Programación Concurrente y de Tiempo Real
Grado en Ingeniería Informática
Examen Final de Prácticas
Febrero de 2018

Apellidos: Nombre:

Grupo (A ó B):

1. Notas

1. Escriba su nombre y apellidos con letra clara y legible en el espacio habilitado para ello.
2. Firme el documento en la esquina superior derecha de la primera hoja y escriba debajo su D.N.I.
3. Dispone de 3 horas para completar el ejercicio.
4. Puede utilizar el material bibliográfico (libros) y copia de API que estime convenientes. Se prohíbe el uso de apuntes, *pen-drives* y similares.
5. Entregue sus productos, utilizando la tarea de entrega disponible en el Campus Virtual, en un fichero (`.rar`, `.zip`) de nombre

   `Apellido1_Apellido_2_Nombre`

   que contendrá una subcarperta por cada enunciado del examen, la cual contendrá a su vez el conjunto de ficheros que den solución a ese enunciado en particular.

2. Criterios de Corrección

1. El examen práctico se calificará de cero a diez puntos y ponderará en la calificación final al 50 % bajos los supuestos recogidos en la ficha de la asignatura.
2. Las condiciones que una solución a un enunciado de examen práctico debe cumplir para considerarse correcta son:

a) Los ficheros subidos a través del Campus Virtual que conforman el examen práctico se ajustan al número, formato y nomenclatura de nombres explicitados por el profesor en el documento de examen.

b) El contenido de los ficheros es el especificado por el profesor en el documento de examen en orden a solucionar el enunciado en cuestión.

c) Los programas elaborados por el alumno, se pueden abrir y procesar con el compilador del lenguaje, y realizan un procesamiento técnicamente correcto, según el enunciado de que se trate.

d) Se entiende por un procesamiento técnicamente correcto a aquél código de programa que compila correctamente sin errores, cuya semántica dá soporte a la solución pedida, y que ha sido escrito de acuerdo a las convenciones de estilo y eficiencia habituales en programación

3. Enunciados

1. (El Mundo de *WaterWorld*, 3.4 puntos) *WaterWorld* es un planeta reticulado de topología toroidal, donde habitan dos especies: rorcuales aliblancos y dragones marinos. El planeta se modela con una retícula de 1000×2000 nodos. Los rorcuales pueden vivir hasta 10 generaciones y los dragones hasta 20. Se entiende por una generación al proceso de obtener el nuevo estado de la retícula a partir del estado anterior. Cada nodo puede estar vacío, o contener un ejemplar de una de las dos especies. Las poblaciones de ambas especies siguen una dinámica predador-presa descrita por las clásicas ecuaciones de *Lotka-Volterra*, que en modo discreto podemos escribir mediante las reglas siguientes, que indican el cambio de estado para cada nodo de la retícula:

   1. Si el nodo actual está vacío entonces: si hay cuatro o más vecinos de la misma especie, tres o más de ellos están en edad de reproducción (los rorcuales se reproducen a partir de las dos generaciones y los dragones a partir de las tres) y hay cuatro o menos vecinos de la otra especie, crear en el nodo actual un nuevo ejemplar de la especie reproductora, con una generación de edad

   2. Si el nodo actual contiene un rorcual que ya tiene diez generaciones de edad, muere de vejez; si el nodo actual contiene un rorcual y está rodeado por cinco o más dragones, el rorcual es comido por ellos y muere; si sus ocho vecinos son también rorcuales, también muere por sobrepoblación; en otro caso, el rorcual incrementa su edad en una generación.

   3. Si el nodo actual contiene un dragón que ya tiene veinte generaciones de edad, muere de vejez; si el nodo actual contiene un dragón y está rodeado por seis o más dragones, no teniendo ningún rorcual como vecino, el dragón muere de hambre; un dragón puede morir por causas aleatorias con una probabilidad $p = 0{,}3$; en otro caso, el dragón incrementa su edad en una generación.

   Se pide modelar el mundo de *WaterWorld* utilizando procesamiento paralelo en lenguaje Java y de acuerdo a las siguientes restricciones:

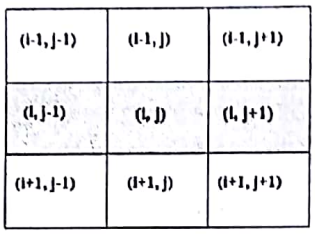

Figura 1: Vecindad de Moore

- El programa efectuará un particionamiento del dominio de datos de manera automatizada, como función del número de *cores* lógicos disponibles en la máquina. Para lograr una ejecución paralela más eficaz, las matrices de lectura y escritura de datos serán diferentes. La matriz de datos original debe inicializarse, también de forma paralela, mediante una carga de estados aleatoria tal que el 50 % de las células contengan un rorcual de una generación de edad, el 25 % contenga un dragón de una generación de edad, y el 25 % restante esté vacía. Cada tarea recibirá información del segmento de datos que debe procesar a través del constructor de clase. Se utilizarán **condiciones de frontera cilíndrica**, para dotar al mundo de topología toroidal. La vecindad de un nodo de la retícula será la de *Moore* (Figura 1).
- Debe utilizar tareas `Runnable` para modelar sus hebras de ejecución, y procesarlas mediante un ejecutor de clase `ThreadPoolExecutor`.
- El programa debe sincronizar a las hebras paralelas mediante un objeto de clase `CyclicBarrier`.
- Si lo necesita, puede ampliar el tamaño del *heap* de la máquina virtual utilizando el *flag* `-Xmx`
- Todo su código estará en un fichero llamado `WaterWorld.java`.
- Dote al código anterior de un corto menú de dos opciones: la primera ejecutará de forma automática durante 1000 pasos de tiempo el modelo, indicando el tiempo total requerido y el *speedup* logrado para un número de tareas paralelas igual al número de *cores* y al doble de ese número; la segunda permitirá al usuario fijar el tamaño de la retícula, inicializarla manualmente, y determinar el número de iteraciones del modelo, que serán procesadas en paralelo. En este caso, la salida será el resultado de imprimir las retículas inicial y final, sin ofrecer ningún dato adicional, en modo texto. Para ello representará los rorcuales mediante cruces (x) y los dragones mediante ceros (0).

AYUDA: Una forma de modelar las generaciones de edad es tomar en la retícula números positivos para los rorcuales y números negativos para los dragones, reservando el cero para indicar un punto vacío.

2. (Línea de Cajas, 3.3 puntos) Una sucursal bancaria dispone de diez puestos de atención personalizada al público. Cuando un cliente llega, va a uno de los puestos libres. Si no hay ninguno, debe esperar hasta que lo haya en una única cola que la sucursal dispone a tal efecto. Se pide escribir un monitor utilizando el API de alto nivel para concurrencia de Java, que modele la situación propuesta, implantando la sincronización requerida. Guárdelo en un fichero `lineaCajas.java`. Modele a los clientes mediante tareas que implementen a la interfaz `Runnable` y guarde el código en un fichero `cliente.java`. Ponga todo a funcionar con un programa que procese a las tareas mediante un objeto de clase `ThreadPoolExecutor`, que guardará en el fichero `usaLineaCajas.java`.

3. (Producto Escalar Concurrente a Futuro, 3.3 puntos) Se desea efectuar el producto escalar de dos vectores de 10^4 componentes de tipo `float` mediante cuatro tareas concurrentes utilizando ejecución asíncrona a futuro. Para ello, debe escribir un programa que soporte las tareas mediante implementación de la interfaz `Callable`, recogiendo los resultados de los productos parciales realizados y devuelto por el método `call()` por cada tarea mediante la interfaz `Future`. Una vez disponibles los cuatro futuros, debe sumarlos para obtener el resultado final, que imprimirá en pantalla. Efectúe el particionamiento de datos de forma manual (por programa) y procese sus tareas a través de un ejecutor `FixedThreadPool`. Guarde todo el código en `prodEscalar.java`.

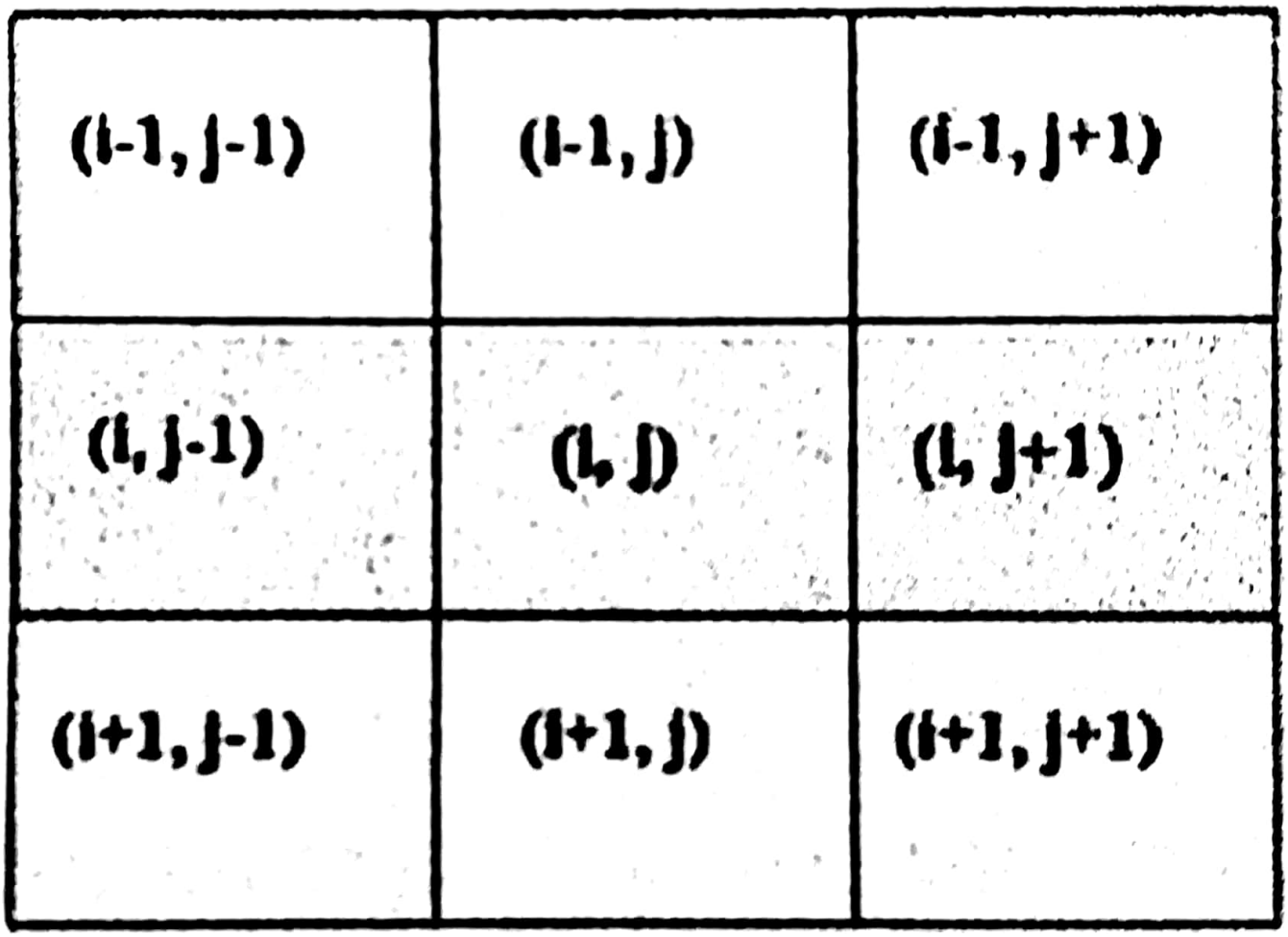

Figura 1: Vecindad de Moore