

Objetivos del Práctico:

Este práctico es una aproximación práctica al diseño e implementación de soluciones, a fin y efecto que el estudiante sea capaz de resolver los problemas reales aplicando el paradigma de memoria compartida a través de la programación con threads.

Temas a tratar:

Programación con Memoria Compartida a través de la biblioteca OpenMP. Directivas. Accesos a Memoria y Sincronizaciones.

Metodología:

Resolución de práctico máquina. A través de distintos problemas simples, algunos resueltos en paralelo y otros a resolver, el alumno se aproximará en el primer caso a las facilidades provistas por la API OpenMP respecto a la programación paralela; y segundo al diseño e implementación de soluciones paralelas a los problemas planteados.



Conceptos Preliminares

En la actualidad, cada vez es más frecuente observar la existencia de sistemas altamente complejos que requieren mayor tiempo de computo. Para afrontar este tipo de problemas se hace uso de la computación de alto desempeño, en donde un único problema es dividido y resuelto de manera simultánea por un grupo de procesadores, donde el principal objetivo es mejorar la velocidad de procesamiento de la aplicación. En estos casos, el modelo de memoria utilizado hace que la programación de aplicaciones paralelas para cada caso sea esencialmente diferente. En este práctico estudiaremos la API de OpenMP, la cual idealmente es utilizada en aplicaciones de memoria compartida donde los procesadores se comunican a través de la memoria.

EJERCICIO 1: Hello World!!!

Desarrollar un Programa OpenMp denominado "Hola" que declare una región paralela y dentro de la misma cada thread en ejecución imprima por pantalla "Hola Mundo Desde El Thread (Nº thread)". Ejecutar variando el número de threads.

EJERCICIO 2: Repartición de tareas

¿Qué diferencia hay en la ejecución de estos dos programas?

¿Qué sucedería si la variable n no fuera privada?

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

```
int main(int argc, char **argv){
    int n,tid;
    omp_set_num_threads(5);
    #pragma omp parallel private (tid,n)
    {
        tid = omp_get_thread_num();
        for(n=0;n<10;n++){
            printf("Hola mundo %d \n",tid);
        }
    }
    return(0);
}
```

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

```
int main(int argc, char **argv){
    int n,tid;
    omp_set_num_threads(5);
    #pragma omp parallel private (tid,n)
    {
        tid = omp_get_thread_num();
        #pragma omp for
        for(n=0;n<10;n++){
            printf("Hola mundo %d \n",tid);
        }
    }
    return(0);
}
```

EJERCICIO 3: Dependencia de datos.

Suponga los siguientes fragmentos de programa e indique que lazos son susceptibles de ser paralelizados y cuáles no.

```
DO i=1,N
    a[i]= a[i+1] + x
END DO
```

```
DO i=1,N
    a[i]= a[i] + b[i]
END DO
```

```
ix = base
DO i=1,N
    a ( ix ) = a ( ix ) - b ( i )
    ix = ix + stride
END DO
```

```
DO i=1, n
    b ( i ) = ( a ( i ) - a (i-1) ) * 0.5
END DO
```

EJERCICIO 4: Bloques estructurados.

Suponga los siguientes fragmentos de programa e indique por que los lazos no son susceptibles de ser paralelizados.

```

for (i=0; i<n; i++) {
    a[i] = 2.3*i;
    if (a[i] < b[i]) break;
}

flag = 0;
for (i=0; (i<n) & (!flag); i++){
    a[i] = 2.3*i;
    if (a[i] < b[i]) flag = 1;
}

```

EJERCICIO 5: Suma de arreglos

Realice un programa OpenMP que realice la suma de dos arreglos componente a componente y deje el resultado en un nuevo arreglo. Ejecute el programa variando el número de threads.

EJERCICIO 6: Multiplicación de matrices

Realice un programa OpenMP que realice la multiplicación de matrices.

EJERCICIO 7: Planificación de bucles

Ejecute los siguientes códigos. Describa cual es la principal diferencia entre ellos.

```

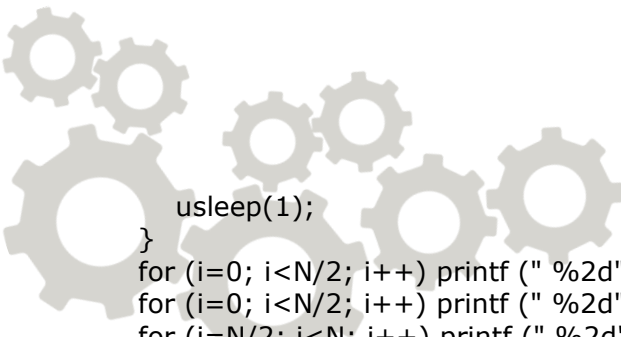
#include <omp.h>
#include <stdio.h>
#include <unistd.h>
#define N 40
main () {
    int tid;
    int A[N];
    int i;
    for(i=0; i<N; i++) A[i]=-1;
    #pragma omp parallel for schedule(static,4) private(tid)
    for (i=0; i<N; i++){
        tid = omp_get_thread_num();
        A[i] = tid;
        usleep(1);
    }
    for (i=0; i<N/2; i++) printf (" %2d", i); printf ("\n");
    for (i=0; i<N/2; i++) printf (" %2d", A[i]);printf ("\n\n");
    for (i=N/2; i<N; i++) printf (" %2d", i); printf ("\n");
    for (i=N/2; i<N; i++) printf (" %2d", A[i]);printf ("\n\n");
}

```

```

#include <omp.h>
#include <stdio.h>
#include <unistd.h>
#define N 40
main () {
    int tid;
    int A[N];
    int i;
    for(i=0; i<N; i++) A[i]=-1;
    #pragma omp parallel for schedule(dynamic,4) private(tid)
    for (i=0; i<N; i++){
        tid = omp_get_thread_num();
        A[i] = tid;
    }
}

```



```
        usleep(1);
    }
    for (i=0; i<N/2; i++) printf (" %2d", i); printf ("\n");
    for (i=0; i<N/2; i++) printf (" %2d", A[i]);printf ("\n\n\n");
    for (i=N/2; i<N; i++) printf (" %2d", i); printf ("\n");
    for (i=N/2; i<N; i++) printf (" %2d", A[i]);printf ("\n\n\n");
}
```

EJERCICIO 8: Secciones

Realice un programa OpenMP el cual disponga de un conjunto de funciones capaces de calcular el mínimo, máximo, la multiplicación y desviación estándar de un conjunto de un millón de elementos contenidos en un arreglo.

Cada una de estas funciones deberá recibir por parámetro los datos a procesar y deberá devolver el valor resultante.

Se pide que realice un programa OpenMP que sea capaz de realizar en paralelo todas estas operaciones. El programa además deberá informar por pantalla que thread ha realizado que operación.

Finalmente se pide que calcule el speedup de esta aplicación. Para realizar este cálculo debe trabajar con los valores promedios obtenidos de 50 ejecuciones tanto para la solución paralela como para la solución secuencial.