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.

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

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

Nota 5: Máquinas y directorios: Vamos a trabajar el cluster del Conicet Rosario, [1] aunque no es obligatorio.

Uno de los nodos del cluster funciona como de servidor de archivos y de punto de entrada al cluster (front-end); su dirección externa es `piluso.rosario-conicet.gov.ar`. Para acceder al cluster primero nos conectamos al `labdcc` :

```
ssh openmpi@labdcc.fceia.unr.edu.ar
```

y luego desde allí nos conectamos al cluster:

```
ssh piluso
```

Cada grupo tiene su propia carpeta de trabajo. El cluster usa Sun Grid Engine (SGE) para el scheduling de trabajos.

Recordar: Para correr con mpi usar el entorno paralelo `mpi` y para usarlo con openmp el entorno `openmp`. Por ejemplo:

[1] cluster.rosario-conicet.gov.ar

```
#$ pe mpi 4
#$ pe openmp 4
```

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[2], 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;
```

[2]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.

```
                }
        }

        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;
        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. 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.

Ej. 7 (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.

Ej. 8 (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. 9 (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. 10 (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. 11 (IO Distribuida). Corra el siguiente programa con diferentes números de procesos y describa que hace.

```
#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;
}
```