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

El proyecto 2, sobre paralelismo de datos y concurrencia declarativa, se realiza en grupos de máximo tres personas. 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`.

Descripción del problema

Un rey generoso de una isla muy lejana, estaba preocupado porque sus ciudadanos estaban aburridos de la monotonía de la pequeña isla. Tuvo la idea de encantar el bosque que está alrededor de su castillo, de tal forma que cada medianoche el bosque cambie y al día siguiente los ciudadanos encuentren un lugar diferente para recrearse. A su mago le pareció genial la idea, pero no sabe qué reglas incluir en el hechizo para que el bosque se mantenga en equilibrio y no llegue a convertirse en un desierto o una selva impenetrable. Si tuvieran alguna forma de ver el efecto de las reglas a futuro, podrían decidir el hechizo con mayor seguridad.

El rey tiene un mapa del bosque como el extracto que puede verse más adelante en el ejemplo de entrada (`map001.txt`). El mapa ilustra lo que hay

en cada metro cuadrado de la isla, a los que les llaman *celdas*. Una celda puede contener un árbol mágico (`'a'`), un trozo de lago encantado (`'l'`), o pradera donde pueden transitar los ciudadanos (`'-'`). Las dimensiones del mapa se encuentran en la primera línea de entrada, indicadas como el número de filas y columnas.

Las reglas que quieren probarse en cada medianoche trabajan en una celda a la vez y consideran las 8 celdas vecinas que tiene alrededor. Las reglas son las siguientes.

1. *Inundación*: Si la celda tiene un árbol y al menos 4 vecinos que son lago encantado, entonces el lago ahoga el árbol, y pasa a ser lago encantado.
2. *Sequía*: Si la celda es lago encantado y tiene menos de 3 vecinos que sean lago encantado, entonces el lago se seca y pasa a ser pradera.
3. *Reforestación*: Si la celda es pradera y tiene al menos 3 vecinos árboles, las semillas tendrán espacio para crecer y la celda se convierte en árbol.
4. *Hacinamiento*: Si la celda es un árbol y tiene más de 4 vecinos árbol, el exceso de sombra evita que crezca y entonces pasa a ser pradera.
5. *Estabilidad*: Cualquier otra situación, la celda permanece como está.

Se quiere que el programa ayude a probar las reglas mágicas anteriores en varios mapas. El programa recibe una orden de trabajo que el mago quiere probar. La orden de trabajo es un archivo cuyo nombre se envía por argumento de línea de comandos. En su contenido, la orden de trabajo lista algunos mapas iniciales de la isla, seguida de un número que indica la cantidad de medianoches en las que quiere probarse el efecto de las reglas del potencial hechizo.

Ejemplo de una orden de trabajo `job001.txt`:

```
map001.txt 2
map002.txt -100
map003.txt -20000
```

En la orden de trabajo, `job001.txt` el número que sucede al nombre de archivo indica la cantidad de noches en las que se debe probar el efecto de las reglas. Si el número es positivo, el programa debe producir un archivo con el estado del mapa en cada amanecer. Por ejemplo, para el primer mapa `map001.txt`, se solicita probar el efecto en `2` noches. En tal caso, el programa produce un archivo de salida con el mapa resultante a la primera medianoche con nombre `map001-1.txt`, y el resultado de la segunda medianoche en el archivo `map001-2.txt`. Estos números positivos permiten al mago rastrear el efecto de las reglas paso a paso.

Cuando el número de noches es negativo, como `-100` para el `map002.txt`, el programa debe aplicar las reglas mágicas esa cantidad de noches, y producir como salida un único archivo con el mapa resultante después de la última medianoche, que para el segundo mapa sería `map002-100.txt`. Estos números negativos permiten al mago conocer si a largo plazo sus reglas producen una topografía razonable o si debe cambiarlas.

Note que el nombre del archivo de salida usa como prefijo el nombre del archivo mapa, seguido por un guión y el número de medianoche. Por ejemplo el archivo `map003.txt` representa el mapa de la isla en su estado actual, lo que equivale a la medianoche cero (`map003-0.txt`). El archivo `map003-20000.txt` representa esa misma isla 20,000 medianoches después (casi 55 años). Su programa podría partir de un mapa cualquiera y visualizar el efecto cierta cantidad de noches posteriores. Por ejemplo, si el mago quisiera conocer cómo se vería la isla 100 años después, podría escribir una orden de trabajo con cualquiera de las dos siguientes líneas

equivalentes:

```
map003.txt -36500
map003-20000.txt -16500
```

La orden de trabajo anterior muestra que una salida del programa puede ser usada como entrada para el mismo. En las dos líneas equivalentes de la orden anterior, naturalmente la segunda es más eficiente, y le permitiría al mago obtener los esperados resultados en menor tiempo.

En cuanto a los archivos de mapa, como `map001.txt` y `map003-20000.txt`, contienen la isla o un trozo de ella representado como una matriz de caracteres. En el primer renglón del archivo se indica la cantidad de filas y columnas de la matriz. Cada carácter en los renglones subsecuentes corresponden a una celda de la matriz (a excepción de los cambios de línea). Por ejemplo:

Contenido del archivo `map001.txt`:

```
7 7
-------
-l--l--
-ll----
-l-----
---laa-
-aa-al-
a-a----
```

Ejemplo de salida `map001-1.txt`:

```
-------
```

```
-------
-ll----
-------
----aa-
-aaaa--
aaaa---
```

Solución serial orientada a objetos [40%]

Se recomienda primero resolver el problema con una solución `serial` enfocada en corrección y utilizando el paradigma de programación orientada a objetos en `C++`. Debe realizarse el diseño acorde en `UML` con diagramas de clases y de interacción entre objetos. Los métodos algorítmicos que implican poca interacción entre objetos pueden diseñarse con pseudocódigo. Algunas ideas a considerar en su diseño:

1. Construir un bosque vacío con dimensiones de filas y columnas arbitrarias dadas por parámetro.
2. Saber la cantidad de filas y columnas que hay en el bosque.
3. Actualizar una celda del bosque en una medianoche. Considere un método privado que recibe por parámetros la fila y la columna que se quiere actualizar. El método actualiza la celda en medianoche de acuerdo a las reglas.
4. Actualizar todo el bosque en una medianoche.
5. Si el bosque es muy grande, permitir a dos o más trabajadores actualizar regiones distintas del bosque sin afectarse entre ellos, o trabajar en bosques distintos. Parte paralela.

Los diseños deben guardarse en la subcarpeta `design/` dentro de la carpeta para el proyecto02 en su repositorio de control de versiones. Exporte sus diseños a imágenes vectoriales (SVG) o escalares (PNG) e incrústelas en un

archivo `design/design.md` (o `design/README.md`). Si lo prefieren pueden usar otros formatos como AsciiDoc o LaTeX. En este documento escriban las anotaciones o explicaciones que ayuden a comprender la solución (los diagramas) y algoritmos (pseudocódigos), sobre todo para la fase de implementación o a futuros miembros del equipo. Además da un aspecto profesional al repositorio cuando se visita la carpeta desde el alojamiento de control de versiones.

Implementen la versión serial en C++ aplicando las buenas prácticas de programación y documentación. La aplicación valida las entradas, como números mal formados, fuera de rango, matrices incompletas o inválidas, son reportadas con mensajes de error, en lugar de caerse o producir resultados incorrectos. Asegúrense de que la aplicación pase los casos de prueba, y no genere diagnósticos del linter, sanitizers o memcheck, como se indica en la sección de "Aspectos transversales".

Optimización concurrente [40%]

Una vez construida su solución serial, realicen una paralelización que incremente el desempeño usando la tecnología OpenMP. Dado que la paralelización es una optimización, apliquen el método sugerido para optimizar visto durante las lecciones.

Paso 1. Se debe medir el rendimiento de la versión serial con un caso de prueba grande, siguiendo el procedimiento indicado en la Tarea03 para realizar mediciones de rendimiento en un nodo esclavo del clúster Arenal con al menos tres corridas. Anotar los resultados en una hoja de cálculo. Nota: si no se implementó una versión serial puede usar la versión paralelizada con un único hilo.

Paso 2. Hacer profiling con *Callgrind* para identificar las regiones de código

que más impactan el consumo de CPU.

Paso 3. Diseñar e implementar la paralelización con OpenMP. Discutan y reflejen la optimización pretetendida en los diseños realizados en la fase serial (pseudocódigo o diagramas). Implementar la paralelización con OpenMP en el código fuente. La cantidad de hilos debe poder ser controlada por argumento de línea de comandos. En caso de omitirse se debe suponer la cantidad de CPU disponibles en el equipo donde corre el programa.

Paso 4. Pasar las pruebas de corrección. La solución debe pasar los casos de prueba, y no tener malas prácticas como espera activa, condiciones de carrera, bloqueos mutuos, inanición, serialización innecesaria, o ineficiencia.

Paso 5. Medir el rendimiento de la versión paralelizada con los mismos métodos que el paso 1. Calcular el incremento de velocidad (*speedup*) y eficiencia. Cree un gráfico que incluya en el eje-x las tres soluciones (serial, concurrente), en el eje-y primario el incremento de velocidad, y en el eje-y secundario la eficiencia. Para la versión concurrente use tantos hilos como núcleos hayan disponibles en la máquina. Incruste su gráfico en una sección "Análisis de rendimiento" de su `README.md` con su correspondiente discusión corta (500 palabras máximo).

Conjeture para este problema cuál método de mapeo debería conseguir el mejor rendimiento. Repita los pasos 3, 4 y 5 con los métodos de mapeo disponibles en la tecnología de paralelización (bloque, cíclico, bloque cíclico, dinámico, y guiado), calcule los resultados y agréguelos al gráfico. Indique cuál de ellos obtuvo el mejor rendimiento y si confirma o rechaza su hipótesis.

Estructura del proyecto [20%]

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

Debe crear un archivo `README.md` en la raíz de la solución que indique el

propósito de la misma. Agregar una sección "Compilar" (en inglés, "Build") al `README.md` que explique cómo compilar y correr su código. Agregar en una sección "Manual de usuario" cómo ejecutar la aplicación, describir los parámetros, y describir la salida esperada. Conviene agregar salidas textuales o capturas de pantalla. Incluir una sección de diseño en el `README.md` con el modelo estático (diagrama de clases) y, a modo opcional, el modelo dinámico (diagrama de secuencia o actividad) de su solución.

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 programa contra los casos 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

Casos de prueba

test_set_1.zip

test_set_2.tar.gz