Prácticas de Paradigmas de Programación II

M. en C. Manuel Almeida Vázquez

Ingeniería en Computación
Unidad Académica: Paradigmas de Programación II
Núcleo: Sustantivo.
Carácter: Obligatoria.
Clave: LINC33.
Período: 4°

Contenido

PRESENTACIÓN

Este manual de prácticas aspira ser un documento de apoyo en el proceso de enseñanza-aprendizaje de la materia: Paradigmas de Programación II.
Dando al estudiante el entrenamiento y habilidades pertinentes al ámbito de la programación concurrente, paralela y distribuida, así como, utilizar este entrenamiento y habilidades adquiridas para el desarrollo de aplicaciones que auxilien en la solución del diseño de programas que implementen algoritmos matemáticos y generales en paralelo y distribuidos.
El material presentado se enfoca a los temas de: programación con hilos a bajo nivel y al uso de la herramienta de POSIX THREADS, que proporciona una biblioteca de funciones para el manejo de los hilos. Para el manejo de hilos a un nivel más alto usamos la API OpenMP, que es es un estándar de la industria, que nos da directivas del compilador, biblioteca de funciones y variables de ambiente para una forma de programación, la cual nos libera de las tareas de detalle del manejo de los hilos, para hacer programación paralela.
Finalmente se aborda el tema de la programación distribuida usando el paradigma de paso de mensajes, usamos la API MPI para construir programas para ejecutar en un ambiente de cluster.
El deseo es que el manual sirva para motivar al educando a repensar sus programas y algoritmos, para reescribirlos y ejecutarlos en paralelo.

EVALUACIÓN:

A continuación, se presenta la rúbrica de evaluación para los programas desarrollados durante las practicas.
Rúbrica para evaluar los productos de la programación.

CATEGORÍA	EXCELENTE	BUENO	REGULAR	DEFICIENTE
Funcionamiento indispensable para evaluar el resto de las puntos. (60%)	Compila y ejecuta el 100% de lo requerido.	Compila y se ejecuta, faltando algo de lo solicitado.	Compila con errores en la ejecución.	No compila.
Aplica correctamente las directivas, funciones y variables de ambiente, para uso de hilos y paralelismo.(20%)	Usa 80%-100% correctamente.	Aplica entre 60%-79%	Utiliza entre el 30% al 59%.	No utiliza correctamente las instrucciones para la programación.
Claridad, legibilidad y lógica del programa. (15%)	Del 90% al 100% del programa es claro legible y la lógica obedece al objetivo del programa.	Es claro y legible del 60%-89%, pero la lógica no es muy clara.	Es claro y legible del 20%-59%, pero la lógica no es clara.	El código no es claro, ni legible no hay lógica.
Documentación del programa.(5%)	Documenta del 90% al 100% con comentarios y nombres de variables y funciones significativas.	Del 50% al 79% con comentarios y nombres de variables y funciones significativas.	Sólo entre el 20% al 49% está comentado, no usa nombres de variables y funciones significativamente.	No documenta el código.

Prácticas unidad 3: Modelos de programación paralela basada en hilos.
Objetivo: Construir programas de computadora mediante modelos de programación paralela basada en hilos, para resolver problemas de procesamiento de datos de manera concurrente.
Los temas correspondientes a la unidad son:

3.1. Concurrencia.
3.2. Hilos y Multihilos.
3.3 Exclusión Mutua.
3.4 Sicronización.

Antecedentes: Hilos
En la UA de sistemas operativos se estudian los procesos y algunas formas para compartir información entre procesos. Ahora veremos cómo usar múltiples hilos de control para efectuar múltiples tareas dentro de un mismo proceso. Un aspecto muy importante al hablar de compartir recursos son los mecanismos de sincronización necesarios para conservar la consistencia de los recursos compartidos.
POSIX significa **P**ortable **O**perating **S**ystem **I**nterface (for **U**nix). Es un estándar orientado a facilitar la creación de aplicaciones confiables y portables. La mayoría de las versiones populares de UNIX (Linux, Mac OS X) cumplen este estándar en gran medida. La biblioteca para el manejo de hilos en POSIX es pthread.

Conceptos de Hilo

Un proceso típico de Unix puede ser visto como un único **hilo de control**: cada proceso hace sólo una tarea a la vez. Con múltiples **hilos de control** podemos hace más de una tarea a la vez cuando cada hilo de hace cargo de una tarea.
Beneficios de hilos:

Se puede manejar eventos asíncronos asignando un hilo a cada tipo de evento.
Luego cada hilo maneja sus eventos en forma sincrónica.

- Los hilos de un proceso comparten el mismo espacio de direcciones y descriptores de archivos.
- Procesos con múltiples tareas independientes pueden terminar antes si estas tareas se desarrollan traslapadamente en hilos separados. De este modo tiempos de espera de la primera tarea no retrasan la segunda.
- Programas interactivos pueden lograr mejor tiempo de respuesta usando hilos para manejar la entrada y salida. Este es un ejemplo del punto previo.

- La creación de un hilo es mucho más rápida y toma menos recursos que la creación de un proceso.

Multihilos se aplica a máquinas con uno o múltiples procesadores o núcleos. Un hilo contiene la información necesaria para representar un contexto de aplicación dentro de un proceso. Ésta es:

- ID del hilo. No son únicos dentro del sistema, sólo tienen sentido en el contexto de cada proceso.
- Stack pointer
- Un conjunto de registros
- Propiedades de planeación (como política y prioridad)
- Conjunto de señales pendientes y bloqueadas.
- Datos específicos del hilo.

Administración de Hilos

Un paquete de manejo de hilos generalmente incluye funciones para: crear y destruir un hilo, planeación, forzar exclusión mutua y espera condicionada.
Los hilos de un proceso comparten variables globales, descriptores de archivos abiertos, y puede cooperar o interferir con otros hilos.
Todas las funciones de hilos del POSIX comienzan con pthread. Entre ellas están:

Función POSIX	Descripción
pthread_equal	verifica igualdad de dos identificados de hilos
pthread_self	retorna ID de propio hilo (análogo a getpid)
pthread_create	crea un hilo (análogo a fork)
pthread_exit	termina el hilo sin terminar el proceso (análogo a exit)
pthread_join	espera por el término de un hilo (análogo a waitpid)
pthread_cancel	Termina otro hilo (análogo a abort)
pthread_detach	Configura liberación de recursos cuando termina
pthread_kill	envía una señal a un hilo

Identificación de Hilos

Así como un proceso tiene un PID (Process Identification), cada hilo tiene un identificador de hilo. Mientras los PID son enteros no negativos, el ID de un hilo es dependiente del SO y puede ser una estructura. Por esto para su comparación se usa una función.

#include <pthread.h> int **pthread_equal**(pthread_t tid1, pthread_t tid2); Retorna: no cero si es igual, cero en otro caso	Compara dos identificados de hilos tid1 y tid2
#include <pthread.h> pthread_t **pthread_self**(void); Retorna: la ID del hilo que la llamó	Para obtener identificador de un hilo

Creación de Hilos

Los procesos normalmente corren como un hilo único. La creación de un nuevo hilo se logra vía pthread_create.

# include <pthread.h> int **pthread_create**(pthread_t * restrict tidp, const pthread_attr_t * restrict attr, void * (* start_routine) (void *), void * restrict arg);	tidp: salida, puntero a id del hilo attr: entrada, para definir atributos del hilo, null para default start_routine: entrada, función a correr por el hilo arg: entrada, argumento de la función del hilo. La función debe retornar un * void, el cual es interpretado como el estatus de término por pthread_join

Para hacer uso de estas funciones incluir <pthread.h>, el ligado debe incluir -lpthread.

Desarrollo: Usando la biblioteca pthreads.h y las funciones PTHREAD_CREATE(), PTHREAD_JOIN(), elaborar un programa que cree 2 hilos que impriman un carácter (*,o) 500 veces.

Término de un Hilo

Si un hilo invoca a exit, _Exit o _exit, todo el proceso terminará.

Un hilo puede terminar de tres maneras sin terminar el proceso: Retornando de su rutina de inicio, cancelado por otro hilo del mismo proceso, o llamando pthread_exit.

#include <pthread.h> void **pthread_exit** (void * rval_ptr);	rval_ptr queda disponible para otros hilos al llamar pthread_join rval_ptr debe existir después del término del hilo.
int **pthread_join**(pthread_t tid, void ** rval_ptr);	El hilo llamante se bloquea hasta el término del hilo indicado. Si el hilo en cuestión es cancelado, rval_prt toma el valor PTHREAD_CANCELED Si no estamos interesados en el valor retornado, poner NULL.

Práctica 3.1 Programación concurrente usando hilos, con POSIX THREADS.
Objetivo: Elaborar un programa en el que use las funciones de POSIX THREADS, PTHREAD_CREATE() y PTHREAD_JOIN() así como la cabecera <pthread.h>, para la creación y manejo de hilos. Observar el funcionamiento del modelo Fork-Join .
Duración. Dos horas.
Requerimientos para el desarrollo:
Se deberá contar con POSIX THREADS en los sistemas tipo LINUX existe por default o instalado en WINDOWS, además del lenguaje C y C++, se deberá trabajar en una computadora con al menos dos núcleos.
Procedimiento:
Instrucciones.
1. Partiendo desde cero escribir las instrucciones para el uso de las cabeceras necesarias para ejecutar un programa en lenguaje C o C++ incluyendo <pthread.h>.
2. Inmediatamente se declarará la función imp_char() tipo void, que escriba un carácter (pej. "*") en la pantalla 600 veces.
3. A continuación escribir la instrucciones correspondientes para definir la función main() tipo void también . En el cuerpo de la función incluiremos un ciclo que imprima un carácter (pej. "o") 500 veces, y la llamada a la función

pthread_create(pthread_id, NULL, &imp_char,NULL)

con sus parámetros correspondientes, para que sea ejecutada la función que imprime un carácter.
4. Ahora incluya la función: pthread_join().
5. Compile y ejecute el programa usando la bandera -lpthread.
Cuestionario. Se deberá llamar la atención sobre lo siguiente:

a) ¿Por qué los caracteres "o" y el "*" aparecen traslapados?
b) ¿Puede ver cuándo escribe una "o" el hilo master? Y ¿cuándo lo hace el hilo hijo?
c) ¿De qué forma se manifiesta el punto donde se hace el fork? Y en ¿cuál donde se hace el join.?,
c) ¿Qué pasó cuando introdujo la función pthread_join().

Productos de la Práctica.
- Un Programa que use las funciones de POSIX THREADS para la generación de hilos, pthread_create() y pthread_join(), y usando una función que imprima un carácter.

Práctica 3.2 Programación usando hilos para mostrar la concurrencia, usando con POSIX THREADS.

Objetivo: Elaborar un programa en el que use las funciones de POSIX THREADS, PTHREAD_CREATE() y PTHREAD_JOIN() así como la cabecera <pthread.h>, para la creación y manejo de hilos de dos hilos, donde se requiere adicionalmente una estructura para pasar parámetros a la función usada..

Duración. Cuatro horas.

Requerimientos para el desarrollo:

Se deberá contar con POSIX THREADS en los sistemas tipo LINUX existe por default o instalado en WINDOWS, además del lenguaje C y C++. Se deberá trabajar en una computadora con al menos dos núcleos.

Procedimiento:

Instrucciones.

1. Partiendo del programa de la práctica 3.1, escribir las instrucciones para el uso de las cabeceras necesarias para ejecutar un programa en lenguaje C o C++ incluyendo <pthread.h>.
2. Inmediatamente se declarará la función imp_char(ch,n) tipo void, que escriba un carácter (pej. "*") en la pantalla n veces, que ahora se le pasarán como parámetros.
3. Crear una estructura con dos campos, uno tipo carácter y otro tipo entero llamada params.
4. A continuación escribir la instrucciones correspondientes para definir la función main() tipo void también . En el cuerpo de la función se crearán dos hilos con la llamada a la función

    pthread_create(pthread_id, NULL, &imp_char, &params)

con sus parámetros correspondientes, para que sea ejecutada la función que imprime un carácter, un número n de veces que se le pasán por mediio de la estructura params, para cada uno de los hilos.

5. Incluya la función: pthread_join().
6. Compile y ejecute el programa usando la bandera -lpthread.

Cuestionario. Se deberá llamar la atención sobre lo siguiente:

a) ¿Por qué los caracteres "o" y el "*" aparecen traslapados?

b) ¿Por qué es importante ahora usar una función y una estructura, para que funcione el hilo?

c) ¿Puede ver cómo funciona la concurrencia?

Productos de la Práctica.

- Un Programa que use las funciones de POSIX THREADS para la generación de dos hilos usando pthread_create() y pthread_join(), usando una función que imprima un carácter n veces y una estructura para pasarle los parámetros a la función..

Práctica 3.3 Programación concurrente usando cuatro hilos aplicando la exclusión mutua por medio de la creación de un MUTEX y de las funciones de POSIX THREADS. Para elaborar un programa que calcule el balance de una cuenta, por medio de la suma de cantidades positivas y negativas de la la cuenta.

Objetivo: Elaborar un programa en el que use las funciones de POSIX THREADS, PTHREAD_CREATE(), THREAD_MUTEX() y PTHREAD_JOIN() así como la cabecera <pthread.h>, para la creación y manejo de hilos, asi como, un MUTEX para coordinar la operación entre los hilos, creando una región crítica usano el mutex para bloquear y desbloquear la operación..

Duración. Cuatro horas.

Requerimientos para el desarrollo:

Se deberá contar con POSIX THREADS en los sistemas tipo LINUX existe por default o instalado en WINDOWS, además del lenguaje C y C++. Se deberá trabajar en una computadora con al menos dos núcleos.

Procedimiento:

Instrucciones.

1. Como en las prácticas anteriores se deberá iniciar escribiendo el esqueleto del programa con las cabecers del programa y de la función main().
2. Ahora será necesario declarar dos variables tipo mutex usando: pthread_mutex_t mutex, e inicializarla, es más directo así:

    pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER.

3. Crear dos hilos, una que adicione las entradas positivas y otro que lo haga con as negativas.
4. Crear una función que suma a una variable llamada TOTAL, y otra que reste a la misama variable TOTAL.
5. En la función se deberá definir la región critica usando: pthread_mutex_lock() y como parámetro un apuntador al mutex declarado para uso del hilo correspondiente, y desbloquear usando: pthread_mutex_unlock().
6. Compile y ejecute el programa usando la bandera -lpthread.

Cuestionario. Se deberá llamar la atención sobre lo siguiente:

a) ¿Cuál sería el resultado si no se usaran los mutex?
b) ¿Por qué es importante que en la función se incluya la región crítica?
c) ¿Puede ver cómo sería la condición de competencia entre los hilos?

Productos de la Práctica.

- Un Programa que use las funciones de POSIX THREADS para la generación de dos hilos usando pthread_create() y pthread_join(), las variables tipo mutex y creación de la región critica, donde se aplican los mutex, para calcular correctamente el balance de una cuenta.

Prácticas unidad 4: Programación Paralela en Memoria Compartida.
Objetivo: Diseñar programas de computadora por medio del modelo de memoria compartida usando la API OpenMP, para implementar soluciones a programas que requieran disminuir sus tiempos de ejecución y utilizar la memoria de manera eficiente.
Los temas son:

4.1 Bases de los algoritmos paralelos.
4.2 Directivas de la API openMP.
4.3 Elaboración de programas.
4.4 Evaluación del rendimiento.

Antecedentes: OpenMP
OpenMP es una interfaz de programación de aplicaciones (API) para la programación multiproceso con memoria compartida en múltiples plataformas. Permite añadir concurrencia los programas escritos en C, C++ y Fortran sobre la base del modelo de ejecución fork-join. Está disponible en muchas arquitecturas incluidas las plataformas de Unix, LINUX y de Microsoft Windows. Se compone de un conjunto de directivas del compilador, biblioteca funciones, y variables de ambiente que influyen en el comportamiento en tiempo de ejecución.
Definido conjuntamente por proveedores de hardware y de software, OpenMP es un modelo de programación portable y escalable que proporciona a los programadores una interfaz simple y flexible para el desarrollo de aplicaciones paralelas, para plataformas que van desde las computadoras de escritorio hasta supercomputadoras. Una aplicación construida con un modelo de programación paralela híbrido se puede ejecutar en un cluster de computadoras utilizando OpenMP y MPI, o a través de las extensiones de OpenMP para los sistemas de memoria distribuida.

Modelo de ejecución

OpenMP se basa en el modelo **fork-join**, paradigma que proviene de los sistemas Unix donde una tarea muy pesada se divide en n hilos (fork) con menor peso, para luego "recolectar" sus resultados al final y unirlos en un solo resultado (join). Cuando se incluye una directiva de compilador **OpenMP** esto implica que se incluye una sincronización obligatoria en todo el bloque. Es decir, el bloque de código se marcará como paralelo y se lanzarán hilos según las características que nos dé la directiva, y al final de ella habrá una barrera para la sincronización de los diferentes hilos (salvo que implícitamente se indique lo contrario con la directiva **nowait**). Este tipo de ejecución se denomina **fork-join**.

Práctica 4.1 Programación paralela usando la API OpenMP.

Objetivo: Elaborar un programa en el que use las: directivas de compilador #pragma omp parallel, #prag omp parallel for, las funciones omp_get_num_procs() y omp_get_num_threads(), así como variables de ambiente OMP_NUM_THREAD de OpenMP, así como la cabecera <omp.h>, para que se imprima el mensaje "Hola mundo", tantas veces como núcleos tenga la computadora.

Duración. Dos horas.

Requerimientos para el desarrollo:

Se deberá contar con la API OpenMP en los sistemas tipo LINUX existe por default o instalado en WINDOWS, además del lenguaje C y C++. Se deberá trabajar en una computadora con al menos dos núcleos.

Procedimiento:

Instrucciones.

1. Partiendo desde cero, escribir las instrucciones para el uso de las cabeceras necesarias para ejecutar un programa en lenguaje C o C++ incluyendo <omp.h>.
2. A continuación escribir la instrucciones correspondientes para definir la función main() tipo void. En el cuerpo de la función se declarará un ciclo "for" paralelo después de la directiva:

    #pragma omp parallel for

3. Compile y ejecute el programa usando la bandera -fopenmp.

Cuestionario. Se deberá llamar la atención sobre lo siguiente:

a) ¿Cuántas veces aparece el mensaje?

b) ¿Cuántos hilos están trabajando?

c) ¿Puede ver cómo nos facilita OpenMP el manejo de los hilos?

Productos de la Práctica.

- Un Programa que use las directivas, funciones y variables de entorno de OpenMP que muestre el mensaje "Hola mundo" tantas veces como núcleos tenga la computadora.

Práctica 4.2 Programación paralela usando la API OpenMP, para calcular el producto de vectores.

Objetivo: Elaborar un programa en el que use las: directivas de compilador #pragma omp parallel, #prag omp parallel for, las funciones omp_get_num_procs() y omp_get_num_threads(), las clausula reduce(), así como variables de ambiente OMP_NUM_THREAD de OpenMP, así como la cabecera <omp.h>, para calcular el producto escalar de dos vectores de tamaño 1000.

Duración. Cuatro horas.

Requerimientos para el desarrollo:

Se deberá contar con la API OpenMP en los sistemas tipo LINUX existe por default o instalado en WINDOWS, además del lenguaje C y C++. Se deberá trabajar en una computadora con al menos dos núcleos.

Procedimiento:

Instrucciones.

1. Partiendo desde cero, escribir las instrucciones para el uso de las cabeceras necesarias para ejecutar un programa en lenguaje C o C++ incluyendo <omp.h>, además de las necesarias para manejar archivos que contengan los datos de los vectores.
2. A continuación escribir la instrucciones correspondientes para definir la función main() tipo void. En la primera versión se puede usar un ciclo for paralelo, con la cláusula reduce para el operador sum, en la variable global Prod.
3. En la segunda versión, se definirá una función que calcule el producto escalar de dos vectores, donde el producto parcial lo acumule en una variable local sumpar. Y que esta variable sumapar se agregue a la variable Prod por medio de la cláusula reduce.
4. Prepare dos archivos que contengan cada uno un vector de mil unos.
5. Divida los datos de los vectores en cuatro, para que se calcule el producto con cuatro hilos.
6. Compile y ejecute el programa usando la bandera -fopenmp, compare el tiempo de ejecución modificando el tamaño de los vectores (use 10,000,100,000).

Cuestionario. Se deberá llamar la atención sobre lo siguiente:

a) ¿Verifique su resultado que deberá ser igual a Prod=1000?
b) ¿Cuántos hilos están trabajando?
c) ¿Puede ver cómo nos facilita OpenMP el manejo de los hilos?
d) ¿De qué manera puede comprobar que el resultado es correcto?
e) Estime el porcentaje de ejecución en paralela y secuencial, usando la ley de Amdahl y la de Gustafson, calcule la aceleración y la máxima aceleración alcanzable por el programa.

Productos de la Práctica.

- Un Programa que use las directivas, funciones y variables de entorno de OpenMP que muestre el resultado del producto escalar de dos vectores de tamaño 1000.
- Medición del tiempo de ejecución con diferentes tamaños de los vectores

Práctica 4.3 Programación paralela usando la API OpenMP,para el producto de matrices.

Objetivo: Elaborar un programa en el que use las: directivas de compilador #pragma omp parallel, #prag omp parallel for, las funciones omp_get_num_procs() y omp_get_num_threads(), las clausula reduce(), así como variables de ambiente OMP_NUM_THREAD de OpenMP, así como la cabecera <omp.h>, para calcular el producto de dos matrices de tamaño 1000x1000.

Duración. Cuatro horas.

Requerimientos para el desarrollo:

Se deberá contar con la API OpenMP en los sistemas tipo LINUX existe por default o instalado en WINDOWS, además del lenguaje C y C++. Se deberá trabajar en una computadora con al menos dos núcleos.

Procedimiento:

Instrucciones.

1. Partiendo desde cero, escribir las instrucciones para el uso de las cabeceras necesarias para ejecutar un programa en lenguaje C o C++ incluyendo <omp.h>, además de las necesarias para manejar archivos que contengan los datos de las matrices.
2. A continuación escribir la instrucciones correspondientes para definir la función main() tipo void. Se puede usar ciclos for paralelos, con la cláusula reduce para calcular los elementos C[i][j] de la matriz producto. En esta variante se tendría que habilitar la forma de for anidados, usando OMP_NESTED, que activa el uso de regiones paralelas anidadas. Si TRUE entonces los miembros de un equipo de hilos pueden crear nuevos equipos de hilos. A partir de la versión 4.5 de OpenMP se puede usar la cláusula COLAPSE, si son reducibles y no hay dependencias entre los for's.
3. En una segunda versión, se puede usar la función que calcula el producto escalar de dos vectores, descomponiendo las matrices en vectores renglón, de este modo reducir el producto de las matrices a productos de vectores renglón y columna.
4. Prepare dos archivos que contengan cada uno una matriz 1000x1000 de unos.
5. Compile y ejecute el programa usando la bandera -fopenmp.

Cuestionario. Se deberá llamar la atención sobre lo siguiente:

a) ¿Verifique su resultado que deberá ser una matriz de 1000x100 con elementos iguales a 1000?

b) ¿Cuántos hilos están trabajando?

c) ¿De qué manera puede comprobar que el resultado es correcto?

e) Estime el porcentaje de ejecución en paralela y secuencial, usando la ley de Amdahl y la de Gustafson, calcule la aceleración y la máxima aceleración alcanzable por el programa.

Productos de la Práctica.

- Un Programa que use las directivas, funciones y variables de entorno de OpenMP que muestre el resultado del producto de dos matrices de tamaño 1000x1000.

- Medición del tiempo de ejecución con diferentes tamaños de las matrices.
- Cálculo de la aceleración y aceleración máxima.

Practicas unidad 5: Programación Paralela en Memoria Distribuida.
Objetivo: Diseñar programas de computadora mediante el modelo programación basada en paso de mensajes, para implementar soluciones que requieran el uso simultáneo y eficiente de recursos.
Temas:

5.1 Bases de algoritmos distribuidos.
5.2 Directivas de la API MPI.
5.3 Elaboración de programas.

5.4 Evaluación del rendimiento.
Antecedentes: Paso de mensajes con MPI
La **interfaz de paso de mensajes** o **MPI** (sigla del inglés *message passing interface*) es un estándar que define la sintaxis y la semántica de las funciones contenidas en una biblioteca de paso de mensajes diseñada para ser usada en programas que exploten la existencia de múltiples procesadores, escritas generalmente en C, C++, Fortran y Ada. La ventaja de MPI sobre otras bibliotecas de paso de mensajes, es que los programas que utilizan la biblioteca son portables (dado que MPI ha sido implementado para casi toda arquitectura de memoria distribuida), y rápidos, (porque cada implementación de la biblioteca ha sido optimizada para el hardware en la cual se ejecuta).
Con MPI el número de procesos requeridos se asigna antes de la ejecución del programa, y no se crean procesos adicionales mientras la aplicación se ejecuta. A cada proceso se le asigna una variable que se denomina *rango*, la cual identifica a cada proceso, en el rango de 0 a p-1, donde p es el número total de procesos. El control de la ejecución del programa se realiza mediante la variable de rango. Esta permite determinar qué proceso ejecuta determinada porción de código. En MPI se define un *comunicador* como una colección de procesos, los cuales pueden enviar mensajes el uno al otro; el comunicador básico se denomina MPI_COMM_WORLD y se define mediante un macro del lenguaje C. MPI_COMM_WORLD agrupa a todos los procesos activos durante la ejecución de una aplicación. Las llamadas de MPI se dividen en cuatro clases:

1. Llamadas utilizadas para inicializar, administrar y finalizar comunicaciones.
2. Llamadas utilizadas para transferir datos entre un par de procesos.
3. Llamadas para transferir datos entre varios procesos.
4. Llamadas utilizadas para crear tipos de datos definidos por el usuario.

La primera clase de llamadas permiten inicializar la biblioteca de paso de mensajes, identificar el número de procesos y el rango de los procesos. La segunda clase de llamadas incluye operaciones de comunicación punto a punto, para diferentes tipos de

actividades de envío y recepción. La tercera clase de llamadas son conocidas como operaciones grupales, que proveen operaciones de comunicaciones entre grupos de procesos. La última clase de llamadas provee flexibilidad en la construcción de estructuras de datos complejos.

Llamadas utilizadas para inicializar, administrar y finalizar comunicaciones

MPI dispone de cuatro funciones primordiales que se utilizan en todo programa con MPI: *MPI_Init*, *MPI_Finalize*, *MPI_Comm_size* y *MPI_Comm_rank*.

Llamadas utilizadas para transferir datos entre dos procesos

La transferencia de datos entre dos procesos se consigue mediante las llamadas MPI_Send y MPI_Recv. Estas llamadas devuelven un código que indica su éxito o fracaso.

MPI_Send

Permite enviar información desde un proceso a otro.

MPI_Recv

Permite recibir información desde otro proceso.

Ambas funciones son bloqueantes, es decir que el proceso que realiza la llamada se bloquea hasta que la operación de comunicación se complete.

Las versiones no bloqueantes de MPI_Send y MPI_Recv son MPI_Isend y MPI_Irecv, respectivamente. Estas llamadas inician la operación de transferencia pero su finalización debe ser realizada de forma explícita mediante llamadas como MPI_Test y MPI_Wait.

MPI_Wait

Es una llamada bloqueante y retorna cuando la operación de envío o recepción se completa.

Práctica 5.1 el paradigma de paso de mensajes, con la API MPI.

Objetivo: Elaborar un programa en el que use las funciones de MPI, MPI_COMM_WORLD, MPI_INIT, MPI_RANK, MPI_SIZE, MPI_FINALIZE así como la cabecera <mpi.h>, para la creación y manejo de hilos. Observar el funcionamiento del modelo Fork-Join .

Duración. Dos horas.

Requerimientos para el desarrollo:

Se deberá contar con la API MPI en los sistemas tipo LINUX existe por default o instalado en WINDOWS, además del lenguaje C y C++, se deberá trabajar en una computadora con al menos dos núcleos.

Procedimiento:

Instrucciones.

1. Partiendo desde cero escribir las instrucciones para el uso de las cabeceras necesarias para ejecutar un programa en lenguaje C o C++ incluyendo <mpi.h>.
2. Definir la función a ejecutar, a continuación escribir la instrucciones correspondientes para definir la función main() tipo void también . En el cuerpo de la función incluiremos dos variables de tipo entero, p y id para contener el número de procesadores y el número de hilos. Usando MPI_INIT(&argc,&argv), MPI_Comm_rank(MPI_COMM_WORLD,&id),
MPI_Comm_size(MPI_COMM_WORLD,&p)
4. También se incluirá un ciclo for para ejecutar la función 1000 veces.
5. Antes del return, se debe incluir MPI_Finalize(), para cerrar MPI.
6. Compile usando mpicc.

Cuestionario. Se deberá llamar la atención sobre lo siguiente:

a) ¿Funcionó su programa?
b) ¿Puede identificar la diferencia entre OpenMP y MPI?
c) ¿Puede proponer una hipótesis del por qué funciona MPI en una computadora multinúcleo?
d) ¿De qué forma se manifiesta el que se usen los núcleos como procesadores independientes?

Productos de la Práctica.

- Un Programa que use las directivas y funciones de MPI para calcular pi, integrando de 0 a 1 la función f(x)=4/(x**2+1).

Referencias bibliograficas

1. http://profesores.elo.utfsm.cl/~agv/elo330/2s08/lectures/POSIX_Threads.html
2. Programming with THREADS. Steve Kleiman, Devan Shah, Bart Smaalders, Sun Microsystems, USA 1996.
3. Taming Java Threads. Allen Holub, Apres Media USA 2000.
4. Concurrent Programming. Stephen J. Hartley Oxford University Press USA 1998.
5. Parallel Programming in C with MPI and OpeMP. Michael J. Quinn.
6. Using OpenMP. Barbara Chapman, Jost, and Van Der Pas. MIT press USA 2007.
7. Using OpenMP-Next Step. Ruud van der Pas, Eric Stotzer and Christian Terboven (2017) MIT press USA 2017.
8. https://es.wikipedia.org/wiki/OpenMP.
9. Advanced Linux Programming. Mark Mitchell, Jeffrey Oldham, and Alex Samuel Pearson Education USA 2001.