4. Programación Paralela

La necesidad que surge para resolver problemas que requieren tiempo elevado de cómputo origina lo que hoy se conoce como computación paralela. Mediante el uso concurrente de varios procesadores se resuelven problemas de manera más rápida que lo que se puede realizar con un solo procesador.
Una computadora paralela es un conjunto de procesadores que son capaces de trabajar cooperativamente para solucionar un problema computacional. Esta definición es muy extensa e incluye supercomputadoras que tienen cientos o miles de procesadores, redes de estaciones de trabajo o estaciones de trabajo con múltiples procesadores.

4.1 Diseño de algoritmos paralelos

Los algoritmos paralelos son extremadamente importantes para solucionar problemas grandes para muchos campos de aplicación. En esta sección se describen las etapas típicas para el diseño de los algoritmos paralelos:

1. Particionamiento. Los cálculos se descomponen en pequeñas tareas. Usualmente es independiente de la arquitectura o del modelo de programación. Un buen particionamiento divide tanto los cálculos asociados con el problema como los datos sobre los cuales opera.

2. Comunicación. Las tareas generadas por una partición están propuestas para ejecutarse concurrentemente pero no pueden, en general, ejecutarse independientemente. Los cálculos en la ejecución de una tarea normalmente requerirán de datos asociados con otras tareas. Los datos deben transferirse entre las tareas y así permitir que los cálculos procedan. Este flujo de información se especifica en esta fase.

3. Aglomeración. Las tareas y las estructuras de comunicación definidas en las dos primeras etapas del diseño son evaluadas con respecto a los requerimientos de ejecución y costos de implementación. Si es necesario, las tareas son combinadas en tareas más grandes para mejorar la ejecución o para reducir los costos de comunicación y sincronización.

4. Mapeo. Cada tarea es asignada a un procesador de tal modo que intente satisfacer las metas de competencia al maximizar la utilización del procesador y minimizar los costos de comunicación.

Estas cuatro etapas se ilustran en la Figura 4.1.

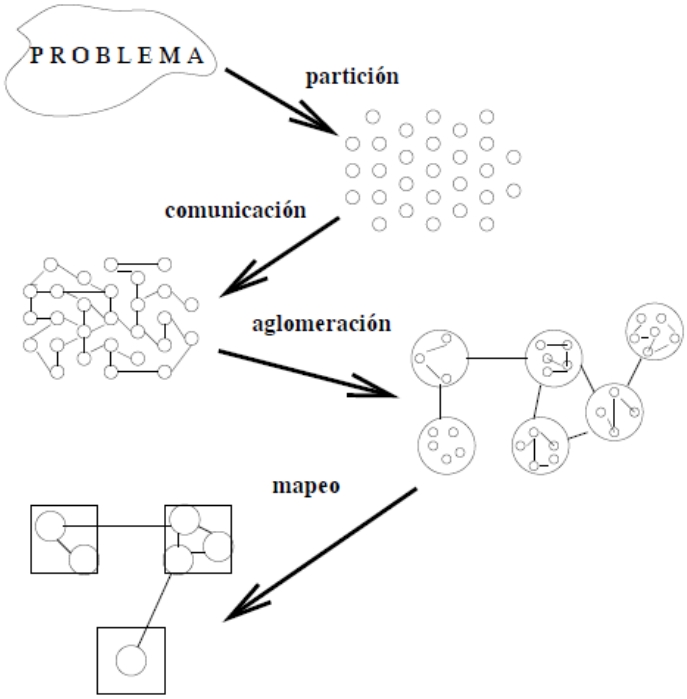

Figura 4.1: PCAM (Particionamiento, Comunicación, Aglomeración y Mapeo): una metodología de diseño para programas paralelos. Iniciando con la especificación de un problema, desarrollando una partición, determinando los requerimientos de comunicación, las tareas aglomeradas, y finalmente el mapeo de tareas a procesadores.

4.2 Arquitecturas

En los sistemas de memoria distribuida, cada procesador tiene acceso a su propia memoria, entonces la programación es más compleja ya que cuando los datos a usar por un procesador están en el espacio de direcciones de otro, es necesario solicitarla y transferirla a través de mensajes. De esta forma, es necesario impulsar la localidad de los datos para minimizar la comunicación entre procesadores y obtener un buen rendimiento. La ventaja que proporcionan estos tipos de sistemas es la escalabilidad, es decir, el sistema puede crecer un número mayor de procesadores que los sistemas de memoria compartida y, por lo tanto, es más adecuado para las computadoras paralelas con un gran número de procesadores.
Sus principales desventajas son que el código y los datos tienen que ser transferidos físicamente a la memoria local de cada nodo antes de su ejecución, y esta acción puede constituir un trabajo adicional significativo. Similarmente, los resultados necesitan transferirse de los nodos al sistema *host*. Los datos son difíciles de compartir. Las arquitecturas de multicomputadoras son generalmente menos flexibles que las arquitecturas de multiprocesadores. Por ejemplo los multiprocesadores podrían emular el paso de mensajes al usar localidades compartidas para almacenar mensajes, pero las multicomputadoras son muy ineficientes para emular operaciones de memoria compartida.
Los programas desarrollados para multicomputadoras pueden ejecutarse eficientemente en

multiprocesadores, porque la memoria compartida permite una implementación eficiente de paso de mensajes. Ejemplos de esta clase de computadoras son la Silicon Graphs Challenge, Sequent Symmetry, y muchas estaciones de trabajo con multiprocesadores.

En la última fase del diseño de algoritmos paralelos (mapeo) lo que se toma en cuenta es la arquitectura de la computadora paralela, debido a que los algoritmos diseñados para la memoria compartida presentan otras características que los diferencia de los algoritmos diseñados para arquitecturas de memoria distribuida. En la siguiente sección se muestra la programación aplicada a cada arquitectura.

4.3 Programación para computadoras paralelas

La programación paralela es diferente para cada arquitectura. La programación con hilos se aplica principalmente a las arquitecturas de multiprocesadores con memoria compartida y la programación con paso de mensajes a las arquitecturas de multicomputadoras. A continuación se hace una revisión de los tipos de programación utilizados en cada arquitectura.

4.3.1 Programación para memoria compartida

En los sistemas de multiprocesadores, cada procesador puede acceder a toda la memoria, es decir, hay un espacio de direccionamiento compartido. Todos los procesadores se encuentran igualmente comunicados con la memoria principal y pueden acceder por medio de un ducto común. En esta configuración se debe asegurar que los procesadores no accedan de manera simultánea a las regiones de memoria de tal manera que se provoque un error. Por ejemplo, si dos o más procesadores desean actualizar una variable compartida se deben establecer procedimientos que permitan acceder a los datos compartidos como exclusión mutua. Esto se logra mediante el uso de candados o semáforos y mecanismos de sincronización explícitos o implícitos como las barreras. Debido a que el modelo de programación en estos sistemas usa un solo espacio de direccionamiento la comunicación entre procesadores es implícita.

La programación en estos sistemas es más sencilla, ya que los datos se pueden colocar en cualquier módulo de la memoria y los procesadores la pueden acceder de manera uniforme. Se pueden utilizar bibliotecas de funciones con un lenguaje secuencial, para programar multiprocesadores, tal es el caso de pthreads, que es una especificación estándar para soporte de *multithreading* a nivel de sistema operativo [5].

En un lenguaje de alto nivel normalmente se programa un hilo empleando procedimientos, donde las llamadas a procedimientos siguen la disciplina tradicional de pila. Con un único hilo, existe en cualquier instante un único punto de ejecución. El programador no necesita aprender nada nuevo para emplear un único hilo. Cuando se tienen múltiples hilos en un programa significa que en cualquier instante el programa tiene múltiples puntos de ejecución, uno en cada uno de sus hilos. El programador puede ver a los hilos como si estuvieran en ejecución simultánea, como si la computadora tuviese tantos procesadores como hilos en ejecución. El programador decide cuando y donde crear múltiples hilos, ayudándose de un paquete de bibliotecas o un sistema de tiempo de ejecución; pero debe

estar consciente de que la computadora no necesariamente ejecuta todos los hilos simultáneamente.
Los hilos se ejecutan en un único espacio de direcciones, lo que significa que el hardware de direccionamiento de la computadora está configurado para permitir que los hilos lean y escriban en las mismas posiciones de memoria. En un lenguaje de alto nivel, esto corresponde normalmente al hecho de que las variables globales (fuera de la pila) son compartidas por todos los hilos del programa. Cada hilo se ejecuta en una pila separada con su propio conjunto de variables locales y el programador es el responsable de emplear los mecanismos de sincronización del paquete de hilos para garantizar que la memoria compartida es accedida de forma correcta.
Un hilo es un concepto sencillo : un simple flujo de control secuencial. Un *thread* (hilo) se define como un proceso ligero que comparte el mismo espacio de memoria y variables globales que el proceso a partir del cual se crea. La creación de hilos es una operación más rápida que la creación de procesos. Un hilo tiene acceso inmediato a variables globales compartidas y su sincronización es más eficiente que la sincronización de procesos, realizándose con una sola variable.
El acceso a datos compartidos tiene que hacerse de acuerdo a las siguientes reglas: primero, la lectura múltiple del valor de una variable no causa conflictos; segundo, la escritura a una variable compartida tiene que hacerse con exclusión mutua. El acceso a secciones críticas se puede controlar mediante una variable compartida. El mecanismo más simple para asegurar exclusión mutua de secciones críticas es el uso de un candado (*lock*). Un *lock* en Pthreads se implementa con variables de exclusión mutua.

4.3.2 Programación con paso de mensajes

Si la memoria está distribuida entre los procesadores, es decir, cada procesador tiene acceso a su propia memoria, entonces la programación es más compleja ya que cuando los datos a usar por un procesador están en el espacio de direcciones de otro, será necesario solicitarla y transferirla a través de mensajes. De este modo, es necesario impulsar la localidad de los datos para minimizar la comunicación entre procesadores y obtener un buen rendimiento.

Los programas con paso de mensajes, crean múltiples tareas; cada tarea encapsula un dato local e interactúa mediante el envío y recibo de mensajes. En los sistemas de multicomputadoras, el código de cada procesador se carga en la memoria local y algunos de los datos requeridos se almacenan localmente. Los programas se particionan en partes separadas y se ejecutan concurrentemente por procesadores individuales. Cuando los procesadores necesitan acceder a la información de otros procesadores, o enviar información a otros procesadores, se comunican mediante el envío de mensajes.

El paso de mensajes se ha usado como un medio de comunicación y sincronización entre una colección arbitraria de procesadores, incluyendo un solo procesador. Los programas que usan paso de mensajes son totalmente estructurados. Frecuentemente, todos los nodos ejecutan copias idénticas de un programa, con el mismo código y variables privadas (modelo SPMD).

La ventaja que presentan estos sistemas es su escalabilidad, es decir, el sistema puede crecer un número mayor de procesadores que los sistemas de memoria compartida y, por lo tanto, es más adecuado para las computadoras paralelas.

Los procedimientos de envío y recepción de mensajes en los sistemas de paso de mensajes frecuentemente tienen la siguiente forma:

send(parámetros)
recv(parámetros)

donde los parámetros identifican los procesadores fuente y destino, y los datos. Estas rutinas pueden dividirse en dos tipos: síncrono o bloqueante y asíncrono o no bloqueante.

Las rutinas síncrono o bloqueante no permiten que los procesos continúen hasta que la operación ha sido completada. Las rutinas asíncronas o no bloqueantes permiten que los procesos continúen a pesar de que la operación no se haya terminado; es decir, las instrucciones que continúan de la rutina se ejecutan a pesar de que la rutina no haya sido concluida. Las rutinas de envío bloqueante esperaran hasta que el mensaje completo haya sido transmitido y aceptado por el proceso receptor. Una rutina de recepción bloqueante esperará hasta que el mensaje esperado es recibido. Las rutinas bloqueantes ejecutan dos acciones: la transferencia de datos y la sincronización de procesos.

Las rutinas de envío de paso de mensajes no bloqueantes permiten a los procesos continuar inmediatamente después de que el mensaje ha sido construido sin esperar por la aceptación o el recibo. Una rutina de recibo no bloqueante no esperará el mensaje y permitirá a los procesos continuar. Las rutinas no bloqueantes generalmente decrementan el proceso de tiempo de ejecución. El paso de mensajes no bloqueante implica que las rutinas tengan buffers para almacenarlos mensajes, pero estos son de longitud finita y podrían llegar a bloquearse cuando el buffer este lleno.
Una de las herramientas para programación con paso de mensajes más utilizada es MPI (Message Passing Interface) que es la especificación estándar de un conjunto de funciones para paso de mensajes.

4.4 Esquemas de comunicación

Se describen a continuación dos tipos de comunicación que se emplean en la paralelización de algoritmos.

4.4.1 Maestro-esclavo o comunicación global

Este tipo de comunicación consiste en la implementación de un componente principal llamado *maestro* que se encarga de recolectar la información procesada por cada *esclavo* (componente secundario) y de distribuirla en su totalidad a cada uno de ellos. Permite el procesamiento en paralelo, pues cada esclavo trabaja independientemente de los demás. También, se le conoce como comunicación global, porque la información se concentra en un procesador (maestro). La Figura 4.2 muestra el comportamiento de este esquema de comunicación.

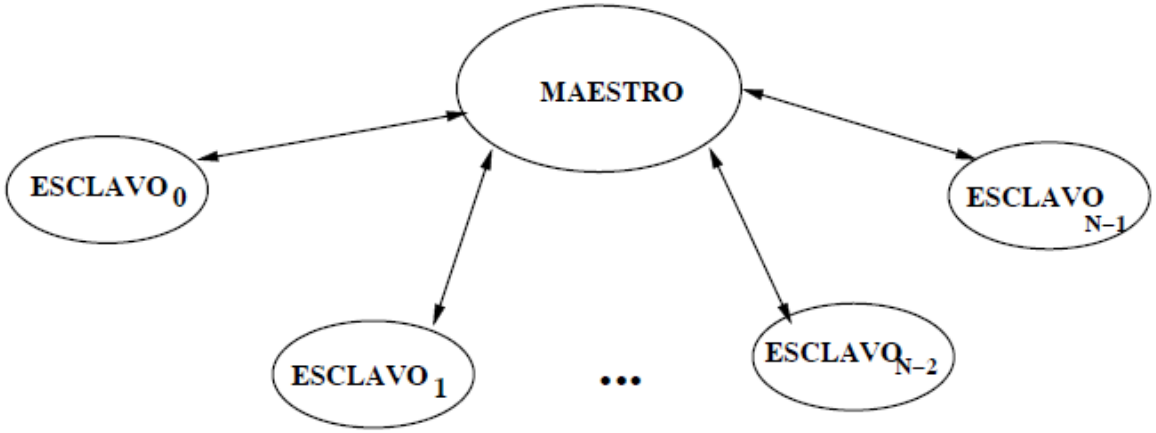

Figura 4.2: Esquema Maestro-Esclavo.

4.4.2 Esquema SPMD o comunicación local

Este esquema utiliza el modelo Single Program Multiple Data (Programas Simples Múltiples Datos). Se escribe únicamente un programa y todos los procesadores ejecutarán el mismo programa. Múltiples datos (MD) se refiere a que los datos se dividen en pedazos, y se le asigna un pedazo a cada procesador.

A diferencia de la comunicación global, al aplicarse este esquema a las estrategias de particionamiento, no existe un proceso maestro. La comunicación del procesador N es sólo con sus vecinos más cercanos, es decir, con el $N - 1$ y el $N + 1$ si no es el procesador 0 o el último. Si es el procesador 0 éste sólo se comunica con el procesador 1 y el último procesador sólo se comunica con el anterior.

Una de las características principales para la aplicación eficientemente del paralelismo es que no deben existir dependencias de datos entre los procesadores, de lo contrario se tendría que usar sincronización para que la evaluación del problema sea correcto. Por lo tanto, se requiere tener una buena orquestación del problema a paralelizar.

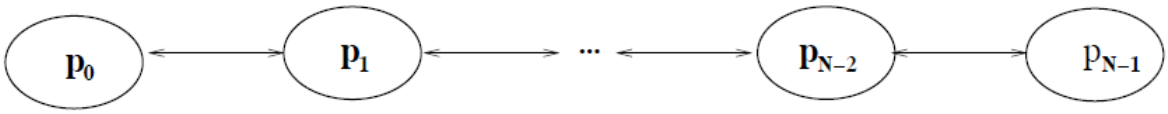

Figura 4.3: Esquema SPMD: Comunicación Local.