1) El problema es que el modelo de memoria de OpenMP, de consistencia relajada no garantiza que cuando una hebra cambia el valor de una memoria compartida, otra hebra vea exactamente este valor. Específicamente, cuando la hebra 0 asigna flag[0] = true, esto lo hace en su visión temporal de la memoria global y es posible que la hebra 1, vea el otro valor, es decir flag[0] = false. Lo mismo sucede con la variable turn. Para solucionar este problema es necesario "flushear" las visiones locales a la vision global, tanto de subida como de bajada.

El algoritmo en sí es perfecto y no tiene probemas de concurrencia, algoritmicamente. El problema surge porque OpenMP maneja un modelo de memoria que no garantiza que todas vean los mismos valores al mismo tiempo. En este sentido, no es necesario "sincronizar" a las hebras, con un barrier. Un barrier induce pérdida de concurrencia y modifica sustancialmente el algoritmo

2) Al ejecutar 4 procesos uno de ellos se demora  un 20% más, significa que ese 20% más es ejecutado por sólo una hebra. Esto es obvio pero importante pues también quiere decir que durante un cierto tiempo no existió paralelismo. Aunque la parte secuencial del programa es 0%, la carga paralela no está balanceada y hay un porcentaje del código paralelo que se ejecuta secuencialmente. Toda la carga de trabajo del programa

es W = 4.2T. donde durante 4T unidades de tiempo se estuvieron usando 4 hebras y durante 0.2T 1 sola hebra.

$S = (0.2T + 4T) / (0.2T + 4T/4)$

$S = 4.2/1.2$

$S = 3.5$

Fíjese que el valor tiene mucho sentido

3) El objeto generador de números aleatorios DEBE ser una corutina con exclusión mutua o una corutina monitor

A continuación pondré los elementos más relevantes en la evaluación

```
_Mutex Corutine LCG {     <-- muy importante
        int x;
        int a;
public:
       LCG(int a, int x0) : a(a), x(x0) {}  <- no muy mportante
       int next();
protected main() {
        for (;;) {
            x = mod( a*x, 4294967295);
            suspend();   <- very important
        }
}

int next() {
        resume();   <- very
        return x;     <- important
}
```

4) En esta pregunta alguno de ustedes confundión SIMD con SSE

SIMD es el paradigma que dice que un mismo flujo de instrucciones opera sobre flujo de datos distintos.

TODOS implementan este paradigma. SSE, OpenMP y uC++, pues en todos es posible construir un programa donde una misma instrucción opera sobre datos distintos. La diferencia principal radica en el tiempo/espacio en donde esto ocurre. En el caso de SSE, la misma operación numérica opera sobre

pares de datos distintos almacenados en un mismo registro, en el mismo ciclo de reloj. En el caso de OpenMP, la misma instrucción es ejecutada por hebras distintas sobre datos almacenados en memoria compartida, pero que pueden ejecutarse en tiempos distintos.

Lo mismo ocurre con uC++. La gran diferencia entre uC++ y OpenMP es que uC++ introduce un nivel de abstracción más elevado (no de más bajo nivel!!!), pues organiza la solución en función de clases (y objetos). En este sentido las Task de OpenMP son muy parecidas a las tareas de uC++. uC++ facilita la construcción de grandes sistemas concurrentes donde las actividades son variadas, como por ejemplo en un sistema basado en servicios, aunque esto lo alejaría de el SIMD. En cambio, OpenMP está orientado a soluciones donde hebras independientes puedan cooperar sobre datos compartidos, pero no necesariamente ejecutarse al mismo tiempo.