Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Escuela de Ciencias Exactas y Naturales
Departamento de Ciencias de la Computación
R-322 — Sistemas Operativos I

Prctica 4

Programación Paralela

Nota 0: Recuerde para los ejercicios de OpenMP compilar con `-fopenmp`. De otra manera, `gcc` ignora los `pragma`s. A veces, `gcc` y `clang` dan distintos resultados de performance. No está de más probar ambos. Para MPI, debe usar el compilador `mpicc` y ejecutar con `mpirun`/`mpiexec`.

Nota 1: Para limitar la cantidad de threads que crea un programa OpenMP, puede usar la variable de entorno `OMP_NUM_THREADS`, e.g.:

```
$ OMP_NUM_THREADS=2 ./prog
```

limita el programa a 2 threads.

Nota 2: En los procesadores con "Hyper-threading" (o algún termino marquetinero similar), puede no observar mejoras significativas al superar la cantidad de "cores" reales. Es decir, en un dual-core, con cualquier cantidad de hilos totales, hay algunos problemas para los cuales no podrá superar un 2x (o apenas más) de ganancia en performance. El `labdcc` tiene un quad-core verdadero, cada uno con un único hilo.

Nota 3: Puede usar el archivo `timing.h` para medir tiempos de cómputo.

Ej. 0. Para calentar motores, adapte a OpenMP su solución del jardín ornamental usando el Algoritmo de la Panadería de Lamport.

Ej. 1 (Suma Paralela). Escriba utilizando OpenMP un algoritmo que calcule la suma de un arreglo de $N = 5 \times 10^8$ `double`s. Compare la performance con la implementación secuencial usando distintos números de hilos. Compare también con una versión paralela que usa un mutex para proteger la variable que lleva la suma.

Ej. 2 (Búsqueda del Mínimo). Escriba utilizando OpenMP un algoritmo que dado un arreglo de $N = 5 \times 10^8$ enteros busque el mínimo. Compare la performance con la implementación secuencial con distinto número de hilos.

Ej. 3 (Primalidad). Escriba utilizando OpenMP una función que verifique si un entero es primo (buscando divisores entre 2 y $\sqrt{N}$). Su solución debería andar igual o más rápido que una versión

secuencial que "corta" apenas encuentra un divisor. Escriba su función para tomar un `long`, i.e. un entero de 64 bits[1], y asegúrese de probarla con números grandes (incluyendo primos, semiprimos, y pares).

Ej. 4 (Multiplicación de Matrices). Implemente en OpenMP la multiplicación de dos matrices en paralelo. Una versión secuencial es:

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

#define N 200
int A[N][N], B[N][N], C[N][N];

void mult(int A[N][N], int B[N][N], int C[N][N])
{
        int i, j, k;
        for (i = 0; i < N; i++)
                for (j = 0; j < N; j++)
                        for (k = 0; k < N; k++)
                                C[i][j] += A[i][k] * B[k][j];
}

int main()
{
        int i, j;

        for (i = 0; i < N; i++) {
                for (j = 0; j < N; j++) {
                        A[i][j] = random() % 1000;
                        B[i][j] = random() % 1000;
                }
        }

        mult(A, B, C);

        return 0;
}
```

a) Compare la performance con la solución secuencial para matrices cuadradas de tamaño 200x200, 500x500 y 1000x1000. ¿Qué relación aproximada puede inferir entre los tiempos en uno y otro caso?

b) Si se cambia el orden de los índices, ¿se puede mejorar el rendimiento? ¿Por qué?

c) Si tuviese que computar la multiplicación de $A \times B^T$, ¿se puede mejorar el rendimiento? ¿Por qué?

Ej. 5 (Quicksort). Recordemos el algoritmo de ordenamiento Quicksort:

```
/* Particion de Lomuto, tomando el primer elemento como pivote */
int particionar(int a[], int N)
{
        int i, j = 0;
```

[1]El tamaño exacto de cada tipo entero en C está definido por la implementación (i.e. el compilador). Para el GCC y Clang, en un sistema Linux de 64 bits, un long ocupa 64 bits.

```
        int p = a[0];
        swap(&a[0], &a[n-1]);

        for (i = 0; i < n-1; i++) {
                if (a[i] <= p)
                        swap(&a[i], &a[j++]);
        }

        swap(&a[j], &a[n-1]);
        return j;
}

void qsort(int a[], int N)
{
        if (N < 2)
                return;

        int p = particionar(a, N);
        qsort(a, p);
        qsort(a + p + 1, n - p - 1);
}
```

Dado que las llamadas recursivas para ordenar las "mitades" del arreglo son independientes, son un claro candidato para paralelizar.

- Como primer intento, escriba una versión que use `pthread_create` para paralelizar las llamadas recursivas. Compare el rendimiento con la versión secuencial para distintos tamaños del array. ¿Hay algún problema? Explique.
- Escriba una versión que paralelice las llamadas usando `sections` de OpenMP. ¿Mejora la performance? ¿Cuánto? Puede usar el servidor `labdcc` para probar en un quad-core.
- Escriba una versión usando `task`s de OpenMP y mida el cambio en rendimiento.

Ej. 6 (Mergesort). Siguiendo la misma idea del ejercicio anterior, implemente un mergesort (sobre enteros) paralelo y compare su performance con la versión secuencial. Puede usar tasks, o escribir una versión bottom-up usando solamente `parallel for`. Su solución debería manejar arreglos de 500 millones de enteros sin problema, y ser lo más eficiente posible.