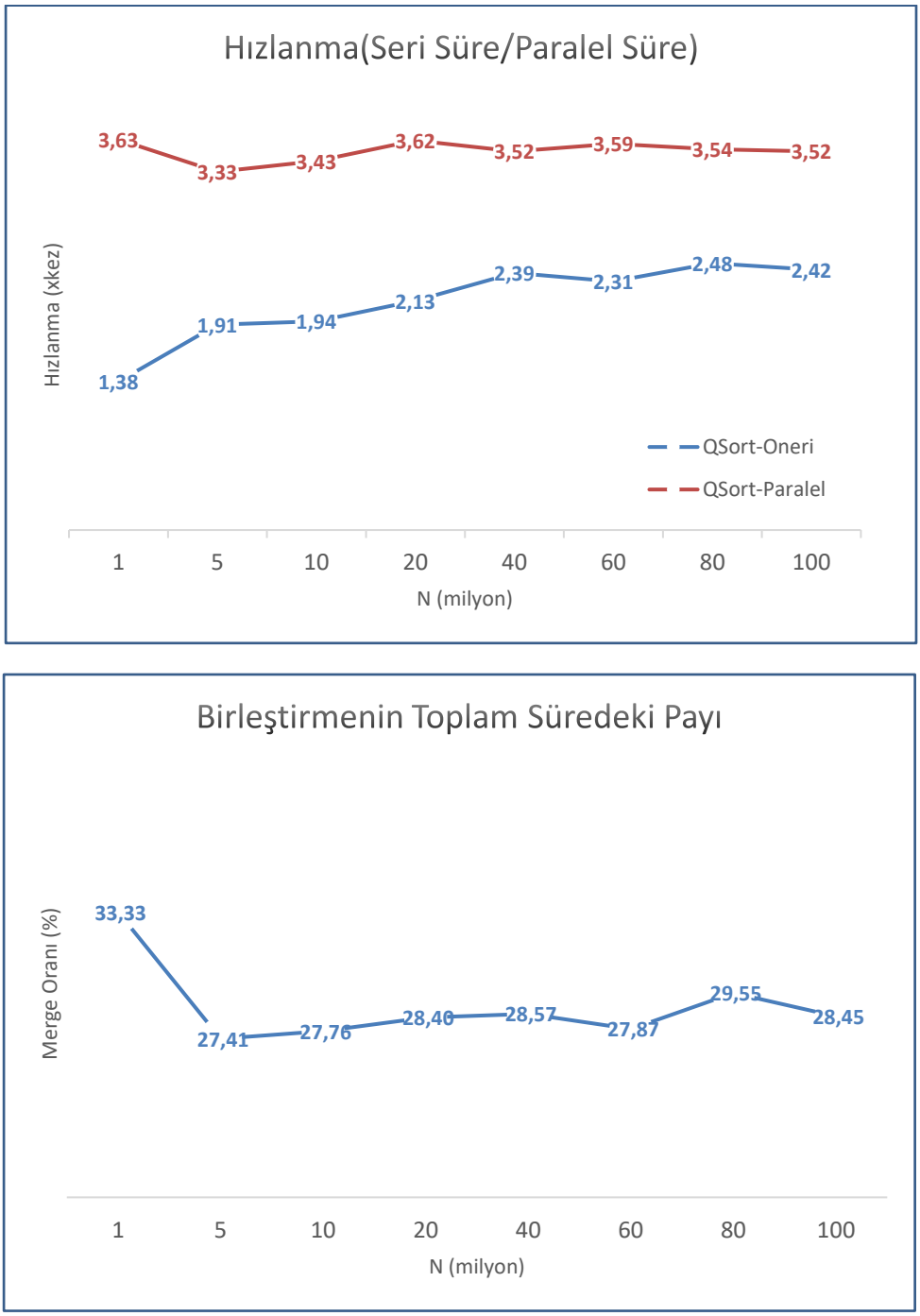
sıralamadaki gibi çok ta önemli bir seviyede olmamaktadır. Bunun sebebi sıralamanın $O(n \lg n)$ karmaşıklığı ile, birleştirmenin $O(n)$ karmaşıklığı arasındaki farkın $\lg(n)$ olmasıdır. Veri büyüse de aradaki fark $\lg(n)$'den dolayı fazla açılmamaktadır.

**Çizelge-3.** Quicksort varyasyonları testleri.

***Table-3.*** *Tests of quicksort variations.*

| | Quicksort varyasyonları ile sıralama süreleri ve hızlanma | | | | | |
|---|---|---|---|---|---|---|
| ***N* (milyon)** | **QSort-Tekli (saniye)** | **QSort-Paralel (saniye)** | **QSort-Paralel Hızlanma** | **QSort-Oneri (saniye)** | **Merge Oranı (%)** | **QSort-Oneri Hızlanma** |
| **1** | 0,098 | 0,071 | 1,38 | 0,027 | 33,33 | 3,63 |
| **5** | 0,450 | 0,235 | 1,91 | 0,135 | 27,41 | 3,33 |
| **10** | 0,901 | 0,464 | 1,94 | 0,240 | 27,76 | 3,43 |
| **20** | 1,833 | 0,861 | 2,13 | 0,507 | 28,40 | 3,62 |
| **40** | 3,718 | 1,560 | 2,39 | 1,057 | 28,57 | 3,52 |
| **60** | 5,623 | 2,431 | 2,31 | 1,568 | 27,87 | 3,59 |
| **80** | 7,553 | 3,050 | 2,48 | 2,132 | 29,55 | 3,54 |
| **100** | 9,464 | 3,911 | 2,42 | 2,620 | 28,45 | 3,52 |






**Şekil-4.** Quicksort algoritmaları için (üstte) tek çekirdek ve 8 çekirdekte OpenMP'nin önerisi ve bu çalışmadaki önerinin saniye olarak süreleri, (ortada) OpenMP'nin önerisi ve bu çalışmadaki önerinin sürelerinin oranları (hızlanma) ve (altta) önerimizdeki birleştirmenin toplam süreye oranı

**Figure-4.** *For the quicksort algorithm, (top) times in seconds of single core and with 8 cores, OpenMP and our suggestions, (middle) rate of times (speed-up), and (below) percentage of merging to total time of our suggestion.*

Birleştirme süresinin toplam süreye oranı %30'un altında kalmaktadır. Bu süreyi azalmak mümkün olmamıştır. Test amacıyla yapılan ve hiç kontrol veya birleştirme yapmadan, sadece bir bellek alanını başka bir alana kopyalama işlemi bile %12 oranında ölçülmüştür. Çekirdek sayısı çok artarsa, sıralamanın birinci aşaması olan parçalı sıralamanın aşırı

hızlanması sonucunda, ikinci aşama olan birleştirmenin bu hıza uyamayıp, toplam sürenin yarısını oluşturacağı açıktır.

## 5. Sonuç

Bu çalışmada ev bilgisayarlarında dahi bulunan ama genelde çalıştırılmayan işlemci çekirdeklerini maksimum seviyede meşgul ederek sıralama algoritmalarının ne kadar hızlanabileceği gösterilmiştir. Ancak literatürde bilinen seri algoritmalar genelde en iyi hallerinde değildir (optimize edilmeleri gerekir). OpenMP'nin kendi önerdiği paralel algoritma bile bu çalışmadaki teknikten yavaş kalmaktadır.

Ekran kartlarından biraz daha hızlı sıralama sonuçları alınabilir. Önerilen yöntem ekran kartlarına yaptırılırsa daha da hızlanacağı düşünülür, fakat ekran kartları hesap yapmada hızlıdırlar, verilerde yer değiştirmek pek te ekran kartına uygun olmayabilir. İyi ekran kartlarının ulaşılabilirliği de az olduğundan, bu çalışmada sadece işlemci çekirdekleri kullanılmıştır.

Kullanılan 8 çekirdekli bilgisayarda hızlanmanın, algoritmanın özelliğine göre 3,5 ile 45 kat aralığında olduğu gözlenmiştir. $O(\ n \lg n\ )$ karmaşıklığa sahip Quicksort algoritmasında çekirdeklere $m = n/8$'lik veri gelmiş ve $O(\ m \lg m\ )$ oranında işlemler bitmiştir. Ancak birleştirme süresi toplam sürenin ortalama %28'ini kaplamaktadır. Bu nedenlerle tam olarak işlemci sayısı kadar hızlanılamamıştır. $O(\ n^2\ )$ sıralama algoritmaları da $8^2 = 64$ kez hızlanamamıştır. Buna hiçbir şekilde paralel çözümü olmayan, seri kalan bölgeler neden olmaktadır. Mükemmel paralellik olsa idi 8-64 aralığına çıkılabilirdi. Günümüzde 4-6-8 çekirdekli bilgisayarlar yaygındır, ancak 16-32 çekirdekli bilgisayarlar yaygınlaştığında önerilen yöntemin etkisi de artacaktır.

## 6. Kaynaklar


Akl S. G. ve Santoro, N., 1987, Optimal Parallel Merging and Sorting Without Memory Conflicts, IEEE Transactions on Computers, vol. C-36, no. 11, 367-1369

Alyasseri, Z.. Parallelize Bubble Sort Algorithm Using OpenMP, Eprint Arşiv: 4(2014):103-110.

Berney K. ve Sitchinava, N., 2020, Engineering Worst-Case Inputs for Pairwise Merge Sort on GPUs, 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 1133-1142

Cho, M., Brand, D., Kulandaisamy, V., Bordawekar, R. ve Puri, R. 2015, PARADIS: An efficient parallel algorithm for in-place radix sort, Proceedings of the VLDB Endowment 8.12(2015):1518-1529

Chowdhury, R.A., Ramachandran, V., Silvestri, F. ve Blakeley, B., Oblivious algorithms for multicores and network of processors, Journal of Parallel & Distributed Computing 73.7(2013), 911-925.

Cole, R., 1988, Parallel merge sort , SIAM Journal on ComputingVol. 17, Iss. 4, 770-785

Cole, R. ve Ramachandran, V. 2017, ACM Transactions on Parallel Computing, Vol.3-4, Article 23, 1–31

Green, O., Mccoll, R. ve Bader, D., 2014. GPU merge path: a GPU merging algorithm. 26th ACM international conference on Supercomputing, 331-340. Web: https://github.com/liuvince/polytech-cuda-project

Herruzo, E. Ruíz, G. ve Plata, O., 2007, A New Parallel Sorting Algorithm based on Odd-Even Mergesort, 15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing (PDP'07), 18-22.

Horowitz, E., Sahnı, S. and Rajasekaran, S., 1998, Fundamentals of Computer Algorithms, Second Edition, Computer Science Press, 154-165

Hou, K., Hao, W. ve Feng, W.C., 2018, A Framework for the Automatic Vectorization of Parallel Sort on x86-Based Processors, IEEE Transactions on Parallel and Distributed Systems, 99

Jeon, M. ve Kim, D., 2003, Parallel Merge Sort with Load Balancing. International Journal of Parallel Programming 31, 21–33

Langr, D. ve Schovánková, K., 2021, A parallel quicksort based on C++ threading, Concurrency and Computation Practice and Experience

Lumunge, E., 2021, Paralel Quick Sort, Web sayfası, ziyaret: 2022, https://iq.opengenus.org/parallel-quicksort/amp/

Odeh, S., Green, O. Mwassi, Z., Shmueli, O. ve Birk Y., 2012, Merge Path - Parallel Merging Made Simple, Parallel & Distributed Processing Symposium Workshops & Phd Forum IEEE.

OpenMP-Sort, Sorting Things Out İnternet Dökümanı, 2016, Ziyaret tarihi: 2022, https://www.openmp.org/wp-content/uploads/sc16-openmp-booth-tasking-ruud.pdf

OpenMP, Ziyaret tarihi: 2022, https://www.openmp.org

Panwar, M., Kumar, M. ve Bhargava, S. 2014, GPU Matrix Sort (An Efficient Implementation of Merge Sort), International Journal of Computer Applications, Vol.89, 9-11.

Satish, N., Harris M. ve Garland, M., 2009, Designing efficient sorting algorithms for manycore GPUs, 2009 IEEE International Symposium on Parallel & Distributed Processing, 1-10

Sedgewick, R. ve Wayne, K., 2011 (1), Algorithms, 4th edition, Princeton University Press, 288-307, Web link: https://algs4.cs.princeton.edu/23quicksort/

Sedgewick, R. ve Wayne, K., 2011 (2), Algorithms, 4th edition, Princeton University Press, 270-287, Web link: https://algs4.cs.princeton.edu/22mergesort/

Sedgewick, R. ve Wayne, K., 2011 (3), Algorithms, 4th edition, Princeton University Press, 248-249, Web link: https://algs4.cs.princeton.edu/21elementary/

Sintorn, E. ve Assarsson, U., 2008, Fast parallel GPU-sorting using a hybrid algorithm. Journal of Parallel Distributed Computing 68(10), 1381-1388

Takayuki, U. ve Shuhei, O., 2015, Performance Comparison of Open-Source Parallel Sorting with OpenMP, Third International Symposium on Computing and Networking (CANDAR), 334-340

Valerievich, B.A., Anatolievna, P.T., Alekseevna, B.M. ve Vladimirovich, S.S., 2017, The implementation on CUDA platform parallel algorithms sort the data, 6th Mediterranean Conference on Embedded Computing (MECO), 1-4

Wikipedia, OpenMP sayfası, Ziyaret tarihi: 2022, https://tr.wikipedia.org/wiki/OpenMP

Wikipedia, Radix Sort sayfası, Ziyaret tarihi: 2022, https://en.wikipedia.org/wiki/Radix_sort

Xiao, S., Li. C., Guo, B. ve Xiao, H., 2020, A radix sorting parallel algorithm suitable for graphic processing unit computing. Concurrency and Computation Practice and Experience

Yu, T. ve Li. W, 2022, A Creativity Survey of Parallel Sorting Algorithm, Cornel Univeristy Press, arXiv preprint

Zhang, K., Li, J., Chen, G. ve Wu, B., 2011, GPU accelerate parallel Odd-Even merge sort: An OpenCL method, Proceedings of the 2011 15th International Conference on Computer Supported Cooperative Work in Design (CSCWD), 76-83