Índice de contenido

Practica 1.- Introducción a OpenMP

Esta práctica pretende introducirnos en la API OpenMP. Para ello lo que se pretende es lo siguiente:

1. En primer lugar se debe instalar OpenMP en los equipos
2. A continuación se va a realizar la típica implementación de "hola mundo" donde se hagan uso de las rutinas (junto con la directiva parallel):
   - omp_set_num_threads
   - omp_get_num_threads
   - omp_get_max_threads
   - omp_get_thread_num
   - omp_get_num_procs
   - omp_in_parallel
   - omp_get_dynamic (modificar omp_set_dynamic y ver cómo cambia el resultado de las rutinas anteriores)
   - omp_get_wtime
3. A continuación vamos a realizar un programa que calcule la suma de n números almacenados en un vector:
   a) para ello en un primer lugar se realizará directamente con la directiva parallel y se aconseja utilizar las cláusulas:
      - private
      - shared
      - if (ejemplo si el número de elementos es menor que 100 no hace falta hacerlo en paralelo)
      - firstprivate (como ejemplo podéis poner unas impresiones por pantalla de los valores del vector de forma que una parte, la mitad por ejemplo, la haga la el hilo padre y el resto la haga cada una de los hilos creados)
   b) Ahora se quitarán todas las impresiones y se realizará directamente la suma con la directiva parallel realizando nosotros la división del trabajo de forma manual, realizando, por ejemplo, la división del vector en tantas partes como hilos tengamos
   c) A continuación realizaremos tantas hebras como elementos del vector y para la suma utilizaremos la cláusula reduction
   d) A continuación modificaremos el apartado b para que openMP haga la paralización automática con la directiva for tanto con la cláusula "nowait" como sin ella.
   e) A continuación modificaremos el apartado anterior para probar la directiva "sections". Para ello vamos a definir dos secciones en cada una de las cuales se llevará a cabo la mitad de la suma del vector.
4. A continuación vamos a realizar pequeñas implementaciones para probar el resto de directivas:
   a) Directiva single: para probar el funcionamiento de esta directiva vamos a escribir un pequeño código que imprima por pantalla un mensaje en el que indiquemos si estamos dentro o fuera de esta directiva e imprima cuál es la hebra que está mostrando el mensaje.
   b) Directiva barrier: para probar su funcionamiento vamos a escribir un pequeño código en el que en la parte paralela haga una pausa proporcional al

número de hilo que la realiza y que finalmente lleguen todos a imprimir un mensaje al mismo tiempo. Para comprobarlo también imprimiremos un mensaje antes y después de hacer la pausa.

c) Directiva master: modificando el código del apartado a), realizar la prueba de esta directiva.
d) Directiva critical: modificando el código del apartado b), realizar la prueba de esta directiva.
e) Directiva ordered: imaginamos una aplicación en la que tenemos un vector a de puntuaciones a rellenar. La forma de rellenarlo será, partiendo de las puntuaciones de las dos primeras posiciones, calculamos la puntuación de la tercera. Una vez calculada esta, a partir de las puntuaciones de las 3 primeras posiciones calculamos la 4ª, y así sucesivamente hasta que lleguemos al final del vector.
f) Directiva flush: para probar el funcionamiento de esta directiva se va a implementar un programa que tendrá un vector compartido de tantas posiciones como hilos creemos en la ejecución. Inicialmente todas las casillas del vector valen 1 y cada hilo va a ir incrementando el valor de la casilla que coincide con su identificador de hilo. Para comprobar que la memoria no es consistente vamos a forzar que el hilo padre (0) cuando llegue a un valor divisible entre 100, aumente este valor y sincronice la memoria. Al mismo tiempo, el resto de los hilos va a ir comprobando el valor de la posición 0 del vector, si en algún momento este valor es divisible entre 100 es porque su valor no es consistente, por lo que mostraremos un mensaje indicando el valor que este hilo ve, sincronizamos y posteriormente mostramos el valor real que tiene.
g) Directiva threadprivate: para comprobarla vamos a implementar varias regiones paralelas, en las cuales vamos a poner como variables persistentes el valor inicial del hilo que se ejecuta. Crearemos otra variable no persistente con este mismo valor inicializándolas ambas en la primera región para después comprobar que no se mantiene el valor de la no persistente en la siguiente región (podemos coger como base el programa de holaMundo)

Practica 2.- Búsqueda de errores en códigos con OpenMP

Esta práctica pretende resaltar posibles errores que podemos cometer a la hora de utilizar OpenMP para nuestros proyectos. Se trata de buscar el error y solucionarlo:

1. omp_bug1.c
2. omp_bug2.c
3. omp_bug3.c
4. omp_bug4.c
5. omp_bug5.c
6. omp_bug6.c

Para el código ver los anexos

Practica 3.- Multiplicación de Matrices con OpenMP

Esta práctica se pretende realizar la multiplicación de matrices de la forma A x B = C, siendo las dimensiones de las matrices de la forma A(x,y), B(y,z) y C(x,z)

Practica 4.- Problema libre y voluntario con OpenMP

Esta práctica será una práctica libre y voluntaria. Vosotros tenéis que pensar un problema, plantearmelo y solucionarlo.

ANEXOS

omp_bug1.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N       50
#define CHUNKSIZE   5

int main (int argc, char *argv[])
{
int i, chunk, tid;
float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;

#pragma omp parallel for     \
  shared(a,b,c,chunk)            \
  private(i,tid)             \
  schedule(static,chunk)
  {
  tid = omp_get_thread_num();
  for (i=0; i < N; i++)
    {
    c[i] = a[i] + b[i];
    printf("tid= %d i= %d c[i]= %f\n", tid, i, c[i]);
    }
  }  /* end of parallel for construct */

}
```

omp_bug2.c

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

int main (int argc, char *argv[])
{
int nthreads, i, tid;
float total;

/*** Spawn parallel region ***/
#pragma omp parallel
  {
  /* Obtain thread number */
  tid = omp_get_thread_num();
  /* Only master thread does this */
  if (tid == 0) {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  printf("Thread %d is starting...\n",tid);

  #pragma omp barrier

  /* do some work */
  total = 0.0;
  #pragma omp for schedule(dynamic,10)
  for (i=0; i<1000000; i++)
     total = total + i*1.0;

  printf ("Thread %d is done! Total= %e\n",tid,total);

  } /*** End of parallel region ***/
}
```

omp_bug3.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N     50

int main (int argc, char *argv[])
{
int i, nthreads, tid, section;
float a[N], b[N], c[N];
void print_results(float array[N], int tid, int section);

/* Some initializations */
for (i=0; i<N; i++)
  a[i] = b[i] = i * 1.0;

#pragma omp parallel private(c,i,tid,section)
  {
  tid = omp_get_thread_num();
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }

  /*** Use barriers for clean output ***/
  #pragma omp barrier
  printf("Thread %d starting...\n",tid);
  #pragma omp barrier

  #pragma omp sections nowait
    {
    #pragma omp section
      {
      section = 1;
      for (i=0; i<N; i++)
        c[i] = a[i] * b[i];
      print_results(c, tid, section);
      }

    #pragma omp section
      {
      section = 2;
      for (i=0; i<N; i++)
        c[i] = a[i] + b[i];
      print_results(c, tid, section);
      }

    }  /* end of sections */

  /*** Use barrier for clean output ***/
  #pragma omp barrier
  printf("Thread %d exiting...\n",tid);

  }  /* end of parallel section */
}
```

```
void print_results(float array[N], int tid, int section)
{
  int i,j;

  j = 1;
  /*** use critical for clean output ***/
  #pragma omp critical
  {
    printf("\nThread %d did section %d. The results are:\n", tid,
section);
  for (i=0; i<N; i++) {
    printf("%e  ",array[i]);
    j++;
    if (j == 6) {
      printf("\n");
      j = 1;
      }
    }
    printf("\n");
  } /*** end of critical ***/

  #pragma omp barrier
  printf("Thread %d done and synchronized.\n", tid);

}
```

omp_bug4.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 1048

int main (int argc, char *argv[])
{
int nthreads, tid, i, j;
double a[N][N];

/* Fork a team of threads with explicit variable scoping */
#pragma omp parallel shared(nthreads) private(i,j,tid,a)
  {

  /* Obtain/print thread info */
  tid = omp_get_thread_num();
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  printf("Thread %d starting...\n", tid);

  /* Each thread works on its own private copy of the array */
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      a[i][j] = tid + i + j;

  /* For confirmation */
  printf("Thread %d done. Last element= %f\n",tid,a[N-1][N-1]);

  }  /* All threads join master thread and disband */

}
```

omp_bug5.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 1000000
#define PI 3.1415926535
#define DELTA .01415926535

int main (int argc, char *argv[])
{
int nthreads, tid, i;
float a[N], b[N];
omp_lock_t locka, lockb;

/* Initialize the locks */
omp_init_lock(&locka);
omp_init_lock(&lockb);

/* Fork a team of threads giving them their own copies of variables */
#pragma omp parallel shared(a, b, nthreads, locka, lockb) private(tid)
  {

  /* Obtain thread number and number of threads */
  tid = omp_get_thread_num();
  #pragma omp master
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  printf("Thread %d starting...\n", tid);
  #pragma omp barrier

  #pragma omp sections nowait
    {
    #pragma omp section
      {
      printf("Thread %d initializing a[]\n",tid);
      omp_set_lock(&locka);
      for (i=0; i<N; i++)
        a[i] = i * DELTA;
      omp_set_lock(&lockb);
      printf("Thread %d adding a[] to b[]\n",tid);
      for (i=0; i<N; i++)
        b[i] += a[i];
      omp_unset_lock(&lockb);
      omp_unset_lock(&locka);
      }

    #pragma omp section
      {
      printf("Thread %d initializing b[]\n",tid);
      omp_set_lock(&lockb);
      for (i=0; i<N; i++)
        b[i] = i * PI;
      omp_set_lock(&locka);
      printf("Thread %d adding b[] to a[]\n",tid);
      for (i=0; i<N; i++)
        a[i] += b[i];
      omp_unset_lock(&locka);
```

```
      omp_unset_lock(&lockb);
      }
    }  /* end of sections */
  }  /* end of parallel region */

}
```

omp_bug6.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define VECLEN 100

float a[VECLEN], b[VECLEN];

float dotprod ()
{
int i,tid;
float sum;

tid = omp_get_thread_num();
#pragma omp for reduction(+:sum)
  for (i=0; i < VECLEN; i++)
    {
    sum = sum + (a[i]*b[i]);
    printf("  tid= %d i=%d\n",tid,i);
    }
}


int main (int argc, char *argv[]) {
int i;
float sum;

for (i=0; i < VECLEN; i++)
  a[i] = b[i] = 1.0 * i;
sum = 0.0;

#pragma omp parallel shared(sum)
  dotprod();

printf("Sum = %f\n",sum);

}
```