OpenMp ile “Hello Word” Programı

OpenMp Nedir ?

OpenMP **(**Open Multi-Processing**),** çok çekirdekli ve çok iş parçacıklı (multithreading) programlamayı kolaylaştıran bir API’dir. **C, C++ ve Fortran** dillerinde kullanılabilir ve özellikle paylaşımlı bellekli (shared memory) sistemler için paralel programlamayı destekler.

Program.c

```c
#include <stdio.h>
#include <omp.h>

int main() {
    // OpenMP paralel bölge kodu
    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num(); // Mevcut thread'in ID'sini alınan kısım
        printf("Hello World from thread %d\n", thread_id);
    }
    return 0;
}
```

Paralel matris çarpımı uygulaması

matrisCarpimi.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define N 4 // Matris boyutu
// Rastgele matris oluşturma
void fill_matrix(int matrix[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            matrix[i][j] = rand() % 10; // 0-9 arasında rastgele sayılar
        }
```

```
    }
}
// Matrisi ekrana yazdırdığımız kısım
void print_matrix(int matrix[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%4d ", matrix[i][j]);
        }
        printf("\n");
    }
}
// Paralel matris çarpımı
void matrix_multiply(int A[N][N], int B[N][N], int C[N][N]) {
    #pragma omp parallel for collapse(2) // 2 döngüyü paralel hale
getirme
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            C[i][j] = 0;
            for (int k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

int main() {
    int A[N][N], B[N][N], C[N][N];

    fill_matrix(A);
    fill_matrix(B);
    printf("Matris A:\n");
    print_matrix(A);
    printf("\nMatris B:\n");
    print_matrix(B);
```

```
    matrix_multiply(A, B, C);
    printf("\nÇarpım Matrisi (C = A x B):\n");
    print_matrix(C);
    return 0;
}
```

OpenMP ile kritik bölgeler (critical sections) ve deadlock

OpenMP'de kritik bölgeler (critical sections), birden fazla iş parçacığının aynı anda eriştiği paylaşılan verilere zarar vermesini önlemek için kullanılır. Deadlock ise, birden fazla iş parçacığının birbirlerini beklemesi sonucu sistemin kilitlenmesi durumudur.

1. OpenMP Critical Section Kullanımı

Birden fazla iş parçacığı (thread) aynı anda bir değişkene yazmaya çalışırsa veri tutarsızlığı (race condition) oluşabilmektedir. Çözüm ise **`#pragma omp critical`** yönergesi ile aynı anda yalnızca bir thread'in belirli bir kod parçasını çalıştırmasını sağlayabiliriz.

criticalSections.c

```c
#include <stdio.h>
#include <omp.h>

int main() {
    int toplam = 0;

    #pragma omp parallel
    {
        for (int i = 0; i < 5; i++) {
            #pragma omp critical
            {
                toplam += 1; // Aynı anda sadece bir thread erişebilir
                printf("Thread %d: toplam = %d\n", omp_get_thread_num(),
toplam);
            }
        }
    }
```

```
    printf("Toplam Son Değer: %d\n", toplam);
    return 0;
}
```

2. OpenMP'de Deadlock

Deadlock, iş parçacıklarının birbirlerini beklemesi sonucu sistemin durmasıdır. OpenMP'de **`#pragma omp critical`** bloklarının iç içe girerek yanlış sıralamayla kullanılması deadlock'a neden olabilir.

Deadlock.c

```
#include <stdio.h>
#include <omp.h>
int main() {
    int a = 0, b = 0;
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            #pragma omp critical(LOCK_A)
            {
                printf("Thread %d: LOCK_A aldı\n", omp_get_thread_num());
                #pragma omp critical(LOCK_B) // Başka bir thread burayı
almış olabilir
                {
                    printf("Thread %d: LOCK_B aldı\n",
omp_get_thread_num());
                    a += 1;
                }
            }
        }

        #pragma omp section
        {
            #pragma omp critical(LOCK_B)
```

```
            {
                printf("Thread %d: LOCK_B aldı\n", omp_get_thread_num());
                #pragma omp critical(LOCK_A) // Diğer thread LOCK_A'da
takılı olabilir
                {
                    printf("Thread %d: LOCK_A aldı\n",
omp_get_thread_num());
                    b += 1;
                }
            }
        }
    }
    printf("Sonuç: a = %d, b = %d\n", a, b);
    return 0;
}
```

1. Thread 1 → `LOCK_A` aldıktan sonra `LOCK_B`'yi almak istiyor.
2. Thread 2 → `LOCK_B` aldıktan sonra `LOCK_A`'yı almak istiyor.
3. İki thread de birbirini bekliyor ve sonsuza kadar kilitli kalıyor.

Yukarıdaki kodda temel sorun yine yukarıda verilen 3 madde ile belirtilmektedir. Peki bunun önüne geçmek için ne yapılabilir. İlk olarak kilitle hep aynı sıraya alınmalı, ikincisi tek bir kritik bölge (#pragma omp critical) kullanılma, aşağıda doğru kullnılması ile ilgili bir kod örneği verilmiştir.

Solvedeadlock.c

```
#include <stdio.h>
#include <omp.h>

int main() {
    int a = 0, b = 0;
    #pragma omp parallel
    {
        #pragma omp critical
        {
            a += 1;
```

```
            b += 1;
        }
    }
    printf("Sonuç: a = %d, b = %d\n", a, b);
    return 0;
}
```

1. `#pragma omp critical`, paylaşılan verilerin güvenli bir şekilde işlenmesini sağlar.
2. Deadlock, iki thread'in birbirini bekleyerek kilitlenmesi sonucu oluşur.
3. Deadlock önlemek için: Kilitleri her zaman aynı sırayla alın, `#pragma omp atomic` kullanın veya mümkünse kilit gerektirmeyen işlemleri tercih edin.