
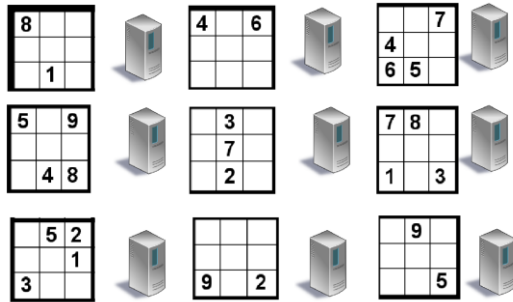


Laboratorio 5

 Sistema	
Memoria base:	768 MB
Procesadores:	4
Orden de arranque:	Disquete, Óptica, Disco duro
Aceleración:	VT-x/AMD-V, Paginación anidada, Paravirtualización KVM

- **¿Qué es una *race condition* y por qué hay que evitarlas?**
Se produce una condición de carrera cuando dos o mas threads pueden acceder a los datos compartidos e intentan cambiarlos al mismo tiempo. Debido a que el algoritmo de programación de threads puede intercambiar entre threads en cualquier momento, no sabe el orden en el que los subprocesos intentaran acceder a los datos compartidos. Por lo tanto, el resultado del cambio en los datos depende del algoritmo de programación de threads, es decir ambos threads están compitiendo por acceder.
- **¿Cuál es la relación entre `pthread` y `clone()`? ¿Hay diferencia al crear *threads* con uno o con otro? ¿Qué es más recomendable?**
Clone es una llamada al sistema de bajo nivel especifica de Linux y se puede utilizar para crear procesos y threads. El fork se utiliza para crear nuevos procesos . pthread se utiliza para multithreading. La mayor diferencia entre los threads y los procesos es que los threads comparten un solo espacio de memoria donde los procesos tienen cada uno su propio espacio de memoria.
En una clase que tiene un objeto thread, hay que determinar los objetos ejecutables, los cuales les indican al sistema si el thread esta listo para ejecutarse.
- **¿Dónde, en su programa, hay paralelización de tareas, y dónde de datos?**
Se paralelizo el programa por medio de OpenMP. En donde se puso “#pragma omp parallel for” justo después de poner for.
El paralelismo de tareas es un paradigma de la programación concurrente que consiste en asignar tareas a cada uno de los procesadores de un sistema de cómputo. Por lo que cada procesador efectúa su propia secuencia de operaciones.
La mejor forma de mostrar la paralelización en un sudoku es la siguiente:



- 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 usa Linux.

Se utilizan 11 threads y 9059 LWP.

- 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* crea OpenMP (en general) por defecto?

Depende de cuantos cores se están ejecutando. N cores debe ser igual a n threads.

Entonces el número de threads que pueden ejecutarse simultáneamente es: `total_threads = num_procs * factor de hyperthreading`.

- 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?

- Compare los resultados de `ps` en la pregunta anterior con los que son desplegados por el método 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*?

Omp teams crea una colección de thread team. Para una target region sin especificar los teams, un solo equipo es creado y su master thread ejecuta el target region. El limite es 002 threads. Hay un thread que se ejecuta desde el principio hasta el final, y ese es el master thread. Las secciones paralelas del programa harán que los threads adicionales hagan fork. Por lo que crean threads esclavos. Aparece un Thread corriendo sin hacer nada debido a que es un thread esclavo. Busy- waiting es cuando un proceso verifica repetidamente una condición, esta "esperando" la condición, pero esta "ocupado" comprobándolo. Esto hará que el proceso consuma CPU.

OpenMp es estrictamente un modelo de thread de fork/join. En algunas implementaciones los threads se crean al inicio de una region paralela y se destruyen al final de la region paralela. Las aplicaciones de OpenMp normalmente tienen varias regiones paralelas con regiones seriales intermedias.

La creación y destrucción de threads para cada región paralela puede resultar en overhead especialmente si la region paralela esta adentro de un loop, por lo tanto OpenMp utiliza thread pools. El pool de worker threads es creado en la primera region paralela. Estos threads existen por la duración de la ejecución de un programa. Mas threads se pueden

agregar si lo pide el programa. Los threads no se destruyen hasta que se ejecute la ultima región paralela.

- **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 el método 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?**
- **Luego de agregar las llamadas `omp_set_num_threads()` a cada método donde se usa OpenMP, y luego de ejecutar su programa con y sin la cláusula `schedule()` en cada `for`, ¿hay más o menos concurrencia en su programa? ¿Es esto sinónimo de un mejor desempeño? Explique.**
- **¿Cuál es el efecto de agregar `omp_set_nested(true)`? Explique.**
Habilita regiones paralelas anidadas, así los miembros del equipo pueden crear nuevos equipos. La función toma el equivalente específico del lenguaje verdadero y falso, donde verdadero habilita el ajuste dinámico de los tamaño del equipo y deshabilita el falso.