Kristen Brandt
171482

Lab 3: Multithreading

1. ¿Qué es una *race condition* y por qué hay que evitarlas?

Un race condition es cuando hay dos o más threads que tienen acceso a datos compartidos y lo tratan de cambiar al mismo tiempo. Hay que evitarlas porque cuando un thread hace check-then-act puede ser que el otro thread cambie algo entre el checking y el act. Esto puede crear problemas con los datos.

2. ¿Cuál es la relación, en Linux, entre `pthreads` y `clone()`? ¿Hay diferencia al crear *threads* con uno o con otro? ¿Qué es más recomendable?

Clone() es una llamada al sistema que puede ser utilizada para crear procesos o threads. Pthreads sirven para el multithreading, estos permiten que un programa pueda controlar varios flujos de trabajo que hacen overlap en el tiempo. Es más recomendable pthreads porque con clone no se puede usar en otros sistemas. Clone también puede ocupar más memoria.

3. ¿Dónde, en su programa, hay paralelización de tareas, y dónde de datos?

En nuestro programa cuando se hacen los forks() se da la paralelización de tareas. La paralelización de datos en nuestro programa es cuando se revisa que no se estén repitiendo números dentro de las filas, columnas o cuadrados.

4. Al agregar los `#pragmas` a los ciclos `for`, ¿cuántos LWP's hay abiertos antes de terminar el `main()` y cuántos durante la revisión de columnas? ¿Cuántos *user threads* deben haber abiertos en cada caso, entonces? ***Hint*:** recuerde el modelo de *multithreading* que usan Linux y Windows.

Antes de terminar el main() hay 1 LWP abierto. En la revisión de columnas cuando se utiliza #pragma omp parallel se le dice que esa parte del código se haga en paralelo. El modelo de multithreading que utiliza Linux es de uno a uno. Por lo tanto si hay un 1 LWP abierto en el main hay 1 user thread por el modelo de multithreading utilizado.

5. Al limitar el número de *threads* en `main()` a uno, ¿cuántos LWP's hay abiertos durante la revisión de columnas? Compare esto con el número de LWP's abiertos antes de limitar el número de *threads* en `main()`. ¿Cuántos *threads* (en general) crea OpenMP por defecto?

Al limitar el número de threads en main() a uno hay un LWP abierto durante la revisión de columnas. En general OpenMP utiliza un hilo para las secciones secuenciales y varios para las secciones paralelas. Por defecto siempre está el master thread. Cuando una parte del código se va a ejecutar en paralelo y esta con #pragma esto forma subprocesos llamados esclavos. En ambos casos hay 1 thread.

6. Observe cuáles LWP's están abiertos durante la revisión de columnas según `ps`. ¿Qué significa la primera columna de resultados de este comando? ¿Cuál es el LWP que está inactivo y por qué está inactivo? ***Hint***: consulte las páginas del manual sobre `ps`.

Los LWP que están abiertos durante la revisión de columnas según ps es F. Esto nos indica que los threads están activos mediante 1 y 0. El LWP que está inactivo es el master thread, este esta inactivos porque los threads esclavos son los que se encuentran activos. Cuando se terminan los esclavos se unen al master. Y cuando todos estos terminan el master sigue con el código.

7. Compare los resultados de `ps` en la pregunta anterior con los que son desplegados por la función de revisión de columnas *per se*. ¿Qué es un *thread team* en OpenMP y cuál es el *master thread* en este caso? ¿Por qué parece haber un *thread* "corriendo", pero que no está haciendo nada? ¿Qué significa el término *busy-wait*? ¿Cómo maneja OpenMP su *thread pool*?

Un thread team en OpenMP es cuando se crea una colección de equipos de subprocesos. El master thread es el que ejecuta la región de los equipos. Parece haber un thread corriendo pero no está haciendo nada porque este es el master thread. El término busy-wait es una técnica que se utiliza para verificar repetidamente si una condición se cumple. Esto puede servir para generar un tipo de wait. Esto sería cuando los sistemas no tenían un wait. OpenMP maneja el thread pool sin especificar equipos, se crea un equipo para una región y el master thread lo ejecuta.

8. Luego de agregar por primera vez la cláusula `schedule(dynamic)` y ejecutar su programa repetidas veces, ¿cuál es el máximo número de *threads* trabajando según la función de revisión de columnas? Al comparar este número con la cantidad de LWP's que se creaban antes de agregar `schedule()`, ¿qué deduce sobre la distribución de trabajo que OpenMP hace por defecto?

El máximo número de threads trabajando según la función de revisión de columnas es el mismo que el número de LWP antes de agregar schedule(). Lo que se puede deducir sobre la distribución de trabajo que hace OpenMP por defecto es que la distribución de tareas/trabajo es bastante equitativa.

9. Luego de agregar las llamadas `omp_set_num_threads()` a cada función donde se usa OpenMP y probar su programa, antes de agregar `omp_set_nested(true)`, ¿hay más o menos concurrencia en su programa? ¿Es esto sinónimo de un mejor desempeño? Explique.

Antes de agregar omp_set_nested(true) hay menos concurrencia en el programa. La concurrencia es la capacidad que tiene el CPU de ejecutar más de un proceso al mismo tiempo. Al agregarlo como se habilita el paralelismo anidado hay más threads. No es sinónimo de un mejor desempeño. Cuando hay concurrencia los procesos que están siendo ejecutados no tienen que estar relacionados, pueden iniciar y terminar cuando sea.

10. ¿Cuál es el efecto de agregar `omp_set_nested(true)`? Explique.

El efecto de agregar omp_set_nested(true) es habilitar el paralelismo anidado. Este establece este tipo de paralelismo al establecer el número de niveles activos de paralelismo que admite la implementación.