2º curso / 2º cuatr.
Grupos A,B,C,D del Grado Ing. Inform.
Doble Grado Ing. Inform. y Mate.

Arquitectura de Computadores (AC)

Examen de Prácticas. 24 de junio de 2014.

Puntuación: 2 puntos **Duración**: 1 hora **Identificación**: DNI

Cuestión 1.(0.5 puntos) Se necesita ejecutar un código OpenMP con 8 threads. Comente todas las alternativas que puede usar con OpenMP para fijar el número de threads a 8 y ordénelas en función de su prioridad (menor a mayor).

Respuesta:

Las alternativas para modificar el número de threads son 3:

- Variable de entorno OMP_NUM_THREADS

  Para fijar el número de threads a 8 se usaría la siguiente sentencia en la terminal:

  export OMP_NUM_THREADS=8
- Función omp_set_num_threads()

  Para fijar el número de threads a 8 se usaría la siguiente sentencia dentro del programa antes de la región paralela:

  Omp_set_num_threads(8)
- Clausula num_threads

  Para fijar el número de threads a 8 se añadiría la siguiente clausula a la cabecera de la región paralela:

  num_threads(8)

  ejemplo de uso: *#pragma omp parallel num_threads(8){…}*

Cuestión 2. (1 puntos) **(a)** (0.5) Implemente un código paralelo OpenMP que calcule el producto escalar de dos vectores, x[] e y[] de N=1000 componentes (declare los componentes como variables globales, implemente en paralelo también la inicialización de los vectores): $z = \sum_{i=0}^{N-1} x(i) \bullet y(i)$.

(b) (0.25) Comente para qué ha usado cada una de las directivas y cláusulas que ha incluido en el código.

(c) (0.05) Indique qué orden o comando usaría para compilar el código desde una ventana de comandos (terminal) si el ejecutable se quiere llamar `pescalar` (utilice en la compilación la opción de optimización con la que ha obtenido mejores tiempos en la práctica de optimización de código).

(d) (0.2) Suponga que debe ejecutar en atcgrid el fichero ejecutable `pescalar` que tiene en el PC del aula de prácticas (o en su portátil), ¿qué haría para ejecutarlo en atcgrid (tenga en cuenta que está en el PC del aula o en su portátil)? ¿Cómo sabría que ya ha terminado la ejecución? ¿dónde podría consultar los resultados de la ejecución?

Respuesta:

a)

Versión 1:

```
#include <stdio.h>
#include <stdlib.h>
#ifdef _OPENMP
#include <omp.h>
#else
#define omp_get_thread_num() 0
#endif

struct tipo{
  int x;
```

Versión 2:

```
#include <stdio.h>
#include <stdlib.h>
#ifdef _OPENMP
#include <omp.h>
#else
#define omp_get_thread_num() 0
#endif

const int N=1000;
int resultado=0,x[N],y[N];
```

```
  int y;
};
const int N=1000;
int resultado=0;
tipo estructura[N];
int main(int argc, char **argv) {
 #pragma omp parallel
 {
 #pragma omp for
  for(int i=0; i<N; i++){
   estructura[i].x=i;
   estructura[i].y=i;
  }
 #pragma omp for reduction(+:resultado)
  for(int i=0; i<N; i++){
   resultado+=estructura[i].x*estructura[i].y;
  }
 }
 printf("Fuera de 'parallel for' resultado=%d\n",resultado);
 return 0;
}
```

```
int main(int argc, char **argv) {
 #pragma omp parallel
 {
 #pragma omp for
  for(int i=0; i<N; i++){
   x[i]=i;
  }
 #pragma omp for
  for(int i=0; i<N; i++){
   y[i]=i;
  }
 #pragma omp for reduction(+:resultado)
  for(int i=0; i<N; i++){
   resultado+=x[i]*y[i];
  }
 }
 printf("Fuera de 'parallel for' resultado=%d\n",resultado);
 return 0;
}
```

b)

he utilizado la primera directiva *"#pragma omp parallel*" para crear las hebras y no hacerlo reiteradamente más adelante. Anidadas dentro de ella están las demás directivas.

Versión 1: la segunda directiva *"#pragma omp for"* la he utilizado para repartir las iteraciones del bucle entre las distintas hebras de forma equitativa (por defecto es static con chunk 1). Esta se encarga de inicializar el vector.

Versión 2: la segunda y la tercera directiva *"#pragma omp for"* la he utilizado para repartir las iteraciones del bucle entre las distintas hebras de forma equitativa (por defecto es static con chunk 1). Estas se encargan de inicializar el vector.

La ultima directiva con la cláusula reduction *"#pragma omp for reduction(+:resultado)"* la he utilizado para repartir las iteraciones del bucle entre las distintas hebras de forma equitativa. La he utilizado con la cláusula reduction para que en cada hebra se genere una variable privada llamada resultado donde cada hebra va a ir almacenando la suma del producto de cada una de sus iteraciones y al final, cuando todas las hebras terminen la cláusula se encargara de unificar todas las variables privadas creadas anteriormente en la variable resultado global usando el operador "+" que es el que se ha especificado en la clausula, realizando correctamente el producto escalar de los vectores.

c)

g++ -fopenmp -O2 pescalar.c -o pescalar

d)

paso 1: conectar con ssh y sftp al front-end de atcgrid

paso2: compilar en pc-local y enviarlo al frontend (en sftp: " *put pescalar* ")

paso3: encolar el ejecutable en la cola ac de torque con el comando " *echo './pescalar' | qsub -q ac* " en shh

paso 4: (una vez haya terminado la ejecución) copiar los resultados al pc-local (en sftp: "*get STDIN**")

paso 5: visualizar el fichero en el pc-local (en terminal con *cat* o en entorno gráfico con *gedit*)

paso 6: eliminar los ficheros generados en el front-end (en shh: "*rm STDIN**")

para comprobar si la ejecución ha terminado o no se usa el comando qstat el cual te devuelve una tabla en la que se

pueden ver los procesos de la cola y algunas características. Una de ellas es el estado (columna S) si tiene valor C es que el proceso ya ha terminado.

Los resultados de la ejecución se generan en dos ficheros:

STDIN.o<identificador>: en este fichero se almacena la salida estándar de la ejecución.

STDIN.e<identificador>: en este fichero se almacena la salida estándar de error de la ejecución.

Cuestión 3. (0.5 puntos) **(a)** (0.25)¿Cuál de los siguientes códigos para C/C++ ofrece mejores prestaciones? ¿Por qué?

```
double m1[n][n], m2[n][n], mr[n][n] ;
…
for (i=0; i<n; i++){
 for (j=0; j<n; j++){
  for(k=0; k<n; k+=4){
   mr[i][j] += m1[i][k] * m2[k][j];
   mr[i][j] += m1[i][k+1] * m2[k+1][j];
   mr[i][j] += m1[i][k+2] * m2[k+2][j];
   mr[i][j] += m1[i][k+3] * m2[k+3][j];
 }
 }
}
…
```

```
double m1[n][n], m2[n][n], mr[n][n] ;
…
for (i=0; i<n; i++){
 for (j=0; j<n; j+=4){
  for(k=0; k<n; k++){
   mr[i][j] += m1[i][k] * m2[k][j];
   mr[i][j+1] += m1[i][k] * m2[k][j+1];
   mr[i][j+2] += m1[i][k] * m2[k][j+2];
   mr[i][j+3] += m1[i][k] * m2[k][j+3];
 }
 }
}
…
```

(b) (0.25) ¿Cuál de los siguientes códigos para C/C++ ofrece mejores prestaciones? ¿Por qué?

```
struct {
        int a;
        int b;
}  s[5000];

main()
{ …
 for (ii=1; ii<=40000;ii++) {
   for(i=0;i<5000;i++)  X1+=2*s[i].a+ii;
   for(i=0;i<5000;i++)  X2+=3*s[i].b-ii;

   … //instrucciones que usan X1 y X2
 }
 …
}
```

```
struct {
        int a;
        int b;
}  s[5000];

main()
{ …
 for (ii=1; ii<=40000;ii++) {
   for(i=0;i<5000;i++) {
      X1+=2*s[i].a+ii;
      X2+=3*s[i].b-ii; }
   … //instrucciones que usan X1 y X2
 }
 …
}
```

Respuesta:

a) el código de la derecha es más eficiente debido a la localidad en memoria.
Si nos fijamos, en el código de la izquierda hay dependencia raw en mr[i][j], además, se produce un fallo de cache en cada línea dentro del bucle for, ya que el segundo operando en cada línea accede a una fila distinta de la matriz. Estos dos problemas se solucionan en el código de la derecha haciéndolo más eficiente.

b) El código de la derecha es más eficiente debido a la localidad en memoria.
Si nos fijamos, en el código de la izquierda se accede a memoria pegando saltitos debido a que las estructuras almacenan sus campos unos contiguos a los otros al igual que los vectores, entonces las posiciones s[0].a y s[0].b estarán contiguas, pero s[0].a y s[1].a no. A demás el código de la izquierda tiene dependencia raw tanto en X1 y X2. Por lo tanto, el código de la derecha es mucho mejor, ya que resuelve los problemas mencionados anteriormente