Universidad de Alcalá

DEPARTAMENTO DE AUTOMÁTICA
ARQUITECTURA Y TECNOLOGÍA DE COMPUTADORES

Grado en Ingeniería de Computadores
COMPUTACIÓN DE ALTAS PRESTACIONES

Práctica 2-4

Detección y corrección del problema de la falsa compartición

OBJETIVO

Cuando se utiliza el paradigma de memoria compartida, se puede incurrir en algunos problemas que pueden impactar sobre el rendimiento de la aplicación de manera extraordinaria. Uno de esos problemas se conoce con el nombre de la falsa compartición. Resumidamente, las diversas hebras de ejecución utilizan variables distintas dentro de la memoria compartida, pero reservadas en zonas tan próximas que se produce constantemente el fenómeno de vaciado de cache en cada uno de los núcleos. Esto produce un severo deterioro del rendimiento.

En esta práctica, se proporciona un código que puede ser compilado en forma ordinaria u optimizada. En el primer caso, nos encontramos con el problema de la falsa compartición, mientras que el código optimizado lo evita mediante la técnica de usar variables locales, totalmente alejadas unas de otras.

Se pide comprobar y medir la diferencia de rendimientos y tratar de averiguar el tamaño de la línea de cache, es decir, la mínima separación entre las variables que evita el problema de la falsa compartición.

DESARROLLO DE LA PRÁCTICA

Presentarse en la máquina

Como en las demás prácticas, hemos de presentarnos en la máquina `sesamo`.

Descarga del código

El código del programa puede descargarse de la página web de la asignatura, en el enlace `computar_pi.c`.

Edición de un programa

Para editar un programa, hay que utilizar algún editor de texto de los proporcionados en el menú de Aplicaciones.

Compilación y ejecución

La compilación se puede realizar directamente usando el comando `make`.

El objetivo del programa consiste en calcular aproximadamente el valor del número π mediante el método de Monte Carlo. En este método utilizamos un generador de números aleatorios que proporciona valores en el intervalo $[0,1]$ de los números reales. Invocamos el generador dos veces y lo asignamos a la coordenada (x,y). Después, comprobamos si $x^2+y^2 \leq 1$ en el cual caso ese punto

estará dentro de un círculo de radio unidad. Si es así, incrementamos la variable que recoge el *número de aciertos*. Después de repetir este experimento muchas veces, la relación entre el número de aciertos y el número total de experimentos es, muy aproximadamente, $\pi/4$.

Es obvio que ese método es fácilmente paralelizable: simplemente, creamos hebras (las que queramos) y hacemos que todas ellas realicen el mismo experimento, acumulando los aciertos en una variable. Finalizada la ejecución, se suman todos los aciertos y se realiza el cómputo de $\pi/4$.

Código

```
/* ========================================================================== */
/*                                                                            */
/* Grado en Ingeniería de Computadores                                        */
/* ``Computación de Altas Prestaciones''                                      */
/*                                                                            */
/* ========================================================================== */
/* ========================================================================== */
/* La idea es calcular Pi por medio del método de Monte Carlo. Además proba-  */
/* mos el problema de la "falsa compartición". Si cada hebra actualiza direc- */
/* tamente el vector de "aciertos", se producen "choques" entre los caches de */
/* los distintos procesadores y el tiempo de computación sube mucho. Si, en   */
/* cambio, usamos una variable local para acumular los "aciertos" y luego la  */
/* "reducimos" globalmente, los resultados son mucho mejores.                 */
/* ========================================================================== */

# define _XOPEN_SOURCE

# include <stdlib.h>
# include <stdio.h>
# include <math.h>
# include <sys/time.h>
# include <pthread.h>

# define MAX_HEBRAS 512

int total_aciertos, total_fallos, aciertos[MAX_HEBRAS],
    n_muestras, n_muestras_hebra, n_hebras;

void *CalculaPi(void *s)
{
unsigned int semilla;
int          i, *p_aciertos;
double       rand_n_x, rand_n_y;
# if CACHEOPT
int          l_aciertos;
# endif

   p_aciertos  = (int *)s;
   semilla     = *p_aciertos;
   *p_aciertos = 0;

# if CACHEOPT
   l_aciertos = 0;
# endif

   for (i = 0; i < n_muestras_hebra; i++)
   {
      rand_n_x = (double)(rand_r(&semilla))/(double)(RAND_MAX);
      rand_n_y = (double)(rand_r(&semilla))/(double)(RAND_MAX);

      if (((rand_n_x - 0.5)*(rand_n_x - 0.5)  +
           (rand_n_y - 0.5)*(rand_n_y - 0.5)) < 0.25)
# if CACHEOPT
/* ========================================================================== */
/* Actualizamos una variable local...                                         */
/* ========================================================================== */
         l_aciertos++;
# else
/* ========================================================================== */
/* Actualizamos directamente el vector de aciertos.                           */
/* ========================================================================== */
         (*p_aciertos)++;
```

```
# endif
   }

# if CACHEOPT
/* ========================================================================= */
/* ...y al final actualizamos el vector de aciertos.                         */
/* ========================================================================= */
   *p_aciertos = l_aciertos;
# endif

   pthread_exit(0);
}

int main(int argc, char *argv[])
{
int i;
pthread_t p_hebras[MAX_HEBRAS];
pthread_attr_t attr;
double pi_calculado;
double tiempo_ini, tiempo_fin;
struct timeval tv;

   if (argc > 3)
   {
      printf("Uso: %s <número de muestras> <número de hebras>\n", argv[0]);
      return 1;
   }

   switch(argc)
   {
      case 3:
         n_muestras = atoi(argv[1]);
         n_hebras   = atoi(argv[2]);
         break;

      case 2:
         n_muestras = atoi(argv[1]);
         printf("Número de hebras: ");
         scanf("%d", &n_hebras);
         break;

      default:
         printf("Número de muestras: "); scanf("%d", &n_muestras);
         printf("Número de hebras: ");   scanf("%d", &n_hebras);
   }

   if (n_hebras > MAX_HEBRAS)
   {
      fprintf(stderr, "ERROR: número de hebras mayor que MAX_HEBRAS = %d\n",
                      MAX_HEBRAS);
      return 1;
   }

   pthread_attr_init(&attr);
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

   gettimeofday(&tv, NULL);
   tiempo_ini = (double)tv.tv_sec +
                (double)tv.tv_usec/1.0e6;

   total_aciertos = 0;

   n_muestras_hebra = n_muestras/n_hebras;

   for (i = 0; i < n_hebras; i++)
   {
      aciertos[i] = i + 1;
      pthread_create(&p_hebras[i], &attr, CalculaPi, (void *)&aciertos[i]);
   }

   for (i = 0; i < n_hebras; i++)
   {
      pthread_join(p_hebras[i], NULL);
      printf("Hebra %d, aciertos %d\n", i, aciertos[i]);
```

```
      total_aciertos += aciertos[i];
   }

   pi_calculado = 4.0*(double)total_aciertos/(double)n_muestras;

   gettimeofday(&tv, NULL);
   tiempo_fin = (double)tv.tv_sec +
                (double)tv.tv_usec/1.0e6;

   printf("PI calculado: %lf\n", pi_calculado);
   printf("Tiempo de cómputo: %lf\n", tiempo_fin - tiempo_ini);
   fprintf(stderr, "%lf %lf\n", log((double)n_hebras)/log(2.0), tiempo_fin - tiempo_ini);

   return 0;
}
```