Karadeniz Teknik Üniversitesi
Mühendislik Fakültesi

Bilgisayar Mühendisliği Bölümü

PARALEL BİLGİSAYAR VİZE PROJESİ

Dr.Öğr.Üyesi İBRAHİM SAVRAN

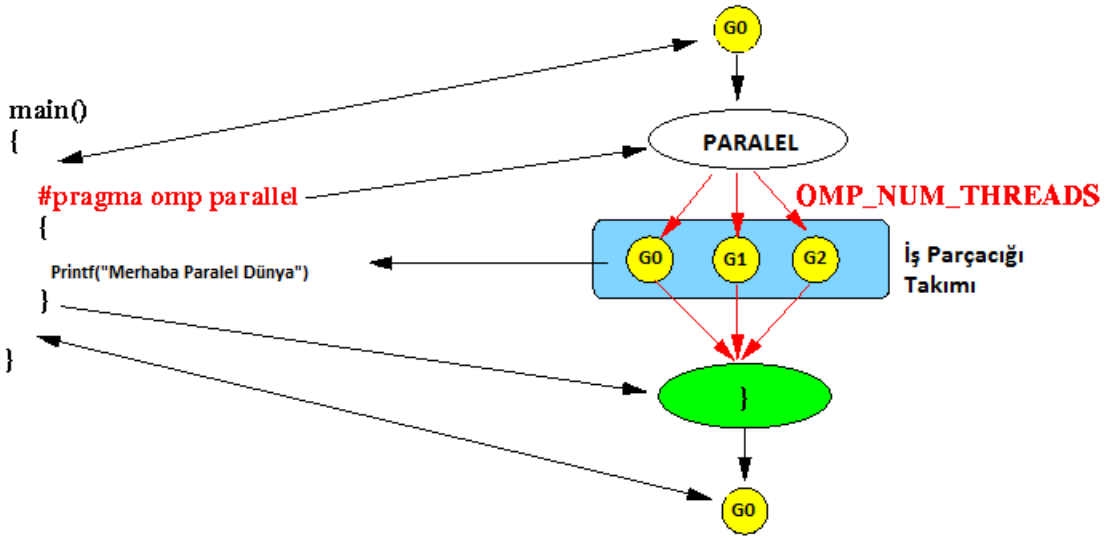
313961
Kemal AYDIN
1.öğretim

KISIM 1 – OpenMP (Open Multiprocessing)

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 işletilmesini 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:

- İş parçacığı tabanlı paralellik
- Çatallan-Birleş (Fork-Join) modeli
- İç içe döngüleri paralel yapabilme
- Dinamik iş parçacıkları
- Derleyici direktifleri yardımıyla paralel programlama

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



KISIM 2 – SİSTEM BİLGİSİ

2.1-) FLOP/s

Saniyedeki kayan noktalı sayı ile yapılan (floating-point number) işlem sayısıdır. Mikroişlemcilerin hız performansını göstermek için kullanılan bir ölçüdür. Özellikle aşırı derecede kayan noktalı sayı işlemleri içeren bilimsel işlemlerde bu ölçüt kullanılır. Kayan noktalı işlemler kesirli sayılarla yapılan işlemleri içerir ve bu işlemler tam sayılar ile yapılan işlemlerden daha uzun sürer ve daha zordur.

GFLOPS → saniyede 1 milyar kayan noktalı sayı işlemi

Projenin geliştirildiği systemin işlemci özellikleri :

➔ Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz [Family 6 Model 61 Stepping 4]

Turbonun devreye girip girmemesine göre değişiklik gösterdiği için tam olarak kestirilemesede yapılan benchmark değerlerine göre sistemin saniyede işlediği kayan noktalı sayı işlemi :

14.06 GFLOPS

2.2-) Cache

Çalışmakta olan programlara ait konutların ya da verilerin geçici olarak saklanması için kullanılan yüksek hızlı hafızalar olarak kullanımı sağlanmaktadır. Bu sayede cihazın daha hızlı bir şekilde istenilen içeriklere ulaşması sağlanmaktadır.

Verilere ulaşılabilmesi için devamlı olarak ana hafızaya erişim sağlanmaması için sıkça kullanılan veriler belirli bir alana kopyalanmaktadır. Düzenlenmemiş ya da düzenli bir şekilde oluşturulmuş olabilir. Her iki durumda da verilere ulaşımı sağlamaktadır. Bu nedenle özellikle istediğiniz programların bilgilerine ulaşımı kolaylaştırmak amacıyla ve daha kısa süre içerisinde açılmasını sağlamak amacıyla kullanımı sağlanmaktadır.

➔ Projenin test edileceği sistemin cache özellikleri :

L1 Data	2 x 32 KByte	8-way
L1 Inst.	2 x 32 KByte	8-way
Level 2	2 x 256 KByte	8-way
Level 3	4 MBytes	16-way

2.3-) 4-way

Küme ilişkilendirme (Set Associativity) erişim yönteminde bir önceki doğrudan erişimden farklı olarak veri kesin ve net bir şekilde bir yere eklenmez. Bunun yerine bir bilginin önbellek üzerinde gidebileceği birden fazla yer bulunur.Bu küme ilişkilendirme yönteminde bir bilginin önbellek üzerinde gidebileceği alana göre sayılar belirtilir. Örneğin ön bellek üzerinde 2 farklı yere eklenebiliyorsa 2 yönlü küme ilişkilendirme (2-way set associativity) veya 4 farklı yere erişilebilirse dört yönlü küme ilişkilendirme (4-way set associativity) ismi verilir.

KISIM 3 – KODLAMA

Projede geliştirme ortamı olarak Visual Studio ve yine bu platformun da desteklediği OpenMP paralelleştirme kütüphanesi kullanılmıştır. Programlama dili olarak C++ seçilmiştir. Kodu derleyip çalıştırmadan önce Visual Studio OpenMp dil desteğinin aktif hale getirildiğinden emin olun.

Kod açıklamaları IDE ortamına aktarımın daha kolay gerçekleştirilebilmesi amacı ile derleyici ortamında yapılıp oradan kod satırlarına gömülmüştür.

Aşağıda OpenMP kullanılarak NxN boyutluk iki matrisin çarpımını paralel olarak gerçekleştiren ve çarpma işleminin başlangıcı ile bitişi arasında geçen süreyi veren C++ fonksiyonu verilmektedir.

```
//a ve b matrisleri başlangıç değerleri daha önceden belirlenmiş olan birbirleriyle
çarpılacak iki matristir. a ve b matrislerinin çarpımı sonucu c matrisine
kaydedilmiştir. a, b, c matrisleri parallelCode fonksiyonuna parametre olarak
verilmiştir.
void parallelCode(float* a, float* b, float* c) {
	//programın başlangıç zaman bilgisi alınır
	clock_t start = clock();

	int i, j, k;
	cout << "Threads calisma sirasi   ==>    ";

	//paralel çalışacak thread sayısı ayarlanır
	omp_set_num_threads(NUM_THREADS);

	//a, b, c matris verileri tüm bloklar tarafından erişilebilirken i, j, k
verileri bloklara özeldir
	//paralel çalışacak olan kod bloğu

	#pragma omp parallel shared(a,b,c) private(i,j,k)
	{
		//threads numaraları alınır
		int ID = omp_get_thread_num();
		cout<<ID;

	//Iterasyonlar varsayılan olarak eşit bir şekilde iş parçacıkları arasında
paylaştırılır.
	#pragma omp for	schedule(static)
		for (i = 0; i < N; i = i + 1) {
			for (j = 0; j < N; j = j + 1) {
				c[i * N + j] = 0;
				for (k = 0; k < N; k = k + 1) {
			c[i * N + j] = (c[i * N + j] + (a[i * N + k] * b[k * N + j]));
				}
			}
		}
	}
	//programın bitiş zaman bilgisi alınır
	double calismaSuresi = static_cast<double>(clock() - start) / CLOCKS_PER_SEC;

	cout << "sure ==>    " << endl << calismaSuresi << endl;
}
```

Seri olarak yazılmış olan kod bloğu, yukarıda açıklaması satır satır yapılmış olan paralel kod bloğunun basitleştirilmiş bir şekli olduğu için tekrardan açıklanmamıştır.

Aşağıda NxN boyutluk iki matrisin çarpımını seri olarak gerçekleştiren ve çarpma işleminin başlangıcı ile bitişi arasında geçen süreyi veren C++ fonksiyonu verilmektedir.

```
void serialCode(float* a, float* b, float* c) {

	clock_t start = clock();

	int i, j, k;

	for (i = 0; i < N; i++) {

		for (j = 0; j < N; j++) {

			c[i * N + j] = 0;

			for (k = 0; k < N; k++) {

				c[i * N + j] += a[i * N + k] * b[k * N + j];
			}
		}
	}

	double calismaSuresi = static_cast<double>(clock() - start) / CLOCKS_PER_SEC;
	cout << "sure ==>    " << calismaSuresi << endl;
}
```

Aşağıda seri ve paralel olarak gerçekleştirilecek olan matris çarpımı öncesinde matrislere ilk değer atamalarını yapan C++ fonksiyonu verilmektedir.

```
void fillingMatrix(float* a, float* b, float* c, float d) {

	int i, j, k;

	for (i = 0; i < N; i++) {

		for (j = 0; j < N; j++) {

			a[i * N + j] = d;
			b[i * N + j] = d;
			c[i * N + j] = 0;
		}
	}
}
```

Aşağıda seri ve paralel olarak çarpma işlemi gerçekleştirebilecek olan fonksiyonların isteğe bağlı olarak çağırılıp çalıştırılabileceği main verilmiştir.

```
#include <iostream> //standart C++ kütüphanesi
#include <omp.h>    //openMP kütüphanesi
#include <ctime>    //tarih ve saat işlemleri için

//matris boyutu N çeşitli kullanım senaryoları için değiştirilmiştir.

#define N 1000      //matrix size
#define NUM_THREADS 8      //paralel thread sayısı

using namespace std;//string veya vektör kullanımı için gerekli

//yukarıda verilen fonksiyon tanımlarının burada yapıldığını unutmayınız.


int main()
{
	//dinamik matris tanımlamaları yapılır
	float* a = new float[N * N];
	float* b = new float[N * N];

	float* c = new float[N * N];	//çarpım sonucunun atanacağı sonuç matrisi tanımlanır

	float veri = 1.0;	//a ve b matrisinin barındıracağı veri

	fillingMatrix(a, b, c, veri);	//matrislere ilk değer atamaları yapılır

	parallelCode(a, b, c);	//a ve b matrisi arasında openMP kullanılarak paralel çarpma işlemi yapılır

	//serialCode(a, b, c);	//a ve b matrisi arasında seri çarpma işlemi yapılır

	//istenildiği takdirde c yani çarpım sonucu yazdırılır fakat N nin çok büyük olduğu durumlarda düzgün bir görsel sonuç elde edilmez
	/*for (int i = 0; i < N; i++) {

		for (int j = 0; j < N; j++) {

			cout << c[i * N + j]<<" ";
		}
		cout << endl;
	}*/

	return 0;

}
```

Yukarıda verilen matris içerikleri float veri türü içermektedir. Test aşamasında bu değerler double veri türüne dönüştürülmüştür. Karmaşıklık yaratmaması için tekrardan rapora eklenmesine gerek görülmemiştir.

KISIM 4 – PROGRAMIN ÇEŞİTLİ SENARYOLARDA TESTİ VE SONUÇ ANALİZİ

Aşağıda paralel ve seri çalışan çarpım programının zaman tablosu verilmiştir.

- Çarpılan veri ilk olarak float, daha sonra da double türündedir.
- N değeri değiştirilerek test edilmiştir. (1000, 2000, 3000, 4000, 5000)
- Tabloda yazan zaman saniye cinsindendir.
- Test işleminin daha rahat yapılabilmesi için a ve b matrisinin her elemanı 1.0 olarak belirlenmiştir.
- N değerinin 5000 olduğu paralel çarpma esnasında CPU ısı artışından dolayı ara sıra işlem yapmayı bırakıp tekrar çalışmıştır.
- N değerinin daha da büyüdüğü durumlarda işlemin sonuçlanması saatleri bulacaktır.
- Çalıştırılan thread sayısının 8 olduğu unutulmamalıdır. Bu değer arttırıldığı taktirde belirli bir seviyeye kadar performans artışı da sağlanabilir.

➔ Tablolar sistemin turbo özelliği aktifken oluşturulmuştur. (max 2.96 GHz)

N x N

	1000x1000	2000x2000	3000x3000	4000x4000	5000x5000
Seri Time (float veri)	6.508sec	55.903sec	241.719sec	442.427sec	1156.64sec
Paralel Time (float veri)	3.017sec	28.578sec	103.813sec	242.674sec	518.658sec

(a ve b matrisinin elemanları float veri türündedir)

N x N

	1000x1000	2000x2000	3000x3000	4000x4000	5000x5000
Seri Time (double veri)	6.988sec	68.675sec	250.764sec	463.946sec	1311.67sec
Paralel Time (double veri)	3.319sec	31.403sec	112.339sec	293.636sec	588.436sec

(a ve b matrisinin elemanları double veri türündedir)

KISIM 5 – SONUÇ ANALİZİ

- Paralel sürüm seri sürüme göre her zaman daha iyi sonuç verdi.

- OpenMP kullanımının kolay ve aynı zamanda oldukça verimli bir paralelleştirme ortamı sunduğu gözlendi.

- Paralel programlama birçok problemin daha hızlı çözüme ulaşmasına yardımcı olur.

- Float ve double sürümler arasında zaman farkı olduğu gözlemleniyor. Float veri türü barındıran matrislerin çarpımının, double veri türü barındıran matrislerin çarpımına göre daha hızlı olduğu tespit edildi. Arada oluşan bu zaman farkının float değişken türünün bellekte 32 bit yer tutarken, double türündeki bir değişkenin 64 bit yer tutması olarak değerlendirilir.

- Sistemin kayan noktalı sayı hesaplama kapasitesinin oldukça altında bir sayı elde ettik. Başlangıçta elde edilen hesap kapasitesi zaten oldukça esnek bir sayıdır. Bu kapasiteye ulaşılamamasının sebepleri; sistemin performansını çalışma esnasında etkileyen faktörler (ısı, arka planda çalışan farklı programlar, sistemin veriye erişim süresi) oldukça fazladır. Bunun yanı sıra sistem performansını kararlı bir şekilde sürdüremedi. Termal ısınmadan kaynaklı işlemcinin kendini yeniden başlatması gözlemlendi.

Hız Sıralaması = Paralel Time (float) > Seri Time (float)

Hız Sıralaması = Paralel Time (double) > Seri Time (double)

Hız Sıralaması = Paralel Time (float) > Paralel Time (double)

Hız Sıralaması = Seri Time (float) > Seri Time (double)

KISIM 6 – PERFORMANS İYİLEŞTİRMELERİ

Burada uygulanan teknik ile matris blok blok getiriliyor. Böylece, matrisin ön belleğe alınan kısmı sürekli olarak değiştirilmiyor. Parçadaki tüm işlemler bittikten sonra diğer parçaya geçiliyor.
Bu da büyük bir performans getirisi sağlıyor. Çünkü dizinin indisinin her artışında, ön bellekteki matris parçasını değiştirmiyoruz.

➔Yapılan test sonucunda da normal paralelleşmiş kısma göre blok veri kullanan parallel çarpma işlemi daha hızlı sonuç vermiştir.

```
void blockData(float* a, float* b, float* c) {

	clock_t start = clock();   //programın başlangıç zaman bilgisi alınır

	int i, j, k, ii, jj, kk;

	cout << "Threads calisma sirasi   ==>    ";

	omp_set_num_threads(NUM_THREADS); //paralel çalışacak thread sayısı ayarlanır

	//a, b, c matris verileri tüm bloklar tarafından erişilebilirken i, j, k
verileri bloklara özeldir
	#pragma omp parallel shared(a,b,c) private(i,j,k,ii,jj,kk)   //paralel çalışacak
olan kod bloğu
	{
		int ID = omp_get_thread_num();    //threads numaraları alınır
		cout << ID;

	#pragma omp for     schedule(static)    //Iterasyonlar varsayılan olarak eşit
bir şekilde iş parçacıkları arasında paylaştırılır.
//Buradaki çarpma işleminin diğer çarpma işlemleriyle arasındaki fark verinin blok
blok getirilmesidir.
		for (ii = 0; ii < N / 10; ++ii)
			for (jj = 0; jj < N / 10; ++jj)
				for (kk = 0; kk < N / 10; ++kk)
					for (i = 10 * ii; i < 10 * (ii + 1); ++i)
						for (j = 10 * jj; j < 10 * (jj + 1); ++j)
							for (k = 10 * kk; k < 10 * (kk + 1);
++k)
								c[i * N + j] += a[i * N + k] *
b[k * N + j];
	}

	double calismaSuresi = static_cast<double>(clock() - start) / CLOCKS_PER_SEC;
	//programın bitiş zaman bilgisi alınır
	cout << endl << "sure ==>  " << endl << calismaSuresi << endl;
	cout << "--"<<c[86252]<<"--";
}
```

➔Bu tablo sistemin sürekli çökmesi (termal ısınma) nedeniyle turbo özelliği kapalıyken oluşturulmuştur. Bu sebeple yukarıda verilen tablodan farklı değerler gözlemlemeniz normaldir.

N x N

	1000x1000	2000x2000	3000x3000	4000x4000	5000x5000
Seri Time (float)	7.426	63.801	273.496	636.989	905.076
Paralel Time (float)	3.613	32.144	133.131	316.608	650.25
Block Data (float)	2.521	19.55	65.368	154.754	325.243

(saniye)

Hız Sıralaması = Block Data (float) > Paralel Time (float) > Seri Time (float)