Unidad 3 / Escenario 6

Lectura fundamental

Programación paralela aplicada

Contenido

Palabras clave: programación paralela, programación distribuida, exclusión mutua, zona crítica, memoria compartida, memoria distribuida, openMP, MPI

Introducción

Esta Lectura fundamental desarrolla los aspectos más relevantes de la computación paralela, un paradigma de programación que corre sobre arquitecturas de computación de alta velocidad como: *grids*, clústeres, SOA, *clouds*, etc., y que implica el uso de la tecnología de red para mejorar el rendimiento, la potencia del *middleware* y el rediseño de algoritmos para utilizar herramientas de desarrollo y aplicaciones de ingeniería de *software* que corran en arquitecturas de hardware paralelo.

Las tecnologías de procesamiento paralelo se han vuelto omnipresentes en la mayoría de los nuevos procesadores, presentes en una amplia gama de equipos informáticos como computadoras para juegos, PC estándar, estaciones de trabajo y supercomputadoras. La razón principal de esta tendencia es que el paralelismo permite un aumento sustancial en la potencia de procesamiento utilizando tecnologías estándar; esto se traduce en una reducción sustancial del costo en comparación con el desarrollo de *hardware* de alto rendimiento. Hoy en día la capacidad de procesamiento de un PC de escritorio con un procesador multinúcleo reemplaza el poder de cómputo de una supercomputadora de hace dos décadas en una fracción del costo.

La utilización de un equipo tan potente requiere un *software* adecuado. En la práctica parece que la construcción de algoritmos paralelos apropiados y el desarrollo del sistema y el *software* de aplicación que pueda aprovechar las ventajas del *hardware* paralelo no es un asunto simple. Estos problemas han sido estudiados durante casi cinco décadas y, aunque se ha avanzado mucho en las áreas de arquitecturas paralelas, algoritmos, *software* y diseño, grandes problemas aún deben ser abordados.

La creciente replicación del procesamiento con elementos en chips y el uso de componentes para el uso de sistemas paralelos que comprenden un gran número de procesadores con el fin de lograr capacidades de procesamiento hasta ahora inalcanzables destacan entre los problemas asociados con la utilización de estas arquitecturas.

Así mismo, combinando el rápido crecimiento con el número de procesadores *multicore* para PC, hay una creciente necesidad de métodos y herramientas para apoyar el desarrollo de *software* en paralelo capaz de utilizar de manera efectiva y eficiente las arquitecturas paralelas.

1. Diferencias entre la programación secuencial y la programación paralela

La programación paralela implica el cálculo simultáneo o la ejecución simultánea de procesos o subprocesos, mientras que la programación secuencial implica una ejecución consecutiva y ordenada de procesos.

Así es que un programa secuencial ejecutará un proceso que deberá esperar la entrada del usuario, y luego se ejecutará otro proceso que será una devolución de acuerdo con la entrada del usuario, creando una serie de eventos en cascada.

En contraste con la computación secuencial, en la programación paralela los procesos pueden ejecutarse simultáneamente, los subprocesos pueden comunicarse e intercambiar señales durante la ejecución y, por lo tanto, los programadores tienen que implementar medidas para permitir tales transacciones.

La programación secuencial sufre de un gran problema cuando requiere ejecutar varios subprocesos porque debe compartir un recurso que se denomina área crítica, se le conoce como exclusión mutua. La exclusividad mutua se logra al colocar bloqueos en la región crítica, lo que permite que se ejecute un solo hilo a la vez, hasta que se realiza todo su trabajo y abre la puerta de la región crítica, dando acceso a otro proceso. Para ello se utilizan diferentes algoritmos que manejan los semáforos que controlan dicha situación.

2. Computación concurrente, paralela o distribuida

Aunque no hay un acuerdo completo sobre la distinción entre los términos paralelo, distribuido y concurrente, muchos autores hacen las siguientes distinciones:

- En la computación concurrente, un programa es aquel en el que se pueden realizar múltiples tareas en cualquier momento.
- En computación paralela, un programa es aquel en el que múltiples tareas cooperan estrechamente para resolver un problema.
- En la computación distribuida, un programa puede necesitar cooperar con otros programas para resolver un problema.

Así que los programas paralelos y distribuidos son concurrentes, pero un programa como un sistema operativo multitarea también lo es, incluso cuando se ejecuta en una máquina con un solo núcleo, ya que múltiples tareas pueden estar en progreso en cualquier momento.

No hay una distinción clara entre programas paralelos y distribuidos, pero un programa paralelo generalmente ejecuta múltiples tareas simultáneamente en núcleos que están físicamente cerca unos de otros y que comparten la misma memoria o están conectados por una red de muy alta velocidad. Por otro lado, los programas distribuidos tienden a estar acoplados más libremente y las tareas pueden ser ejecutadas por múltiples computadoras que están separadas por grandes distancias o por programas que fueron creados independientemente.

Pero cuidado, no hay un acuerdo general sobre estos términos. Por ejemplo, muchos de los autores consideran que los programas de memoria compartida son paralelos y los de memoria distribuida son programas para ser distribuidos. En resumen: programas paralelos son aquellos en los que las tareas estrechamente relacionadas cooperan para resolver un problema.

3. *Hardware* paralelo

Los ingenieros químicos, los físicos, los matemáticos y otros profesionales disciplinares son personas que, sin haber estudiado programación de computadores, se arriesgan y crean programas paralelos. Sin embargo, para escribir programas paralelos eficientes, se necesita cierto conocimiento del *hardware* subyacente y del *software* del sistema.

3.1. Aplicaciones de la arquitectura SIMD

El *hardware* y el *software* en paralelo se ha desarrollado a partir del *hardware* y el *software* en serie convencional, el que ejecuta un solo trabajo a la vez.

De acuerdo con el modelo de Von Neumann, *hardware* paralelo es aquel que permite modificar fácilmente su código fuente para explotarlo, es decir que es operado por código programable.

Así mismo, en computación paralela, la taxonomía de Flynn se usa con frecuencia para clasificar arquitecturas de computadora; clasifica un sistema según la cantidad de secuencias de instrucciones y la cantidad de secuencias de datos que puede administrar simultáneamente. Por lo tanto, un sistema clásico de Von Neumann es un flujo de instrucciones único, un flujo de datos único o un sistema SISD, ya que ejecuta una sola instrucción a la vez y puede recuperar o almacenar un elemento de datos en cada oportunidad.

Los sistemas de instrucción única, datos múltiples o SIMD son sistemas paralelos. Como su nombre indica, los sistemas SIMD operan en múltiples flujos de datos aplicando la misma instrucción a múltiples elementos. Son ideales para la paralelización de simples bucles que operan en grandes matrices de datos, pues el paralelismo se obtiene al dividir los datos entre los procesadores y al aplicar las mismas instrucciones a todos los procesadores para sus subconjuntos de datos.

Los mapas y los modelos geográficos se desarrollan con base en tres elementos: el punto, la línea y los polígonos. Con esos tres elementos se desarrollan los mapas, los gráficos y, en general, toda la computación gráfica que usan los sistemas distribuidos de renderizado. Una de las aplicaciones distribuidas de *software* libre, utilizadas para trabajar en renderizado es el DrQueue. (http://www.drqueue.org). Este tipo de aplicaciones se ejecutan en arquitecturas SIMD y en lo posible con procesadores GPU (Unidad de Procesamiento Gráfico).

Las GPU pueden optimizar el rendimiento utilizando el paralelismo SIMD; esto se obtiene al incluir un gran número de ALU (Unidades Aritmético Lógicas) en cada procesador GPU. El procesamiento de una sola imagen puede requerir grandes cantidades de datos (*frames)*, y cientos de megabytes de datos. Por lo tanto, las GPU deben mantener tasas de movimiento de datos muy altas y evitar bloqueos en los accesos a la memoria.

Las GPU se están volviendo cada vez más populares para la informática general de alto rendimiento, y se han desarrollado varios idiomas que permiten a los usuarios explotar su poder.

3.2. Aplicaciones de la arquitectura MIMD

Los sistemas de instrucción múltiple y datos múltiples o MIMD son compatibles con múltiples secuencias de instrucciones simultáneas que funcionan en múltiples flujos de datos. Por lo tanto, los sistemas MIMD consisten típicamente en una colección de unidades o núcleos de procesamiento totalmente independientes, cada uno de los cuales tiene su propia unidad de control y su propia ALU.

En muchos sistemas MIMD no hay un reloj global y puede que no haya relación entre los tiempos del sistema en dos procesadores diferentes. De hecho, a menos que el programador imponga alguna sincronización, incluso si los procesadores están ejecutando exactamente la misma secuencia de instrucciones, en cualquier momento pueden estar ejecutando diferentes declaraciones.

En un sistema de memoria compartida, los procesadores generalmente se comunican implícitamente al acceder a estructuras de datos compartidas en la memoria principal. Por su parte, en un sistema de memoria distribuida, cada procesador está emparejado con su propia memoria privada, y los pares de memoria de procesador se comunican a través de una red de interconexión. Así, en los sistemas de

memoria distribuida los procesadores usualmente se comunican explícitamente mediante el envío de mensajes MPI (*Message Passing Interface*) o mediante el uso de funciones especiales que brindan acceso a la memoria de otro procesador.

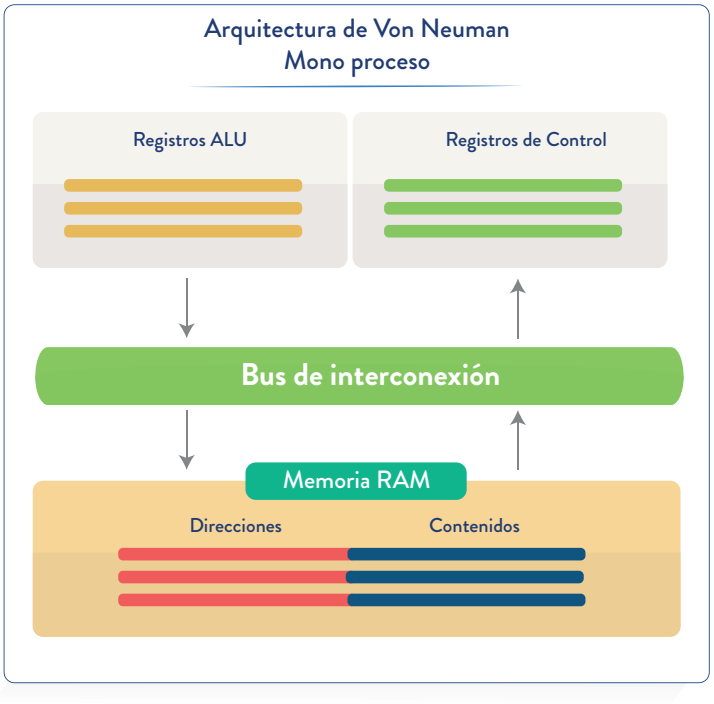


Figura 1. Arquitectura de Von Neumman
Fuente: elaboración propia

3.3. Proceso, multitareas e hilos

El sistema operativo es una pieza importante de *software* cuyo propósito es administrar los recursos de *hardware* y *software* en una computadora, así como determinar qué programas pueden ejecutarse y cuándo pueden hacerlo. También controla la asignación de memoria a los programas en ejecución y el acceso a dispositivos periféricos como discos duros y tarjetas de interfaz de red.

Cuando un usuario ejecuta un programa, el sistema operativo crea un proceso, una instancia de un programa de computadora que se está ejecutando. Este proceso consta de varias entidades:

- El programa de lenguaje de máquina ejecutable.

- Un bloque de memoria, que incluirá el código ejecutable, una pila de llamadas que mantiene seguimiento de funciones activas en grupos y algunas otras ubicaciones de memoria.
- Descriptores de recursos que el sistema operativo ha asignado al proceso, por ejemplo, descriptores de archivos.
- Información de seguridad, por ejemplo, información que especifica a qué recursos de *hardware* y *software* puede acceder el proceso.
- Información sobre el estado del proceso, por ejemplo, si el proceso está listo para ejecutarse o está esperando por algún recurso. Equivale el contenido de los registros e información sobre la memoria del proceso.

La mayoría de los sistemas operativos modernos son multitarea. Esto significa que el sistema operativo proporciona soporte para la ejecución simultánea aparente de múltiples programas. Esto es posible incluso en un sistema con un solo núcleo, ya que cada proceso se ejecuta durante un pequeño intervalo de tiempo (generalmente, unos pocos milisegundos).

Después de que un programa en ejecución se haya ejecutado durante un intervalo de tiempo, el sistema operativo ejecuta un programa diferente. Un sistema operativo multitarea puede cambiar el proceso en ejecución muchas veces por minuto, aunque el proceso en ejecución pueda llevar mucho tiempo.

En un sistema operativo multitarea, si un proceso necesita esperar un recurso, por ejemplo, necesita leer datos de un almacenamiento externo, se bloqueará. Esto significa que dejará de ejecutarse y que el sistema operativo puede ejecutar otro proceso. Sin embargo, muchos programas pueden continuar haciendo un trabajo útil, aunque la parte del programa que se está ejecutando actualmente deba esperar por el recurso. Por ejemplo, un sistema de reservas de una aerolínea que está bloqueado esperando un mapa de asientos para un usuario podría proporcionar una lista de vuelos disponibles a otro.

La creación de hilos proporciona un mecanismo para que los programadores dividan sus programas en tareas más o menos independientes con la propiedad que cuando se bloquea un hilo se puede ejecutar otro. Además, en la mayoría de los sistemas es posible cambiar entre subprocesos mucho más rápido de lo que es posible cambiar entre procesos. Esto se debe a que los hilos son más livianos.

Los subprocesos están contenidos dentro de los procesos, por lo que pueden usar el mismo ejecutable y, por lo general, comparten la misma memoria y los mismos dispositivos de E/S. De hecho, dos subprocesos que pertenecen a un proceso pueden compartir la mayoría de los recursos.

3.4. Memoria distribuida

En un sistema de memoria distribuida, cada procesador está emparejado con su propia memoria privada, y los pares de memoria de procesador se comunican a través de una red de interconexión. Así que en los sistemas de memoria distribuida los procesadores usualmente pueden comunicarse de forma explícita mediante el envío de mensajes o mediante el uso de funciones especiales que proporcionan acceso a la memoria de otro procesador.

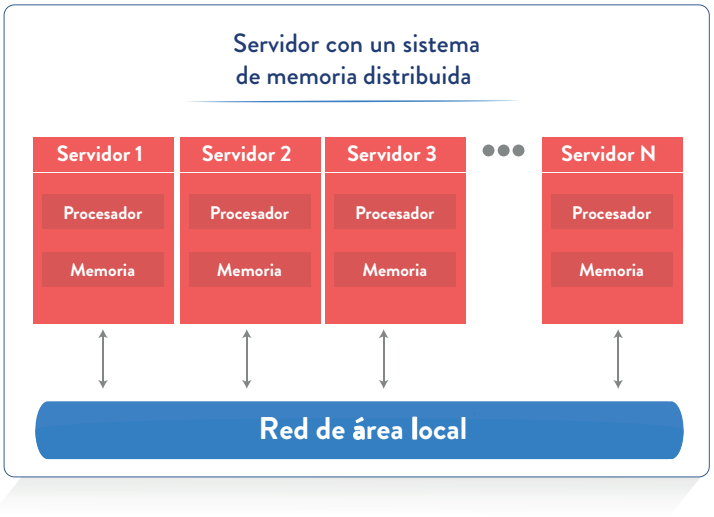


Figura 2. Sistema de memoria distribuida o débilmente acoplada

Fuente: elaboración propia

Los sistemas de memoria distribuida se conocen como clústeres. Se componen de una colección de elementos básicos de *hardware*, por ejemplo, servidores o PC conectados por una red de área local LAN, como Ethernet. De hecho, los nodos de estos sistemas son unidades computacionales individuales unidas por la red de comunicación. Suelen ser sistemas de memoria compartida con uno o más procesadores multinúcleo.

Las mallas computacionales proveen la infraestructura necesaria para convertir grandes redes de computadoras distribuidas geográficamente en un sistema unificado de memoria distribuida. En general, dicho sistema será heterogéneo, es decir, los nodos individuales se pueden construir a partir de diferentes tipos de *hardware*.

Los programas de memoria distribuida generalmente se ejecutan al iniciar múltiples procesos en lugar de múltiples subprocesos. Esto se debe a que los subprocesos de ejecución típicos en un programa de memoria distribuida pueden ejecutarse en CPU independientes con sistemas operativos independientes, y es posible que no haya una infraestructura de *software* para iniciar un solo proceso distribuido y que ese proceso se bifurque en uno o más subprocesos en cada nodo del sistema.

3.5. Memoria compartida

Un sistema de memoria compartida es una colección de procesadores autónomos que están conectados a un sistema de memoria a través del bus, y cada procesador puede acceder a cada ubicación de memoria. Así, los procesadores generalmente se comunican implícitamente al acceder a estructuras de datos compartidas.

Los programas de memoria compartida generalmente se ejecutan en un súper servidor con varios procesos de múltiples subprocesos (hilos). Esto se debe a que los subprocesos de ejecución típicos en un programa de memoria compartida pueden ejecutarse en una sola CPU con el mismo sistema operativo.

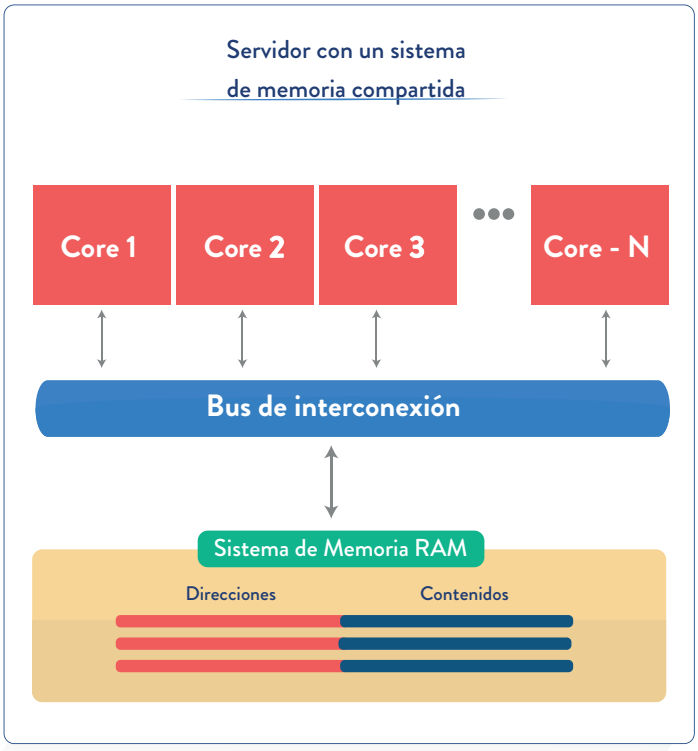


Figura 3. Sistema de memoria compartida o fuertemente acoplada

Fuente: elaboración propia

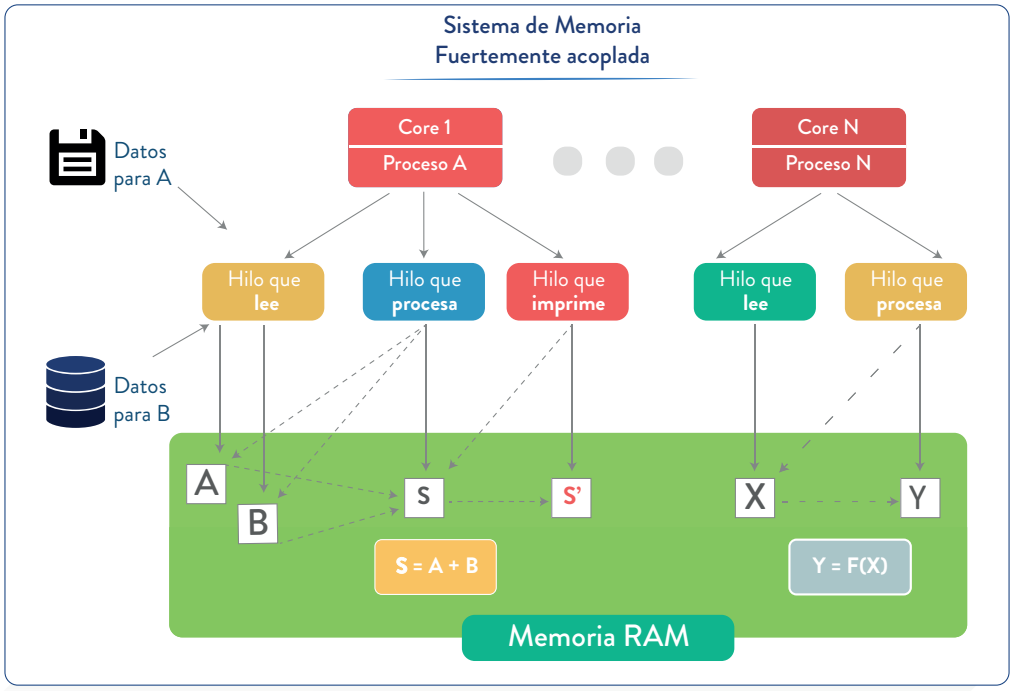


Figura 4. Sistema de memoria fuertemente acoplada - Hilos threads

Fuente: elaboración propia

3.6. Memoria distribuida Vs. memoria compartida

Los recién llegados a la computación paralela a veces se preguntan por qué todos los sistemas MIMD no son de memoria compartida, ya que la mayoría de los programadores encuentran el concepto de coordinación implícita, donde el trabajo de los procesadores se hace a través de estructuras de datos compartidas, más atractivo que el envío explícito de mensajes.

En este contexto, el problema principal del *hardware* es el costo de escalar la interconexión, pues a medida que se agregan procesadores a un bus (un super servidor) o a un clúster (muchos servidores) aumenta la posibilidad de que haya conflictos por el acceso al bus, por lo que los estos son adecuados para sistemas con solo unos pocos procesadores. Las barras cruzadas grandes son muy caras, por lo que también es poco común encontrar sistemas con interconexiones de barras cruzadas grandes.

Por otro lado, las interconexiones de memoria distribuida como el hipercubo y la malla toroidal son relativamente baratas, y se han construido sistemas de memoria distribuida con miles de procesadores que utilizan estas y otras interconexiones.

Por lo tanto, los sistemas de memoria distribuida a menudo son más adecuados para problemas que requieren grandes cantidades de datos o computación.

4. *Software* paralelo

El desarrollo de un programa necesita como mínimo abordar los problemas de equilibrio de carga, comunicaciones y sincronización entre los procesos o hilos. En los programas de memoria compartida, los hilos individuales pueden tener memoria privada y compartida y la comunicación se realiza habitualmente a través de variables compartidas.

En la mayoría de las API de memoria compartida, se da la exclusión mutua en una sección crítica que se puede imponer con un objeto llamado bloqueo de exclusión mutua o mutex. Las secciones críticas deben hacerse lo más pequeñas posible, ya que un mutex permita solo un hilo a la vez para ejecutar el código en la sección crítica.

La API más común para la programación de sistemas de memoria distribuida es paso de mensajes MPI. En el paso de mensajes hay, al menos, dos funciones distintas: envío *(send)* y recepción *(receive)*. Cuando los procesos necesitan comunicarse, uno llama el envío y el otro llama al recibidor.

Hay una variedad de comportamientos posibles para estas funciones. Por ejemplo, la función *send* puede bloquearse o esperar hasta que la función correspondiente de *receive* comience, o el *software* de paso de mensajes puede copiar los datos del mensaje en su propio almacenamiento, y el proceso de envío puede volver antes de que *receive* haya empezado a recibir. El comportamiento más común para recibir es bloquear hasta que el mensaje ha sido recibido. El sistema de paso de mensajes más utilizado se llama interfaz de paso de mensajes o MPI y proporciona una gran funcionalidad más allá de simplemente enviar y recibir.

La metodología de Foster proporciona una secuencia de pasos que se pueden utilizar para diseñar programas paralelos:

- La partición del problema para identificar tareas.
- La comunicación entre las tareas, la aglomeración o la agregación para agrupar las tareas.
- La asignación para asignar tareas agregadas a los procesos / subprocesos.

Recuerde que, en los programas de paso de mensajes, un programa que se ejecuta en la memoria central se suele denominar proceso, y dos procesos se pueden comunicar llamando a las funciones de envío y de recepción.

4.1. Lenguajes y métodos de compilación y ejecución

Rocks

Es una distribución especial de *software* libre sobre Linux para poder montar un clúster. Miles de investigadores de todo el mundo usan Rocks para poder montar sus clústeres.

Cuando no existía Rocks, era necesario instalar Scientific Linux y sobre ese sistema se tenía que montar un parche en el Kernel del sistema operacional que permitiera incrustar el módulo hpc que contiene el MPI. Luego, se montaba un monitor de discos como nagios, luego el oneSIS que es un paquete de *software* de código abierto destinado a simplificar la administración de clústeres sin disco. Después, se montaba el cobbler para aprovisionar servicios como dns, http, dhcp, etc. Luego, era necesario algún *software* libre para crear y administrar las redes, un monitor de procesos como nmon o ganglia, los compiladores gcc, fortran, c++, java, etc. Al final, se recompilaba todo el sistema operacional y se probaba hasta que funcionara.

Hoy con Rocks (http://www.rocksclusters.org/) el proceso es mucho más sencillo, pero además existen otras distribuciones como PelicanHPC (https://pelicanhpc.org), con las que se puede instalar un sistema HPC sin tanto desgaste.

MPI define una biblioteca de funciones que se pueden llamar desde los programas Java, Python, C, C ++ y Fortran. Y entre estos, los compiladores más utilizados para desarrollar programación paralela son: C, C ++ y Fortran.

Por lo general, los compiladores se ejecutan desde el sistema operacional Linux con una distribución acondicionada para procesamiento paralelo como Rocks.

Para compilar un programa sobre un ambiente HPC (*High performance Computing*), se ejecuta un *script* cuyo propósito principal es compilar un programa en de C, C++, Fortran, etc., que le dice al compilador dónde encontrar los archivos de encabezado necesarios y qué bibliotecas vincular con el archivo objeto.

Ejemplo: Compilar el programa HolaMundo.f (Fortran 90)

```
$ mpif90 HolaMundo.f   -o holamundo
```

Ejemplo: Compilar el programa HolaMundo.c

```
$ mpicc -g  -Wall  HolaMundo.c   -o holamundo
```

mpicc: indica que se va a compilar un programa desarrollado en gcc, (C para Linux GNU) que va a trabajar en un clúster.

- El signo $ es el indicador del Shell de Linux. Generalmente, se usan las siguientes opciones para el compilador:
- g. Crear información que permita utilizar un depurador.
- Wall. Emitir un montón de advertencias.
- o <out ile>. Nombre del archivo ejecutable

Ejemplo: Correr el programa ejecutable: holamundo, en un clúster de 6 nodos

Sintaxis: mpirun -np (número de servidores) . /ruta/objeto, donde np es el número de servidores (en este caso 6).

```
$ mpirun -np 4 ./home/proyecto/holamundo
```

Con cuatro procesos la salida del programa sería:

Los estoy saludando desde el proceso 0 de 4

Los estoy saludando desde el proceso 1 de 4

Los estoy saludando desde el proceso 2 de 4

Los estoy saludando desde el proceso 3 de 4

Los estoy saludando desde el proceso 4 de 4

Suponiendo que el programa imprime el mensaje "Saludos desde el proceso x de y".

Muchos sistemas también admiten la ejecución del programa con mpiexec, así:

```
$ mpiexec -n <número de procesos> ./ holaMundo
```

Nota: Para usar el compilador MPI de Java, se debe comprar la librería MPI para Java, dado que el SDK (Java estudio) no la trae.

4.2. Programación en memoria distribuida MPI

Para trabajar con desarrollo de programas de memoria distribuida, se debe empezar por crear un programa en gcc, para imprimir un Hola Mundo, como un programa serial y luego se hace el Hola Mundo utilizando MPI.

Hola Mundo serial o en programación clásica

```
#include <stdio.h> //Librería de entrada y salida

// este es un programa clásico de holaMundo en C, para un solo computador
en un solo Core

int main(void)
    {  printf("Hola mundo   \n");
       return 0;
   }
holaMundo1.c
```

Se compila así:

```
$ gcc holaMundo1.c   -o holamundito
```

Como se puede observar es un programa sencillo, pero solo va a correr por un servidor con un solo *core*.

Ahora si se crea el mismo programa holaMundo.c para una arquitectura paralela, el proceso cambia. En lugar de hacer que cada proceso simplemente imprima un mensaje, se designa un proceso para hacer la salida y los otros procesos le enviarán mensajes, que se imprimirán.

En la programación paralela es común que los procesos se identifiquen mediante rangos enteros no negativos. Entonces, si hay “k” procesos, los procesos tendrán rangos 0, 1, 2,..., k-1.

Para este "hola Mundo.c" paralelo, se plantea que el proceso 0 sea el proceso maestro, y los otros procesos le enviarán mensajes.

```
1 #include <stdio.h>
2 #include <string.h>  / * para strlen * /
3 #include <mpi.h>     / * Para las funciones MPI, etc * /
4
5  const int LONG_MENSAJE = 100;
6
7  int main(void) {
8            char saludando[LONG_MENSAJE];
9     int tamanno;       / * Número de procesos * /
10    int mi_rango;     / * Mi rango de proceso * /
11
12    MPI_Init(NULL, NULL);
13    MPI_Comm_size(MPI_COMM_WORLD, &tamanno);
14    MPI_Comm_rank(MPI_COMM_WORLD, &mi_rango);
15
16    if (mi_rango  = 0)  /* dijimos que el proceso 0, hace de maestro */
17                        /* aquí toma el mando el core maestro y recibe
      los mensajes de los nodos y los imprime */
18                        /* tamanno  es el número de cores que se
      especifican cuando se ejecute el proceso con */
18                        /* mpirun -np <número de procesos > ./holamundo */
20          { printf(""Saludos desde el proceso %d of %d! \n", mi_rango,
            tamaño);
21       for (int j = 1; j < tamanno; j++) {
22             MPI_Recv(saludando, MAX_STRING, MPI_CHAR, j, 0, MPI_COMM_
       WORLD, MPI_STATUS_IGNORE);
23           printf("%s \n",saludando);
24            }
25     esle /*NO SOY EL MAESTRO, SOY UN NODO */
26       { sprintf(saludando, "Los estoy saludando desde el proceso %d de
         f %d!", mi_rango, tamanno);
27         MPI_Send(saludando, strlen(saludando)+1, MPI_CHAR, 0, 0, MPI_COMM_
           WORLD);
28       }
29
30    MPI_Finalize();
31     return 0;
32
33  } /* end main*/
```

Programa 2. HolaMundo.c

Al revisar el programa 2, lo primero que hay que observar es que este es un programa en C. Por ejemplo, incluye los archivos de cabecera C estándar stdio.h (para manejar los procesos de entrada y salida I/O) y string.h. También tiene una función principal como cualquier otro programa de C. Sin embargo, hay muchas partes del programa que son nuevas.

Se incluye el archivo de cabecera mpi.h en la línea 3. Contiene los prototipos de las funciones MPI, definiciones de macros, definiciones de tipo, etc. Contiene todas las definiciones y declaraciones necesarias para compilar un programa MPI.

Lo segundo que hay que observar es que todos los identificadores definidos por MPI comienzan con la cadena MPI. La primera letra que sigue al subrayado se escribe con mayúscula para los nombres de las funciones y los tipos definidos por MPI. Todas las letras en macros y constantes definidas por MPI están en mayúsculas, por lo que no hay duda sobre lo que está definido por MPI y lo que está definido por el programa de usuario.

4.3. Librerías MPI

A continuación se van a explicar paso a paso las librerías que se utilizan en la construcción de un programa en paralelo, como las que se expusieron en el programa "HolaMundo.c" .

Los tipos de datos más usados en C para MPI son:

Tabla 1. Tipos de datos MPI

MPI_CHAR	con signo char
MPI_SHORT	con signo corto int
MPI_INT	con signo int
MPI_LONG	con signo LONG int
MPI_LONG_LONG	con signo LONG LONG int
MPI_UNSIGNED	CHAR Sin signo char
MPI_UNSIGNED	SHORT Sin signo short int
MPI_UNSIGNED	Sin signo int

MPI_UNSIGNED	LONG Sin signo long int
MPI_FLOAT	Float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long doble
MPI_BYTE	
MPI_PACKED	

Fuente: elaboración propia

4.3.1. MPI_Init y MPI_Finalize

En la línea 12 del programa HolaMundo.c de la página 15 de esta lectura, la llamada a MPI_Init le dice al sistema MPI que realice toda la configuración necesaria.

Por ejemplo, podría asignar almacenamiento para los buffers de mensajes y podría decidir qué proceso obtiene qué rango. Como regla general, no se debe llamar a ninguna otra función MPI antes de que el programa llame a MPI_Init. Su sintaxis es:

```
int MPI_Init (  int_argc_p /* in/out */,
                char_ar    /* in/out */
  );
```

Los argumentos, argc_p y argv_p son punteros a los argumentos de main, argc y argv. Sin embargo, cuando nuestro programa no usa estos argumentos, solo se puede pasar NULL para ambos. Como la mayoría de las funciones de MPI, MPI_Init devuelve un código de error int, en la mayoría de los casos estos códigos de error se ignoran.

En la línea 30, la llamada a MPI_Finalize() le dice al sistema MPI que se ha terminado con MPI y que se pueden liberar todos los recursos asignados para MPI. La sintaxis es bastante sencilla:

```
int MPI_Finalize (void);
```

En general, no se debe llamar a las funciones MPI después de finalizar la llamada a MPI.

4.3.2. Comunicadores MPI_Comm_size y MPI_Comm_rank

En MPI, un comunicador es una colección de procesos que pueden enviarse mensajes entre sí. Uno de los propósitos de MPI_Init es definir este comunicador que consiste en todos los procesos iniciados por el usuario cuando inició el programa.

Este comunicador se llama MPI_COMM_WORLD. Las llamadas de función en las Líneas 13 y 14 del programa HolaMundo.c están obteniendo información sobre MPI_COMM_WORLD. Su sintaxis es:

```
int MPI_Comm_size(
  MPI_Comm comm      /* in  */,
  int*   com_sz_p   /* out */);
int MPI_Comm_rank(
  MPI_Comm comm      /* in  */,

  int*   my_rank_p  /* out */);
```

Para ambas funciones, el primer argumento es un comunicador y tiene el tipo especial definido por MPI para comunicadores MPI_Comm. El tamaño de MPI_Comm devuelve en su segundo argumento el número de procesos en el comunicador, y el rango de MPI_Comm, devuelve en su segundo argumento la clasificación del proceso de llamada en el comunicador. A menudo se usa la variable
`comm_sz` para el número de procesos en MPI_COMM_WORLD y la Variable mi_rango para el rango de proceso.

En el programa HolaMundo.c, en las líneas 26 y 27 cada uno de los core, que no son máster, crean un mensaje que enviará al proceso 0 (al maestro). (La función sprintf es muy similar a printf, excepto que en lugar de escribir en stdout, escribe en una cadena). La Línea 26 en realidad envía el mensaje al proceso 0. El proceso 0, por otro lado, simplemente imprime su mensaje usando printf, y luego usa un bucle ”for” para recibir e imprimir los mensajes enviados por los procesos 1, 2,..., tamanno o comm_zs_p. La línea 22, recibe el mensaje enviado por el proceso j, para j = 1, 2,.., comm_sz_p.

4.3.3. MPI_Send

Los envíos ejecutados por los procesos 1, 2,..., tamaño, son bastante complejos, así que demos un vistazo más de cerca. Cada uno de los envíos se realiza mediante una llamada a MPI_Send, para enviar lados de respuesta al máster o del máster a los nodos. La sintaxis es:

```
int MPI_Send(
  void*         msg_buf_p      /* in  */,
  int           msg_size       /* in  */,
  MPI_Datatype  msg_type       /* in  */,
  int           dest           /* in  */,
  int           tag            /* in  */,
  MPI_Comm      communicator   /* in  */);
// "in" quiere decir: de entrada
```

Los primeros tres argumentos, msg_buf_p, msg_size y msg_type, determinan el contenido del mensaje. Los argumentos restantes, dest, tag y comunicador determinan el destino del mensaje.

El primer argumento, msg_buf_p, es un puntero al bloque de memoria que contiene el contenido del mensaje. En nuestro programa, este es solo la cadena que contiene el mensaje, saludo. (Recuerde que en C una matriz, como una cadena, es un puntero).

El segundo y tercer argumento, tamaño de mensaje y tipo de mensaje, determinan la cantidad de datos que se enviarán. En nuestro programa, el tamaño del argumento del mensaje es el número de caracteres en el mensaje más un carácter para el carácter "\0" que finaliza las cadenas en C. El tipo del argumento msg es MPI_CHAR. Estos dos argumentos juntos le dicen al sistema que el mensaje contiene strlen (saludo)+1 caracteres.

Como los tipos C (int, char, etc.) no se pueden pasar como argumentos a funciones, MPI define un tipo especial, el tipo de datos MPI, que se usa para el argumento de tipo msg.

MPI también define una serie de valores constantes para este tipo. Los más usados (y algunos otros) se enumeran en la Tabla 1.

Observe en el programa HolaMundo.c que el tamaño de la cadena saludo, no es el mismo que el tamaño de los mensajes especificados y el tipo de mensaje de los argumentos msg. Por ejemplo, cuando ejecutamos el programa con seis procesos, la longitud de cada uno de los mensajes es de 3 caracteres, mientras que hemos asignado almacenamiento para 100 caracteres en saludos. Por supuesto, el tamaño del mensaje enviado debe ser menor o igual a la cantidad de almacenamiento en el búfer, en nuestro caso, la cadena saludo.

El cuarto argumento, dest, especifica el rango del proceso (es la cédula del proceso que le asignó el Core máster) que debe recibir el mensaje. El quinto argumento, la etiqueta, es un int no negativo. Puede usarse para distinguir mensajes que, de lo contrario, son idénticos. Por ejemplo, supongamos que el proceso 1 está enviando flotadores al proceso 0. Algunos de los flotadores deben imprimirse, mientras que otros deben usarse en un cálculo. Luego, los primeros cuatro argumentos de MPI_Send no proporcionan información sobre qué flotantes deben imprimirse y cuáles deben utilizarse en un cálculo. Por lo tanto, el proceso 1 puede usar, digamos, una etiqueta de 0 para los mensajes que deben imprimirse y una etiqueta de 1 para los mensajes que deben usarse en un cálculo.

El argumento final para enviar MPI es un comunicador. Todas las funciones MPI que involucran la comunicación tienen un argumento comunicador. Uno de los propósitos más importantes de los comunicadores es especificar universos de comunicación; recuerde que un comunicador es una colección de procesos que pueden enviarse mensajes entre sí. A la inversa, un mensaje que se envía mediante un proceso que utiliza un comunicador no puede ser recibido por un proceso que utiliza un comunicador diferente. Como MPI proporciona funciones para crear nuevos comunicadores, esta función se puede usar en programas complejos para asegurar que los mensajes no se reciban "accidentalmente" en el lugar equivocado.

4.3.4. MPI_Recv

Los primeros seis argumentos de MPI_Recv corresponden a los primeros seis argumentos de MPI_Send:

```
int MPI_Recv(
  void*          msg_buf_p      /* out */,
  int            buf_size       /* in  */,
  MPI_Datatype   buf_type       /* in  */,
  int            source         /* in  */,
  int            tag            /* in  */,

  MPI_Status*    status_p       /* out */);
```

Figura 7. MPI _Recv
Fuente: elaboración propia

Por lo tanto, los tres primeros argumentos especifican la memoria disponible para recibir el mensaje: msg_buf_p apunta al bloque de memoria, el tamaño de buf_size determina el número de objetos que se pueden almacenar en el bloque y el tipo de buf_type indica el tipo de los objetos. Los siguientes tres argumentos identifican el mensaje. El argumento fuente (*source*) especifica el proceso y el *Core* que lo envió o sea desde el cual se debe recibir el mensaje. El argumento de etiqueta (*tag*) debe coincidir con el argumento de etiqueta del mensaje que se está enviando y el argumento del comunicador debe coincidir con el comunicador utilizado por el proceso de envío.

Hablaremos sobre el argumento del status_p en breve. En muchos casos, no será utilizado por la función de llamada y, como en nuestro programa de "saludos", la constante MPI especial MPI_STATUS_IGNORE se puede pasar.

4.3.5. Coincidencia de mensajes

Supongamos un proceso “q” llama MPI_Send con:

```
MPI_Send (send_buf_p, send_buf_sz, send_type, dest, send_tag, send_comm);
```

Ahora suponga que el proceso “r” llama a MPI_Recv con:

```
MPI_Recv (recv_buf_p, recv_buf_sz, recv_type, src, recv_tag, recv_comm, &status);
```

Entonces, el mensaje enviado por “q” con la llamada anterior a MPI_Send puede ser recibido por ”r” con la llamada a MPI_Recv, sí:

- `Recv_comm = send_comm,`
- `Recv_tag = send_tag,`
- `dest =r,`
- `src = q`

Sin embargo, estas condiciones no son suficientes para que el mensaje se reciba con éxito. Los parámetros especificados por los tres primeros pares de argumentos, send_buf_p/recv_buf_p, send_buf_sz/recv_buf_sz, y send_type/recv_type, deben especificar buffers compatibles. La mayoría de las veces, la siguiente regla será suficiente:

Si ***recv_type = send_type y recv_buf_sz >= send_buf_sz***, enconces el mensaje enviado por ***"q"*** puede ser recibido por ***"r"*** de manera correcta.

Puede suceder que un proceso esté recibiendo mensajes de otros procesos y no sepa cual es el orden, entonces se confunde, para evitar esto, MPI provee una librería que nos ayuda en este sentido, veamos un ejemplo:

```
for ( i = 1; i < comm_sz; i++){
    MPI_Recv (result, results_sz, result_type, MPI_ANY_SOURCE,
  result_tag, com, MPI_STATUS_IGNORE);
    Process_result(result);
}
```

De manera similar, es posible que un proceso pueda recibir múltiples mensajes con etiquetas diferentes de otro proceso, y el proceso de recepción no sepa el orden en que se enviarán los mensajes. Para esta circunstancia, MPI proporciona la constante especial MPI_ANY_TAG que se puede pasar al argumento de etiqueta de MPI_Recv.

Un par de puntos deben destacarse en relación con estos argumentos de "comodín":

1. Solo un receptor puede usar un argumento comodín. Los remitentes deben especificar un rango de proceso y una etiqueta no negativa
2. No hay comodines para los argumentos del comunicador; Tanto los remitentes como los receptores siempre deben especificar comunicadores.

4.3.6. El argumento STATUS

Note que un receptor puede recibir un mensaje sin saberlo:

1. La cantidad de datos en el mensaje,
2. El remitente del mensaje, o
3. La etiqueta del mensaje.

Entonces, ¿cómo puede el receptor averiguar estos valores?. Recuerde que el último argumento de MPI_Recv tiene el tipo MPI_Status *. El tipo MPI MPI_Status, es una estructura con al menos los tres miembros MPI_SOURCE, MPI_TAG y MPI_ERROR. Supongamos que nuestro programa contiene la definición: estado de:

MPI_status status;

Luego, después de una llamada a MPI_Recv en la que &status se pasa como el último argumento, podemos determinar el remitente y la etiqueta examinando el estado de los dos miembros.

status.MPI_SOURCE

status.MPI_TAG

y de esta manera se soluciona el problema.

Nota: a continuación, se presenta un taller de autoaprendizaje, realícelo de forma autónoma para que usted mismo pueda comprobar los conocimientos adquiridos hasta el momento, tenga en cuenta que este taller no es evaluativo ni tendrá una retroalimentación durante el desarrollo del Módulo.

Utilizando los fundamentos del cálculo de un área bajo la curva, que consiste en dividir el área de la curva de la figura 5.a, en trapecios sucesivos como se muestra en la figura 5b, mediante sumas sucesivas, lo cual nos lleva al uso de la integración de la función entre los límites a y b de f(x)dx.

Para que aprenda haciendo, usted deber calcular la integral de la función y = f(x)dx como se muestra en la figura 5, entre los límites a y b.

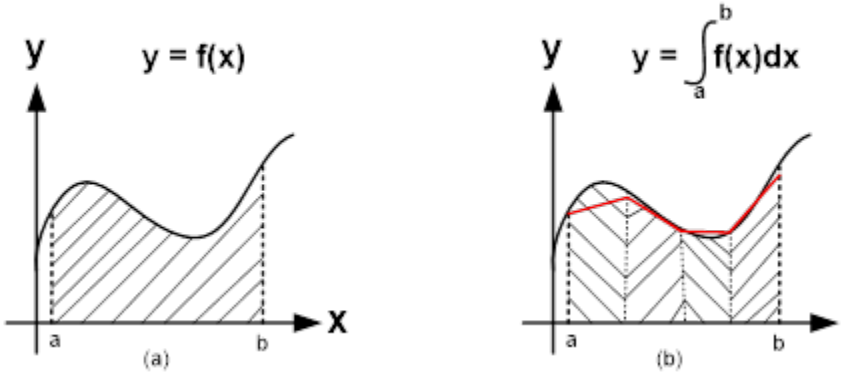


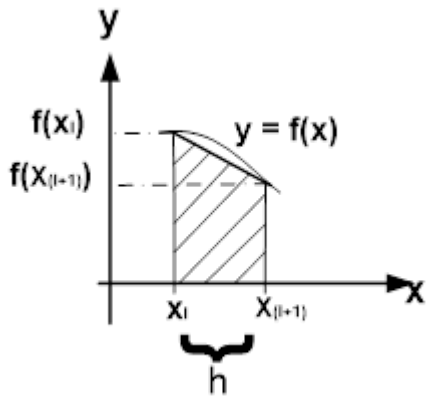
Figura 5. Regla trapezoidal usando integración

Fuente: elaboración propia

Recuerde que podemos usar la regla trapezoidal para aproximar el área entre la gráfica de una función y = f(x) entre los límites a y b.

El área de un trapecio se calcula como:

Área = h/2[f(x(i)) + f(x(i+1))]



Parámetros:

a = valor inicial del intervalo a evaluar

b = valor final del intervalo a evaluar

n = número total de trapecios

h = base de cada trapecio

Figura 6. Un trapecio del área bajo la curva

Fuente: elaboración propia

Revisando la figura 6, se ve que se tienen "n" subintervalos para que todos los trapecios tengan la misma longitud de la altura h, donde x(i) y x(i+1) que son las líneas verticales, son las bases del trapecio, entones h = (b – a) / 2. Por lo tanto, si se llama el punto más a la izquierda x(0), y el punto extremo más a la derecha x(n), se tiene:

$$x_0 = a, x_1 = a + h, x_2 = a + 2h, \dots, x_n - 1 = a + (n-1)h, x_n = b$$

Se calculan las áreas de los trapezoides, se suman y obtenemos una aproximación al área total, así que visto como funciones, la suma es:

$$áreas = h[f\frac{x_0}{2} + f(x_1) + f(x_2) + \cdots + f(x_n - 1) + f(x_n)/2]$$

Por lo tanto, el pseudocódigo para un programa en serie de la figura 5a, podría tener este aspecto:

```
 /* Input: a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++){
   x_1 = a + i*h;
   approx += f(x_1)
}

approx = h*aprrox;
```

Quienes escriben programas paralelos usan el verbo "paralelizar" para describir el proceso de conversión de un programa o algoritmo serial en un programa paralelo. Se puede diseñar un programa paralelo usando cuatro pasos básicos:

1. Particionar la solución del problema en tareas.
2. Identificar los canales de comunicación entre las tareas.
3. Agregue las tareas (programa serial) en tareas compuestas.
4. Mapear las tareas compuestas a los núcleos.

En la fase de partición, generalmente se intenta identificar tantas tareas como sea posible. Para la regla trapezoidal, podemos identificar dos tipos de tareas: un tipo es encontrar el área de un solo trapecio y el otro es calcular la suma de estas áreas. Luego, los canales de comunicación unirán cada una de las tareas del primer tipo a la tarea única del segundo tipo.

Entonces, ¿cómo se pueden agregar las tareas y mapearlas a los núcleos? Por las reglas de las integrales se sabe que entre más trapecios se usan, más precisa será la estimación. Es decir, se deben

usar muchos trapezoides y, a menudo, serán muchos más trapezoides que núcleos. Por lo tanto, es necesario agregar el cálculo de las áreas de los trapezoides en grupos. Una forma natural de hacer esto es dividir el intervalo [a, b] en subintervalos de com_sz. Si comm_sz se divide uniformemente en "n" número de trapecios, y es posible simplemente aplicar la regla trapezoidal con "n = com_sz" trapezoides a cada uno de los subintervalos de comunicación.

Para finalizar, se puede tener uno de los procesos, proceso 0, y agregar las estimaciones. Suponga que la comunicación simplificada divide n. Entonces, el pseudocódigo para el programa podría tener un aspecto similar al siguiente:

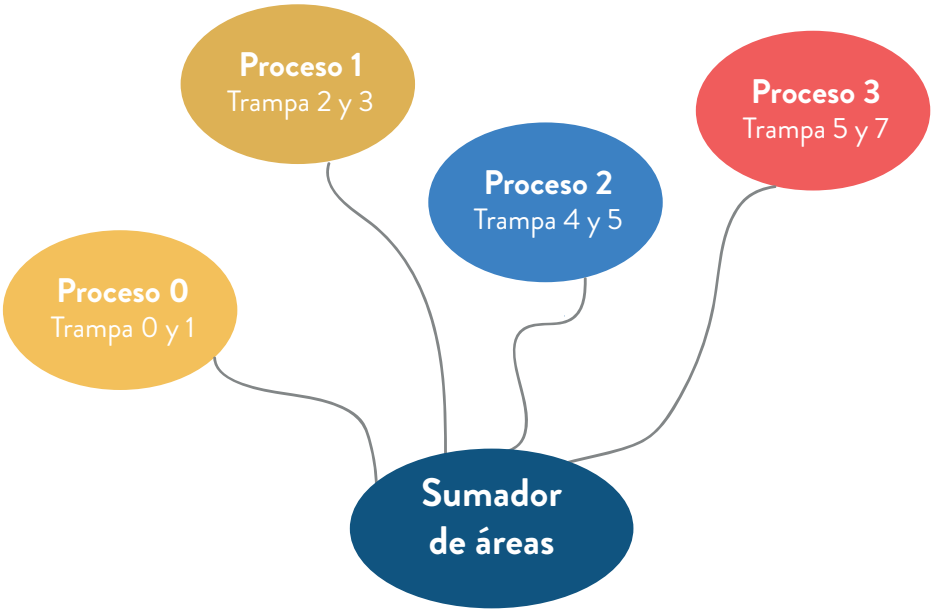


Figura 7. Trampas para agrupar áreas por proceso

Fuente: elaboración propia

```
1      Get a, b, n;
2      h = (b-a)/n;
3      local_n = n/comm_sz;
4      local_a = a + my_rank*local_n*h;
5      local_b = local_a + local_n*h;
6      local_integral = Trap(local_a, local_b, local_n, h);
```

En la Figura 7, se puede observar que solo se tiene un conjunto de 4 servidores, pero que se hizo un rango de n = 8, por lo cual es necesario enviar a cada procesador, dos tareas que se agrupan mediante un parámetro llamado trampa y cuando se hayan calculado las dos áreas en cada servidor, se van enviando al sumador maestro para que calcule el área total.

```
7      if (my_rank != 0)
8         Send local_integral to process 0;
9      else /* muy_rank == 0 */
10        total_integral = local_integral:
11        for (proc = 1; proc > comm_sz; proc +++){
12          Receive locla_integral form proc;
13            total_integral += local_integral;
14        }
15     }
16     if (my_rank == 0)

17        print result;
```

Reto. Revise las referencias y en su infraestructura virtual para un máster y dos cores (tres máquinas virtuales), convierta el seudocódigo en código en C paralelo y hágalo correr.

4.4. Programación en memoria compartida openMP

Ya aprendió a modelar, desarrollar y ejecutar un programa construido bajo el paradigma de programación paralela que debe correr en varios servidores, para lo cual se usa MPI como orquestador para el paso de mensajes, que corre sobre arquitecturas como la expuesta anteriormente.

Para desarrollar usando hilos paralelos, es decir, para desarrollar *software* paralelo para arquitecturas diferentes no se puede usar MPI, porque no se trata de sistemas débilmente acoplados, sino en un servidor con muchos *cores*, por lo cual se debe usar openMP en vez de MPI, lo que quiere decir que en el encabezado del programa, no se emplea <mpi.h> sino que se incluye la librería <ptrhread>.

4.4.1. Ejecución de un programa para memoria compartida en openMP

El programa se compila como un programa C normal, con la posible excepción que es posible que debamos vincular en la biblioteca de Pthreads. Observe:

```
$ gcc -g  -Wall holaMundoPthread.c   -o holaMunditoHilos
```

Se ejecuta con el comando: $ ./programa_ejecutable <número de cores>

```
$  ./holaMunditoHilos 6
```

A continuación, en la Figura 8 se muestran las imágenes de un programa holaMundoHilos.c creado en un editor C en línea, para mostrar la manera como se escribe un simple programa de hola mundo en C, para correr con hilos en paralelo llamados Pthreads.

```c
/******************************************************************************
 * Online C Compiler.
 *                Code, Compile, Run and Debug C program online.
Write your code in this editor and press "Run" button to compile and execute it.

*******************************************************************************/

#include <stdio.h>
 #include <stdlib.h>
 #include <pthread.h>

 /* variables glonales: acceso a todos los threads */
 int thread count;

void* Hello(void* rank); /*  funcion Thread */

 int main(int argc, char* argv[ ]) { long thread; /* Use long in caso de un sistema de  64 bits */
 pthread t* thread handles;

  /* tome el numero de hilos de la linea de comandos */
  thread count = strtol(argv[1], NULL, 10);

  thread handles = malloc (thread count*sizeof(pthread t));

  for (thread = 0; thread < thread count; thread++)
  pthread create(&thread handles[thread], NULL,  Hello, (void*) thread);

  printf("Salaundado desde el hilo \n");

  for (thread = 0; thread < thread count; thread++)
  pthread join(thread handles[thread], NULL);

  free(thread handles);
  return 0;
  }  /* fin del main */

  void* Hello(void* rank) {
      long my rank = (long) rank
/* Use long un caso de un sistema de64 bits */

  printf("Saludando desde el hilo %ld of %d \n", my rank, thread count);
   return NULL;
 } /* fin HolaMundo*/
```

Figura 8. Programa holaMundoHilos.c usando Pthreads

Fuente: Rocks

Referencias bibliográficas

Tesis de programación paralela: http://www.josecc.net/archivos/tesis/tesis_html1/node13.html

Prototipo de un programa de trapecios en paralelo.

http://selkie-macalester.org/csinparallel/modules/MPIProgramming/build/html/trapezoidIntegration/trapezoid.html

Dirección para bajar el proyecto de software libre de Rocks clúster. http://www.rocksclusters.org/

Robert, Y y Shende, S et al. (2011). *Encyclopedia of Parallel Computing.* Recuperado de: https://books.google.com.co/books?id=Hm6LaufVKFEC&lpg=PA592&ots=uEBUeWzbiR&dq=Parallel%20computing%20IBM%20REDBOOK&hl=es&pg=PA878#v=onepage&q&f=false

Thomas Rauber, G. (s.f.) *Parallel Programming for Multicore and Cluster Systems-Springer*. Verlag Berlin Heidelberg.

INFORMACIÓN TÉCNICA

FACULTAD DE
INGENIERÍA, DISEÑO
E INNOVACIÓN

Módulo: Sistema Distribuidos

Unidad 3: Computación en cluster y programación paralela

Escenario 6: Procesamiento y computación paralela

Autor: Alexis Rojas

Asesor Pedagógico: Jeimy Lorena Romero Perilla

Diseñador Gráfico: Brandon Steven Ramírez Carrero

Asistente: Maria Elizabeth Avilán Forero