1 Primitivas básicas de OpenMP

Consultar la página oficial de la plataforma OpenMP http://www.openmp.org/drupal/

Pragmas

Es una directiva para el compilador que permite la definición de nuevas directivas

```
#pragma omp parallel for [private (<listaVariables>)]
                         [firstprivate (<listaVariables>)]
                         [lastprivate (<listaVariables>])
```

En su versión más simple, intenta distribuir el número de iteraciones del *for* entre el número máximo de *threads* existentes (en principio número de procesadores). Si no se indica la cláusula *private*, la única variable local a los *threads* que ejecutan el *for* será el propio índice de dicho *for*. A continuación, se muestra una inicialización de un vector con valores aleatorios:

```
#pragma omp parallel for
  for (i=0; i<numElementos; i++) V[i] = random();
```

La cláusula *firstprivate* fuerza que las variables indicadas se inicialicen con los valores que tenían las correspondientes variables globales tan sólo al principio del lazo –sólo una vez por *thread*-.

La cláusula *lastprivate* fuerza que el *thread* que ejecuta la última iteración del *for*, copie los valores de las variables locales indicadas en las correspondientes variables globales.

```
#pragma omp critical
```

Define una región crítica. A continuación se ve un ejemplo de uso:

```
#pragma omp parallel for private (x)
  for (i=0; i<N; i++) {
    x = (i+0.5) / N;
#pragma omp critical
    area = += 4.0 / (1.0 + x*x);
  }
  pi = area / N;
```

```
#pragma omp parallel for [reduction (<op>:<variable>)]
```

Esta cláusula adicional del pragma "*parallel*", permite especificar una operación de reducción sobre una variable global. En realidad lo que hace es operar internamente con variables locales y luego componer los valores locales en la variable global mediante la operación que se indique. La operación se restringe a: +, *, &, |, ^, && y ||.

El trozo de código que utilizamos para explicar el pragma "*critical*", puede ahora reescribirse como:

```
#pragma omp parallel for private (x) reduction (+:area)
  for (i=0; i<N; i++) {
    x = (i+0.5) / N;
    area = += 4.0 / (1.0 + x*x);
  }
  pi = area / N;
```

```
#pragma omp parallel [private (<listaVariables>)]
```

Con este pragma se indica que el bloque que le sigue sea ejecutado por todos los threads activos. A continuación se muestra un ejemplo:

```
#pragma omp parallel private (numero)
{
  numero = random();
  printf ("Hola, mi numero ha sido el %d\n, numero);
}
```

1.1 Funciones

```
int omp_set_num_threads (int cuantos)
```

Fija el número de threads que se desea estén activos para el resto de regiones a ejecutar en paralelo. En principio, está fijado al número de CPU's de la máquina y será el utilizado en las regiones paralelas cuando no se fije otro de forma explícita.

```
int omp_get_num_threads (void)
```

Devuelve el número de threads que están activos en ese punto.

```
int omp_get_thread_num (void)
```

Devuelve el identificador del thread. Los identificadores van de 0 a omp_get_num_threads().

```
omp_set_num_threads( 4 );
```

```
#pragma omp parallel
{
  int yo;

  yo = omp_get_thread_num();
  if (yo == 0) maestro();
  else         esclavo();
}
```

Ejercicio: Cálculo de PI con el método de MonteCarlo

El método de MonteCarlo consiste en realizar un experimento aleatorio un determinado número de veces. Como sabemos que la frecuencia con que ocurre un suceso se acerca a su probabilidad, a medida que aumenta el número de ensayos nos iremos acercando más y más al valor buscado. Puede utilizarse para estimar probabilidades que serían muy difíciles de calcular de forma teórica, o para corroborar que ocurrirá lo que nosotros esperamos de un determinado experimento.

Para calcular el número π, podemos seguir los siguientes pasos:

- Inscribir un círculo en un cuadrado de lado 2. El radio del círculo será 1.
- Elegir al azar un punto del cuadrado y observar si este punto pertenece o no al círculo.
- La probabilidad de que el punto esté dentro del circulo es la razón entre las áreas, π/4.
- Multiplica por 4 la frecuencia de este suceso y tendrás una aproximación de π.
- La aproximación será mejor cuanto mayor sea el número de ensayos.

Versión Secuencial

```
// piUno.c: Programa que calcula el numero PI mediante el metodo
//          de Monte Carlo basado en circulo de radio 1 inscrito en
//          cuadrado de lado 2.
//          Terminacion controlada por: Numero de iteraciones
#include <assert.h>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
//--------------------------------------------------------------------
int main( int   argc, char *argv[] ) {

  int i, iteraciones, enCirculo=0;
  double x, y, pi;
  struct timeval t0, t1, t;

  // Control parametro => Numero de iteraciones
  if (argc != 2) {
    printf ("Uso: piUno numIteraciones \n");
    exit(0);
  }
  iteraciones = atoi(argv[1]);

  assert (gettimeofday (&t0, NULL) == 0);
  for (i=1; i<=iteraciones; i++) {
    x = (double) random() / (double) RAND_MAX;
    y = (double) random() / (double) RAND_MAX;
    if ((x*x + y*y) <= 1.0) enCirculo++;
  }
  assert (gettimeofday (&t1, NULL) == 0);
  timersub(&t1, &t0, &t);
  pi = (4.0 * enCirculo) / (double) i;
  printf ("Valor de PI = %.6lf\n", pi);
  printf ("Error       = %.6lf\n", fabs(pi-M_PI));
  printf ("Tiempo      = %ld:%ld(seg:mseg)\n", t.tv_sec, t.tv_usec/1000);
  return 0;
}
```

Transforme la versión secuencial del cálculo de PI usando el paradigma de programación de memoria compartida.

Ejercicio: Multiplicación de Matriz usando Programación de Memoria Compartida