Tarea 1.2

Implementación de algoritmos paralelos

Publicada el 4 de septiembre de 2018

Fecha de entrega (parte 2): martes 25 de septiembre de 2018, 23:59 hrs., en Git.
Código base: Ingresar a GitHub Classroom en `https://classroom.github.com/g/rSIYA8od`

1. Objetivo

En esta segunda parte de la primera tarea, deberás implementar una versión paralela del algoritmo de ordenación Quicksort[1], utilizando los conceptos sobre procesos concurrentes y sincronización vistos en el curso, y su implementación a través de la API de *threads* POSIX.

2. Prerequisitos

Para esta segunda parte de la tarea no es necesario haber completado la parte anterior, pues el algoritmo Quicksort opera en forma completamente independiente del algoritmo de búsqueda binaria. De todos modos, es importante continuar repasando programación en C, incluyendo entrada y salida en consola, control de flujo, arreglos, strings, punteros y manejo de memoria. Para esto, puedes revisar tus apuntes del curso anterior de Programación de Bajo Nivel, y algún libro como "*Essential C*". El capítulo 14 del libro de Arpaci-Dusseau describe el funcionamiento de la API de memoria en C, el capítulo 27 la API de `threads`, y el 28 algunas primitivas de sincronización útiles.

Además, es indispensable contar con un ambiente de programación que incluya las herramientas de desarrollo GNU, como `gcc` (compilador de C), `valgrind` (análisis de memoria), `helgrind` (análisis de concurrencia), `gdb` (depurador), `make` (scripts para compilación), y también `Git` para versionar adecuadamente tu código. El sistema operativo oficial para las tareas del curso será Ubuntu 16.04 LTS `http://www.ubuntu.com/download/desktop`. Puedes instalar Ubuntu rápidamente en tu computador como una máquina virtual con productos como Virtualbox, VMWare o Parallels. Sin embargo, es factible desarrollar la tarea en otros ambientes, como en Cygwin y en Mac OS X, mientras el código finalmente compile correctamente en Ubuntu 16.04 LTS.

3. Descripción y Requisitos

La tarea aquí pedida consistirá en desarrollar un programa que implemente una versión paralela del algoritmo Quicksort. A continuación describimos la versiones serial y paralela de este algoritmo.

[1] Ver `https://en.wikipedia.org/wiki/Quicksort`

3.1. Quicksort Serial

El algoritmo Quicksort serial puede ser el descrito por Cormen et al. (2009), cuyo pseudocódigo es el siguiente:

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p - 1 )
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
    pivot := A[hi]
    i := lo
    for j := lo to hi - 1 do
        if A[j] < pivot then
            swap A[i] with A[j]
            i := i + 1
    swap A[i] with A[hi]
    return i
```

Notar que la llamada inicial a la función `quicksort debe ser quicksort(A, 0, length(A) - 1)`. Esto permite ordenar un arreglo `A` desde la posición inicial a la final.

3.2. Quicksort Paralelo

El algoritmo Quicksort paralelo a implementar está descrito en el capítulo 9.4.3 del libro "Introduction to Parallel Computing", disponible en `http://parallelcomp.uw.hu/`. Se adjunta a este enunciado dicho capítulo en formato PDF. A continuación se resume el funcionamiento:

- Se define un número arbitrario p de *threads*. Se puede considerar p como el doble de núcleos de CPU disponibles. El i-ésimo *thread* lo llamaremos $P_i, i = 0 \ldots p-1$.
- El arreglo A ordenar de tamaño n se particiona en trozos de tamaño n/p. Llamaremos A_i al subarreglo (bloque) asignado al proceso P_i.
- Un *thread* elige un valor pivote aleatorio dentro de A, y lo comunica a todos los demás *threads*.
- Cada proceso P_i mueve los valores en su bloque A_i, de manera que en la mitad izquierda quedan los valores menores al pivote (sub-bloque S_i), y en el lado derecho los valores mayores (sub-bloque L_i). Esto es similar a lo que hace el procedimiento `partition` dentro de su ciclo en el algoritmo quicksort serial.

- Se hace reordenamiento global de los ítems del arreglo A creando un arreglo A', en donde se copian los elementos en orden descrito de acuerdo a la figura 9.19 en el libro mencionado arriba.
- El arreglo A' tiene dos secciones; una que contiene los valores menores al pivote, S, y otra sección contigua a la derecha que contiene los valores mayores, L.
- El algoritmo se repite recursivamente, asignando los primeros $\lceil |S|p/n + 0{,}5 \rceil$ *threads* a la sección S y los *threads* restantes a la sección L.
- La recursión termina cuando un sub-bloque de A con todos los valores mayores (o menores) al pivote es asignado a un proceso. En este caso, el proceso ejecuta la versión serial del algoritmo Quicksort sobre el bloque que recibe.

3.3. Funcionamiento del Programa

El programa principal, llamado `quicksort`, debe recibir como argumentos por línea de comando (¡usar `getopt` para esto![2]) un valor E ($E \geq 1$), el cual indica la cantidad de ejecuciones secuenciales que deben realizarse del algoritmo quicksort paralelo. Además, el valor T ($3 \leq T \leq 9$) indica la potencia de 10 correspondiente al tamaño del conjunto de datos a generar para cada ejecución del algoritmo. A diferencia de la parte anterior de la tarea, ahora se requiere un nuevo conjunto de valores para cada ejecución. Ejemplo de ejecución en la línea de comando:

```
$./quicksort -T 2 -E 3
```

Quicksort es ejecutado tres veces con conjuntos de $10^2 = 100$ valores.

Como en la parte anterior de la tarea, para generar los datos para cada ejecución, el proceso `quicksort` deberá comunicarse a través de un *domain socket* de Unix con el mismo proceso `datagen`, el cual en esta oportunidad proveerá un flujo aleatorio de valores sin orden particular. Un nuevo proceso para `datagen` debe ser iniciado por `quicksort` cada vez que ejecute. Además `quicksort` debe encargarse de terminar `datagen`, y esperar su terminación para cerrarse. Se provee la implementación del proceso `datagen`, el cual escucha conexiones en el socket, y atiende peticiones del proceso `quicksort`. Las comunicaciones entre `quicksort` y `datagen` se describen en la sección siguiente.

4. Comunicación Inter-Procesos

El proceso `quicksort` deberá iniciar el proceso `datagen`, y luego comunicarse con éste a través de un Unix *domain socket* en `/tmp/dg.sock`. El proceso datagen acepta una conexión desde `quicksort`, y recibe sus instrucciones a través del socket. Las instrucciones son los siguientes mensajes (strings) con texto ASCII:

[2]Ver `https://www.gnu.org/software/libc/manual/html_node/Example-of-Getopt.html#Example-of-Getopt`

- `BEGIN U <T>`: A diferencia de la parte anterior de la tarea, notar que se usa caracter `U` en vez de `S`. El valor `<T>` corresponde al recibido por `quicksort` desde la línea de comando. El proceso `datagen` responderá con un string con el texto `OK\n\n`, y luego comenzará a generar un flujo aleatorio de enteros sin signo de 32 bits, de tamaño 10^T que el proceso `binsearch` podrá leer desde su socket cliente. Notar que la lectura desde el socket se debe realizar utilizando un arreglo de bytes (de unos 1000 bytes sería recomendable), y desde allí se debe extraer el mensaje `OK\n\n`, y luego los enteros de 4 bytes sin signo.
- `END`: Este mensaje es enviado por `quicksort` a `datagen` para terminarlo.

5. Reporte de resultados

5.1. Implementación de la salida en `quicksort`

Para cada una de las E ejecuciones del algoritmo quicksort paralelo, el programa `quicksort` debe imprimir en consola el arreglo A original con los valores obtenidos desde `datagen`, en una sola línea de texto, con los valores separados por coma. En la línea siguiente, deben aparecer los mismos valores pero ordenados utilizando el algoritmo Quicksort, de menor a mayor.

El formato de ambas líneas tiene un prefijo indicando el número de la ejecución, y un caracter indicando si la línea es entrada (`E`) o salida (`S`). Ejemplo:

```
E1:67,4,12,56,23,45,6,2,6,5,542...
S1:2,4,5,6,6,12,23,45,56,67,542...
```

5.2. Informe

Junto con el código de su tarea, debe entregar un informe escrito que debe contener las siguientes secciones:

- Problemas encontrados en la implementación
- Funcionalidad pendiente sin implementar

El informe debe encontrarse en un archivo de texto (o formato *markdown*) disponible en el directorio raíz del repositorio de código. Además, **el informe debe contener los nombres de los integrantes del grupo**.

6. Consideraciones Técnicas

El sistema operativo oficial para desarrollar esta y la siguientes tareas del curso es Ubuntu Linux 16.04 LTS. Si se ejecuta en una máquina virtual, se debe tener cuidado de asignar todos los

núcleos de CPU disponibles al sistema operativo huesped (*guest*), a fin de que el algoritmo paralelo implementado utice todos los núcleos disponibles.

Para desarrollar esta tarea necesitarás usar funciones de la librería estándar de C, y también funciones de la interfaz de programación de llamadas de sistema[3]. Algunas de las llamadas de sistema y funciones relevantes de la biblioteca estándar de C a utilizar podrían ser las siguientes (revisar la documentación de cada una):

- `read`, `write`: operaciones para leer y escribir desde/a un socket.
- `pthread_create`, `pthread_join`, `pthread_broadcast`: biblioteca de *threads* POSIX.
- `fork`, `wait`, `getpid`, `pipe`: creación y administración de procesos y *sockets*.
- `malloc`, `realloc`, `free`: manejo de memoria dinámica.
- `strnlen`, `strncmp`, `strncpy`, `snprintf`: manejo de *strings*.
- `perror`, `strerror`: manejo de errores.

7. Evaluación

7.1. Inspección de código y funcionamiento (65 %)

Se revisará el código de la tarea en forma exhaustiva y se realizarán pruebas de funcionamiento con los distintos parámetros involucrados.

7.2. Manejo de errores (10 %)

Debe existir código con adecuado manejo de errores en situaciones como las siguientes:

- Error al crear socket.
- Invocación a `fork` retorna `-1`.
- Error al iniciar *thread*
- Invocación a `malloc` retorna `NULL`.

7.3. Manejo de memoria (15 %)

Es importante que toda la memoria pedida por el programa al sistema operativo a través de invocaciones a funciones como `malloc` y `realloc` sea correctamente liberada llamando a `free`, y que además, todo el acceso a la memoria a través de punteros sea implementado en forma correcta.

[3] *POSIX system calls API*, hay una buena lista en `http://www.tutorialspoint.com/unix_system_calls/index.htm`

7.4. Informe (10 %)

La tarea debe incluir el informe descrito anteriormente en un archivo de texto de nombre `README`, ubicado en el directorio raíz del código de la tarea.

8. Modalidad de Trabajo

La tarea debe ser desarrollada en grupos de 2 (parejas). Cada integrante debe contar con una cuenta de usuario en GitHub. El desarrollo debe realizarse por ambos integrantes, y deben quedar claramente registradas sus operaciones de *commit* en el sistema, con comentarios descriptivos en cada una de estas operaciones.

El profesor publicará en Canvas el medio a través del cual se deben inscribir los grupos e informar las cuentas de usuario en GitHub.

Se proporcionará junto con este enunciado un código base con algún trabajo adelantado. Para compilar la tarea basta invocar `make` en el directorio en donde se encuentra el código. El nombre del programa ejecutable es `bank`.

9. Entrega

La tarea debe compilar; en caso contrario, la nota máxima será un 3,9 – apelable – si el error de compilación es un detalle mínimo.

El lenguaje C tiene una serie de dialectos y extensiones según la versión de las herramientas que se esté usando para desarrollar. Para evitar confusiones, la evaluación será hecha compilando con `gcc` (la versión incluída en la distribución de Ubuntu de referencia), usando los parámetros `-std=gnu11 -Wall -Werror -lpthread`.

La entrega de la tarea debe hacerse hasta las 23:59 hrs. en el repositorio Git del grupo. Para hacer efectiva esta restricción, se evaluará la última revisión del repositorio del grupo previa a la hora de cierre. Si un grupo hace su primer *commit* y/o *push* después de la hora de cierre, tendrá derecho a nota máxima 4,0.