I - S - 2021 - RRF - Programación Paralela y Concurrente - 002

Proyecto01-Avance02: Aplicación concurrente

Nota: Para optar por la mejora en la calificación del avance 01 tras la revisión, el equipo debe crear los *issues* y aplicar las correcciones siguiendo los mismos lineamientos para este aspecto indicados en las tareas individuales.

En el avance 1 se construyó un *servidor web* concurrente que podía atender una cantidad arbitraria de clientes de forma simultánea, los cuales solicitan obtener sumas de Goldbach calculadas por una *aplicación web* serial. Sin embargo, este diseño no es óptimo. Por ejemplo, si en una máquina con 8 núcleos de procesador se levanta un servidor web capaz de atender 1000 conexiones concurrentes, y en efecto se conectan 1000 usuarios que solicitan una cantidad considerable de trabajo cada uno, el servidor tendrá 1000 hilos (ó "1000 calculadoras") compitiendo por los 8 núcleos, lo que crea una gran competencia y cambio de contexto por ellos, además de mayor consumo de memoria principal. En este avance se creará una solución que haga mejor uso de los recursos de la máquina.

Entregar a más tardar el domingo 11 de julio a las 23:59. Las entregas tardías se tratarán como se estipula en la carta al estudiante. Deben entregar su solución en el repositorio privado que para ese efecto ya creó cada grupo, dentro de una carpeta exclusiva para este proyecto, dado que el repositorio albergará los dos proyectos del curso. Deben

asegurarse de que los docentes a cargo tenga acceso a dicho repositorio. Todos los miembros del grupo deben demostrar un nivel similar de participación en el desarrollo de la solución. Para esto se les recomienda hacer `commits` frecuentes con comentarios breves pero descriptivos de lo que representa cada `commmit`.

Diseño [30%] ✓

Esta segunda entrega consiste en modificar su *aplicación web* para que no haga un consumo agresivo de los recursos de la máquina. Esta modificación consiste en hacer a la aplicación web concurrente, para que pueda controlar la cantidad de hilos que realmente consumen CPU en el cálculo de las sumas de Goldbach, para que estos **no superen** la cantidad de núcleos de procesador disponibles en el sistema.

El equipo debe primero diseñar una solución que cumpla lo especificado anteriormente. Este diseño puede verse como una cadena de producción, que recibe listas de números en solicitudes HTTP, éstas pasan por un optimizado proceso de fabricación de respuestas, con las que se responden a los usuarios.

El diseño de este proceso de producción debe considerar que la cantidad de hilos "calculadoras" de Goldbach debe estar limitado a la óptima para el sistema. El rol de estos hilos debe estar claramente definido en el patrón concurrente visto en el curso (productores, consumidores, ensambladores, o repartidores). El diseño debe contemplar cómo se comunican las "calculadoras" con otros hilos.

El diseño debe indicar los datos y las estructuras en toda fase de la cadena de producción. Es decir, para cada cola de espera debe indicarse el tipo de datos almacenado en ellas como una clase en UML

(su nombre, atributos, y métodos). Esto es muy importante, dado que el diseño debe balancear la carga de trabajo lo más equitativamente posible entre las CPUs disponibles en el sistema. Por tanto, deberán tomar decisiones de descomposición y mapeo que afectan a la naturaleza de las solicitudes. Es decir, las solicitudes serán descompuestas en unidades más finas de trabajo, y es necesario poder llevar rastro de ellas para poder reensamblar las respuestas que serán enviadas a los clientes.

El diseño concurrente al que llegue el equipo deberá ser expresado una red de Petri, pseudocódigo, o preferiblemente ambas, complementado con diagramas UML para los objetos. La red de Petri sirve para dar una vista de bosque a la cadena de producción. El pseudocódigo sirve para detallar las operaciones que se realizan en las transiciones de la red de Petri. Los diagramas de clases sirven para diseñar los valores que se almacenarán en las colas de la cadena de producción.

Los diseños deben guardarse en la carpeta `design/`. En el archivo `design/design.md` (o `design/README.md`) incrustar las imágenes de las redes de Petri, algoritmos (pseudocódigo), y diagramas de clase. Este documento permite al equipo escribir anotaciones o explicaciones que les ayude en la fase de implementación o a futuros miembros del equipo poder comprender el diseño de la solución. Además da un aspecto profesional al repositorio cuando se visita la carpeta desde el alojamiento de control de versiones.

Implementación [60%]

Implementar el codigo y procesar varios numeros

Se debe implementar el diseño de la *aplicación web* concurrente usando siguiendo el *patrón productor-consumidor* visto durante las lecciones del curso y el código C++ dado correspondiente.

Su solución debe realizar los cálculos correctamente. Es decir, la aplicación web responde con una página que tiene las sumas de Goldbach correctas para las entradas solicitadas y en el mismo orden.

Al igual que el avance anterior, la aplicación permite ingresar listas de números separados por comas, positivos o negativos, tanto en el URI (barra de direcciones del navegador), como el formulario web. Además valida entradas. Números mal formados, fuera de rango, o entradas mal intencionadas son reportados con mensajes de error (en la página web resultado), en lugar de caerse o producir resultados con números incorrectos.

Al finalizar el servidor web (con `Ctrl+C` o el comando `kill`), éste debe reaccionar a la señal y hacer la limpieza debida. Por ejemplo, detenerse de aceptar más conexiones de clientes, avisar a los hilos secundarios para que terminen su ejecución, tanto los que atienden conexiones como los que encuentran sumas de Goldbach ("calculadoras"), esperar que terminen el trabajo pendiente, y finalmente liberar recursos como estructuras de datos en memoria dinámica, archivos, u otros.

La implementación de la solución debe apegarse al diseño elaborado por el equipo. Se verificará que haya una correspondencia entre los elementos del diseño y los elementos de la implementación.

Repositorio, documentación y estructura de archivos [10%] ✓

Buen uso del repositorio de control de versiones. Debe tener *commits* de cada integrante del equipo. Debe haber una adecuada distribución de la carga de trabajo, y no desequilibrios como que "un miembro del equipo sólo documentó el código". Cada *commit* debe tener un cambio

con un significado, un mensaje significativo, y "no romper el build".

Las clases, métodos, y tipos de datos, deben estar documentados con Doxygen. La salida de Doxygen debe estar en una carpeta `doc/` y ésta NO se debe agregar a control de versiones. Documentar el código no trivial en los cuerpos de las subrutinas.

Debe mantenerse la estructura de archivos y directorios característica de un proyecto de Unix, resumido en la siguiente tabla.

Recurso	Descripción	Versionado
`bin/`	Ejecutables	No
`build/`	Código objeto (.o) temporal	No
`design/`	Diseño de la solución en redes de Petri o pseudocódigo o ambos	Sí
`doc/`	Documentación generada por doxygen	No
`src/`	Código fuente (.hpp y .cpp)	Sí
`test/`	Casos de prueba	Sí
`Makefile`	Compila, prueba, genera documentación, limpia	Sí
`Doxyfile`	Configura Doxygen para extraer documentación del código fuente	Sí
`README.md`	Presenta el proyecto, incluye el manual de usuario	Sí
`.gitignore`	Ignora los recursos que NO deben estar en control	

	de versiones (columna Versionado)	Sí

Diseño de la *aplicación web* distribuida [0%]

Para el tercer avance del proyecto 1, la *aplicación web* debe escalar la concurrencia y calcular las sumas de Goldbach de forma distribuida, tratando de descomponer y mapear el trabajo lo más equitativamente posible entre procesos ubicados en máquinas conectadas por una red de computadoras. El equipo debe preparar redes de Petri, pseudocódigo, UML, o una combinación de ellos, de cómo planean implementar el cálculo de sumas de forma distribuida para lograr este ideal. La intención es que los docentes podamos ayudarles durante la revisión del segundo avance a continuar con la tercera fase del proyecto.

Aspectos transversales

El peso en la evaluación de cada componente del avance se encuentra en los títulos anteriores. En cada uno de ellos se revisa de forma transversal:

1. La corrección de la solución. Por ejemplo: comparar la salida del servidor web contra un caso de prueba.

2. La completitud de la solución. Es decir, si realiza todo lo solicitado.

3. La calidad de la solución ejecutable. Por ejemplo, no generar condiciones de carrera, espera activa, bloqueos mutuos, inanición (puede usar herramientas como `helgrind`, `tsan`). Tampoco accesos inválidos ni fugas de memoria (`memcheck`, `asan`, `msan`,

`ubsan`).

4. La calidad del código fuente, al aplicar las buenas prácticas de programación. Por ejemplo, modularizar (dividir) subrutinas o clases largas, y modularizar los archivos fuente (varios `.hpp` y `.cpp`). Apego a una convención de estilos (`cpplint`). Uso identificadores significativos. Inicialización de todas las variables. Reutilización de código