Karadeniz Teknik Üniversitesi
Mühendislik Fakültesi
Bilgisayar Mühendisliği Bölümü

PARALEL BİLGİSAYARLAR

Yüksek Performanslı Hesaplama/Yüksek Başarımlı Hesaplama/ Paralel Bilgisayarlar/Paralel Hesaplama/Multi-Core System Programming Programlama Ödevi

SİSTEM BİLGİSİNİ İNCELEME MATRİS ÇARPMA

2019-2020 Bahar Dönemi

Öğretim	1. Öğretim ☒	2. Öğretim ☐
Numara	330104	
Ad ve Soyad	Onur ERDAŞ	
Dersin Sorumlusu	Dr. Öğr. Üyesi İbrahim SAVRAN	

İçindekiler

1.Projenin Amacı

Projenin amacı bilgisayarın özelliklerini incelemek ve bu özelliklerin Matris çarpma işlemine etkisini gözlemlemektir. Matris çarpımını belirtilen boyutlarda Visual Studio 2019 C++ ile gerçekleştirilmektedir. Sonrasında seri kodlama olarak koşulan program paralel bir platformda koşulmalı ve sonuçların farkları karşılaştırılmalıdırlar. Matris çarpım kodu ile geçen sürenin minimuma indirilmesi amaçlanmaktadır. Bu amaç doğrultusunda değişken tiplerini değiştirerek ve blok ve ardışıl programlama yöntemleri kullanarak sürenin azaltılması hedeflenmektedir. Ayrıca sistemin GFLOPS performansının yüzde kaçına ulaşıldığı gözlemlenmektedir.

2.Projenin Yapım Aşamaları

Gerekli sistem bilgileri verilmiştir ve açıklamalar yapılmıştır. NxN boyutuna sahip iki matris için seri bir şekilde matris çarpımının kodu yazılmıştır. N boyutu belirtilen boyutlarda değiştirilerek sistemin çarpımı yapabilmesi için geçen süreler not alınmıştır. Sonrasında kod paralel olarak koşulmalıdır. Bunun için Visual Studio 2019 içerisinden proje adına sağ tıklanarak özellikler kısmına gidilmiştir. Buradan yapılandırma özellikleri->Dil kısmından OpenMP desteği açılmıştır ve uyumluluk modu hayır olarak gerekli ayarlar yapılmıştır. Kod kısmına “omp.h” kütüphanesi eklenmiştir. Gerekli deyimler yazılarak kod paralleştirilmiştir. Matrisin boyutları değiştirilerek geçen süreler not alınmıştır. Burada sürelerin farkları gözlenmiştir. Matrislerin tipleri değiştirilerek FLOAT ve DOUBLE tipleri için 5 kez seri ve 5 kez paralel toplamda 20 kez kod koşulup geçen süreler not alınmıştır. Burada FLOAT ve DOUBLE için sürelerin farkları gözlemlenmiştir ve FLOAT tipinde matrislerin daha hızlı gerçekleştirdiği gözlemlenmiştir. Sonrasında sistemde belirtilen GFLOPS performansının yüzde kaçına ulaşıldığı hesaplanmış ve gözlemlenmiştir. En son ise paralel olarak koşulan kod blok veri paylaşımı ve ardışıl veri paylaşımına göre kodlanmış ve geçen sürelerin farkları gözlemlenmiştir. Kodlama kısmının altında kullanılan platformlar ile ilgili açıklamalar yapılmıştır.

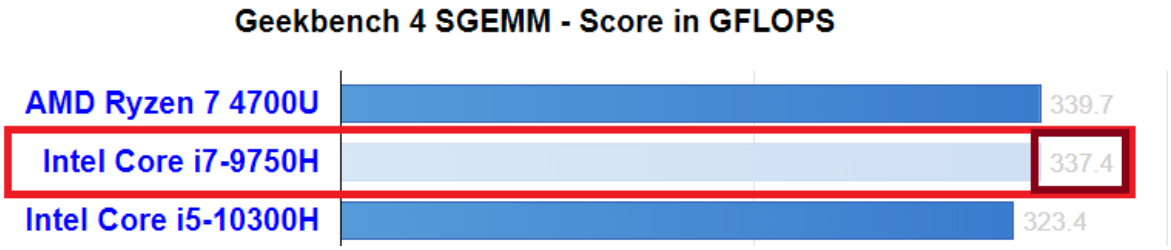
3.Sistem Bilgileri

3.1. Bilgisayarınızın FLOP/s kapasitesi nedir?

Bilgisayarımın CPU için FLOP/s kapasitesi 337,4 GFLOP/s kapasiteye sahiptir. Aşağıdaki şekilde görüldüğü gibi Geekbench tarafından ölçülen değeri 337,4 GFLOP/s olmaktadır. Ayrıca bir sistemdeki FLOPS aşağıdaki formülle hesaplanabilir.

FLOPS = soket * (her soket için core sayısı) * (saniyedeki clock cycle sayısı) * (her cycle için floating point sayısı)

GPU için FLOP/s kapasitesi farklı olmaktadır bunun sebebi grafik kartının daha fazla çekirdeğe sahip olmasıdır. Grafik kartım Nvidia GTX 1660Ti olduğundan Nvidia sitesinin yayınına göre saniyede 11TFLOP/S işlem yapabilmektedir.



3.2. Sistemindeki Cache belleğin özelliklerini açıklayınız, I-Cache, D-Cache kapasiteleri nedir?

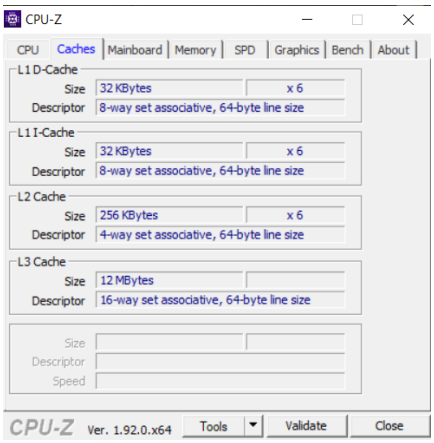
Ön bellek, işlemcideki iç işlemleri hızlandırmak ve yavaş bellekteki komutları yürütürken harcanan zamanı en aza indirmek için geliştirilmiştir. Ön bellek çalışmakta olan programa ait komutların, verilerin geçici olarak saklandığı yüksek hızlı hafızalardır.

- L1 ön bellek (cache) : İşlemcinin daha hızlı ulaşabilmesi amacıyla önemli veriler ve kodlar bellekten buraya kopyalanır. L1 ön bellek kapasitesi 2 KB ile 256 KB arasında değişmektedir. L1 ön bellek 2 bölüme ayrılır (Instruction cache ve Data cache). Instruction cache ve Data cache farklı zamanda yeterliliğe sahiptirler. Bu nedenle ikiye bölünmüşlerdir.

- L2 ön bellek (cache) : L2 ön bellek kapasiteleri 256 KB ile 2 MB arasında değişir. L2 ön bellek çekirdeğin dışında ve işlemciyle aynı yapıda kullanılmaktadır. Bu kısa geçiş döneminden sonraysa L2 önbellek işlemci çekirdeklerine entegre edilmiştir. Her soket için ayrı bir L2 cache olmaktadır.

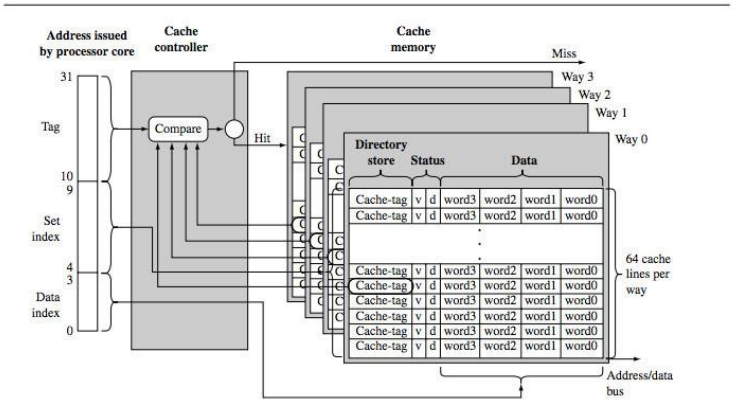
- L3 ön bellek (cache): L3 ön bellek kapasiteleri 2MB ile 256 MB arasında değişir. İşlemcideki en büyük ön bellektir. Yeni bir teknolojidir. Çok çekirdekli işlemcilerde bütün çekirdeklere tek bir bellekle hizmet vermek akıllıca bir yaklaşım olacağı düşüncesiyle geliştirilmiştir.

I-Cache ve D-Cache kapasiteleri aşağıdaki CPU-Z programının çalıştırılması ile elde edilen şekilden elde edilebilir. D-Cache kapasitesi 6x32 KBytes ve I-Cache kapasitesi 6x32 KBytes olmaktadır. Ayrıca aşağıdaki şekilde aynı zamanda sistemin L1,L2 ve L3 ön bellek boyuları görüntülenmektedir.



3.3. Tampon bellekler için 4-yönlü (4-way) ne demektir?

İşlemci bir bloğa erişmek istediğinde önce onun hangi kümede olduğunu belirler. Küme önbellek ilişkilendirmesinde önbellek kümelere ayrılır. Küme içerisinde paralel bir şekilde arama yapılır. Küme ilişkilendirme yönteminde bilgi kesin ve net bir şekilde bir yere eklenmez. Tampon bellekler üzerinde bir bilginin gideceği alanlara göre sayılar olmaktadır. Bu sayı 4 ise tampon bellek 4 yönlü olmaktadır. Yani bilginin gideceği farklı yer sayısı tampon belleklerdeki yön sayısını belirlemektedir. 4 yönlü küme ilişkilendirmesi olan bir tampon bellek her satırında 2048 bloğa sahiptir. Aşağıda



4.Kodlama Kısmı

4.1.Projede NxN Boyutlu 2 Matrisin Çarpımının Yapılması

```
#define N 2000

double dMultiplicationArray[N][N] = { 0 };
double dMatrix1[N][N];
double dMatrix2[N][N];

float fMultiplicationArray[N][N] = { 0 };
float fMatrix1[N][N];
float fMatrix2[N][N];
```

Yukarıdaki kod bölümünde N değeri matrisin boyutunu ifade etmektedir. 1000, 2000, 3000, 4000, 5000 değerlerini buradan değiştirip kodu çalıştırmaktayım. Float ve Double tipleri için ayrı matrisler tanımladım ve fonksiyonlarda bu dizileri kullandım.

4.2.Projenin Seri ve Paralel Kodlanması

Aşağıdaki kod bölümünde double türünden dizilerin seri çarpımı ile ilgili kod bulunmaktadır. Öncelikle çarpımı yapılacak matrisleri 1.0 değeriyle doldurdum. Sonrasında iç içe 3 for kullanarak iki matrisin çarpma işlemini gerçekleştirdim. Burada en üstteki for döngüsü satır , ortadaki for döngüsü sütun ve en içteki for döngüsü ise toplama işlemlerini gerçekleştirmektedir. Açıklama satırına alınmış kod ise çarpılan iki matrisin sonucunu ekranda göstermek için yazılmıştır. Gerekli testleri yapıp çarpım işleminin doğru olduğunu kontrol ettim.

```
void Multiplication_Serial_Double() {

    int i, j, k;
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
        {
            dMatrix1[i][j] = 1.0;
            dMatrix2[i][j] = 1.0;
        }
    for (i = 0; i < N; ++i)
    {
        for (j = 0; j < N; ++j)
        {
            for (k = 0; k < N; ++k)
            {
                dMultiplicationArray[i][j] += dMatrix1[i][k] * dMatrix2[k][j];
            }
        }
    }

    /*for (i= 0; i< N; i++)
    {
        for (j= 0; j< N; j++)
        {
            cout<<dMultiplicationArray[i][j]<<" ";
        }
        cout<<endl;
    }
    */
    cout << "done ";
}
```

Aşağıdaki kod bölümünde ise double türünden dizilerin paralel çarpımı ile ilgili kod bulunmaktadır. Öncelikle çarpımı yapılacak matrisleri 1.0 değeriyle paralel bir şekilde doldurdum. Burada kullanılan collapse deyimiyle iç içe iki for döngüsünün olduğunu belirledim. Tek for döngüsü koşacak olsaydım bu deyimi yazmama gerek kalmayacaktı. Kullanılan private değeri ile her bir iş parçacığı yerel bir kopyaya sahip olacak ve o değişkeni geçici olarak kullanacaktır. Private değişken ilklendirilmez ve değeri paralel bölge dışında kullanılmaz. Varsayılan olarak döngü iterasyon sayacı özeldir. Bu nedenle i, j ve k değerleri yalnızca threadlere özel olduğundan paylaşılmaz. Bu sayede her iterasyonu belirli bir thread gerçekleştirecektir. Shared değeri ile paralel bölgede veri paylaşılır. Paylaşımın anlamı tüm threadler tarafından erişilebilir ve görülebilirdir. Varsayılan olarak iş paylaşım bölgesindeki tüm değişkenlerin döngü iterasyon sayacı dışında hepsi paylaşılır. Matris değerlerine tüm threadlerin ulaşması gerektiğinden matrisleri shared deyimiyle yazdım. Bu değer güncellenebilir ve optimum sayı kullanılabilir. Bu for döngüsünün altındaki yazım seri kodlama ile benzerdir ve alt kısmında olan kod ise çarpılan matrislerin sonuç matrisi ekrana yazılması kodudur.

```
void Multiplication_Parallel_Double()
{
    int i, j, k;

#pragma omp parallel for collapse(2) private(i,j) shared(dMatrix1,dMatrix2)
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++)
        {
            dMatrix1[i][j] = 1.0;
            dMatrix2[i][j] = 1.0;
        }
    }
#pragma omp parallel for collapse(2) private(i,j,k) shared(dMatrix1,dMatrix2,dMultiplicationArray)
    for (i = 0; i < N; ++i) {
        for (j = 0; j < N; ++j) {
            for (k = 0; k < N; ++k) {
                dMultiplicationArray[i][j] += dMatrix1[i][k] * dMatrix2[k][j];
            }
        }
    }
    /*#pragma omp parallel for private(i,j) shared(multiplicationArray)
    for (i= 0; i< N; i++)
    {
        for (j= 0; j< N; j++)
        {
            cout<<dMultiplicationArray[i][j]<<" ";
        }
        cout<<endl;
    }*/
    cout << "done ";
}
```

4.3.Çarpma İşleminin Sonucunu Kolay Test Etmek İçin Matrisleri “1.0” ile Doldurma

Seri kısım için matrisin içini aşağıdaki kodda görüldüğü gibi doldurdum.

```
int i, j, k;
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
    {
        dMatrix1[i][j] = 1.0;
        dMatrix2[i][j] = 1.0;
    }
```

Paralel kısım için matrisin içini aşağıdaki kodda görüldüğü gibi doldurdum.

```
    int i, j, k;

#pragma omp parallel for collapse(2) private(i,j) shared(dMatrix1,dMatrix2)
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++)
        {
            dMatrix1[i][j] = 1.0;
            dMatrix2[i][j] = 1.0;
        }
    }
```

4.4.Matris Boyutlarını Sırasıyla N=1K, 2K, ... 5K (K:bin) Boyutlarında Olacak Şekilde Çalıştırma

5 seri ve 5 paralel çarpma işlemi yaparken geçen süreler aşağıdaki gibidir.

	1000x1000	2000x2000	3000x3000	4000x4000	5000x5000
SERİ	4.156sn	42.835sn	161.258sn	388.273sn	788.956sn
PARALEL	0.605sn	6.019sn	21.581sn	58.043sn	122.161sn

4.5.FLOAT VE DOUBLE Tipleri İçin Matrislerin Çarpımı

Seri kod ile koşulduktan sonra geçen sürenin çıktısı aşağıdaki tablodaki gibidir.

	1000x1000	2000x2000	3000x3000	4000x4000	5000x5000
FLOAT	3.433sn	34.362sn	109.35sn	338.349sn	710.609sn
DOUBLE	4.172sn	42.416sn	160.351sn	387.391sn	787.485sn

Paralel kod ile koşulduktan sonra geçen sürenin çıktısı aşağıdaki tablodaki gibidir.

	1000x1000	2000x2000	3000x3000	4000x4000	5000x5000
FLOAT	0.587sn	5.276sn	20.503sn	50.886sn	105.904sn
DOUBLE	0.611sn	5.874sn	22.081sn	57.737sn	121.742sn

4.5.1.FLOAT VE DOUBLE Tipleri İçin Matrislerin Çarpımındaki Süre Farkları

Dizilerin tipini FLOAT tanımladığımda dizilerin çarpım işlemleri için geçen süre, DOUBLE ile tanımladığımdaki dizilerin çarpım işlemleri için geçen süreden azdır. Bunun sebebi tanım aralığıdır. DOUBLE, FLOATa göre 2 kat daha fazla hassiyete sahiptir. DOUBLE daha büyük bir tanım aralığına sahip olduğundan DOUBLE türünden dizilerin çarpımı için harcanılan süre daha fazla olmaktadır. Aşağıda DOUBLE ve FLOAT değişkenlerinin özellikleri verilmiştir.

Değişken	Boyu	Açıklaması	Değer Aralığı
FLOAT	4	Kesirli sayı	3.4e +/- 38 (7 basamak)
DOUBLE	8	Geniş ve fazla duyarlıklı kesirli sayı	1.7e +/- 308 (15 basamak)

4.5.2.Sistemde Ulaşılan FLOP/s Performans Limitinin Açıklanması

Floating point işlemleri en içteki for döngüsünün altındaki 1 toplama ve 1 çarpma işlemidir. Her adımda 1 toplama ve 1 çarpma işlemi vardır. Toplamda üç for döngüsü bulunduğundan, toplam floating point işlem sayısı 2*n*n*n olmaktadır.

```
for (i = 0; i < N; ++i)
{
    for (j = 0; j < N; ++j)
    {
        for (k = 0; k < N; ++k)
        {
            dMultiplicationArray[i][j] += dMatrix1[i][k] * dMatrix2[k][j];
        }
    }
}
```

Big-O notasyonuna göre baştaki sayının önemi yoktur. Algoritma karmaşıklığı O(n^3) olarak belirtilmektedir. Aşağıdaki tabloda floating point işlem sayısını, sistemdeki GFLOP/S kapasitesini ve sistemdeki belirtilen FLOP/s performans limitinin yüzde kaçına ulaşabildiğini görebiliriz.

Matris	1000x1000	2000x2000	3000x3000	4000x4000	5000x5000
FLOP sayısı	2*10^9	8*10^9	18*10^9	32*10^9	50*10^9
Geçen süre	0.587	5.276	20.503	50.886	105.904
Ulaşılan GFLOP/s	3.40	1.51	0.87	0.62	0.47
Sistem (GFLOP/s)	337,4	337,4	337,4	337,4	337,4
Ulaşılan FLOP/s değeri	%1.0	%0.44	%0.25	%0.18	%0.13

Kod seri bir şekilde CPU'da koştuğumda ulaşabileceğim performans limitlerini yukarıdaki tablodan inceleyebiliriz.

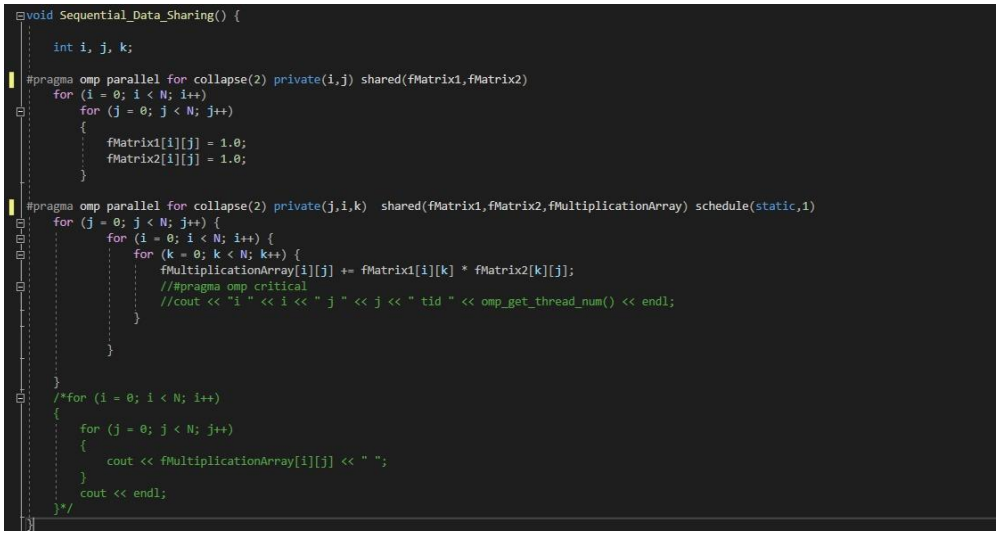
4.6.Veri Paylaşım Türlerine Göre Uygulamanın Geliştirilmesi

```
void Block_Data_Sharing() {

    int i, j, k;
#pragma omp parallel for collapse(2) private(i,j) shared(fMatrix1,fMatrix2)
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
        {
            fMatrix1[i][j] = 1.0;
            fMatrix2[i][j] = 1.0;
        }
#pragma omp parallel for  collapse(2) private(i,j,k)  shared(fMatrix1,fMatrix2,fMultiplicationArray) schedule(static,1)
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            for (k = 0; k < N; k++) {
                fMultiplicationArray[i][j] += fMatrix1[i][k] * fMatrix2[k][j];
            }
        }

    }
    /*for (i = 0; i < N; i++)
    {
        for (j = 0; j < N; j++)
        {
            cout << fMultiplicationArray[i][j] << " ";
        }
        cout << endl;
    }*/
}
```

Yukarıdaki kod bölümünde Blok Veri Paylaşımı modelini kodladım. Blok Veri Paylaşımı modelinin kodlanmasında satırları threadlere bölerek diğer matrisle çarparak sonuç matrisini oluşturdum. Paralelleştirme işlemine ek olarak Schedule deyimini ekledim. Schedule deyimi ile ilk döngüdeki for işlemleri threadlere bölündü. Değişken olarak tanımladığım i değeri satır işlemlerini tutmaktadır. İlk for döngüsüne Schedule deyimi ekledim bu nedenle satırları oluşan threadlere böldüm. Örneğin 1000x1000 matris için 10 thread oluştuğunu düşünelim. 1000 satır olduğundan ilk 100 satır ilk gelen threade ikinci 100 satır ikinci gelen threade ve diğerlerini de bu şekilde paylaştırdım. İlk gelen threadin işi birinci matrisin ilk 100 satırı ile ikinci matrisi çarpmaktır. Bu şekilde threadlere iş paylaşımı yapılmaktadır. Kodun paralel koşma performansı artmaktadır. Aşağıdaki tabloda blok veri paylaşımı modelinin kodlanması ile geçen süre ve normal paralelleştirme arasındaki süre ilişkileri verilmektedir.



```
void Sequential_Data_Sharing() {

    int i, j, k;

#pragma omp parallel for collapse(2) private(i,j) shared(fMatrix1,fMatrix2)
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
        {
            fMatrix1[i][j] = 1.0;
            fMatrix2[i][j] = 1.0;
        }

#pragma omp parallel for collapse(2) private(j,i,k)  shared(fMatrix1,fMatrix2,fMultiplicationArray) schedule(static,1)
    for (j = 0; j < N; j++) {
            for (i = 0; i < N; i++) {
                for (k = 0; k < N; k++) {
                    fMultiplicationArray[i][j] += fMatrix1[i][k] * fMatrix2[k][j];
                    //#pragma omp critical
                    //cout << "i " << i << " j " << j << " tid " << omp_get_thread_num() << endl;
                }

            }

    }
    /*for (i = 0; i < N; i++)
    {
        for (j = 0; j < N; j++)
        {
            cout << fMultiplicationArray[i][j] << " ";
        }
        cout << endl;
    }*/
}
```

Yukarıdaki kod bölümünde Ardışıl Veri Paylaşımı modelini kodladım. Ardışıl Veri Paylaşımı modelinin kodlanmasında ikinci matrisin sütunlarını threadlere bölerek diğer matrisle çarparak sonuç matrisini oluşturdum Aynı şekilde paralelleştirme işlemine ek olarak Schedule deyimini ekledim. Değişken olarak tanımladığım j değeri sütun işlemlerini tutmaktadır. İlk for döngüsüne Schedule deyimi ekledim bu nedenle sütunları oluşan threadlere böldüm. Örneğin 1000x1000 matris için 10 thread oluştuğunu düşünelim. 1000 sütun olduğundan ilk 100 sütun ilk gelen threade ikinci 100 sütun ikinci gelen threade ve diğerlerini de bu şekilde paylaştırdım. İlk gelen threadin işi birinci matris ile ikinci matrisin ilk 100 sütununu çarpmaktır. Bu şekilde threadlere iş paylaşımı yapılmaktadır. Kodun paralel koşma performansı artmaktadır. Aşağıdaki tabloda ardışıl veri paylaşımı modelinin kodlanması ile geçen süre, blok veri paylaşımı modelinin kodlanması ile geçen süre ve normal paralelleştirme arasındaki süre ilişkileri verilmektedir.

	1000x1000	2000x2000	3000x3000	4000x4000	5000x5000
PARALEL NORMAL	0.587sn	5.276sn	20.503sn	50.886sn	105.904sn
PARALEL ARDIŞIL	0.334sn	5.325sn	18.794sn	47.127sn	98.583sn
PARALEL BLOK	0.269sn	2.053sn	6.469sn	14.958sn	29.370sn

5. Programlama Platformları

Programlama platformu olarak OpenMP kullandım. Kodlarımı Visual Studio 2019'da derledim ve çalıştırdım.

OpenMP, paylaşımlı bellekli çoklu işlemcili mimariler için geliştirilmiş ve derleyici direktifleri yardımıyla paralel programlama yapan bir uygulama geliştirme arayüzüdür. Derleyicide derlenen program komutlarını paralel çoklu işlemciye/çekirdeğe sahip sistemlerde dağıtır ve paralel olarak işletilmesinin sağlar. OpenMP uygulamaları C/C++ ve Fortran dilleri ile geliştirilebilir ve bu dillere ait birçok derleyici OpenMP desteği vermektedir. OpenMP programlama modelinin sahip olduğu bazı temel özellikler şöyledir:

- Thread tabanlı paralellik
- Fork-Join modeli
- İç içe döngüleri paralel yapabilme
- Dinamik threadler
- Derleyici direktifleri yardımıyla paralel programlama

OpenMP'nin çalışma biçimi temel fork-join programlama modeline dayanır. Uygulamalarda, çalışan bir program tek bir thread olarak çalışmaya başlar. Programcı eş zamanlılığı kullanmak istediğinde, threadleri oluşturulur ve bu threadleri takım halinde verilen görevi yerine getirirler. Takım halinde çalışan threadler de sadece paralel alan dediğimiz alan içerisinde paralel olarak çalışır. Paralel alan bittiğinde, threadler tüm threadler bitene kadar beklerler ve birleştirilirler (join). Bu noktadan sonra master thread yeni bir paralel bir alan tanımı ile karşılaşıncaya kadar tek olarak çalışmaya devam eder.

6. Elde Edilen Sonuçlar ve Çarpma İşlemlerindeki Performans Yorumu

Matris çarpımını seri kodlama ile gerçekleştirdiğimde geçen süreyi gördüm ve paralel kodlama ile bu sürelerin ciddi bir şekilde azalmakta olduğunu gözlemledim. Problemin boyutu arttıkça geçen sürenin ciddi anlamda büyük süreler olacağından kodun paralel bir şekilde kodlanması ve bazı paralel kodlama yöntemleriyle iyileştirilmesi geçen süreyi oldukça azaltmaktadır. Paralel kodlama yöntemlerinden birisi dizinin tipinin FLOAT olarak belirlemektir. Yukarıdaki tablodan elde edilen sonuçlara göre FLOAT tipindeki matrislerin çarpımı DOUBLE tipine göre hızlıdır. Bu fark örneğin 2000x2000lik matris için yaklaşık 1 saniye olarak gözükse de büyük problemlerde bu fark oldukça fazla olacaktır. Paralel kodlama yaparken virgüllü işlemler yapılacaksa tip olarak FLOAT seçilmelidir. Bir sonraki yöntemlerden birisi ise kodu blok veya ardışıl biçimde programlamaktır. Threadlere iş dağılımları eşit ve mantıklı bir şekilde paylaştırıldığında geçen süre oldukça azalacaktır. Ayrıca bir diğer yöntem ise iki matris çarpılırken satır sütun şeklinde çarpım yapmaktansa ikinci matrisin transpozunu alarak çarpım işlemini yapmaktır. Bu şekilde veriler belleğe sıralı bir şekilde yerleştirildiğinden çarpım işlemi çok daha hızlı gerçekleştirilecektir. Paralel programlama ile kodlamanın çok önemli olduğunu ve tüm kodlamaların paralel bir şekilde gerçekleştirilmesinin en mantıklısı olduğunu düşünüyorum.