# Paradigmas de interacción entre procesos

Hay 3 esquemas básicos de interacción entre procesos.

Ahora estos esquemas básicos se pueden combinar generando aún más paradigmas o modelos de interacción entre procesos.

**Paradigma 1 – master/worker:** Implementación distribuida del modelo Bag of task

El concepto de bag of task usando variables compartidas supone que un conjunto de workers comparten una bola con tareas independientes. Sacan la tarea de una bolsa, la ejecutan y posiblemente crean nuevas tareas que ponen en la bolsa.

La mayor virtud de este enfoque es la escalabilidad y la facilidad para equilibrar la carga de trabajo de los workers.

Implementándolo en PM un proceso manager implementa la bolsa comunicándose con los workers y detectando fin de tareas. Se trata de un esquema C/S

**Paradigma 2: – algoritmos heartbeat** Los procesos periódicamente deben intercambiar data con mecanismos send/receive.

Es útil para soluciones iterativas que se quieren paralelizar. Usando un esquema divide and conquer se distribuye la carga entre los workers donde cada uno es responsable de actualizar una parte.

Los nuevos valores dependen de los mantenidos por los workers o sus vecinos inmediatos y cada paso debe significar un progreso hacia la solución.

Formato general de los worker:

Texto

Descripción generada automáticamente

**Ejemplo de la topología de una red (otra vez ejemplos la puta madre)**

Los procesadores están conectados por canales bidireccionales y cada uno se comunica solo con sus vecinos y conoce esos links, como puede cada procesador determinar la topología completa de la red?

La solución es que los vecinos interactúen para intercambiar información local. Con el algoritmo heartbeat se expande enviando información y luego se contrae incorporando nueva información.

Cada vecino tiene un grafo de booleanos con true o false para saber si son o no vecinos, es similar a determinar los vecinos de un grafo.

Entre ellos se van a ir mandando la matriz con los vecinos completados hasta el momento, es muy ayed la verdad.

El problema con esto es que terminas teniendo un huevo de mensajes y que tampoco sabemos cómo cada proceso puede darse cuenta de que ya conoce todo. Lo ideal sería que cada nodo se de cuenta de forma local que terminó.

Para esto se agrega que un proceso se da cuenta que la topología está completa cuando cada fila de la matriz completa tiene algún valor true, y una vez tiene eso le manda la topología a los vecinos (para que sepan que termino y que no va a mandar más info y que se caguen) y termina.

Eso sí, no en todos los algoritmos se puede hacer que los procesos terminen localmente.

**Paradigma 3 – algoritmo pipeline:** La información recorre una serie de procesos utilizando alguna forma de receive/send

Un pipeline es un arreglo lineal de procesos filtro que reciben datos de un puerto de entrada y entregan resultados por un canal de salida.

Estos procesos pueden estar en procesadores que operan en paralelo en un primer esquema a lazo abierto (W1 en el input, Wn en el output).

Un segundo esquema es el pipeline circular donde Wn se conecta con W1, este esquema sirve en procesos iterativos o bien donde la aplicación no se resuelve en una pasada por el pipe.

En un tercer esquema existe un proceso coordinador que maneja la realimentación entre Wn y W1.

**Paradigma 4 – probes (send) y echoes (receive):** La interacción entre los procesos permite recorrer grafos o árboles diseminando y juntando info.

Las arquitecturas distribuidas pueden asimilar nodos de grafos y árboles, con canales de comunicación que los vinculan (serían las aristas).

DFS es uno de los paradigmas clásicos para visitar todos los nodos de un árbol o grafo, este paradigma es la forma concurrente de DFS.

Se hace una prueba-eco que es un envío de mensajes ‘’probe’’ de un nodo al sucesor y se espera un mensaje de respuesta “echo”. Los probes se envían en paralelo a todos los sucesores

Estos algoritmos son interesantes cuando se trata con redes donde no hay o no se conoce un número fijo de nodos activos.

**Paradigma 5 – algoritmos broadcast:** Permiten alcanzar una información global en una arquitectura distribuida. Sirven para toma de decisiones descentralizadas.

En las LAN cada procesador se conecta directamente con otros, y normalmente soportan la primitiva broadcast ch(m). Los mensajes broadcast de un proceso se encolan en los canales en el orden de envío. Broadcast no es atómico entonces los mensajes enviados al mismo tiempo podrían ser recibidos por otros en distinto orden.

Se puede usar broadcast para diseminar información o para resolver problemas de sincronización distribuida.

**Paradigma 6 – token passing:** En muchos casos la arquitectura recibe una info global a través del viaje de tokens de control o datos. También permite la toma de decisiones distribuidas.

La idea es tener un tipo especial de mensaje token que va pasando por los distintos procesos y ese token se puede usar para otorgar un permiso para realizar una acción o para recoger información global entre la arquitectura distribuida. Un ejemplo del primer caso es controlar la exclusión mutua distribuida.

Ahora, el problema de la SC puede darse en memoria distribuida cuando hay algún recurso compartido que puede ser usado de a uno a la vez. Generalmente es el componente de un problema más grande, como asegurar la consistencia en una BD.

**Paradigma 7 – servidores replicados:** Los servidores manejan recursos compartidos como dispositivos o archivos a través de múltiples instancias.

Se puede usar cuando hay múltiples instancias de un recurso compartido y podemos tener un servidor para administrar cada recurso. También se puede usar la replicación para darle a los clientes la sensación de un único recurso cuando en realidad hay varios.

Usando el problema de los filósofos como ejemplo:

● Centralizado: los filósofos se comunican con un único proceso mozo (servidor) que decide el acceso a los recursos.

● Distribuido: se supone 5 procesos mozo, cada uno maneja un tenedor y un filósofo puede comunicarse con 2 mozos, solicitando y devolviendo los recursos. Los mozos no se comunican entre ellos.

● Descentralizado: cada filósofo ve un único mozo, donde los mozos se comunican entre ellos, cada mozo se comunica con sus dos vecinos para decidir el manejo del recurso asociado a su filósofo asignado.

# Librería para PM MPI

Voy a tirar un par de datos que da la teoría

El send y el receive puede ser bloqueantes.

Para asegurar que un mensaje se envíe un proceso puede no devolver el control del send hasta que el dato a transmitir este seguro

Esto tiene costos y pros.

Como pro me aseguro que el valor llega a destino, como costo hay mas ociosidad del proceso.

Hay dos posibilidades, puede ser bloqueante con buffering o sin buffering. Sin buffering es parecido a un PMS, ya que no hay lugar donde dejar el mensaje entre medio, con buffering yo dejo una copia del mensaje a enviar en un buffer (PMA).

El problema con el buffer es que necesito algún lugar en memoria donde poner el buffer, que le corresponda a un proceso ya que no hay una memoria compartida donde acceder.

En no bloqueante se devuelve el control de la operación inmediatamente para evitar el overhead generado. Esto igual requiere un posterior chequeo para asegurarse la finalización de la comunicación. Está en manos del programador asegurar la semántica del SEND.

Mismo que antes, sin buffering el dato está inseguro, se puede modificar la variable antes que se reciba el mensaje, enviándolo cualquier cosa. Con buffering el dato solo estará inseguro mientras se realiza la copia en el buffer

MPI define una librería estándar que puede ser empleada desde C y Fortran.

El estándar MPI define la sintaxis y la semántica de mas de 125 rutinas.

Hay implementaciones de MPI de la mayoría de los proveedores de hardware.

Básicamente con 6 rutinas podemos escribir programas paralelos basados en pasaje de mensajes.

**Inicio y finalización**

* MPI\_INIT: Se invoca en todos los procesos antes que cualquier otro llamado a rutinas MPI. Sirve para inicializar el entorno MPI.
* MPI\_finalize: Se invoca como último llamado a rutinas. A partir de ahí no se pueden llamar a más rutinas MPI.

**Comunicadores**

Un comunicador define el dominio de comunicación, cada proceso puede pertenecer a muchos comunicadores. Hay un comunicador que incluye a todos los procesos de la aplicación.

Son variables del tipo MPI\_Comm y en cada operación de transferencia se debe indicar sobre el que se va a realizar.

**Adquisición de información**

* MPI\_Comm\_size: Cantidad de procesos en el comunicador
* MPI\_Comm\_rank: Indica el identificador del proceso dentro de ese comunicador. Cada proceso puede tener un rank diferente en cada comunicador.

**Tipos de datos que se pueden comunicar**

**Tabla

Descripción generada automáticamente**

El packed es un registro en el que se pueden poner varios tipos de datos en la misma comunicación.

Se usa un MPI\_Send como rutina básica para enviar datos

MPI\_Recv se usa como rutina básica para recibir datos a otro proceso, si o si espera que haya al menos un mensaje en esa comunicación para seguir.

Ejemplo: Intercambio de valores

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Comunicación no bloqueante

Comienza la operación de comunicación e inmediatamente devuelven el control ( no se asegura que la comunicación finaliza correctamente.

MPI\_Isend y MPI\_Irecv se usan para esto.

Para ver si la comunicación finalizó uso MPI\_Test. Para bloquear al proceso hasta que finaliza la operación puedo usar MPI\_Wait.

No necesariamente puedo mandar o recibir con no bloqueantes, puedo mandar con un bloqueante y recibir con un no bloqueante o viceversa.

Este tipo de comunicación permite solapar cómputo con comunicación. Evita overhead de manejo de buffer y deja en manos del programador asegurar que se realice la comunicación correctamente.

MPI tiene uso de comunicaciones colectivas

MPI tiene un conjunto de funciones para realizar operaciones colectivas, sobre un grupo de procesos asociado a un comunicador. Todos los procesos del comunicador deben llamar a la rutina colectiva:

Texto

Descripción generada automáticamente con confianza media

Este tipo de conexión es super eficiente a comparación de otras librerías.

También tiene para reducir todos los mensajes a uno combinando los elementos enviados por cada uno de los procesos aplicando una cierta operación.

Existe la operación gather que recolecta el vector de datos de todos los procesos y los concatena en orden para dejar el resultado en un único proceso.

Por ejemplo, 4 procesos envían un mensaje cada uno, el gather guarda los cuatro mensajes en uno solo ( por ejemplo el 0 guarda los 4 mensajes )

Scatter es lo opuesto al gather, reparte un vector de datos entre todos los procesos inclusive el mismo dueño del vector. El normal los divide de forma equitativa, el scatterV los divide con una cantidad distinta a cada uno.

Si tengo un proceso con 4 mensajes el scatter manda 1 mensaje a cada uno por ejemplo.

Minimizando overheads de comunicación

Algunos tips:

Tabla

Descripción generada automáticamente con confianza media