Laboratorio #3

Fecha de Entrega: 23 de marzo, 2022.

Descripción: en este laboratorio se empleará *multithreading* por medio de pthreads y OpenMP para desarrollar un verificador de soluciones para *sudokus* de nueve por nueve. Los entregables serán todo el código escrito, así como un documento que responda a las preguntas planteadas al final. Se recomienda auxiliar sus respuestas con *screenshots* de la ejecución de su programa.

Materiales: una máquina virtual con Linux o Windows, pero que tenga GCC, y documentación sobre OpenMP.

Contenido:

OpenMP busca paralelizar (no sólo ejecutar de forma concurrente), por lo que su funcionamiento requiere de más de un procesador. Si usará una máquina virtual, comience por asegurarse de que su máquina cuente con cuatro procesadores (en VirtualBox, ventana *Settings*, menú *System*, *tab Processor*). La cantidad se especifica para uniformizar las respuestas a las preguntas que se plantean al final.

Cree un programa en C llamado SudokuValidator.c. En él escriba tres funciones que se encarguen de revisar que todos los números del uno al nueve estén:

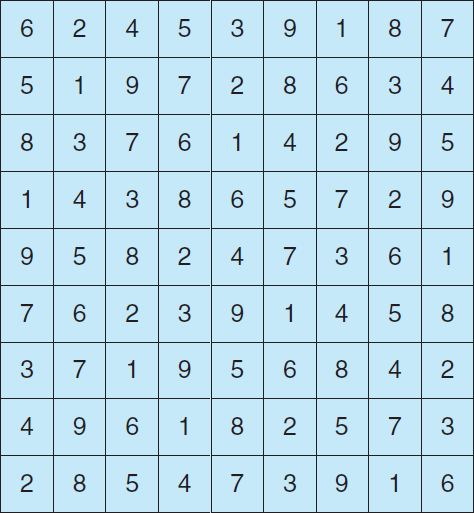
* En cada columna de un arreglo de nueve por nueve.
* En cada fila de un arreglo de nueve por nueve.
* En un subarreglo de tres por tres dentro de un arreglo de nueve por nueve.

Las funciones para verificación de filas y columnas serán iguales exceptuando un intercambio de índices al recorrer el arreglo. La función de revisión de subarreglos debe recibir una fila y una columna para ubicar la esquina superior izquierda de un cuadrado de tres por tres, donde iniciará la revisión dentro del arreglo de nueve por nueve. Todas estas funciones se deben basar en ciclos for obligatoriamente.

Este programa recibirá, en terminal, la ubicación de un archivo (sólo el nombre, si está en el mismo directorio que SudokuValidator.c) que contiene una solución a un *sudoku* de nueve por nueve. El formato de las soluciones debe ser un único *string* de ochenta y un dígitos, en la primera línea, comenzando por la celda superior izquierda del *sudoku* y avanzando de izquierda a derecha, por filas. Para este laboratorio se provee una solución de ejemplo en el archivo “*sudoku”*.

Lo primero que su main() deberá hacer es abrir el archivo usando open() y *mappearlo* a su memoria usando mmap(). Luego debe ejecutar un for en el que se copie cada símbolo del *string* en el archivo de solución a un arreglo bidimensional de nueve por nueve, de modo que le quede una grilla lógica como la que se muestra en la página siguiente. Se recomienda que su arreglo bidimensional sea global (es decir, que esté declarado fuera del main()) para que sea accesible por varios *threads*. Luego de llenar la grilla, escriba un for que haga la revisión, con su función, de los subarreglos de tres por tres que conforman el arreglo de nueve por nueve (**nota**:

revise los subarreglos de tres por tres cuya primera posición (si comenzamos desde 1) sea [𝑖, 𝑖] para 𝑖 ∈ {1,4,7}).



*Grilla lógica ejemplar*

Luego de lo anterior, obtenga el número de proceso (no el de *thread*) y ejecute un fork(). En el proceso hijo convierta el número del proceso padre (no el de *thread*) a texto, y ejecute por medio de execlp() el siguiente comando:

ps –p <#proc> -lLf

donde <#proc> es el número del proceso padre. Este comando permite ver información relacionada al proceso <#proc> que incluye los *lightweight processes* que tenga asociados.

En el proceso padre:

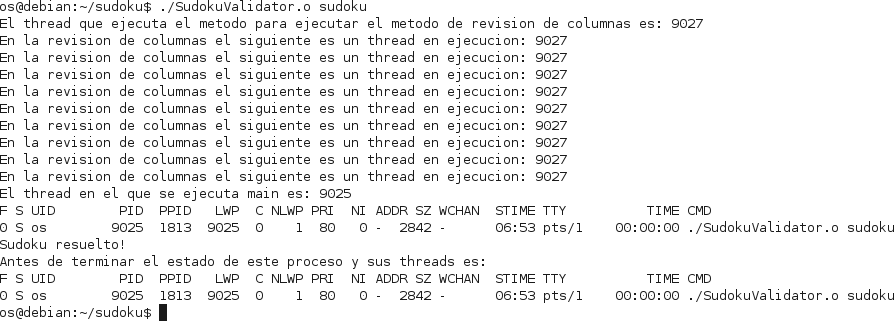
* Cree un pthread que haga su revisión de columnas.
* Ejecute pthread\_join() y luego despliegue el número de *thread* en ejecución. Para lograrlo debe #incluir <sys/syscall.h> en su programa y ejecutar syscall(SYS\_gettid) (el resultado de esta llamada de sistema es el *id* del *thread*).
* Espere a que concluya el hijo que está ejecutando ps.
* Realice su revisión de filas.
* Despliegue si la solución al *sudoku* es válida o no.
* Ejecute un nuevo fork() y ejecute el comando ps en el proceso hijo, tal como se describe en instrucciones anteriores. Esto servirá para comparar el número de LWP’s asociados al proceso

padre cuando se está realizando la revisión de columnas y cuando (el padre) está a punto de terminar.

* Espere al hijo y retorne 0.

Observe que la creación de un *thread* que ejecute la revisión de columnas implica la creación de una función que sea asignable a un *thread* en el cual, a su vez, se ejecute su función de revisión de columnas. Es decir, una función que tenga tipo de retorno void\* y que termine con pthreads\_exit(0). En esa función tipo void\* también despliegue el número de *thread* en ejecución.

El siguiente es un ejemplo de cómo podría verse el *output* de su programa hasta este momento:

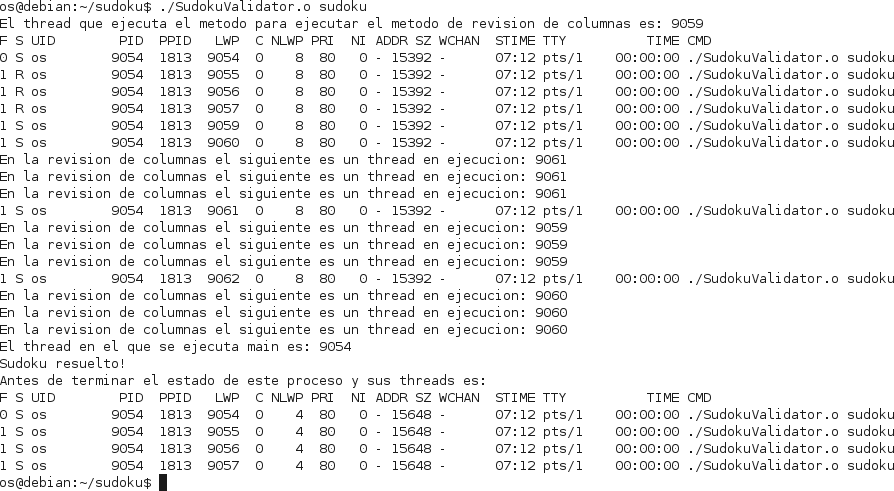


Como siguiente paso deberá paralelizar todos los ciclos for que pueda (vea la nota **importante**) usando OpenMP. Para ello simplemente es necesario que la siguiente línea preceda inmediatamente a la del for en cada caso:

#pragma omp parallel for

**Importante**: evite las *race conditions*. Investigue el uso de la directiva private de OpenMP para auxiliarse en este aspecto. No todos los ciclos for deberán ser precedidos por la directiva.

Ejecutar su programa ahora deberá resultar en un *output* similar al siguiente:



Anote el número de LWP’s que se tienen durante la revisión de columnas y antes de terminar el programa. Agregue la siguiente instrucción al principio de main():

omp\_set\_num\_threads(1);

Ejecute su programa y note el resultado de las ejecuciones de ps. También anote los números de *thread* desplegados durante la revisión de columnas.Ahora, agregue la siguiente directiva a todas las líneas #pragma que incluyó anteriormente:

schedule(dynamic)

Ejecute su programa varias veces y observe los números de *thread* que se despliegan durante la revisión de columnas. Compárelos con el resultado de ps que se despliega durante la ejecución del pthread y anote sus observaciones.

Como siguiente paso, agregue una llamada a omp\_set\_num\_threads() al inicio de cada función donde se ejecute un for paralelo, determinando el número de *threads* adecuados (*e.g.*, si su función ejecuta un for paralelo de nueve iteraciones, posiblemente el número de *threads* deba ser nueve). Ejecute su programa varias veces y anote los efectos sobre los *threads* en los resultados de ps. Repita el procedimiento comentando la cláusula schedule() en el primer for paralelo de su revisión de columnas. Finalmente agregue la siguiente instrucción al principio de cada función que use OpenMP:

omp\_set\_nested(true);

Ejecute su programa y anote los efectos sobre el resulta

Responda las siguientes preguntas:

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

*Races conditions* o condiciones de carrera ocurren cuando dos más threads tienen acceso a una variable compartida al mismo tiempo, entonces los threads acceden al valor de la variable, y luego de una serie de operaciones la modifican. Generando una “competencia “ entre los threads para ver quien modifica al final la variable compartida. Básicamente ambos threads están compitiendo entre accesar y modificar la información (Yu, 2022). Un ejemplo de esto sucedió durante el laboratorio, ya que para hacer la validación por filas o columnas se utilizó que la suma de todos los números en dicha fila (o columna) debe de ser 45. La condición de carrera ocurría al momento de hacer la validación de la suma, como la variable que almacenaba la suma acumulada era compartida entre todos los threads mientras un thread estaba validando que la suma fuera igual a 45 otro thread en ese mismo instante podía estar modificando el valor de la variable, ocasionando que el resultado de la validación fuera falso. Es por ello, que en ocasiones cuando se ejecutaba el programa este retornaba que la solución era invalida y en otras que era válida.

La principal razón por la cual es necesario evitar las condiciones de carrera es por la incoherencia de información, además de la inconsistencia de los datos. Ya que como se pudo leer en el ejemplo anterior, aunque el algoritmo sea correcto, sin embargo, por las condiciones de carrera se pueden llegar a resultados incorrectos o inesperados. Ya que la información de las variables que se comparten no es consistente entre threads sino más bien el último thread que logra acceder/modificar la variable puede modificar el valor asignado a la variable por el thread anterior.

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

Tanto pthreads como clone permiten la creación de nuevos threads, sin embargo, crear un thread con clone, en términos del contexto y recursos de thread, no es igual a crearlo con pthread, principalmente esta diferencia ocurre ya que pthread realiza más configuraciones al momento de crear los threads que permiten que utilizar los threads sea más fácil.

Como mencioné anteriormente, la principal diferencia es el proceso de configuración, al menos el realizado por el usuario, en el caso de pthread únicamente debe de llamar a una función que se encargue de la creación del thread, sin embargo, en el caso de clone, es necesario crear un *stack* para almacenar la información del thread, y para ello también es necesario asignar memoria para el *stack*, esto comúnmente se hace mediante *mmap*. Es importante recalcar que existen varias formas de crear el thread mediante clone (i.e. mediante libc o syscall). Como menciona Obrien en *Process, Threads and the Clone Syscall* algo interesante acerca de pthread (lo cual no se puede realizar con clone, o al menos no en ese artículo) es que al momento de crear un thread mediante pthread, printf es *threadsafe* es decir, al momento de “imprimir” algo en pantalla no ocasionará resultados inesperados.

Qué opción se debe de utilizar, depende del contexto en el cual se utilizarán los threads, sin embargo, a manera de uso general, se puede decir que, debido a su portabilidad, al conjunto de configuraciones ya preestablecidas con las que cuenta y a la facilidad de implementación (por parte del usuario) pthread es más recomendable. (Starynkevitch, 2012)

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

Como bien se sabe la paralelización de tareas se refiere a asignar distintas tareas a cada uno de los procesadores existentes en el sistema, un ejemplo de esto dentro del programa es al ejecutar en paralelo la validación de subarreglos y la validación de columnas. (colaboradores de Wikipedia, 2019)

Por otro lado, la paralelización de datos ocurre cuando se subdivide un conjunto de datos a manera que cada uno de los procesadores o cores ejecute una parte de esos datos. Esto en el programa se puede observar al hacer la validación de columnas, ya que la paralelización no se realiza cuando se manda a llamar la función que hace la validación, sino dentro de la validación en sí, es decir al momento de ir validando columna por columna, y cada columna es validada por un thread distinto. (colaboradores de Wikipedia, 2021)

1. **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.**

Text

Description automatically generated

**Figura 1.** Ejecución del programa al agregar #pragma a los ciclos for

Como se puede observar en la **figura 1** la cantidad de LWP’s que se tienen antes de terminar el main es de 4 y durante la revisión de columnas se tienen 8.

Como bien se sabe, el modelo de multithreading que utiliza Linux es el de uno:uno, es decir, un thread de kernel por thread de usuario. Por lo tal, se tendrían 8 threads en la revisión de columnas y 4 threads de usuario antes de terminar el main.

1. **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?**

Graphical user interface, text

Description automatically generated**Figura 2.** Ejecución del programa de validación de sudoku al limitar el número de threads a 1

Como se puede observar al limitar el número de threads en main() a uno ahora únicamente se tienen 5 LWP´s durante la revisión de columnas, a diferencia de antes cuando se tenían 8 threads. En general, la cantidad de threads que crea OpenMP será igual a la cantidad de cores con los que se cuenta, en este caso, la cantidad de cores en la máquina virtual son 4. Por lo tal, se puede notar que al especificar la cantidad de threads a 1 en main en el primer #pragma que se realiza, ya no se crean los 3 threads restantes de antes, para completar los 4 sino más bien solo se queda con el que esté en ejecución. Sin embargo, al momento de crear el otro thread con Pthreads y llamar a la función de revisión por columnas, debido a que este nuevo thread no tiene definido explícitamente la cantidad de threads que debe de crear, se basa en la cantidad de procesadores o cores. De esta forma, la cantidad de threads que realiza la revisión por columnas son 4 (de aquí que la cantidad de threads abiertos durante la revisión de columnas sea 5) sin embargo, al terminar dicha revisión, estos threads terminan y únicamente queda el thread original (es por ello que al terminar de ejecutar el main ya solo se muestra un thread).

1. **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.**

Graphical user interface, text

Description automatically generated

**Figura 3.** LWP´s abiertos durante la revisión de columnas según ps

La primera columna de ps es el *header*, es decir indica qué muestra cada columna devuelta por *ps,* cada uno de los headers significa lo siguiente:

* + **S**: es el estado del proceso o thread
  + **UID**:ID del usuario
  + **PID**: ID del proceso
  + **PPID**: ID del proceso padre
  + **LWP**: ID del thread
  + **C**: uso del procesador
  + **NLWP**: números de threads
  + **PRI**: prioridad del proceso. A más grande el número significa menor prioridad
  + **NI**:valor *nice*
  + **ADDR**:
  + **SZ**: tamaño, en páginas físicas de la imagen del proceso
  + **WCHAN**: dirección de la función kernel en la cual el proceso/thread está esperando. Las tareas en ejecución muestran un *dash* (-).
  + **STIME**: tiempo de inicio
  + **TTY**: controlando terminal
  + **TIME**: tiempo acumulado de CPU
  + **CMD**: nombre de comando

En la **figura 3** especialmente en el resultado de ejecutar *ps* al hacer la validación por columnas se puede notar que el thread con PID 3055 se encuentra en estado S, el cual significa que está esperando a que un evento se complete. Es importante destacar que este es el thread en el cual se ejecuta main, por lo tal, este es el thread “original” o el thread que está asociado al proceso padre en el instante que se inicia la ejecución, y básicamente este proceso se encuentra “inactivo” ya que está esperando a que el proceso hijo finalice su ejecución.

1. **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*?**

Text

Description automatically generated with medium confidence

**Figura 4.** Ejecución del programa para validar solución a un sudoku.

En OpenMP un thread team hace referencia a un grupo de threads que están ejecutando un programa (Yliluoma, 2016). Cuando el programa inicia, el *team* consiste básicamente en un solo thread que se conocer como *master thread* y es el *thread* que se ejecuta desde el inicio hasta el final del programa, por otro lado, existen los *slave threads* los cuales son los que trabajan en las secciones paralelas. En este caso, el *master thread* será aquél que está asociado al proceso desde el inicio de la ejecución del programa hasta el final (es como el *thread* padre) para el caso particular de la **figura 4** sería el *thread* con ID 4578. (*Intro to Parallel Programming with OpenMP*, s. f.)

En la **figura 4** se puede notar que existe un thread que si bien su estado es *running* sin embargo se encuentra bloqueado o sleeping (esto se puede ver en que WCHAN tiene valor de futex\_\_). Esto puede estar sucediendo a causa de que el trhead está esperando que otro thread libere un recurso compartido dentro del *for* de validación por columnas.

Diagram

Description automatically generated

**Figura 5.** Ejemplo gráfico de *busy-waiting*. (Baeldung, 2021)

El término *busy-waiting* es una técnica de sincronización de procesos, en la cual el proceso o thread espera y constantemente valida una condición que debe de satisfacerse antes de su ejecución. En la “espera en loop” un proceso/thread ejecuta una instrucción que verifica sin la entrada a la condición se satisface para poder salir del “area de espera”, como se puede observar en la **figura 5** el proceso 2 quiere acceder al recurso compartido, sin embargo, debido a que el recurso compartido está siendo usado por el proceso 1, el proceso 2 debe de hacer *busy-waiting*. Algo importante a recalcar en el caso de *busy-waiting* es que debido a que está constantemente validando si la condición se satisface o no eso implica que estará haciendo uso del procesador. (Baeldung, 2021)

Un *thread-pool* es un conjunto de threads anónimos inactivos que están listos para ejecutar una tarea, son threads que pueden ser usados como *slaves* en regiones paralelas. Entonces cuando se ejecuta un bloque de código de forma paralela y se debe de crear un *team* de más de un *thread* entonces se revisa el *pool* y se toman aquellos *threads*  que estén inactivados (Oracle, s. f.). Su uso es preferible donde se tiene un gran número de tareas cortas ya que así se evita el *overhead* de crear threads, una gran cantidad de veces (Blrfl, 2012). En el caso de OpenMP este tiene los threads y les va asignando tareas a ejecutar, cuando los threads han finalizado de ejecutar la tarea, este regresar a un *“dock waiting”* a esperar que sea llamado a ejecutar otra tarea. (Yliluoma, 2016)

1. **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?**

Graphical user interface, text

Description automatically generated

**Figura 6.** Ejecución del programa al utilizar *Schedule(dynamic)*

Con base en las ejecuciones realizadas y según la función de revisión de columnas la cantidad de threads que están trabajando son tres. La cantidad de threads que se crean, según la **figura 6** y **figura 5** tanto antes como después de utilizar *Schedule(dynamic)* se crean 5 threads. Sin embargo, en la **figura 5** en la validación de las columnas se puede notar que existen 4 threads activos.

Schedule(dynamic) según la definición de dinámico es que hay algo que escoge en cada ejecución, que no es algo estático o definido. En las diversas ejecuciones realizadas en ocasiones se observaba como la mayoría de las iteraciones era ejecutada por un solo thread, u 8 de las 9 iteraciones se ejecutaban por un solo thread y la última iteración se ejecutaba por otro thread. Con esto se puede inferir que lo que hace la calendarización dinámica, es dividir, valga la redundancia, de manera dinámica la cantidad de iteraciones que ejecutará cada thread.

1. **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.**

Text

Description automatically generated

**Figura 7.** Ejecución del programa con 9 *threads* en la función de validación por columnas y sin definir la cantidad de threads en el programa principal.

Debido a que la cantidad de *threads* a incrementando de 4 a 9, sin embargo, la cantidad de cores o procesos sigue siendo 4 esto implica que la concurrencia ha incrementado ya que 4 *threads* de hardware tendrán que ejecutar 9 *threads* de kernel, lo que implica que para dar la simulación que se están ejecutando de manera concurrente se tendrá que cambiar de un thread a otro para procesar pequeñas partes de todos.

No se podría decir con certeza si esto es o no sinónimo de un mejor desempeño, sin embargo, como se vio en clase, el cambio de contexto de los threads recurre en menor overhead que el cambio de contexto de procesos, de aquí que se conozcan como LightWeightProcess, ya que no implican todo el proceso de guardar el estado del proceso y registros en el PCB. Por lo tal, se podría decir que sí existe una mejora en el desempeño ya que ahora cada iteración del *for* está siendo ejecutada por un thread.

1. **¿Cuál es el efecto de agregar omp\_set\_nested(true)? Explique.**

Graphical user interface

Description automatically generated

**Figura 8.** Ejecucición del programa con *nested* *parallelism* activado.

Al comparar la **figura 8** con la **figura 7** se puede encontrar que no existe cambio alguno en cuanto a la cantidad de *thread* totales creados. Ya que, en ambas ejecuciones en total, se crearon 13 *threads*.

Investigando un poco más acerca del funcionamiento de omp\_set\_nested(true) encontré que básicamente lo que hace es la paralelización de bloques ya paralelizados, básicamente como indica el nombre, trabaja con paralelismo anidado. Por ejemplo, si se tiene paralelismo de 2-niveles la cantidad de *threads* totales será N^2, donde N es la cantidad de threads almacenada en OMP\_NUM\_THREADS (Conor, 2010).

Dentro del código elaborado, no se cuentan con bloques paralelos anidados, sino más bien con varios bloques paralelos, pero todos al mismo nivel, y por lo anterior mencionado se sabe que omp\_set\_nested(true) necesita de bloques paralelos anidados para que tenga un efecto en la ejecución, por lo tal, la **figura 8** y la **figura 7** en cuanto a la cantidad de *thread*s totales creados es la misma ya que no encontró bloques paralelos en los cuales hiciera la paralelización anidada.

**Bibliografía**

Baeldung. (2021, 7 agosto). *What Does “Busy Waiting” Mean in Operating Systems?* Baeldung on Computer Science. https://www.baeldung.com/cs/os-busy-waiting

Blrfl. (2012, 4 noviembre). *What is a thread pool?* Software Engineering Stack Exchange. https://softwareengineering.stackexchange.com/questions/173575/what-is-a-thread-pool

CB. (2010, 29 noviembre). *OpenMP Thread count question*. Stack Overflow. https://stackoverflow.com/questions/4301396/openmp-thread-count-question

colaboradores de Wikipedia. (2019, 30 julio). *Paralelismo de tareas*. Wikipedia, la enciclopedia libre. https://es.m.wikipedia.org/wiki/Paralelismo\_de\_tareas

colaboradores de Wikipedia. (2021, 11 septiembre). *Paralelismo de datos*. Wikipedia, la enciclopedia libre. https://es.m.wikipedia.org/wiki/Paralelismo\_de\_datos

Conor, T. (2010, 30 noviembre). *OpenMP: What is the benefit of nesting parallelizations?* Stack Overflow. https://stackoverflow.com/questions/4317551/openmp-what-is-the-benefit-of-nesting-parallelizations

*Intro to Parallel Programming with OpenMP*. (s. f.). tildesites. https://tildesites.bowdoin.edu/%7Eltoma/teaching/cs3225-GIS/fall17/Lectures/openmp.html#:%7E:text=When%20run%2C%20an%20OpenMP%20program,it%27s%20called%20the%20master%20thread

Jakas. (2016, 13 enero). *OpenMP: For & Scheduling*. jakascorner. http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

Obrien, S. (2021, 28 febrero). *Processes, Threads and the Clone Syscall*. Wayofthepipe. https://blog.wayofthepie.dev/linux/linux-procs-and-threads/

Oracle. (s. f.). *ChapterÂ 4 Nested Parallelism (Sun Studio 12: OpenMP API User’s Guide)*. https://docs.oracle.com/cd/E19205-01/819-5270/aewbc/index.html

Oracle. (2012, 1 octubre). *What is a Thread Pool? - Multithreaded Programming Guide*. https://docs.oracle.com/cd/E26502\_01/html/E35303/ggedh.html

Starynkevitch, B. (2012, 26 julio). *When is clone() and fork better than pthreads?* Stack Overflow. https://stackoverflow.com/questions/11662781/when-is-clone-and-fork-better-than-pthreads

Yliluoma, J. (2016, junio). *Guide into OpenMP: Easy multithreading programming for C++*. bisqwit. https://bisqwit.iki.fi/story/howto/openmp/

Yu, H. (2022, 26 enero). *Race conditions and deadlocks - Visual Basic*. Microsoft Docs. https://docs.microsoft.com/en-us/troubleshoot/developer/visualstudio/visual-basic/race-conditions-deadlocks