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

Práctica 4

Programación Paralela y Distribuida

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.

Nota 4: En cálculo paralelo interesa el tiempo transcurrido, total, para ejecutar el programa. Para un programa que corre en paralelo, con p procesadores, se introducen dos definiciones: "aceleración" (speedup) y eficiencia.

$$\textbf{Speedup} : S_p = t_s/t_p$$

$$\textbf{Eficiencia} : E_p = S_p/p$$

Donde t_s es el tiempo de ejecución secuencial y t_p el tiempo de ejecución pararlelo.

Ej. 1. Para calentar motores, adapte a OpenMP su solución del jardín ornamental usando el Algoritmo de la Panadería de Lamport.

Ej. 2 (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. 3 (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. 4 (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. 5 (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. 6 (Quicksort). Recordemos el algoritmo de ordenamiento Quicksort:

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

[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 i, j = 0;
        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 `tasks` de OpenMP y mida el cambio en rendimiento.

Ej. 7 (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.

Ej. 8 (Servidor de Turnos). Reimplemente el servidor de tickets de la Práctica 2 en Erlang. Puede usar el siguiente esqueleto para manejar conexiones TCP en Erlang. El mismo acepta conexiones TCP en *modo activo*, haciendo que el proceso que realizar el `accept` de una conexión reciba mensajes con los datos recibidos por la misma. También puede usar el *modo pasivo* si así lo desea, cambiando las opciones pasadas a `listen`. Asegúrese también de que el servidor es robusto: debe manejar correctamente conexiones cerradas por el cliente y también tener en cuenta que los pedidos pueden llegar fragmentados o "pegados" (TCP no tiene concepto de mensaje ni de "borde"), entre otras cosas.

```
-module(turnos).
-export([server/0]).

server() ->
        {ok, ListenSocket} = gen_tcp:listen(8000, [{reuseaddr, true}]),
```

```
        wait_connect(ListenSocket, 0).

wait_connect(ListenSocket, N) ->
        {ok, Socket} = gen_tcp:accept(ListenSocket),
        spawn (fun () -> wait_connect (ListenSocket, N+1) end),
        get_request(Socket).

get_request(Socket) ->
        io:fwrite("Esperando mensajes de ~p~n", [Socket]),
        receive
                _X -> ok,
                      get_request(Socket)
        end.
```

a) Compare el servidor en PThreads y el actual con el cliente dado anteriormente, para 200, 2000 y 20000 conexiones simultáneas. Puede usar el cliente `turno_cliente.c`.

b) ¿Ve una diferencia importante en el consumo de memoria de los dos servidores? ¿A qué cree que se puede deber?

c) ¿Puede cada servidor aceptar 50000 conexiones simultáneas?

Nota: para conseguir aceptar tantas conexiones, seguramente tenga que aumentar el `ulimit` de FDs abiertos que impone el sistema operativo. Correr `ulimit -n 1000000` debería bastar. Ver también `help ulimit`.

Ej. 9 (Lanzar Procesos en Anillos). Escriba un programa que lance N procesos en anillos. Cada proceso recibirá dos clases de mensajes:

- `{msg, N}` donde `N` es un entero. Deberá decrementarlo y enviarlo al siguiente proceso en el anillo si `N` es mayor que cero. En caso contrario deberá enviar un mensaje `exit` y terminar cuando todos los demás lo hayan hecho.
- `exit` cuando el proceso debe terminar.

Modifique el programa para que el mensaje enviado gire una vez alrededor del anillo y sea descartado por el que inició el envío.

Ej. 10 (Suma Distribuida). Implemente en MPI un programa distribuido que compute la suma de un array distribuyendo segmentos del mismo. Su solución debe ser robusta si varía el tamaño del array y/o la cantidad de procesos involucrados.

Corra en programa con distintas cantidad de procesos: 4, 8, y 16.

Ej. 11 (Suma y Consenso por Rotación). Considere en MPI un anillo de N procesos (con N configurable) en el que cada proceso tiene algún valor privado (ej. su rango). Queremos computar la suma de todos los valores, y que la misma resulte disponible en cada proceso. Implemente esto haciendo que

- Como primer paso, cada proceso envía su valor hacia el siguiente proceso del anillo.
- Cada proceso recibe el valor y lo agrega a su suma.
- Cada proceso reenvía el mismo valor que recibió hacia el siguiente.

Al hacer esto, luego de $N - 1$ pasos, cada proceso debería tener la suma total computada.

Ej. 12 (Suma y Consenso). En el ejercicio anterior, logramos que N procesos sumen sus variables privadas y todos conozcan el resultado en $N-1$ pasos (paso = envío/recepción de un mensaje). Si N es muy grande (ej. miles de procesos) esto puede introducir una latencia muy alta. Diseñe una manera de realizarlo en $\lg_2 N$ pasos (puede asumir que N es potencia de 2). Todo proceso debe usar memoria $O(1)$. Verifique que su solución es robusta.

Ej. 13 (Suma y Consenso en Erlang). Reimplementar la suma por consenso en Erlang.

Compare esta nueva implementación con la de MPI:

a) ¿Qué similitudes y diferencias encontró en cuanto a la implementación?

b) Cómpare la performance y la robustez. ¿Qué pasa si un proceso muere en cada caso?

Nota: En linux puede usar el comando `/usr/bin/time -v PROGRAMA` para ver el uso de memoria. Por ejemplo haciendo `/usr/bin/time -v erl` y luego saliedo del interprete podemos tener una noción de cuanta memoria usa Erlang para levantar su entorno de ejecución.

Para un profiling más detallado se puede usar algunas de las herramientas que provee Erlang para profiling[2].

Ej. 14 (Producto distribuido). El siguiente fragmento de código permite calcular el producto de una matriz cuadrada por un vector, ambos de la misma dimensión

```
#include <stdio.h>
#include <stdlib.h>
#define N  // definir

int main(int argc, char **argv)
{
    int i, j;
    int A[N][N], v[n], x[n];

    /*Leer A y v*/
    for (i=0;i<n;i++) {
        x[i]=0;
        for (j=0;j<n;j++)
            x[i] += A[i][j]*v[j];
    }
    /*Escribir x */
    return 0;
 }
```

a) Escriba un programa MPI que realice el producto en paralelo, teniendo en cuenta que el proceso 0 lee la matriz A y el vector v, realiza una distribución de A por bloques de filas consecutivas sobre todos los procesos y envía v a todos los procesos. Asimismo, al final el proceso 0 debe obtener el resultado.

b) Calcular el speed up y la eficiencia.

Ej. 15 (IO Distribuida). Corra el siguiente programa con diferentes números de procesos y describa que hace.

[2] `https://www.erlang.org/docs/22/efficiency_guide/profiling.html`

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

#define tambuf 4*32

int main(int argc, char **argv)
{
    int pid, npr;
    int i, numdat;
    int buf[tambuf], buf2[tambuf], modoacceso;
    MPI_File dat1;
    MPI_Status info;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    MPI_Comm_size(MPI_COMM_WORLD, &npr);
    numdat = 4;

    if (pid == 0)
        for(i=0; i<npr*numdat; i++) buf[i] = i*i;
    if (pid == 0){
        modoacceso = (MPI_MODE_CREATE | MPI_MODE_WRONLY);
        MPI_File_open(MPI_COMM_SELF, "dat1", modoacceso, MPI_INFO_NULL, &dat1);
        MPI_File_seek(dat1, 0, MPI_SEEK_SET);
        MPI_File_write(dat1, buf, npr*numdat, MPI_INT, &info);
        MPI_File_close(&dat1);
        printf("\n El master escribió %d datos, desde 0 hasta %d \n\n",
                                    npr*numdat, buf[npr*numdat-1]);
    }

    sleep(3);
    modoacceso = MPI_MODE_RDONLY;
    MPI_File_open(MPI_COMM_WORLD, "dat1", modoacceso, MPI_INFO_NULL, &dat1);
    MPI_File_seek(dat1, pid*numdat*sizeof(int), MPI_SEEK_SET);
    MPI_File_read(dat1, buf2, numdat, MPI_INT, &info);
    MPI_File_close(&dat1);
    printf(" > %d ha leido %5d %5d %5d %5d \n",
                                pid, buf2[0], buf2[1], buf2[2], buf2[3]);

    MPI_Finalize();
    return 0;
}
```

Ej. 17 (Servidor de Difusión). Implemente un proceso servidor que distribuya los mensajes que recibe entre todos sus suscriptores. El servidor tiene las siguientes operaciones:

Suscribirse:	El proceso llamado es incluido en el conjunto de suscriptores.
Enviar mensaje:	El mensaje recibido debe ser reenviado a todos los suscriptores.
Desuscribirse:	El proceso llamado es eliminado del conjunto de suscriptores.

Cada operación debe tener una función que la implemente, por ejemplo `suscribir/1`. El servidor puede iniciarse con una función que retorne un descriptor del mismo, o puede registrarse globalmente (en cuyo caso, `suscribir` tiene aridad cero). Si hay paso de mensajes, el mismo debe estar abstraído detrás de esa interfaz. En cada difusión, los suscriptores deben recibir el mensaje una única vez. Una vez suscrito, no debería tener efecto suscribirse nuevamente, y desuscribirse siempre tiene efecto inmediato (i.e. las suscripciones no son recursivas).

Ej. 18 ("Hello" tolerante a fallas). El siguiente programa crea un proceso que imprime "Hello" a intervalos regulares. Por una falla desconocida termina al poco tiempo con un error.

```
-module(hello).
-export([init/0]).

hello() ->
        receive after 1000 -> ok end,
        io:fwrite("Hello ~p~n", [case rand:uniform(10) of 10 -> 1/uno; _ -> self() end]),
        hello().

init() -> spawn(fun () -> hello() end).
```

Reemplace este proceso por dos, donde el segundo deba levantar al proceso que imprime "Hello" cada vez que se caiga.

Nota: puede ser de ayuda utilizar `process_flag(trap_exit, true)`.

Ej. 19 (Cambio en Caliente). El cliente está satisfecho con el servicio de salutación, pero le gustaría que lo salude en castellano y no en inglés. Modifique el código de manera que, una vez levantado el servicio, se pueda cambiar el mensaje por "Hola" sin darlo de baja. Es decir, que se pueda hacer lo siguiente:

```
2> hello:init().
...
Hello <0.XX.0>
Hello <0.XX.0>
```

Después reemplazar `Hello` por `Hola` en `hello.erl`, en la misma EShell hacer:

```
3> c(hello).
Hello <0.XX.0>
Hola <0.XX.0>
...
```

Notar que el PID del proceso no debería cambiar si el proceso no muere por el error.

Ej. 20 (Ping Pong distribuido). Los sistemas Erlang que se comunican entre sí en diferentes computadoras deben tener la misma *magic cookie*. La forma más sencilla de lograr esto es tener un archivo llamado *.erlang.cookie* en el directorio de inicio en todas las máquinas en las que se va a ejecutar el sistema Erlang comunicándose entre sí. Con respecto a los permisos, el archivo sólo debe poder ser leído por el dueño del archivo (`chmod 400 .erlang.cookie`).

Para iniciar un sistema Erlang que va a comunicarse con otros sistemas Erlang, hay que darle un nombre:

```
erl -sname my_name
```

a) Implemente el programa en Erlang de ping-pong visto en clase para que pueda correr de manera distribuida.

b) Pruebe su programa en forma simultánea con 2 pcs del laboratorio, de modo que una ejecute el ping y la otra el pong.

c) Documente la forma de lanzar los programas y la salida de cada terminal.