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

El proyecto 1, sobre concurrencia de tareas, se realiza en grupos de máximo tres personas. Entregar a más tardar el domingo 13 de Junio 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 el docente 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`.

Análisis [5%]

El objetivo del proyecto es desarrollar un servidor web concurrente y distribuido que permita a sus visitantes obtener las sumas de Goldbach de números en los que están interesados. En este primer avance el servidor web debe ser concurrente, y la aplicación web debe ser orientada a objetos. Los detalles del avance fueron especificados oralmente por los docentes, tal como se estipula en la carta al estudiante, con el fin de emular la práctica en la industria donde los profesionales en informática recaban los requerimientos de los clientes.

Al revisar las grabaciones o las notas de clase, los miembros del equipo recaban los requerimientos y los anotan en un archivo `README.md` en formato Markdown en la carpeta para el proyecto 01 en el repositorio de control de

versiones. Al leer el README.md debe quedar claro a un lector ajeno al proyecto, **qué** problema éste resuelve, y quienes son los integrantes del equipo. Indicar además cómo compilar el servidor web y cómo correrlo (por ejemplo, explicar los argumentos en línea de comandos que recibe).

Una vez implementada la aplicación de Goldbach, es opcional agregar una captura de pantalla de una consulta hecha a través de un navegador. Esta es una práctica común en los repositorios de control de versiones que provee un efecto más llamativo y profesional del proyecto. De hacerlo, guarden las imágenes en una subcarpeta `img/` y no en la raíz del repositorio.

Diseño [15%]

Esta primera entrega consiste en modificar el programa que se les dio para que el *servidor web* se comporte de manera concurrente, es decir, que sea capaz de aceptar y atender múltiples conexiones de manera concurrente. El *servidor web* debe recibir solicitudes de usuarios las cuales consisten de listas de números para los cuales una *aplicación web* calculará las sumas de Goldbach, tal y como se les ha solicitado en las tareas individuales 1 y 2.

Su solución debe tener un buen diseño concurrente expresado en pseudocódigo o una red de Petri o ambas. El diseño debe centrarse en la lógica concurrente y no los detalles algorítmicos de más bajo nivel (tales como cálculo de las sumas de Goldbach, establecimiento de conexiones de red entre clientes y servidor, entre otros). Los diseños deben guardarse en la carpeta `design/`

Quienes gusten, pueden agregar un archivo `design/design.md` (o `design/README.md`) que incruste imágenes de las redes de Petri o haga enlace a los algoritmos. 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.

Servidor web concurrente [40%] ✓

El código base brindado por los docentes, implementa un servidor web que se comporta de manera serial. El funcionamiento del programa así como el diseño interno del código fuente ha sido explicado durante las clases y los vídeos se encuentran disponibles. Como es sabido, los servidores de software son de naturaleza concurrente. El reto del equipo es convertir a este servidor web serial en uno concurrente para que naturalmente pueda atender varias conexiones de clientes al mismo tiempo.

Al iniciar, el *servidor web* recibe por parámetro un valor que puede llamar `max_connections` que se refiere al número máximo de conexiones de cliente que el *servidor web* debe permitir de manera concurrente. En cada conexión, el *servidor web* debe ser capaz de procesar las solicitudes que tenga el usuario para la *aplicación web*.

Para lograr este propósito se debe modificar la clase `HttpServer` ubicada en la carpeta `http/` del código base dado para implementar un patrón productor consumidor. El productor será el hilo principal, y los consumidores serán objetos que atienden conexiones de clientes (se sugiere el nombre `HttpConnectionHandler` para estos consumidores). Sólo una cola *thread-safe* debe mediar entre el productor y los consumidores, la cual aloja conexiones de clientes pendientes de ser procesadas (copias de objetos `Socket`).

El código en la carpeta `prodcons/` servirá de base crear nuevas clases de productores y consumidores. Puede examinar como ejemplo el programa

`simulation/` que también se describió durante las lecciones del curso. El código brindado tiene además documentación y anotaciones de tipo `TODO:` que proporcionan una guía sobre lo que se debe implementar.

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, esperar que terminen el trabajo pendiente, y finalmente liberar recursos como estructuras de datos en memoria dinámica, archivos, u otros.

Aplicación web orientada a objetos [30%] ✓

En este primer avance no se requiere que la *aplicación web* (la que resuelve las sumas de Goldbach) sea concurrente. Es decisión del equipo si usan una versión serial (Tarea 01) o concurrente (Tarea 02). Sin embargo, la *aplicación web* debe estar implementada en el paradigma de programación a objetos, en coherencia con el resto de la base de código que ya consta de varias clases. El código nuevo del servidor web, la aplicación web, y el cálculo de sumas, deben estar en objetos separados. Se recomienda una separación en al menos las siguientes clases de C++:

1. `HttpConnectionHandler` que se encarga de atender una conexión con un cliente en un hilo consumidor. Cada vez que obtiene una solicitud de su cliente, ensambla los objetos `HttpRequest` y `HttpResponse` y los pasa al `WebServer::handleHttpRequest()`. El *servidor web* se encargará de enrutar la solicitud y si es para la *aplicación web* de Goldbach, la pasará al objeto `GoldbachWebApp`.

2. `GoldbachWebApp` que se encarga de recibir las solicitudes HTTP que realiza el cliente, extraer los números, pasarlos al `GoldbachCalculator`,

obtener las sumas de éste, ensamblar y despachar las respuestas HTTP.

3. `GoldbachCalculator` que se encarga de encontrar las sumas de primos de números solicitados y almacenarlas en estructuras de datos en memoria.

Para su información, la separación anterior se adecua al patrón de diseño de software Modelo-Vista-Controlador (MVC). Note que el objeto de *aplicación web* se encarga de encontrar los números resultados. El *servidor web* no debe realizar ningún cálculo de sumas de Goldbach.

Su solución debe realizar los cálculos correctamente. Es decir, la aplicación web responde con una página que tiene las sumas de Goldbach. Para efectos de este proyecto, el servidor web produce un único mensaje de respuesta por cada solicitud HTTP del cliente, la cual contiene todos los resultados de los números solicitados. Esto implica que si el cálculo de las sumas toma tiempo al servidor, el cliente deberá esperar.

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.

La aplicación valida entradas. Números mal formados, fuera de rango, o entradas mal intencionadas son reportadas con mensajes de error (en la página web resultado), en lugar de caerse o producir resultados con números incorrectos.

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* concurrente [0%]

Para el segundo avance del proyecto 1, la *aplicación web* debe calcular las

sumas de Goldbach de forma concurrente, tratando de descomponer y mapear el trabajo lo más equitativamente posible entre los hilos y sin saturar los núcleos disponibles de la máquina. El equipo debe preparar redes de Petri, pseudocódigo, o ambos, de cómo planean implementar el cálculo de sumas de forma concurrente para lograr este ideal. La intención es que los docentes podamos ayudarles durante la revisión del primer avance a continuar con la próxima 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