Universidad Nacional de San Antonio Abad del Cusco

Departamento Académico de Informática

ALGORITMOS PARALELOS Y DISTRIBUIDOS

Práctica Nº 3

PARALELIZACION DE BUCLES

1. OBJETIVO.

- Conocer directivas para paralelizar bucles con OpenMP
- Reconocer problemas en los que es necesario aplicar paralelización de bucles.

2. INTRODUCCION.

Uno de los casos más recurrentes dentro de las tareas de programación son los ciclos. A través de cientos, miles o millones de iteraciones de ciclos se logra procesar información valiosa que en otros tiempos parecía imposible calcular. Sin embargo, si los cálculos son muy grandes también las computadoras llegan a su límite. Por lo tanto, es bueno repartir esta carga de procesamiento en diferentes hilos. Por ejemplo el problema de encontrar números primos es uno de los problemas más antiguos, y no se ha encontrado manera alguna para poder encontrarlos a todos mediante una formula. La único que es posible hacer es comprobar número por número si es divisible entre algún otro número. Esta tarea es tediosa y larga y en ciertos momentos incomputable.

3. DESARROLLO DE LA PRÁCTICA

3.1. ACTIVIDAD 1.

3.1.1. Directiva for.

La directiva *for* causa que la iteraciones del ciclo inmediato se ejecutarán en paralelo. En tiempo de ejecución las iteraciones del ciclo son distribuidas entre todos los hilos. Las cláusulas que soporta la directiva *for* son las siguientes:

- private(lista)
- firstprivate(lista)
- lastprivate(lista)
- reduction(lista)
- ordered
- Schedule([kind], chunk_size)
- Nowait.

3.1.2. Ejercicio 1.

En este ejercicio vemos que el grupo de procesadores se reparten el trabajo. Cada hilo comienza a ejecutar un pedazo del ciclo de manera que todos los procesadores están ocupados.

```
#include<stdio.h>
#include<omp.h>
int main()
{
  int i, n;
  n = 10;
   #pragma omp parallel shared(n) private (i)
```

```
  {
   #pragma omp for
     for (i = 0; i < n; i++)
       printf("El hilo %d esta ejecutando el ciclo %d \n",
              omp_get_thread_num(), i);
   }
 getchar();
 return 0;
}
```

3.1.3. Ejercicio 2.

En este ejercicio, al crear dos ciclos *for* dentro de la región paralela, cada uno de los bucles ciclos tendrán un orden según se hayan escrito secuencialmente dentro de la región paralela. Primero se ejecuta el ciclo 1 y posteriormente el ciclo dos.

```
#include<stdio.h>
#include<omp.h>
int main()
{
 int i, n;
 n = 10;
  #pragma omp parallel shared(n) private (i)
 {
    #pragma omp for
    for (i = 0; i < n; i++)
       printf("Region uno : El hilo %d esta ejecutando el
              ciclo %d \n", omp_get_thread_num(), i);
    #pragma omp for
    for (i = 0; i < n; i++)
       printf("Region dos : El hilo %d esta ejecutando el
               ciclo %d \n", omp_get_thread_num(), i);
 }
 getchar();
 return 0;
}
```

3.2. ACTIVIDAD 2.

3.2.1. Directiva section.

La directiva *section* permite ejecutar una serie de bloques de código de manera paralela, junto a otros bloques. De esta forma, no u ‘nicamente se puede repartir las tareas de un ciclo, si no que diversas secciones enteras de código se ejecutaran de manera paralela. Primero se debe llamar a la directiva *parallel*, para después entrar a un bloque de secciones, lo cual se indicará mediante *#pragma_omp_sections*. Cada sección de código separada, a su vez, será´ llamada desde *#pragma_omp_section*. La llamada será´ a una función, donde su salida se convertirá a *void.* Las instrucciones contenidas en las diferentes secciones se ejecutan cada una en un hilo, por los que el procesamiento de los tres bloques de código se hace al mismo tiempo.

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

void func(int num)
```

```
{
  printf("En la funcion no %d que se ejecuta 1\n", num);
  printf("En la funcion no %d que se ejecuta 2\n", num);
  printf("En la funcion no %d que se ejecuta 3\n", num);
  printf("En la funcion no %d que se ejecuta 4\n", num);
}
void funcA()
{
  printf("In funcA: esta section es ejecutada por el hilo %d\n",
  omp_get_thread_num());
}

void funcB()
{
  printf("In funcB: esta section es ejecutada por el hilo %d\n",
        omp_get_thread_num());
}


int main()
{
  int i, n;
  n = 10;
  #pragma omp parallel shared(n) private (i)
  {
      #pragma omp sections
        {
          #pragma omp section
             (void)funcA();
          #pragma omp section
             (void)funcB();
          #pragma omp section
             (void)func(3);
        }
  }
  getchar();
  return 0;
}
```

3.3. ACTIVIDAD 3.

3.3.1. Directiva single.

Existen ocasiones en que dentro de un bloque paralelo se requiere que un determinado bloque de código sea ejecutado de manera secuencial. Para este propósito tenemos lo cláusula *single*

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

int main()
{
  int n, i, a, b;
  #pragma omp parallel shared (a,b) private(i)
  {
    #pragma omp single
     {
      a = 10;
      printf("Esto fue ejecutado por el hilo %d \n", omp_get_thread_num());
      printf("Esto fue ejecutado por el hilo %d \n", omp_get_thread_num());
```

```
      printf("Esto fue ejecutado por el hilo %d \n", omp_get_thread_num());
      printf("Esto fue ejecutado por el hilo %d \n", omp_get_thread_num());
       printf("Esto fue ejecutado por el hilo %d \n", omp_get_thread_num());

    } //--una barrera es insertada automáticamente aquí
    #pragma omp for
     for (i = 0; i < 10; i++)
      printf("Ejecutado %d desde for en el hilo %d \n", i,
              omp_get_thread_num());
  }
  getchar();
  return 0;

}
```

3.4. ACTIVIDAD 4.

3.4.1. Directiva critical.

Cuando existen datos que se comparten entre diferentes hilos existe la posibilidad de entrar en problemas de competencia por los datos. Para evitar que se pierdan datos o se produzcan errores por la competencia, es posible indicar que la región debe ser ejecutada únicamente por un hilo a la vez, cuando entra a la sección crítica. El siguiente ejemplo muestra la suma de diferentes hilos que se realiza de manera efectiva, sin perder información.

```
#include<iostream>
#include<stdio.h>
#include<omp.h>
#include<ctime>
#define TAM 10000
int main()
{
  int sum = 0;
  int n, TID, sumLocal, i;
  int a[TAM];
  n = TAM;
  //--Cargar un vector con números aleatorios
  srand(time(0)); //se establece el número semilla
  for (int x = 0; x<TAM; x++) {
        a[x] = rand() %  10000;
  }//for
    #pragma omp parallel shared (n, a, sum) private (TID, sumLocal )
   {
      TID = omp_get_thread_num();
      sumLocal = 0;
      #pragma omp for
        for (i = 0; i < n; i++)
              sumLocal += a[i];
      #pragma omp critical
        {
              sum += sumLocal;
              printf("TID=%d : sumLocal=%d sum=%d \n", TID, sumLocal, sum);
        }
     }//Fin de la región paralela
    printf("Valor de suma despu_es de la region paralela : %d", sum);
    getchar();
    return 0;

  }
```

3.4.2. Directiva atomic.

En algunas máquinas se incluye optimización para manejar las competencias. Únicamente la parte izquierda de la operación de asignación se protege de competencias, no la parte derecha. Y únicamente se permite una operación de asignación con la directiva. Si no se encuentra una aceleración hardware para *atomic*, se utiliza como una sección critica.

```
#include<stdio.h>
#include<omp.h>
#define TAM 10000
int main()
{
  int n, i;
  int ic = 0;
  n = TAM;
 #pragma omp parallel shared (n, ic) private (i)
  for (i = 0; i < n; i++)
  {
     #pragma omp atomic //--acceso exclusivo al contador
         ic += 1;
  }//Fin de la región paralela
  printf("Contador : %d \n", ic);
  getchar();
  return 0;
}
```

3.5. PROBLEMAS

3.5.1. Búsqueda Secuencial.

Ahora mostraremos el mejoramiento de la velocidad al tratar datos grandes mediante un programa sencillo de búsqueda secuencial. En este caso utilizamos la paralelización con la cláusula *for* de *parallel*, y lo comparamos para cada número de datos en secuencial.

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include "omp.h"

#define MAX_SIZE_VISIBLE 100
int main(int argc, char* argv[])
{
   int *a;
   unsigned long long int size, i;
   double start, end;
   double parallel, sequential;
   int n, j;
   // Es necesario tener dos argumentos en la linea de comandos
   // de lo contrario se interrumpira el programa y mostrará
   // un mensaje explicando el uso del mismo .
   if (argc <= 2)
   {
   printf("Faltan argumentos, el uso correcto es :\n\ tseq[tamaño de lista][numero a buscar] \n");
    return 1;
   }
  // Transformamos los argumentos en numeros
  size = strtoull(argv[1], NULL, 10);
  n = atoi(argv[2]);
  printf("Elementos en la lista : %llu \n Número a buscar : %d \n", size, n);
  // Alojando dinamicamente un arreglo para guardar los valores
  // aleatorios .
  a = (int*)malloc(size * sizeof(unsigned long long int));
  // Generacion de numeros aleatorios .
  printf("Generando numeros aleatorios en la lista . . . \n");
  srand(time(NULL));
```

```
    for (i = 0; i < size; i++)
    {
      a[i] = rand() / 1000;
    }
    if (size < MAX_SIZE_VISIBLE)
    {
      for (i = 0; i < size; i++)
          {
            printf(" %llu : %d \n", i, a[i]);
          }
     }
    else
     {
         printf("Lista demasiado grande para mostrarse en pantalla. \n");
         {
          //Inicio de busqueda secuencial
          printf("Buscando . . . \n");
          j = 0;
          start = omp_get_wtime();
          for (i = 0; i < size; i++)
           {
            if (a[i] == n)
            {
             // Impresion de resultados
                 printf("%d se encuentra en la posicion %llu \n", n, i);
                 j = 1;
            }
           }
         }
     if (!j)
      {
         printf("No se encontró el numero \n");
      }
     end = omp_get_wtime();
     printf("Procesamiento secuencial %f \n", end - start);
     sequential = end - start;
     j = 0;
     start = omp_get_wtime();
     int i;
     #pragma omp parallel for default(none) firstprivate(size, n, a) private(i) shared(j)
     for (i = 0; i < size; i++)
     {
         if (a[i] == n)
         {
            // Impresion de resultados
            printf("%d se encuentra en la posicion  %llu \n", n, i);
            j = 1;
         }
     }
     if (!j)
     {
       printf("No se encontro el numero \n");
     }
     end = omp_get_wtime();
    printf("Procesamiento paralelo %f \n", end - start);
    parallel = end - start;
    printf("\n\n Tamaño n tSecuencial n tParalelo \n");
    printf("Resul tado : %llu nt %f nt %f \n", size, sequential, parallel);
    //Liberando la memoria reservada
    free(a);
    return 0;
   }
}
```