

1. Defina el concepto de granularidad. ¿Qué relación existe entre la granularidad de programas y de procesadores?

La granularidad de una aplicación está dada por la relación entre el cómputo y la comunicación. Una aplicación es de Grano fino es aquella que requieren muchas comunicaciones y poco cálculo, En caso contrario es de Grano grueso.

En una arquitectura de grano fino existen muchos procesadores pero con poca capacidad de procesamiento por lo que son mejores para programas que requieran mucha comunicación. Entonces, los programas de grano fino son los apropiados para ejecutarse sobre esta arquitectura ya que se dividen en muchas tareas o procesos que requieren de poco procesamiento y mucha comunicación.

Por otro lado, en una arquitectura de grano grueso se tienen pocos procesadores con mucha capacidad de procesamiento por lo que son más adecuados para programas de grano grueso en los cuales se tienen pocos procesos que realizan mucho procesamiento y requieren de menos comunicación.

2. Explique sintéticamente los 7 paradigmas de interacción entre procesos en programación distribuida. Ejemplifique en cada caso. Indique ¿qué tipo de comunicación por mensajes es más conveniente y qué arquitectura de hardware se ajusta mejor? Justifique sus respuestas.

Servidores replicados: Los servidores manejan (mediante múltiples instancias) recursos compartidos tales como dispositivos o archivos. Su uso tiene el propósito de incrementar la accesibilidad de datos o servicios donde cada servidor descentralizado interactúa con los demás para darles la sensación a los clientes de que existe un único servidor. Un ejemplo de servidores replicados es la resolución descentralizada al problema de los filósofos donde cada proceso mozo interactúa con otros para obtener los tenedores pero esto es transparente a los procesos filósofos.

Algoritmos de heartbeat: es un esquema “divide & conquer” se distribuye la carga (datos) entre los workers; cada uno es responsable de actualizar una parte. Los nuevos valores dependen de los mantenidos por los workers o sus vecinos inmediatos. Cada “paso” debiera significar un progreso hacia la solución. Ejemplos de problemas que se pueden resolver son computación de grillas (labeling de imágenes) o autómatas celulares (el juego de la vida).

Algoritmos de pipeline: La información recorre una serie de procesos utilizando alguna forma de receive/send donde la salida de un proceso es la entrada del siguiente. Ejemplos son las redes de filtros o tratamiento de imágenes.

Algoritmos probe (send)/echo (receive): se basa en el envío de un mensaje (“probe”) de un nodo al sucesor, y la espera posterior del mensaje de respuesta (“echo”). Los probes se envían en paralelo a todos los sucesores. Los algoritmos de prueba-eco son particularmente interesantes cuando se trata de recorrer redes donde no hay (o no se conoce) un número fijo de nodos activos (ejemplo: redes móviles).

Algoritmos de Broadcast: La idea del algoritmo es cual transmitir un mensaje de un procesador a todos los otros. Podemos usar broadcasts para diseminar o reunir información; También podemos usarlo para resolver muchos problemas de

sincronización distribuidos. Un ejemplo típico es la sincronización de relojes en un Sistema Distribuido de Tiempo Real.

Algoritmos Token passing: En muchos casos la arquitectura distribuida recibe una información global a través del viaje de tokens de control o datos. Permite realizar exclusión mutua distribuida y la toma de decisiones distribuidas. Un ejemplo podría ser el de determinar la terminación de un proceso en una arquitectura distribuida cuando no puede decidirse localmente.

Master/workers: Implementación distribuida del modelo de bag of tasks que consiste en un proceso controlador de tareas y múltiples procesos workers que acceden a él para poder obtener una tarea para ejecutarlas en forma distribuida. Por ejemplo: se utiliza para resolver multiplicación de matrices ralas.

En una arquitectura de memoria distribuida, tiene que usarse alguna forma de MP, explícita o implícita. En este caso, si el flujo de información es unidireccional (por ejemplo, en redes de filtros -pipeline-) entonces el mecanismo más eficiente es AMP. También AMP es el más eficiente y el más fácil de usar para peers interactuantes (por ejemplo en algoritmos heartbeat, probe/echo, servers replicados, token passing). Broadcast es más difícil de usar con SMP. Sin embargo, para interacción cliente/servidor, la comunicación bidireccional obliga a especificar dos tipos de canales (requerimientos y respuestas). RPC y Rendezvous son técnicas de comunicación y sincronización entre procesos que suponen un canal bidireccional. Por esto son ideales para programar aplicaciones C/S (por ejemplo manager/workers).

3. Describa los 5 paradigmas de resolución de programas concurrentes: paralelismo iterativo, paralelismo recursivo, productores-consumidores, clientes-servidores, y peers. Ejemplifique en cada caso.

En el paralelismo recursivo: el problema puede descomponerse en procesos recursivos que trabajan sobre partes del conjunto total de datos (Dividir y conquistar). Por ejemplo, el sorting by merging.

En el paralelismo iterativo: un programa consta de un conjunto de procesos (posiblemente idénticos) cada uno de los cuales tiene 1 o más loops. Cada proceso es un programa iterativo. Los procesos cooperan para resolver un único problema, pueden trabajar independientemente, comunicarse y sincronizar por memoria compartida o pasaje de mensajes. Generalmente, el dominio de datos se divide entre los procesos siguiendo diferentes patrones. Un ejemplo es la multiplicación de matrices.

En el esquema de Productores- Consumidores, los procesos deben comunicarse y la comunicación fluye en una dirección por lo que generalmente se organizan en pipes donde la salida de uno será la entrada de su sucesor. Es decir, cada proceso es un filtro que consume la salida de su predecesor y produce una entrada para su sucesor. Por ejemplo, secuencias de filtros sobre imágenes.

En el esquema Cliente- Servidor, que es el predominante en los sistemas distribuidos, el servidor es un proceso que espera pedidos de servicio de múltiples clientes. Este tipo de esquema requiere de comunicación bidireccional y permite que clientes y servidores puedan ejecutarse en diferentes procesadores. La atención de clientes puede ser de a uno por vez (Rendezvous) o varios simultáneamente (RPC). Por ejemplo, la mayoría de los servicios en internet son de tipo cliente/servidor.

Con pares interactuantes los procesos (que forman parte de un programa distribuido) resuelven partes de un problema, intercambian información mediante mensajes y a veces deben sincronizar para llegar a una solución. Este tipo de esquema permite mayor grado de asincronismo que el esquema de cliente-servidor. Existen diferentes configuraciones posibles: grilla, pipe circular, uno a uno, arbitraria.

4. ¿Cuál es el objetivo de la programación paralela?

El objetivo principal es reducir el tiempo de ejecución, o resolver problemas más grandes o con mayor precisión en el mismo tiempo, usando una arquitectura multiprocesador en la que se pueda distribuir la tarea global en tareas que puedan ejecutarse en distintos procesadores.

5. Define las métricas de speedup y eficiencia. ¿Cuál es el significado de cada una de ellas (que miden) y su rango de valores? Ejemplifíquelo.

Ambas son métricas asociadas al procesamiento paralelo.

El speedup es una medida de la mejora de rendimiento (performance) de una aplicación al aumentar la cantidad de procesadores (comparado con el rendimiento al utilizar un solo procesador). El mismo se calcula:

$$\text{Speedup} \Rightarrow S = T_s / T_p$$

S es el cociente entre el tiempo de ejecución secuencial del algoritmo secuencial conocido más rápido (T_s) y el tiempo de ejecución paralelo del algoritmo elegido (T_p).

El rango de valores de S va desde 0 a p, siendo p el número de procesadores.

La eficiencia es una medida relativa que permite la comparación de desempeño en diferentes entornos de computación paralela. La misma se calcula:

$$\text{Eficiencia} \Rightarrow E = S/P$$

Es el cociente entre speedup y número de procesadores. El valor está entre 0 y 1, dependiendo de la efectividad en el uso de los procesadores. Cuando es 1 corresponde al speedup perfecto.

6. ¿En qué consiste la “ley de Amadhi”?

La ley de Amadhi dice que para un problema dado existe un máximo speedup alcanzable independiente del número de procesadores. Esto significa que es el

algoritmo el que decide la mejora de velocidad dependiendo de la cantidad de código no paralelizable y no del número de procesadores, llegando finalmente a un momento que no se puede paralelizar más el algoritmo.

7. Mencione las tres técnicas fundamentales de la computación científica. Ejemplifique.

Entre las diferentes aplicaciones de cómputo científicas y modelos computacionales existen tres técnicas fundamentales:

(1) Computación de grillas (soluciones numéricas a PDE, imágenes). Dividen una región espacial en un conjunto de puntos.

(2) Computación de partículas. Modelos que simulan interacciones de partículas individuales como moléculas u objetos estelares.

(3) Computación de matrices. Sistemas de ecuaciones simultáneas.

8. ¿Qué significa que un problema sea de “exclusión mutua selectiva”?

En los problemas de exclusión mutua selectiva, cada proceso compite por sus recursos no con todos los demás procesos sino con un subconjunto de ellos. Dos casos típicos de dicha competencia se producen cuando los procesos compiten por los recursos según su tipo de proceso o por su proximidad. Por ejemplo los problemas de lectores/escritores y filósofos.

9. ¿El problema de los lectores-escritores es de exclusión mutua selectiva? ¿Por qué?

Si, ya que la exclusión mutua selectiva se da entre clases de procesos: los procesos lectores como clase de proceso compiten con los escritores, esto se debe a que cuando un lector está accediendo a la BD otros lectores pueden acceder pero se excluye a los escritores.

El problema de los lectores-escritores es de exclusión mutua selectiva, ya que los lectores compiten solamente con los escritores para acceder al recurso.

10. Si en el problema de los lectores-escritores se acepta sólo 1 escritor o 1 lector en la BD, tenemos un problema de exclusión mutua selectiva? ¿Por qué?

En este caso, al ser un único proceso lector el que compite el problema se convierte en un problema de exclusión mutua; Porque compiten entre todos los procesos por el acceso a un recurso compartido ya que no existen más lectores para acceder a la BD y lectores y escritores por definición se excluyen mutuamente.

Si en el problema de los lectores-escritores se acepta sólo 1 escritor o 1 lector en la BD, no tenemos un problema de exclusión mutua selectiva ya que ambos compiten entre sí (o sea entre todos los procesos).

11. ¿De los problemas de los baños planteados en teoría, cuál podría ser de exclusión mutua selectiva? ¿Por qué?

Opción 1: Un baño único para varones o mujeres (excluyente) sin límite de usuarios. Exclusión mutua selectiva porque tanto las mujeres o los hombres pueden seguir entrando después que uno de su clase obtuvo el acceso.

Opción 2: Un baño único para varones o mujeres (excluyente) con un número máximo de ocupantes simultáneos (que puede ser diferente para varones y mujeres). Es exclusión mutua general ya que cuando alguna de las clases que está ocupando un baño llega a su valor máximo de ocupantes simultáneos empieza a excluir a los de su propia clase, es decir a competir entre todos.

Opción 3: Dos baños utilizables simultáneamente por un número máximo de varones (K_1) y de mujeres (K_2), con una restricción adicional respecto que el total de usuarios debe ser menor que K_3 ($K_3 < K_1 + K_2$). Similar a la opción 2 hay un tope en las cantidades de cada clase por lo que en algún momento se empezaran a excluir entre todos.

12. ¿Por qué el problema de los filósofos es de exclusión mutua selectiva? ¿Si en lugar de 5 filósofos fueran 3, el problema seguiría siendo de exclusión mutua selectiva? ¿Por qué?

Un problema es de exclusión mutua selectiva cuando cada proceso compite por el acceso a los recursos no con todos los demás procesos sino con un subconjunto de ellos. El problema de los filósofos es de exclusión mutua selectiva ya que para comer un filósofo no compite con todos los demás sino que solo compite con sus adyacentes. En el caso de que fueran 3 filósofos y no 5 como en la definición del problema general, no sería de exclusión mutua selectiva ya que un proceso compite con todos los demás procesos y no solo con un subconjunto de ellos, dado que el resto de los procesos son sus adyacentes.

13. ¿El problema de los filósofos resuelto en forma centralizada y sin posiciones fijas es de exclusión mutua selectiva? ¿Por qué?

Sin posiciones fijas se refiere a que cada filósofo solicita cubiertos y no necesariamente los de sus adyacentes, el coordinador o mozo se los da si es que hay dos cubiertos disponibles sin tener en cuenta vecinos, el problema no sería de exclusión mutua selectiva ya que competirían entre todos por poder acceder a los tenedores para poder comer.

El problema de los filósofos resuelto en forma centralizada y sin posiciones fijas no es de exclusión mutua selectiva ya que al no haber posiciones fijas cualquiera puede agarrar los cubiertos para comer, por lo tanto todos los procesos compiten entre si.

14. ¿A qué se denomina propiedad de programa? ¿Qué son las propiedades de vida y seguridad? Ejemplifique.

Una propiedad de un programa es un atributo que es verdadero en cualquier posible historia del programa, y por lo tanto de todas las ejecuciones del programa. Cada propiedad puede ser formulada en términos de dos clases especiales de propiedades: seguridad (safety) y vida (liveness). Una propiedad de seguridad asegura que el programa nunca entra en un estado malo (es decir uno en el que algunas variables tienen valores indeseables). Una propiedad de vida asegura que el programa eventualmente entra en un estado bueno (es decir, uno en el cual todas las variables tienen valores deseables).

La corrección parcial es un ejemplo de una propiedad de seguridad. Asegura que si un programa termina, el estado final es correcto (es decir, se computó el resultado correcto). Si un programa falla al terminar, puede nunca producir la respuesta correcta, pero no hay historia en la cual el programa terminó sin producir la respuesta correcta. Terminación es un ejemplo de propiedad de vida. Asegura que un programa eventualmente terminará (es decir, cada historia del programa es finita). Corrección total es una propiedad que combina la corrección parcial y la terminación. Asegura que un programa siempre termina con una respuesta correcta.

La exclusión mutua es otro ejemplo de propiedad de seguridad. Asegura que a lo sumo un proceso a la vez está ejecutando su sección crítica. El estado malo en este caso sería uno en el cual las acciones en las regiones críticas en distintos procesos fueran ambas elegibles para su ejecución. La ausencia de deadlock es otro ejemplo de propiedad de seguridad. El estado malo es uno en el que todos los procesos están bloqueados, es decir, no hay acciones elegibles. Finalmente, un entry eventual a una sección crítica es otro ejemplo de propiedad de vida. El estado bueno para cada proceso es uno en el cual su sección crítica es elegible.

15. Defina fairness. Relacione con las políticas de scheduling (NO DESCRIBA LOS DISTINTOS TIPOS DE FAIRNESS).

En un sistema concurrente fairness es el equilibrio en el acceso a recursos compartidos por todos los procesos. Trata de asegurar que un proceso tenga la posibilidad de avanzar sin importar lo que hagan el resto de los procesos. Dado que el avance de un proceso está dado por la elección de la próxima sentencia atómica que él debe ejecutar, el concepto de fairness está relacionado estrechamente con las políticas de scheduling ya que estas son las que deciden cual es la próxima sentencia atómica a ejecutarse.

16. Describa los distintos tipos de fairness.

Fairness incondicional: Toda acción atómica incondicional que es elegible es eventualmente ejecutada.

Fairness débil: Es incondicionalmente fair y toda acción atómica condicional que se vuelve elegible es ejecutada, asumiendo que su guarda se pone en true y se mantiene así hasta que es vista por el proceso que ejecuta la acción atómica condicional.

Fairness fuerte: Es incondicionalmente fair y toda acción atómica condicional que se vuelve elegible es ejecutada si su guarda es true con infinita frecuencia.

17. ¿Describa las propiedades que debe cumplir un protocolo de E/S a una sección crítica? ¿Cuáles son de seguridad y cuáles de vida?

Estos protocolos deben cumplir con 4 propiedades:

- Exclusión mutua: Solo un proceso a la vez puede estar ejecutando su sección crítica. Es una propiedad de seguridad.
- Ausencia de deadlock: Si uno o más procesos están intentando entrar a su sesión crítica al menos uno de ellos tendrá éxito. Es una propiedad de seguridad.
- Ausencia de demora innecesaria: si un proceso está intentando acceder a su sesión crítica y ningún otro proceso está ejecutando su sesión crítica o está en su sesión NO crítica o terminaron, el primero no está impedido de ingresar. Es una propiedad de seguridad.
- Eventual entrada: Un proceso que está intentando entrar a su sesión crítica eventualmente lo hará. Es una propiedad de vida.

18. ¿Cuáles son los defectos que presenta la sincronización por busy waiting? Diferencie esta situación respecto de los semáforos.

La sincronización por busy waiting es ineficiente cuando los procesos son implementados por multiprogramación ya que se mantendrá ocupado un procesador realizando un spinning cuando este podría ser usado por otro proceso. También tiene el problema de que al estar chequeando por una variable compartida que es modificada por otros procesos muchas veces ese valor se encuentra invalidado en la caché lo que requiere que se acceda a memoria; este problema es muy notorio en implementaciones de acceso a sesión crítica.

Busy waiting y semáforos son herramientas que sirven para sincronización pero son distintas ya que semáforos evitan los problemas que tiene busy waiting. Al bloquearse en un semáforo un proceso no consume tiempo de procesamiento hasta que no tenga posibilidad de ejecutarse, en cuyo caso, es puesto en la cola de listos para poder usar el procesador.

Problemas busy waiting: Los protocolos de sincronización que usan solo busy waiting pueden ser difíciles de diseñar, entender y probar su corrección. La mayoría de estos protocolos son bastante complejos, no hay clara separación entre las variables usadas para sincronización y las usadas para computar resultados.

19. Explique la semántica de la instrucción de grano grueso AWAIT y su relación con instrucciones tipo Test & Set o Fetch&Add.

<AWAIT B -> S;> Sintaxis de la sentencia AWAIT de grano grueso.

Semánticamente, B es una expresión booleana que especifica una condición de demora y S es una secuencia de sentencias que se garantiza que terminan. También, se garantiza que B es true al momento de ejecutar S, y ningún estado interno de S es visible para los otros procesos.

La sentencia await es muy poderosa ya que puede ser usada para especificar acciones atómicas arbitrarias coarse grained. Esto la hace conveniente para expresar sincronización tanto por condición como exclusión mutua. Este poder expresivo también hace a await muy costosa de implementar. Sin embargo, hay casos en que puede ser implementada eficientemente, como por ejemplo con las instrucciones T&S y F&A. Estas últimas están presentes en casi todos los procesadores y pueden ser utilizadas para implementar la instrucción AWAIT.

```
process SC [i=1..n]
{ while (true)
  {await (not lock) lock=true;
   sección crítica;
   lock=false;
   sección no crítica;
  }
}
```

```
bool lock=false;      # lock compartido
process SC[i=1 to n] {
  while (true) {
    while (TS(lock)) skip ; # protocolo de entrada
    sección crítica;
    lock = false;           # protocolo de salida
    sección no crítica;
  }
}
```

```

bool TS(bool lock) {
    bool initial = lock; # guarda valor inicial
    lock = true; # setea lock
    return initial; # devuelve valor inicial
}

```

20.¿En qué consiste la técnica de “passing the baton” y cuál es su utilidad? ¿Qué relación tiene con “passing the condition”?

Passing the baton proveer exclusión y controlar cuál proceso demorado es el próximo en seguir. Puede usarse para implementar sentencias await arbitrarias y así implementar sincronización por condición arbitraria. La técnica también puede usarse para controlar precisamente el orden en el cual los procesos demorados son despertados.

El funcionamiento es el siguiente: cuando un proceso está ejecutando su sección crítica, puede pensarse que posee la "posta" o "batón", esto significa que tiene permiso de ejecución. Cuando este proceso termina, pasa la posta a otro. Si alguno está esperando por una condición que es ahora verdadera, la posta se le otorga al mismo. Este a su vez, ejecuta su SC y la pasa a otro. Cuando no hay procesos esperando por la condición verdadera, la posta se pasa al próximo proceso que trate de entrar en su SC por primera vez.

Ambas técnicas (“**passing the baton**” y “**passing the condition**”) tienen la misma utilidad que es la de proveer un orden en la ejecución de procesos.

La relación que tiene Passing the Baton con Passing the Condition reside en que ambas técnicas permiten determinar qué proceso es quien puede acceder a la sección crítica. Passing the baton se utiliza en semáforos, mientras que passing the condition con monitores.

a. Aplicar este concepto a la resolución del problema de lectores y escritores:

Los contadores dr y dw representan la cantidad de lectores y escritores esperando, respectivamente.

Los contadores nr y nw representan la cantidad de lectores y escritores en la bd, respectivamente.

El semáforo de entrada e, controla el acceso a variables compartidas.

El semáforo r representa la condición de demora del lector ($nw = 0$), y w representa la condición de demora del escritor ($nr = 0 \wedge nw = 0$).

<pre> int nr = 0, nw = 0, dr = 0, dw = 0; sem e = 1, r = 0, w = 0; process Lector [i = 1 to M] { while(true) { P(e); if (nw > 0) {dr = dr+1; V(e); P(r); } nr = nr + 1; if (dr > 0) {dr = dr - 1; V(r); } else V(e); lee la BD; P(e); nr = nr - 1; if (nr == 0 and dw > 0) {dw = dw - 1; V(w); } else V(e); } } </pre>	<pre> process Escritor [j = 1 to N] { while(true) { P(e); if (nr > 0 or nw > 0) {dw=dw+1; V(e); P(w);} nw = nw + 1; V(e); escribe la BD; P(e); nw = nw - 1; if (dr > 0) {dr = dr - 1; V(r); } elseif (dw > 0) {dw = dw - 1; V(w); } else V(e); } } </pre>
--	---

En esta solución, la última sentencia if en los escritores es no determinística. Si hay lectores y escritores demorados, cualquiera podría ser señalizado cuando un escritor termina su protocolo de salida. Además, cuando finaliza un escritor, si hay más de un lector demorado y uno es despertado, los otros son despertados en forma de "cascada". El primer lector incrementa nr, luego despierta al segundo, el cual incrementa nr y despierta al tercero, etc. El batón se va pasando de un lector demorado a otro hasta que todos son despertados.

21. ¿En qué consiste la comunicación guardada (introducida por CSP) y cuál es su utilidad?

Con frecuencia un proceso se quiere comunicar con más de un proceso, quizás por distintos canales y no sabe el orden en el cual los otros procesos podrían querer comunicarse con él. Es decir, podría querer recibir información desde diferentes canales y no querer quedar bloqueado si en algún canal no hay mensajes. Para poder comunicarse por distintos canales se utilizan sentencias de comunicación guardadas.

Las sentencias de comunicación guardada soportan comunicación no determinística: B; C->S;

B es una expresión booleana que puede omitirse y se asume true.

B y C forman la guarda.

La guarda tiene éxito si B es true y ejecutar C no causa demora.

La guarda falla si B es falsa.

La guarda se bloquea si B es true pero C no puede ejecutarse inmediatamente.

22. Describa cómo es la ejecución de sentencias de alternativa e iteración que contienen comunicaciones guardadas.

Las sentencias de comunicación guardada soportan comunicación no determinística:

B; C → S;

- ***B*** puede omitirse y se asume *true*.
- ***B*** y ***C*** forman la guarda.
- La guarda tiene éxito si ***B*** es *true* y ejecutar ***C*** no causa demora.
- La guarda falla si ***B*** es *falsa*.
- La guarda se bloquea si ***B*** es *true* pero ***C*** no puede ejecutarse inmediatamente.

Sea la sentencia de alternativa con comunicación guardada de la forma:

```
If B1; comunicacion1 -> S1
  □ B2; comunicacion2 -> S2
  Fi
```

Primero, se evalúan las expresiones booleanas, Bi y la sentencia de comunicación.

- Si todas las guardas fallan (una guarda falla si B es false), el if termina sin efecto.
- Si al menos una guarda tiene éxito, se elige una de ellas (no determinísticamente).
- Si algunas guardas se bloquean (B es true pero la ejecución de la sentencia de comunicación no puede ejecutarse inmediatamente), se espera hasta que alguna de ellas tenga éxito.

Segundo, luego de elegir una guarda exitosa se ejecuta la sentencia de comunicación asociada.

Tercero, y último, se ejecutan las sentencias S relacionadas.

La ejecución de la iteración es similar realizando los pasos anteriores hasta que todas las guardas fallen.

23. Describa brevemente en qué consisten los mecanismos de RPC y Rendezvous. ¿Para qué tipo de problemas son más adecuados?

RPC solo provee un mecanismo de comunicación y la sincronización debe ser implementada por el programador utilizando algún método adicional. En cambio, Rendezvous es tanto un mecanismo de comunicación como de sincronización.

En RPC la comunicación se realiza mediante la ejecución de un CALL a un procedimiento, este llamado crea un nuevo proceso para ejecutar lo solicitado. El llamador se demora hasta que el proceso ejecute la operación requerida y le devuelva los resultados. En Rendezvous la diferencia con RPC está en cómo son atendidos los pedidos, el CALL es atendido por un proceso existente, no por uno nuevo. Es decir, con RPC el proceso que debe atender el pedido no se encuentra activo sino que se activa

para responder al llamado y luego termina; en cambio, con Rendezvous el proceso que debe atender los requerimientos se encuentra continuamente activo. Esto hace que RPC permita servir varios pedidos al mismo tiempo y que con Rendezvous solo puedan ser atendidos de a uno por vez.

Estos mecanismos son más adecuados para problemas con interacción del tipo cliente servidor donde la comunicación entre ellos debe ser bidireccional y sincrónica, el cliente solicita un servicio y el servidor le responde con lo solicitado ya que el cliente no debe realizar ninguna otra tarea hasta no obtener una respuesta del servidor.

RPC (Remote Procedure Call) y Rendezvous: técnicas de comunicación y sincronización entre procesos que suponen un canal bidireccional ideales para programar aplicaciones C/S. RPC y Rendezvous combinan una interfaz “tipo monitor” con operaciones exportadas a través de llamadas externas (CALL) con mensajes sincrónicos (demoran al llamador hasta que la operación llamada se termine de ejecutar y se devuelvan los resultados).

Diferencias entre RPC y Rendezvous

Difieren en la manera de servir la invocación de operaciones:

Un enfoque es declarar un procedure para cada operación y crear un nuevo proceso (al menos conceptualmente) para manejar cada llamado (RPC porque el llamador y el cuerpo del procedure pueden estar en distintas máquinas). Para el cliente, durante la ejecución del servicio, es como si tuviera en su sitio el proceso remoto que lo sirve(Ej: JAVA).

El segundo enfoque es hacer rendezvous con un proceso existente. Un rendezvous es servido por una sentencia de Entrada (o accept) que espera una invocación, la procesa y devuelve los resultados (Ej:Ada).

24. ¿Por qué es necesario proveer sincronización dentro de los módulos en RPC? ¿Cómo puede realizarse esta sincronización?

Por sí mismo, RPC es puramente un mecanismo de comunicación. Aunque un proceso llamador y su server sincronizan, el único rol del server es actuar en nombre del llamador. Conceptualmente, es como si el proceso llamador mismo estuviera ejecutando el llamado, y así la sincronización entre el llamador y el server es implícita.

También necesitamos alguna manera para que los procesos en un módulo sincronicen con cada uno de los otros. Estos incluyen tanto a los procesos server que están ejecutando llamados remotos como otros procesos declarados en el módulo. Como es habitual, esto comprende dos clases de sincronización: exclusión mutua y sincronización por condición. Hay dos aproximaciones para proveer sincronización en módulos, dependiendo de si los procesos en el mismo módulo ejecutan con exclusión mutua o ejecutan concurrentemente. Si ejecutan con exclusión (es decir, a lo sumo hay uno activo por vez) entonces las variables compartidas son protegidas

automáticamente contra acceso concurrente. Sin embargo, los procesos necesitan alguna manera de programar sincronización por condición. Para esto podríamos usar automatic signalig (await B) o variables condición.

Si los procesos en un módulo pueden ejecutar concurrentemente (al menos conceptualmente), necesitamos mecanismos para programar tanto exclusión mutua como sincronización por condición. En este caso, ada módulo es en sí mismo un programa concurrente, de modo que podríamos usar cualquiera de los métodos descriptos anteriormente. Por ejemplo, podríamos usar semáforos dentro de los módulos, monitores locales, o podríamos usar Rendezvous (o usar MP).

25.¿Qué elementos de la forma general de Rendezvous no se encuentran en el lenguaje ADA?

Rendezvous provee la posibilidad de asociar sentencias de scheduling y de poder usar los parámetros formales de la operación tanto en las sentencias de sincronización como en las sentencias de scheduling. Ada no provee estas posibilidades.

Otra:

ADA no provee la expresión de Scheduling "ei" la cual se usa para alterar el orden de servicio de invocaciones por default en las guardas.

26.Describa los mecanismos de comunicación y sincronización provistos por MPI, Ada, Java y Linda.

Linda utiliza memoria compartida como mecanismo de comunicación y pasaje de mensajes asincrónico como mecanismo de sincronización. En la memoria compartida hay tuplas que pueden ser activas (tareas) y pasivas (datos) pero el núcleo de LINDA es el espacio de tuplas compartido (TS) que puede verse como un único canal de comunicaciones compartido. Si bien hablamos de memoria compartida el espacio de tuplas puede estar distribuido en una arquitectura multiprocesador.

Java provee concurrencia mediante el uso de threads, los threads comparten la zona de datos por lo que el mecanismo que utilizan para comunicarse es memoria compartida. Como método de sincronización java permite utilizar la palabra clave synchronized sobre un método lo que brinda sincronización y exclusión mutua. También permite la sincronización por condición con los métodos wait, notify y notifyAll, similares al "wait" y "signal" de los monitores, que deben usarse siempre dentro de un bloque synchronized. La semántica que utiliza el notify es de signal and continue.

Ada utiliza Rendezvous como mecanismo de sincronización y comunicación. En Ada un proceso realiza pedidos haciendo un call de un entry definido por otro proceso, este llamado bloquea al proceso llamador hasta que termina la ejecución del pedido. Para servir los llamados a sus entries una task utiliza la sentencia accept que lo que hace es demorar a la tarea hasta que haya una invocación para el entry asociado a esa sentencia. Cuando recibe una invocación copia los parámetros y ejecuta la lista de sentencias correspondiente; al terminar copia los parámetros de salida y tanto el

como el invocador pueden continuar su ejecución. Además, provee tres clases de sentencias select: wait selectivo que permite comunicación guardada, entry call condicional y entry call timed.

Wait selectivo: Select when B1 => Sentencia accept; sentencias

Or...

Or when Bn => sentencia accept; sentencias

End select.

Se puede poner una sentencia else que es seleccionada si ninguna otra alternativa puede serlo, sentencia delay o sentencia terminate.

Entry call condicional: select entry call, sentencias

Else sentencias

End select

Solo se selecciona el entry call si puede ser ejecutado inmediatamente sino se selecciona el else.

Entry call timed: Select entry call; sentencias

Or delay; sentencias

End select

En este caso, el entry call se selecciona si puede ser ejecutado antes de que expire el intervalo delay.

27. ¿Qué significa el problema de “interferencia” en programación concurrente? ¿Cómo puede evitarse?

La interferencia se da cuando un proceso toma una acción que invalida alguna suposición hecha por otro proceso. Para evitar la interferencia entre procesos se usa la sincronización cuyo rol es restringir las historias (interleaving de sentencias) de un programa concurrente sólo a las deseables. Para hacerlo existen dos mecanismos de sincronización: exclusión mutua o por condición.

28. ¿En qué consiste la propiedad de “A lo sumo una vez” y qué efecto tiene sobre las sentencias de un programa concurrente? De ejemplos de sentencias que cumplan y de sentencias que no cumplan con ASV.

Referencia crítica en una expresión es la referencia a una variable que es modificada por otro proceso.

Instrucción atómica es aquella cuya ejecución no puede ser interrumpida.

Una sentencia de asignación $x = e$ satisface la propiedad de “A lo sumo una vez” si:

- I. e contiene a lo sumo una referencia crítica y x no es referenciada por otro proceso, o
- II. e no contiene referencias críticas, en cuyo caso x puede ser leída por otro proceso

En una expresión que no se encuentra en sentencias de asignación, se dice que cumple la propiedad de a lo sumo una vez si dicha expresión no contiene más de una referencia crítica.

El efecto que tiene sobre las sentencias de un programa concurrente es que si una sentencia de asignación cumple la propiedad ASV, entonces su ejecución parece atómica, pues la variable compartida será leída o escrita sólo una vez.

Ejemplos:

- int x=0, y=0;
co x=x+1 // y=y+1 oc; No hay ref. críticas en ningún proceso.
 En todas las historias $x = I$ e $y = I$

- int x = 0, y = 0;
co x=y+1 // y=y+1 oc; El 1er proceso tiene 1 ref. crítica. El 2do ninguna.
 Siempre $y = I$ y $x = I$ o 2

- int x = 0, y = 0;
co x=y+1 // y=x+1 oc; Ninguna asignación satisface ASV.
 Posibles resultados: $x = I$ e $y = 2$ / $x = 2$ e $y = I$
 Nunca puede ocurrir $x = I$ e $y = 1$

29. Defina el concepto de “sincronización barrier”. ¿Cuál es su utilidad? Mencione alguna de las soluciones posibles usando variables compartidas.

Como muchos problemas pueden ser resueltos con algoritmos iterativos paralelos en los que cada iteración depende de los resultados de la iteración previa es necesario proveer sincronización entre los procesos al final de cada iteración, para realizar esto se utilizan las barreras. Las barreras son un punto de encuentro de todos los procesos luego de cada iteración, aquí se demoran los procesos hasta que todos hayan llegado a dicho punto y cuando lo hagan recién pueden continuar su ejecución.

Es útil para poner un punto de encuentro entre los procesos de un programa concurrente a la espera de una determinada condición que le permita proseguir su curso normal. Dicha condición puede ser para mantener la integridad de los datos o bien para realizar una nueva iteración de los procesos.

La sincronización barrier establece que el punto de demora al final de cada iteración es una barrera a la que deben llegar todos antes de permitirles pasar.

Soluciones posibles usando variables compartidas:

I. Contador Compartido: emplea un entero compartido, cantidad, el cual es inicialmente 0. Asumimos que hay n procesos worker que necesitan encontrarse en una barrera. Cuando un proceso llega a la barrera, incrementa cantidad. Por lo tanto, cuando cantidad es n, todos los procesos pueden continuar.

Usando la instrucción Fetch-and-Add lo podemos implementar de la siguiente forma:

```
FA(cantidad,1);
while (cantidad <> n) skip;
```

problemas: resetear cantidad a 0 y “memory contention”, todos los procesos consultan sobre la misma variable cantidad.

II. Flags y Coordinadores: Se dispone de dos arreglos de enteros inicializados en 0 (arribo[1:n] y continuar[1:n]) llamadas flags variables y un proceso coordinador adicional. Dado un proceso p[i] al llegar a la barrera setear arribo[i] en 1, y luego se demora esperando que continuar[i] sea seteada en 1. El proceso Coordinador espera a que todos los elementos de arribo se vuelvan 1, luego setea todos los elementos de continuar en 1.

```
int arribo[1:n] = ([n] 0), continuar[1:n] = ([n] 0);
# arribo y continuar son "flags"
process Worker[i=1 to n] {
    while (true) {
        código para implementar la tarea i;
        arribo[i] = 1;
        < await (continuar[i] == 1); >
        continuar[i] = 0;
    }
}
process Coordinador {
    while (true) {
        for [i=1 to n] {
            < await (arribo[i] == 1); >
            arribo[i] = 0;
        }
        for [i = 1 to n] continuar[i] = 1;
    }
}
```

Mejora: los procesos esperan que distintas variables sean seteadas y estas variables podrían estar almacenadas en distintas unidades de memoria (evita “memory contention”).

Problema: requiere proceso extra, coordinador.

III. Árboles: Cada proceso hace labores de coordinador y están organizados en un árbol. Los procesos envían señales de arribo hacia arriba en el árbol y señales de continuar hacia abajo. En particular, un nodo proceso primero espera a que sus hijos arriben, luego le dice a su nodo padre que él también arribó. Cuando el nodo raíz sabe que sus hijos han arribado, sabe que todos los otros procesos también lo han hecho. Por lo tanto la raíz puede decirle a sus hijos que continúen; estos a su vez pueden decirle a sus hijos que continúen, etc.

```

Nodo hoja l; arribo[l] := 1
    ( await continuar[l] = 1 ); continuar [l]:= 0

Nodo interno i; ( await arribo[left] = 1 ); arribo[left] := 0
    ( await arribo[right] = 1 ); arribo[right] := 0
    arribo[i] := 1
    ( await continuar [i] = 1 ); continuar [i]:= 0
    continuar [left] := 1; continuar [right] := 1

nodo raiz r : ( await arribo[left] = 1 ); arribo[left] := 0
    ( await arribo[right] = 1 ); arribo[right] := 0
    continuar [left] := 1; continuar [right] := 1

```

Ventaja: Más eficiente en la transmisión de mensaje de continuar, sobre todo cuando n es grande.

Desventaja: Los procesos juegan diferentes roles.

IV. Butterfly Barrier: En esta técnica se combinan las barreras para dos procesos para construir una barrera n-proceso. Para la sincronización entre 2 procesos dados cada proceso tiene un flag que setea cuando arriba a la barrera. Luego espera a que el otro proceso setee su flag y finalmente limpia la bandera del otro.

$W[i]:: \langle$ await (arribo[i] == 0); \rangle arribo[i] = 1; \langle await (arribo[j] == 1); \rangle arribo[j] = 0;	$W[j]:: \langle$ await (arribo[j] == 0); \rangle arribo[j] = 1; \langle await (arribo[i] == 1); \rangle arribo[i] = 0;
---	---

Una butterfly barrier tiene $\log_2 n$ etapas. Cada proceso sincroniza con un proceso distinto en cada etapa. En particular, en la etapa s un proceso sincroniza con un proceso a distancia $2s-1$. Se usan distintas variables flag para cada barrera de dos procesos. Cuando cada proceso pasó a través de $\log_2 n$ etapas, todos los Workers deben haber arribado a la barrera y por lo tanto todos pueden seguir. Esto es porque cada proceso ha sincronizado directa o indirectamente con cada uno de los otros.

30. ¿Qué es una barrera simétrica?

Una barrera simétrica es un conjunto de barreras entre pares de procesos que utilizan sincronización barrier. En cada etapa los pares de procesos que interactúan van cambiando dependiendo a algún criterio establecido.

Otra:

Una barrera simétrica para n procesos se construye a partir de pares de barreras simples para dos procesos. Consiste en que cada proceso tiene un flag que setea cuando arriba a la barrera. Luego espera a que el otro proceso setee su flag y finalmente limpia la bandera del otro.

31. Describa “combining tree barrier” y “butterfly barrier”. Marque ventajas y desventajas en cada caso.

En un combining tree barrier los procesos se organizan en forma de árbol y cumplen diferentes roles. Básicamente, los procesos envían el aviso de llegada a la barrera hacia arriba en el árbol y la señal de continuar cuando todos arribaron es enviada de arriba hacia abajo. Esto hace que cada proceso debe combinar los resultados de sus hijos y luego se los pase a su padre.

Con butterfly barrier lo que se hace es en cada etapa diferente (son $\log_2 n$ etapas) cada proceso sincroniza con uno distinto. Es decir, si s es la etapa cada proceso sincroniza con uno a distancia $2s-1$. Así al final de las $\log_2 n$ etapas cada proceso habrá sincronizado directa o indirectamente con el resto de los procesos.

Combining tree barrier

Ventajas: Su implementación es más sencilla.

Desventajas: Los procesos no son simétricos ya que cada uno cumple diferentes roles, por lo que los nodos centrales realizarán más trabajo que los nodos hoja y la raíz.

Butterfly barrier

Ventajas: La implementación de sus procesos es simétrica ya que todos realizan la misma tarea en cada etapa que es la de sincronizar con un par a distancia $2s-1$.

Desventajas: Su implementación es más compleja que la del combining tree barrier y además cuando n no es par puede usarse un n próximo par para sustituir a los procesos perdidos en cada etapa; sin embargo esta implementación no es eficiente por lo que se usa una variante llamada Dissemination barrier.

32. Defina programa concurrente, programa paralelo y programa distribuido.

Un programa concurrente consiste en un conjunto de tareas o procesos secuenciales que pueden ejecutarse intercalándose en el tiempo y que cooperan para resolver un problema. Básicamente, es un concepto de software que dependiendo de la arquitectura subyacente da lugar a las definiciones de programación paralela o distribuida.

Un programa concurrente es un programa que tiene más de un hilo de ejecución o flujo de control, es decir, está dividido en tareas que pueden ejecutarse en paralelo o simultáneamente. No depende de un número determinado de procesadores ni de una arquitectura particular.

La programación paralela consiste en un programa concurrente que se ejecuta sobre múltiples procesadores que pueden tener una memoria compartida y que son utilizados para incrementar la performance de un programa concurrente.

La programación distribuida es un caso especial de la programación concurrente en la que se cuenta con varios procesadores pero no se posee una memoria compartida. Los procesos se interconectan por una red de comunicaciones, para interactuar se comunican mediante pasaje de mensajes.

33. Compare la comunicación por mensajes sincrónicos y asincrónicos en cuanto al grado de concurrencia y posibilidad de entrar en deadlock.

Con pasaje de mensajes asincrónico los canales de comunicación son colas ilimitadas de mensajes. Un proceso agrega un mensaje al final de la cola de un canal ejecutando una sentencia send. Dado que la cola conceptualmente es ilimitada la ejecución de send no bloquea al emisor. Un proceso recibe un mensaje desde un canal ejecutando la sentencia receive que es bloqueante, es decir, el proceso no hace nada hasta recibir un mensaje en el canal. La ejecución del receive demora al receptor hasta que el canal este no vacío, luego el mensaje al frente del canal es removido y almacenado en variables locales al receptor. El acceso a los contenidos de cada canal es atómico y se respeta el orden FIFO, es decir, los mensajes serán recibidos en el orden en que fueron enviados. Se supone que los mensajes no se pierden ni se modifican y que todo mensaje enviado en algún momento puede ser leído. Los procesos comparten los canales.

En el pasaje de mensajes sincrónico tanto send como receive son primitivas bloqueantes. Si un proceso trata de enviar a un canal se demora hasta que otro proceso este esperando recibir por ese canal. De esta manera, un emisor y un receptor sincronizan en todo punto de comunicación. La cola de mensajes asociada a un send sobre un canal se reduce a un mensaje. Esto significa menor memoria. Los procesos no comparten los canales. El grado de concurrencia se reduce respecto a pasaje de mensaje asincronico. Como contrapartida los casos de falla y recuperación de errores son más fáciles de manejar. Con SMP se tiene mayor probabilidad de deadlock: el programador debe ser cuidadoso para que todos los send y receive hagan matching.

34. Defina el concepto de “continuidad conversacional” entre procesos.

Es un modo de interacción entre clientes y servidores en File Servers, en los cuales contamos con un canal abrir compartido por cualquier FS, donde cada canal puede tener un solo receptor; por lo tanto para mantener una continuidad conversacional necesitamos un alocador de archivos separado que reciba pedidos de abrir y aloque un FS libre a un cliente. Cuando estos son liberados se debe avisar de dicho acontecimiento al alocador.

35. Indique por qué puede considerarse que existe una dualidad entre los mecanismos de monitores y pasaje de mensajes. Ejemplifique.

Existe dualidad entre monitores y PM porque cada uno de ellos puede simular al otro. Por ejemplo los simularemos usando procesos servidores y MP, como procesos activos en lugar de como conjuntos pasivos de procedures.

Dualidad entre Monitores y Pasaje de Mensajes

Programas con Monitores

Variables permanentes
Identificadores de procedures
Llamado a procedure
Entry del monitor
Retorno del procedure
Sentencia wait
Sentencia signal
Cuerpos de los procedure

Programas basados en PM

Variables locales del servidor
Canal request y tipos de operación
send request(); receive respuesta
receive request()
send respuesta()
Salvar pedido pendiente
Recuperar/ procesar pedido pendiente
Sentencias del “case” de acuerdo a la clase de operación.

36. Defina sincronización entre procesos y mecanismos de sincronización.

La sincronización entre procesos puede definirse como el conocimiento de información acerca de otro proceso para coordinar actividades. Esta coordinación de actividades puede darse cuando se necesita acceder a valores compartidos, esperar resultados de otro proceso, etc.

Existen dos mecanismos de sincronización:

Por exclusión mutua: asegurar que solo un proceso tenga acceso a un recurso compartido en un determinado instante de tiempo. Si un programa tiene secciones críticas que pueden ser compartidas por más de un proceso, este mecanismo evita que varios procesos puedan encontrarse en la misma sección crítica en el mismo tiempo.

Por condición: Permite bloquear la ejecución de un proceso hasta que se cumpla una determinada condición.

37. Defina el concepto de no determinismo. Ejemplifique.

El concepto no determinístico refiere al hecho de que no se puede saber de antemano cuál será el comportamiento del programa concurrente. Esto se debe a que en la programación concurrente se ejecutan varios procesos simultáneamente. Dentro de cada proceso la ejecución es secuencial, pero el intercalado de la ejecución de las instrucciones de todos los procesos en curso puede variar. De este modo no podemos determinar a priori cuál será el resultado exacto. A lo sumo, podemos determinar algunos posibles resultados.

Tomemos como ejemplo el siguiente código:

Ejemplo:

```
process imprime10 {  
    for [i=1 to n]  
        write(i);  
}  
  
process imprime1 [i=1 to 10] {  
    write(i);  
}
```

Al ejecutarse concurrentemente una vez el proceso imprime10 y los 10 procesos imprime1 los resultados en pantalla se intercalarán. Si se vuelve a ejecutar una segunda vez, puede darse que los resultados se intercalen de otra manera distinta, por lo tanto estaríamos frente a una situación de no determinismo.

38. ¿Por qué las propiedades de vida dependen de la política de scheduling? ¿Cuándo una política de scheduling es fuertemente fair?

Las propiedades de vida en los programas concurrentes hacen referencia a pedidos de servicio que eventualmente serán atendidos, que un mensaje eventualmente alcanzará su destino, y que un proceso eventualmente entrará a su sección crítica. Es por eso que se ven afectadas por las políticas de scheduling, ya que estas son las que determinan cuáles acciones atómicas elegibles son las próximas en ejecutarse.

Una política de scheduling es fuertemente fair si:

- I. es incondicionalmente fair (es decir que toda acción atómica incondicional que es elegible eventualmente es ejecutada) y
- II. toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada si su guarda es true con infinita frecuencia.

39. Describa cómo es la ejecución de sentencias de alternativa e iteración que contienen sentencias guardada.

Las sentencias de alternativa (if) e iterativa (do) contienen una o más sentencias guardadas, cada una de la forma:

B -> S

Aquí, B es una expresión booleana llamada guarda y S es una sentencia simple o compuesta. La expresión booleana B “guarda” a S en el sentido de que S no se ejecuta a menos que B sea verdadera.

Una sentencia alternativa contiene una o más sentencias guardadas:

```

if B1 → S1
□ ...
□ Bn → Sn
fi

```

Las guardas son evaluadas en algún orden arbitrario. Si la guarda B_i es verdadera, entonces se ejecuta la sentencia S_i. Si más de una guarda es verdadera, la elección de cuál se ejecuta es no determinística. Si ninguna guarda es verdadera, la ejecución del if no tiene efecto.

La sentencia iterativa es similar a la alternativa, excepto que las sentencias guardadas son evaluadas y ejecutadas repetidamente hasta que todas las guardas sean falsas. La forma del do es:

```

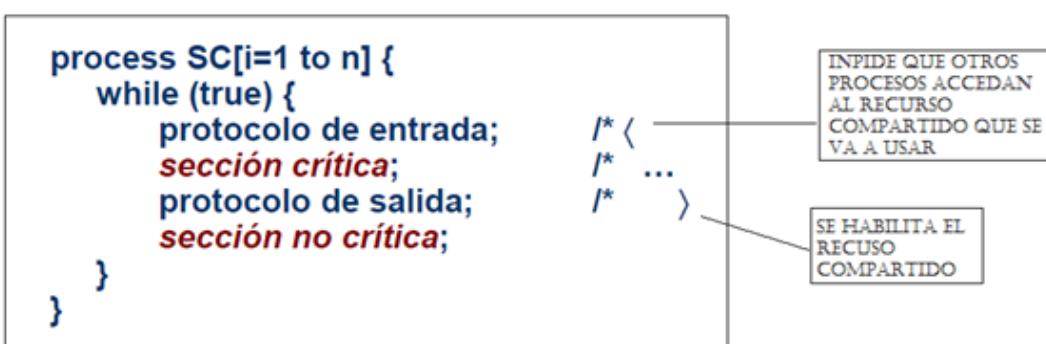
do B1 → S1
□ ...
□ Bn → Sn
od

```

Como en el if, las guardas se evalúan en algún orden arbitrario. Si al menos una guarda es verdadera, se ejecuta la correspondiente sentencia, y se repite el proceso de evaluación. Como el if, el do es no determinístico si más de una guarda es verdadera. La ejecución termina cuando no hay más guardas verdaderas.

40. Defina el problema de sección crítica. Compare los algoritmos para resolver este problema (spin locks, Tie Breaker, Ticket, Bakery). Marque ventajas y desventajas de cada uno.

En este problema, n procesos repeticamente ejecutan una sección crítica de código, luego una sección no crítica. La sección crítica está precedida por un protocolo de entrada y seguido de un protocolo de salida.



Spin locks

Se utiliza una variable booleana, llamada lock, que indica cuando un proceso está en su SC. Se utilizan instrucciones especiales como Test & Set (TS), Fetch & Add (FA) o Compare & Swap, presentes en casi todas las máquinas, que permiten implementar el protocolo de entrada a la sección critico.

Por ejemplo la instrucción TS toma una variable compartida lock como argumento y devuelve un resultado booleano. Accion= reserva de recurso, de forma atómica.

```
bool TS(bool lock) {  
    <bool initial = lock; # guarda valor inicial  
    lock = true;          # setea lock  
    return initial; >    # devuelve valor inicial  
}
```

Se utilizan loops que no terminan hasta que lock es false, y por lo tanto TS setea initial a falso. Si ambos procesos están tratando de entrar a su SC, solo uno puede tener éxito en ser el primero en setear lock en true; por lo tanto, solo uno terminará su protocolo de entrada. Cuando se usa una variable de lockeo de este modo, se lo llama spin lock pues los procesos “dan vueltas” (spinning) mientras esperan que se libere el lock.

```
bool lock=false;      # lock compartido  
process SC[i=1 to n] {  
    while (true) {  
        while (TS(lock)) skip ; # protocolo de entrada  
        sección crítica;  
        lock = false;          # protocolo de salida  
        sección no crítica;  
    }  
}
```

Funcionamiento:

Si el recurso esta libre, entonces $TS(\text{lock}=\text{false})$. Al setear $\text{lock} = \text{true}$, lo estamos reservando. Como devuelve el valor original de lock, false, al retornar se sale del while interno, ingresando a la sección critica del proceso.

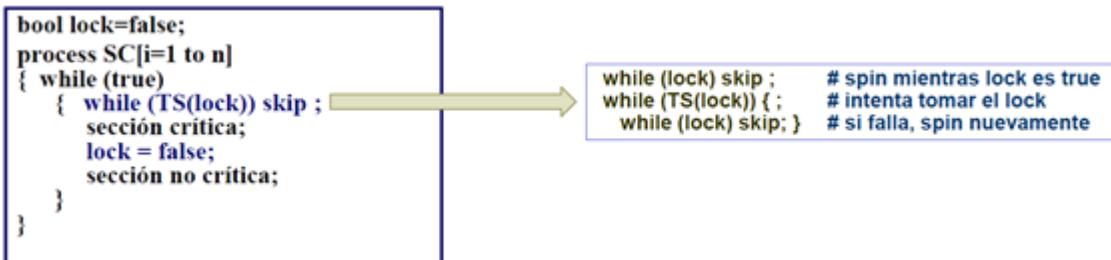
Si el recurso esta ocupado, entonces $TS(\text{lock}=\text{true})$. El seteo no produce ningún cambio. Al retornar con un valor true, el while interno sigue iterando.

Desventaja:

- 1) En multiprocesadores puede llevar a baja performance si varios procesos están compitiendo por el acceso a una SC. Esto es porque lock es una variable compartida y todo proceso demorado continuamente la referencia. Esto causa “memory contention”,

lo que degrada la performance de las unidades de memoria y las redes de interconexión procesador-memoria.

- 2) El scheduling debe ser fuertemente fair para asegurar la eventual entrada
- 3) La instrucción TS escribe en lock cada vez que es ejecutada, aun cuando el valor de lock no cambie. Dado que la mayoría de los multiprocesadores emplean caches para reducir el tráfico hacia la memoria primaria, esto hace a TS mucho más cara que una instrucción que solo lee una variable compartida (al escribir un valor en un procesador, deben invalidarse o modificarse los caches de los otros procesadores). Aunque el overhead por invalidación de cache puede reducirse modificando el entry protocol para usar un protocolo test-and-test-and-set como sigue:



Tie-Breaker

Si hay n procesos, el protocolo de entrada en cada proceso consiste de un loop que itera a través de $n-1$ etapas. En cada etapa, determinamos si el proceso o un segundo proceso avanzan a la siguiente etapa. Si aseguramos que a lo sumo a un proceso a la vez se le permite ir a través de las $n-1$ etapas, entonces a lo sumo uno a la vez puede estar en su SC.

Sean $in[1:n]$ y $ultimo[1:n]$ arreglos enteros, donde $n > 1$. El valor de $in[i]$ indica cuál etapa está ejecutando $p[i]$; el valor de $ultimo[j]$ indica cuál proceso fue el último en comenzar la etapa j . Estas variables son usadas de la siguiente manera:

```

int in[1:n] = ([n] 0), ultimo[1:n] = ([n] 0);
process SC[i = 1 to n] {
  while (true) {
    for [j = 1 to n] {  # protocolo de entrada
      # el proceso i está en la etapa j y es el último
      ultimo[j] = i; in[i] = j;
      for [k = 1 to n st i <> k] {
        # espera si el proceso k está en una etapa más alta
        # y el proceso i fue el último en entrar a la etapa j
        while (in[k] >= in[i] and ultimo[j]==i) skip;
      }
    }
    sección crítica;
    in[i] = 0;
    sección no crítica;
  }
}

```

Ventajas:

- 1) Solo requiere scheduling incondicionalmente fair para satisfacer la propiedad de eventual entrada.
- 2) No requiere instrucciones especiales del tipo Test-and-Set

Desventajas:

- 1) El algoritmo es mucho más complejo que la solución spin lock.

Ticket

Se basa en repartir tickets (números) y luego esperar turno. Este algoritmo es implementado por un repartidor de números y por un display que indica qué cliente está siendo servido.

Sean numero y próximo enteros inicialmente 1, y sea turno[1:n] un arreglo de enteros, cada uno de los cuales es inicialmente 0. Para entrar a su SC, el proceso P[i] primero setea turn[i] al valor corriente de número y luego incrementa número. El proceso P[i] luego espera hasta que el valor de próximo es igual a su número.

Algunas máquinas tienen instrucciones que retornan el viejo valor de una variable y la incrementan o decrementan como una operación indivisible simple. Como ejemplo específico, Fetch-and-Add es una instrucción con el siguiente efecto:

FA(var,incr): < temp := var; var := var + incr; return(temp) >

Solucion Ticket con instrucción Fetch-and-add

```
int numero = 1, proximo = 1, turno[1:n] = ( [n] 0 );
process SC [i: 1..n]
{ while (true)
    { turno[i] = FA (numero, 1);
      while (turno[i] <> proximo) skip;
      sección crítica;
      proximo = proximo + 1;
      sección no crítica;
    }
}
```

Ventaja:

- 1) Es más fácil de comprender que el algoritmo Tie-Breaker.

Desventaja:

- 1) Si el algoritmo corre un tiempo largo, se puede alcanzar un overflow en las variables numero y próximo. Aunque se puede resolver reseteando los contadores a un valor chico (digamos 1) cada vez que sean demasiado grandes. Si el valor más grande es al

- menos tan grande como n, entonces los valores de turno[i] se garantiza que son únicos.
- 2) Limitado porque es necesario utilizar instrucciones de máquina como Fetch-and-add. Aunque se podría resolver la sección crítica con un algoritmo spin locks o tie-breaker

Bakery

Es un algoritmo de tipo ticket, donde los procesos puede obtener el mismo número, pero hay una técnica de desempate en caso de que efectivamente ocurra eso.

Como en el algoritmo ticket, sea turno[1:n] un arreglo de enteros, cada uno de los cuales es inicialmente 0. Para entrar a su SC, el proceso p [i] primero setea turno[i] a uno más que el máximo de los otros valores de turno. Luego p[i] espera hasta que turno[i] sea el más chico de los valores no nulos de turno.

$$(\forall j : 1 \leq j \leq n, j \neq i : \text{turn}[j] = 0 \vee \text{turn}[i] < \text{turn}[j] \\ \vee (\text{turn}[i] = \text{turn}[j] \wedge i < j)) \}$$

```
P[i: 1..n] :: var j : int
    do true →
        turn[i] := 1; turn[i] := max(turn[1:n]) + 1
        fa j := 1 to n st j ≠ i →
            do turn[j] ≠ 0 and (turn[i],i) > (turn[j],j) → skip od
        af
        critical section
        turn[i] := 0
        non-critical section
    od
```

Ventaja:

- 1) No necesita de instrucciones de maquina especiales.

41. Defina procesamiento secuencial, concurrente y paralelo. Ejemplifique en cada caso. Marque especialmente ventajas e inconvenientes en cada uno de ellos.

Procesamiento secuencial: Cuando se tiene un único hilo de ejecución o control.

Procesamiento concurrente: Cuando se tiene más de un hilo de ejecución o control, no implicando la existencia de más de un procesador.

Procesamiento paralelo: Procesamiento concurrente sobre más de un procesador, ejecutando cada uno un hilo diferente en un momento dado.

En Programación Concurrente los procesos no son completamente independientes y comparten recursos. La necesidad de utilizar mecanismos de exclusión mutua y sincronización agrega complejidad a los programas & menor confiabilidad.

Los procesos iniciados dentro de un programa concurrente pueden NO estar "vivos". Esta pérdida de la propiedad de liveness puede indicar deadlocks o una mala distribución de recursos.

Hay un no determinismo implícito en el interleaving de los procesos concurrentes. Esto significa que dos ejecuciones del mismo programa no necesariamente son idénticas ☺ dificultad para la interpretación y debug.

La comunicación y sincronización produce un overhead de tiempo, inútil para el procesamiento. Esto en muchos casos desvirtúa la mejora de performance buscada

La mayor complejidad en la especificación de los procesos concurrentes significa que los lenguajes de programación tienen requerimientos adicionales. ☺ mayor complejidad en los compiladores y sistemas operativos asociados

Aumenta el tiempo de desarrollo y puesta a punto respecto de los programas secuenciales. También puede aumentar el costo de los errores ☺ mayor costo de los ambientes y herramientas de Ingeniería de Software de sistemas concurrentes.

La paralelización de algoritmos secuenciales NO es un proceso directo, que resulte fácil de automatizar.

Para obtener una real mejora de performance, se requiere adaptar el software concurrente al hardware paralelo.

42. Explique el concepto de broadcast y sus dificultades de implementación en un ambiente distribuido, con mensajes sincrónicos y asincrónicos.

El concepto de broadcast consiste en enviar concurrentemente un mismo mensaje a varios procesos.

En un ambiente compartido es más fácil que la información de un proceso pueda ser conocida por todos los demás al poner el dato, por ejemplo, en una variable compartida por todos ellos. En un ambiente distribuido esto no se puede ya que no hay variables compartidas por lo que sí o sí se debe mandar la información por medio de mensajes.

Con PMA existen dos opciones:

1) Utilizar un único canal desde donde todos los procesos pueden tomar el dato por lo que el emisor deberá enviar tanto mensajes como procesos receptores existan.

2) Utilizar un canal privado para comunicarse (enviar el mensaje) con cada uno de los procesos receptores.

El primer caso es mejor porque no hay demoras innecesarias ya que ni bien haya un mensaje en el canal este puede ser tomado por un receptor. En cambio, con la segunda opción un proceso que está listo para recibir el mensaje debe esperar a que el emisor le deposite el mensaje en su canal.

Con PMS no se puede usar un canal compartido porque todos son punto a punto, por lo tanto es más complejo e ineficiente. Es similar a la segunda opción con PMA ya que se debe enviar si o si un mensaje a cada proceso y además entre cada envío se debe esperar (SEND bloqueante) que el proceso lo haya recibido. Hay formas de optimizarlo como poner un buffer por cada proceso, y el emisor se lo envía a cada uno de esos buffer por si el proceso real está haciendo otro trabajo.

43. Defina scheduling, deadlock, fairness, inanición y overloading. Ejemplifique cada uno.

La inanición se refiere a algún proceso en particular que queda postergado permanentemente por algún motivo, como baja prioridad.

Ocurre deadlock cuando dos (o más) procesos se quedan esperando que el otro libere un recurso compartido.

Con overloading de un proceso, nos referimos a cuando la carga asignada excede su capacidad de procesamiento.

fairness se refiere al equilibrio en el acceso a los recursos compartidos por todos los procesos.

Las políticas Scheduling plantean estrategias para determinar que proceso será el siguiente en ejecutarse.

44. Analice que tipo de mecanismo de pasaje de mensajes son más adecuados para resolver problemas de tipo cliente-servidor, pares que interactúan, filtros y productores-consumidores. Justifique

Pares que interactúan, filtros y productor-consumidor: Es más adecuado el uso PM ya que el flujo de comunicación es unidireccional. Además con RPC los procesos no pueden comunicarse directamente por lo que la comunicación debe ser implementada y con Rendezvous aunque los procesos si pueden comunicarse pero no pueden ejecutar una llamada seguida de una entrada de datos por lo que deberían definirse procesos asimétricos o utilizar procesos helpers.

Dependiendo de tipo de problema particular dependerá si es más adecuado PMA que provee buffering implícito y mayor grado de concurrencia o PMS que provee mayor sincronización.

Cliente-servidor: Es más adecuado el uso de RPC o Rendezvous ya que el problema requiere de un flujo de comunicación bidireccional el cliente solicita un servicio y el servidor responde a la solicitud, es decir que no solo el cliente le envía información para llevar a cabo la operación sino que el servidor le debe devolver los resultados de dicha ejecución, también no es necesario que el cliente siga procesando ya que antes de realizar otra tarea requerirá los resultados por parte del servidor. Ambos mecanismos proveen este flujo de comunicación bidireccional con PM deberían utilizarse dos canales uno para las solicitudes y otro para las respuestas.

Otra:

Los mecanismos de Remote Procedure Call [RPC] y Rendezvous son ideales para interacciones cliente/servidor. Ambas combinan aspectos de monitores y pasaje de mensaje sincrónico (SMP). Como con monitores, un módulo o proceso exporta operaciones, y las operaciones son invocadas por una sentencia call. Como con las sentencias de salida en SMP, la ejecución de call demora al llamador. La novedad de RPC y Rendezvous es que una operación es un canal de comunicación bidireccional desde el llamador al proceso que sirve el llamado y nuevamente hacia el llamador. En particular, el llamador se demora hasta que la operación llamada haya sido ejecutada y

se devuelven los resultados. Es por esto que son más adecuados para problemas de tipo cliente/servidor ya que se precisa de una comunicación bidireccional.

El pasaje de mensajes se ajusta bien a los problemas de filtros y pares que interactúan, ya que se plantea la comunicación unidireccional.

Para resolver C/S la comunicación bidireccional obliga a especificar dos tipos de canales (requerimientos y respuestas). Además cada cliente necesita un canal de reply distinto. RPC y Rendezvous son técnicas de comunicación y sincronización entre procesos que suponen un canal bidireccional. Por esto son ideales para programar aplicaciones C/S como así también productores y consumidores.

45. Describa el paradigma “bag of tasks”. ¿Cuál es la principal ventaja del mismo?

El concepto de bag of tasks supone que un conjunto de workers comparten una “bolsa” con tareas independientes. Los workers sacan una tarea de la 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.

46. Defina monitores. Diferencie monitores y semáforos.

Los monitores son un mecanismo de abstracción de datos: encapsulan las representaciones de recursos abstractos y proveen un conjunto de operaciones que son los únicos medios por los cuales la representación es manipulada. En particular, un monitor contiene variables que almacenan el estado del recurso y procedimientos que implementan operaciones sobre el recurso. Un proceso puede acceder las variables en un monitor solo llamando a uno de los procedures del monitor.

SEMÁFORO	MONITOR
Las variables facilitad s son globales a todos los procesos	El monitor es compartido por los procesos y el es el que resguarda y limita el acceso a las variables compartidas. Solo los nombres de los procedures son visibles.

Las sentencias que acceden a variables compartidas pueden estar dispersas en un programa.	Las sentencias que acceden a las variables compartidas están representadas por los procedures del monitor. Están implementadas una única vez y son invisibles a los procesos. Ello sólo hacen uso de la interfaz del monitor.
Exclusión mutua y sincronización por condición son conceptos distintos, se programan de forma similar.	<p>La ejecución de procedures en el mismo monitor no se superponen, esto brinda la exclusión mutua. Mientras que la sincronización por condición es provista por un mecanismo de bajo nivel llamado variables condición.</p> <p>Así, la exclusión mutua en monitores es provista implícitamente y la sincronización por condición es programada explícitamente.</p>
Con P, el proceso solo duerme si el semáforo es 0.	Con WAIT, el proceso siempre se duerme.
Con V, se incrementa el semáforo para que un proceso dormido o que hará un P continue. No sigue ningún orden al despertarlos.	Con SIGNAL, si hay procesos dormidos despierta al primero de ellos. En caso contrario no tiene efecto posterior.

47. ¿Qué relación encuentra entre el paralelismo recursivo y la estrategia de dividir y conquistar? ¿Cómo aplicaría este concepto a un problema de ordenación de un arreglo?

En el paralelismo recursivo el problema se llama a sí mismos una o más veces. En cada nivel de la recursión, se aplica la estrategia de dividir y conquistar, que consta de tres pasos.

- Se divide el problema en varios subproblemas similares al problema original pero de menor tamaño (cada subproblema es asignado a un procesador distinto).
- Si los tamaños de los subproblemas son suficientemente pequeños, entonces se los resuelven de manera directa. Sino, se continúa la recursión sobre los subproblemas.
- Se combinan las soluciones para crear la solución al problema original.

Para ordenar un arreglo con el concepto mencionado podemos utilizar un algoritmo sorting by merging, se seguirán los siguientes pasos:

1. Se asigna a un procesador el proceso de ordenar el arreglo con un algoritmo sorting by merging.
2. Si la longitud del arreglo es 0 ó 1, entonces ya está ordenada. En otro caso:
3. Se divide el arreglo desordenado en dos sub-arreglos de aproximadamente la mitad del tamaño. Asignando cada uno de esos sub-arreglos a un procesador distinto.
4. Ordenar cada sublista recursivamente aplicando el ordenamiento por mezcla. Mezclar las dos sublistas en una sola lista ordenada.

48. Analice conceptualmente la resolución de problemas con memoria compartida y memoria distribuida. Compare en términos de facilidad de programación.

La resolución de problemas con memoria compartida requiere el uso de exclusión mutua o sincronización por condición para evitar interferencias entre los procesos mientras que con memoria distribuida la información no es compartida si no que cada procesador tiene su propia memoria local y para poder compartir información se requiere del intercambio de mensajes evitando así problemas de inconsistencia. Por lo tanto, resulta mucho más fácil programar con memoria distribuida porque el programador puede olvidarse de la exclusión mutua y muchas veces de la necesidad de sincronización básica entre los procesos que es provista por los mecanismos de pasajes de mensajes. Igualmente la facilidad de programación con uno o con otro modelo también está dada por el problema particular que deba ser resuelto.

49. Analice conceptualmente los modelos de mensaje sincrónico y asincrónico. Compare en términos de concurrencia y facilidad de programación.

Con pasaje de mensajes asincrónico (AMP) los canales de comunicación son colas ilimitadas de mensajes. Un proceso agrega un mensaje al final de la cola de un canal ejecutando una sentencia send. Dado que la cola conceptualmente es ilimitada la ejecución de send no bloquea al emisor.

Un proceso recibe un mensaje desde un canal ejecutando la sentencia receive que es bloqueante, es decir, el proceso no hace nada hasta recibir un mensaje en el canal. La ejecución del receive demora al receptor hasta que el canal este no vacío, luego el mensaje al frente del canal es removido y almacenado en variables locales al receptor.

El acceso a los contenidos de cada canal es atómico y se respeta el orden FIFO, es decir, los mensajes serán recibidos en el orden en que fueron enviados. Se supone que los mensajes no se pierden ni se modifican y que todo mensaje enviado en algún momento puede ser leído. Los procesos comparten los canales. Es ideal para algoritmos del tipo HeartBeat o Broadcast.

En el pasaje de mensajes sincrónico (SMP) tanto send como receive son primitivas bloqueantes. Si un proceso trata de enviar a un canal se demora hasta que otro proceso este esperando recibir por ese canal. De esta manera, un emisor y un receptor sincronizan en todo punto de comunicación.

La cola de mensajes asociada a un send sobre un canal se reduce a un mensaje. Esto significa menor memoria. Los procesos no comparten los canales. El grado de

conurrencia se reduce respecto a AMP. Como contrapartida los casos de falla y recuperación de errores son más fáciles de manejar. Con SMP se tiene mayor probabilidad de deadlock. El programador debe ser cuidadoso para que todos los send y receive hagan matching. Es ideal para algoritmos del tipo Cliente/Servidor o de Filtros.

50. Marque diferencias y similitudes entre los mecanismos de sincronización y comunicación de ADA, OCCAM, TURING PLUS, SR y LINDA.

En ADA el rendezvous es el único mecanismo de sincronización y también es el mecanismo de comunicación primario.

Las declaraciones de entry tienen la forma entry identificador (formales). Los parámetros del entry pueden ser in, out o in out.

Ada también soporta arreglos de entries, llamados familias de entry.

Si la task T declara el entry E, otras tasks en el alcance de la especificación de T pueden invocar a E por una sentencia call: call T.E(reales). Como es usual, la ejecución de call demora al llamador hasta que la operación E terminó (o abortó o alcanzó una excepción).

La task que declara un entry sirve llamados de ese entry por medio de la sentencia accept. Esto tiene la forma general: accept E(formales) do lista de sentencias end;

La ejecución de accept demora la tarea hasta que haya una invocación de E, copia los argumentos de entrada en los formales de entrada, luego ejecuta la lista de sentencias. Cuando la lista de sentencias termina, los formales de salida son copiados a los argumentos de salida. En ese punto, tanto el llamador como el proceso ejecutante continúan.

Para controlar el no determinismo, Ada provee tres clases de sentencias select: wait selectivo, entry call condicional, y entry call timed.

La sentencia wait selectiva soporta comunicación guardada. La forma más común de esta sentencia es:

```
select when B1 &lt;> sentencia accept E1; sentencias1  
or ...  
or when Bn &lt;> sentencia accept En; sentenciasn  
end select
```

Cada línea (salvo la última) se llama alternativa. Las Bi son expresiones booleanas, y las cláusulas when son opcionales. Una alternativa se dice que está abierta si Bi es true o se omite la cláusula when. Esta forma de wait selectivo demora al proceso ejecutante hasta que la sentencia accept en alguna alternativa abierta pueda ser ejecutada, es decir, haya una invocación pendiente del entry nombrado en la sentencia accept. Dado que cada guarda Bi precede una sentencia accept, no puede referenciar los parámetros de un entry call. Además, Ada no provee expresiones de scheduling, lo cual hace difícil resolver algunos problemas de sincronización y scheduling.

La sentencia wait selectiva puede contener una alternativa opcional else, la cual es seleccionada si ninguna otra alternativa puede serlo. En lugar de la sentencia accept, el

programador puede también usar una sentencia delay o una alternativa terminate. Una alternativa abierta con una sentencia delay es seleccionada si transcurrió el intervalo de delay; esto provee un mecanismo de timeout. La alternativa terminate es seleccionada esencialmente si todas las tasks que rendezvous con esta terminaron o están esperando una alternativa terminate.

Estas distintas formas de sentencias wait selectiva proveen una gran flexibilidad, pero también resultan en un número algo confuso de distintas combinaciones. Para “empeorar” las cosas, hay dos clases adicionales de sentencia select.

Un entry call condicional se usa si una task quiere hacer polling de otra. Tiene la forma:

```
select entry call; sentencias adicionales  
else  sentencias  
end select
```

El entry call es seleccionado si puede ser ejecutado inmediatamente; en otro caso, se selecciona la alternativa else.

Un entry call timed se usa si una task llamadora quiere esperar a lo sumo un cierto intervalo de tiempo. Su forma es similar a la de un entry call condicional:

```
select entry call; sentencias  
or   sentencia de delay; sentencias  
end select
```

En este caso, el entry call se selecciona si puede ser ejecutado antes de que expire el intervalo de delay. Esta sentencia soporta timeout, en este caso, de un call en lugar de un accept.

Ada provee unos pocos mecanismos adicionales para programación concurrente. Las tasks pueden compartir variables; sin embargo, no pueden asumir que estas variables son actualizadas excepto en puntos de sincronización (por ej, sentencias de rendezvous). La sentencia abort permite que una tarea haga terminar a otra. Hay un mecanismo para setear la prioridad de una task. Finalmente, hay atributos que habilitan para determinar cuándo una task es llamable o ha terminado o para determinar el número de invocaciones pendientes de un entry.

En OCCAM no se puede compartir variables es por esto que para comunicarse y sincronizar se usa canales, en particular se usan pasaje de mensajes sincrónicos. Una declaración de canal tiene la forma: CHAN OF protocol name. El protocolo define el tipo de valores que son transmitidos por el canal. Pueden ser tipos básicos, arreglos de longitud fija o variable, o registros fijos o variantes.

Los canales son accedidos por los procesos primitivos input (?) y output (!). A lo sumo un proceso compuesto puede emitir por un canal, y a lo sumo uno puede recibir por un canal.

LINDA no es en sí mismo un lenguaje de programación sino un pequeño número de primitivas que son usadas para acceder lo que llamamos espacio de tupla (tuple space,

TS). El TS es una memoria compartida, asociativa, consistente de una colección de registros de datos tagged llamados tuplas. TS es como un canal de comunicación compartido simple, excepto que las tuplas no están ordenadas.

Linda generaliza y sintetiza aspectos de variables compartidas y AMP. La operación para depositar una tupla (out) es como un send, la operación para extraer una tupla (in) es como un receive, y la operación para examinar una tupla (rd) es como una asignación desde variables compartidas a locales. Una cuarta operación, eval, provee creación de procesos. Las dos operaciones finales, inp y rdp, proveen entrada y lectura no bloqueante.

En SR hay una variedad de mecanismos de comunicación y sincronización. Los procesos pueden comunicarse y sincronizar también usando semáforos, AMP, RPC, y rendezvous. Así, SR puede ser usado para implementar programas concurrentes tanto para multiprocesadores de memoria compartida como para sistemas distribuidos.

Las operaciones son declaradas en declaraciones op como con RCP. Tales declaraciones pueden aparecer en especificaciones de recurso, en cuerpos de recurso, y aún dentro de procesos. Una operación declarada dentro de un proceso es llamada operación local. El proceso declarante puede pasar una capability para una operación local a otro proceso, el cual puede entonces invocar la operación. Esto soporta continuidad conversacional.

Una operación es invocada usando call sincrónico o send asincrónico. Para especificar qué operación invocar, una sentencia de invocación usa una capability de operación o un campo de una capability de recurso. Dentro del recurso que la declara, el nombre de una operación es de hecho una capability, de modo que una sentencia de invocación puede usarla directamente. Las capabilities de recurso y operación pueden ser pasadas entre recursos, de modo que los paths de comunicación pueden variar dinámicamente.

Una operación es servida o por un procedure (proc) o por sentencias de entrada (in). Un nuevo proceso es creado para servir cada llamado remoto de un proc. Todos los procesos en un recurso ejecutan concurrentemente, al menos conceptualmente. La sentencia de entrada soporta rendezvous. Puede tener tanto expresiones de sincronización como de scheduling que dependen de parámetros. La sentencia de entrada también puede contener una cláusula opcional else, que es seleccionada si ninguna otra guarda tiene éxito.

Una declaración process es una abreviación para una declaración op y un proc para servir invocaciones de la operación. Una instancia del proceso se crea por un send implícito cuando el recurso es creado. El cuerpo de un process con frecuencia es un loop permanente. (También pueden declararse arreglos de procesos). Una declaración procedure es una abreviación para una declaración op y un proc para servir invocaciones de la operación.

Dos abreviaciones adicionales son la sentencia receive y los semáforos. En particular, receive abrevia una sentencia de entrada que sirve una operación y que solo almacena los argumentos en variables locales. Una declaración de semáforo (sem) abrevia la declaración de una operación sin parámetros. La sentencia P es un caso especial de receive, y la sentencia V un caso especial de send.

51. Analice las arquitecturas multiprocesador, vector processor, array processor, data flow y transputer. Relacionelos con los tipos de problemas que son más adecuados para resolver en cada una de ellas.

Procesador Vectorial: son usados para operaciones con vectores. Es un diseño de CPU capaz de ejecutar la misma operación sobre múltiples datos de forma simultánea. En este tipo de procesadores, existe una memoria global compartida por los procesadores locales. El sincronismo es tal que todos los procesos tardan tiempos semejantes. La eficiencia se incrementa cuando los vectores a procesar son de dimensión mayor que el número de elementos de procesamiento.

Los procesadores vectoriales son muy comunes en el área de la computación científica

Array processor: gráficamente es una grilla de procesadores SIMD conectados formando una arquitectura SIMD. La misma operación se ejecuta simultáneamente en un conjunto de procesadores con datos diferentes. Cada procesador tiene memoria local y capacidad de E/S local y puede comunicarse con procesadores adyacentes. El control es centralizado generalmente por un host SIMD.

Data flow: se compone de N procesadores con arquitectura MIMD, con bajo acoplamiento. Las instrucciones no se ejecutan en secuencia ni bajo la coordinación de ninguna unidad de control: al estar disponibles el dato que requieren y algún procesador, son activadas y ejecutadas. La arquitectura global es la pipeline circular, donde los paquetes de instrucciones se van acoplando. Requieren lenguajes y compiladores especiales.

Transputer: un transputer es un microprocesador RISC de alta performance con memoria en el chip y capacidad de memoria externa. Un transputer no representa una arquitectura paralela distribuida. La arquitectura es paralela, varios transputer sobre un bus común bajo supervisión, o estructurando una grilla de transputers vinculados por comunicación de alcance localizado y coordinados por un master. La coordinación de aplicaciones que exploten el paralelismo requiere de lenguajes orientados para la arquitectura. Características de diseño: simplicidad, alto grado de escalabilidad, alta performance, alta conectividad.

52. ¿Qué es una lógica de programación y cuáles son sus componentes? ¿Qué es un invariante?

Lógica de Programación: es un sistema lógico formal (SLF) para desarrollar y analizar programas. Incluye:

- . Predicados: caracterizan estados de programa.
- . Relaciones: caracterizan el efecto de la ejecución.

Cualquier SLF consta de reglas definidas en términos de:

- . Un conjunto de símbolos.

- . Un conjunto de fórmulas construidas a partir de los símbolos.
- . Un conjunto de axiomas (formulas distinguidas).
- . Un conjunto de reglas de inferencia.

Invariante: es un predicado que se mantiene verdadero antes y después de la iteración o ejecución de una acción.

53. Mencione las características de la notación de primitivas múltiples. Ejemplifique.

Es una notación que combina RPC, Rendezvous y PMA en un paquete coherente. Provee un gran poder expresivo combinando ventajas de las 3 componentes, y poder adicional.

Como con RCP se estructura los programas con colecciones de módulos.

Una operación visible es especificada en la declaración del módulo y puede ser invocada por procesos de otros módulos pero es servida por un proceso o procedure del módulo que la declara. También se usan operaciones locales, que son declaradas, invocadas y servidas solo por el módulo que la declara.

Una operación puede ser invocada por **call** sincrónico o por **send** asincrónico y las sentencias de invocación son de la siguiente manera:

```
call Mname.op(argumentos)
send Mname.op(argumentos)
```

Como por RCP y Rendezvous el call termina cuando la operación fue servida y los resultados fueron retornados. Como con AMP el send termina tan pronto como los argumentos fueron evaluados.

En la notación de primitivas múltiples una operación puede ser servida por un procedure (**proc**) o por rendezvous (sentencias **in**). La elección la toma el programador del modulo que declara la operación. Depende de si el programador quiere que cada invocación sea servida por un proceso diferente o si es más apropiado rendezvous con un proceso existente.

En resumen, hay dos maneras de invocar una operación (call o send) y dos maneras de servir una invocación (proc o in).

54. Definir el problema general de alocación de recursos y su resolución mediante una política SJN. ¿Minimiza el tiempo promedio de espera? ¿Es fair? Si no lo es, plantee una alternativa que lo sea.

La alocación de recursos es el problema de decidir cuándo se le puede dar a un proceso acceso a un recurso. En programas concurrentes, un recurso es cualquier cosa

por la que un proceso podría ser demorado esperando adquirirla. Esto incluye entrada a una SC, acceso a una BD, un slot en un buffer limitado, una región de memoria, el uso de una impresora, etc.

Existen variadas formas de resolver el problema de alocación de recursos, entre las que se encuentra SJN (shortest job next). Esta técnica en particular muestra cómo controlar explícitamente cuál proceso toma un recurso cuando hay más de uno esperando.

En el problema de alocación de recursos un proceso pide una o más unidades del recurso ejecutando un REQUEST con la cantidad de unidades que solicita y su identificación. Un REQUEST solo puede ser satisfecho si la cantidad de unidades disponibles es suficiente para satisfacer el REQUEST de lo contrario se demora hasta que haya suficientes unidades disponibles. Después de usar los recursos alocados, un proceso los retorna ejecutando la operación RELEASE.

Las operaciones de REQUEST y RELEASE deben ser atómicas ya que ambas acceden a la representación de los recursos. Se puede utilizar la técnica de PASSING THE BATON para el REQUEST y el RELEASE:

```
REQUEST (parámetros): P (e)
    if (REQUEST no puede satisfacerse) {DELAY}
        Tomar las unidades;
        SIGNAL;
RELEASE (parámetros): P (e);
    Retorna unidades
    SIGNAL;
```

Su resolución mediante SJN se basa en varios procesos que requieren un único recurso compartido utilizando REQUEST como antes pero con los parámetros id y time siendo este el tiempo que retendrá el recurso para su uso. Por lo tanto, al ejecutar un REQUEST si el recurso está libre es inmediatamente alocado al proceso y en caso contrario, el proceso se demora. Cuando el recurso es liberado, utilizando el RELEASE, es alocado al proceso demorado (si lo hay) que tiene el mínimo valor de time. Si dos o más procesos demorados tienen el mismo valor de time se le asigna al que más tiempo ha estado demorado.

La política de SJN minimiza el tiempo promedio de ejecución pero es unfair ya que un proceso podría ser demorado para siempre si hay una corriente continua de REQUEST con time menor. Esta política puede modificarse para evitar esto utilizando la técnica de aging que consiste en darle preferencia a un proceso que ha estado demorado un largo tiempo. (Poner un valor fijo para determinar si un proceso espero mucho o implementar la variante del SJN, Highest Response Ratio Next que pone la prioridad como $1 + \text{tiempo espera} / \text{tiempo ejecución}$).

55. Describir la solución usando la criba de Eratóstenes al problema de hallar los números primos entre 2 y n. ¿Cómo termina el algoritmo? ¿Qué modificaría para que no termine de esa manera?

Se comienza por el primer número en la lista, en este caso el 2, y se borran todos los números que son múltiplos de este. Los números que quedan todavía son candidatos a ser primos así que se repite el mismo proceso una y otra vez hasta que todos los números son considerados. Por lo tanto, los números que queden en la lista al final serán todos los números primos entre 2 y n.

La solución a este problema utiliza un pipeline de filtros; particularmente, cada proceso filtro del pipeline recibe un stream de números de su predecesor y envía un stream de números a su sucesor realizando la eliminación de los múltiplos correspondiente. Hay un problema con esta solución y es que termina en deadlock ya que los procesos reciben dentro de un loop los valores de stream y no tienen forma de darse cuenta cuando no recibirán nuevos valores y se quedarán todos esperando un nuevo valor desde su predecesor pero esto puede solucionarse utilizando centinelas. Básicamente el uso de centinelas consiste en agregar un valor especial, EOS, al stream para indicar que ya se han recibido todos los valores.

56. Sea el problema en el cual N procesos poseen inicialmente cada uno un valor V, y el objetivo es que todos conozcan cual es el máximo y cuál es el mínimo de todos los valores.

Plantee conceptualmente posibles soluciones con las siguientes arquitecturas de red: Centralizada, simétrica y anillo circular. No implementar.

En una arquitectura centralizada cada nodo o proceso le envía su dato al procesador central que es el encargado de ordenar los N valores y de reenviar el máximo y el mínimo valor a todos los demás procesos.

En la arquitectura simétrica hay un canal entre cada par de procesos y todos ejecutan el mismo algoritmo por lo tanto cada uno le envía su dato V a los n-1 procesos restantes y recibe n-1 valores. De este modo en paralelo todos están calculando el máximo y el mínimo.

Por último, en una arquitectura de anillo cada P[i] recibe mensajes de su predecesor P [i-1] y envía mensajes a su sucesor P [i+1]. Este esquema consta de 2 etapas; en la primera cada proceso recibe dos valores y los compara con su valor local, trasmitiendo un máximo local y un mínimo local a su sucesor. En la segunda etapa todos deben recibir la circulación del máximo y el mínimo global.

Analice las soluciones anteriores desde el punto de vista del número de mensajes y la performance global del sistema.

Para la primera solución con arquitectura centralizada se requieren 2(n-1) mensajes, n-1 para enviar los valores desde los procesos hacia el coordinador y n-1 enviados por el coordinador a los procesos con el máximo y el mínimo. Los mensajes al coordinador se envían casi al mismo tiempo por lo que solo el primer RECEIVE del coordinador demora mucho. Puede reducirse a N mensajes si el proceso central cuenta con una instrucción de BROADCAST.

En cambio, con una arquitectura simétrica se envían $n(n-1)$ mensajes que pueden transmitirse en paralelo si la red soporta transmisiones concurrentes pero el OVERHEAD de comunicación acota el SPEEDUP. Aunque es la más corta y sencilla utiliza la mayor cantidad de mensajes si no existe una instrucción de BROADCAST pero si la hay solo se necesitan n mensajes.

En la arquitectura de anillo se requieren la misma cantidad que para la centralizada incluso si existe instrucción BROADCAST serán $2(n-1)$ mensajes. El último proceso debe esperar a que todos los otros (uno por vez) reciban un mensaje, hagan poco cómputo y envíen su resultado. Estos mensajes circulan 2 veces completas por el anillo por lo que la solución es lineal y lenta para este problema pero puede funcionar si cada proceso realiza mucho cómputo.

Tanto la arquitectura centralizada como en la arquitectura de anillo usan un número lineal de mensajes pero tienen distintos patrones de comunicación que llevan a distinta performance el patrón de pipeline circular de la arquitectura anillo hace que la solución sea más lenta.

57. ¿Qué se entiende por arquitecturas de grano fino? ¿Son más adecuadas para programas con mucha o poca comunicación?

Una arquitectura de grano fino posee muchos procesadores pero con poca capacidad de procesamiento esto hace que las arquitecturas de grano fino sean más adecuadas para programas con mucha comunicación, es decir programas de grano fino donde existen muchos procesos que realizan poco cómputo pero donde se requiere de mucha comunicación.

58. Analice conceptualmente los modelos de mensajes sincrónicos y asincrónicos. Compárelos en términos de concurrencia y facilidad de programación.

Con PMS el send es bloqueante por lo que el emisor no puede realizar otras operaciones mientras espera que el mensaje sea recibido pero esta primitiva bloqueante provee un punto de sincronización.

En términos de concurrencia PMA la maximiza con respecto a PMS ya que su send es no bloqueante por lo que un proceso puede seguir realizando tareas después de enviar un mensaje aunque este no haya sido recibido, para lograr esto PMA provee de buffering implícito. Además PMS tiene más riesgo de deadlock, que PMA por la semántica de la sentencia send, e implica muchas veces implementar procesos asimétrico o utilizar comunicación guardada.

Desde el punto de vista de facilidad de programación es mejor PMA aunque la sincronización deba ser implementada por el programador en muchos casos. Igualmente la elección de un tipo u otro con respecto a la facilidad dependerá del tipo de problema que se requiera resolver ya que mucho problemas pueden por ejemplo ser resueltos con ambos mecanismos pero uno de ellos se adapta mejor haciendo más fácil la resolución.

Otra respuesta:

En cuanto a la facilidad de la programación no varía mucho, lo que es importante en este aspecto es que si programamos con mensajes sincrónicos, el emisor del mensaje se va a quedar bloqueado hasta cuando el receptor lo reciba, por lo que hay que saber manejar de la manera más óptima las situaciones de comunicación para no generar demoras innecesarias, en el caso del uso de mensajes asincrónicos, el emisor del mensaje deposita el mensaje y posteriormente sigue su curso normal, no hay retraso.

En términos concurrente, el grado de concurrencia se reduce respecto de la sincronización por Pasaje de Mensaje Asincrónico, ya que siempre un proceso se bloquea. Pero en términos de memoria, utiliza menos memoria, debido a que la cola de mensajes asociada a un canal se reduce a 1 mensaje.

59. ¿En qué consisten las arquitecturas SIMD y MIMD? ¿Para qué tipo de aplicaciones es más adecuada cada una?

SIMD, Single instruction multiple data: Todos los procesadores ejecutan las mismas instrucciones pero sobre diferentes datos. Son para aplicaciones muy específicas de problemas regulares por ejemplo la multiplicación de matrices.

MIMD, multiple instruction multiple data: Cada procesador tiene su propio flujo de instrucciones y su propio conjunto de datos, es decir, cada uno ejecuta su propio programa. Pueden utilizar memoria compartida o memoria distribuida. Además pueden subclasicarse en MPMD donde cada procesador ejecuta su propio programa (PVN) y SPMD donde cada procesador ejecuta una copia del programa (MPI). Cualquier aplicación que no debe hacer lo mismo sobre distintos datos. Por ejemplo un pipeline o un master/worker o donde el trabajo es irregular, es decir que la cantidad de tiempo que lleva cada tarea depende de los datos en sí y no del tamaño de los mismos como es la ordenación de vectores.

- a) Resuelva el problema de encontrar la topología de una red utilizando mensajes asincrónicos. Muestre con un ejemplo la evolución de la matriz de adyacencia para una red con al menos 7 nodos y de diámetro al menos 4.**
- b) Compare conceptualmente con una solución utilizando PMS.**

La solución con PMS dificultaría el algoritmo de forma tal que incrementaría la demora ya que se debe recibir en orden uno por uno los valores de los vecinos y si uno de ellos se retrasa no podemos continuar hasta que no recibamos el valor. Además de que se deberían usar algoritmos asimétricos de forma tal de evitar deadlock o utilizar comunicación guardada tanto para las sentencias de envío como para las de recepción. Por lo que en caso de resolver este algoritmo con PMS usando comunicación guardada sería más eficiente ya que solo recibiría de los canales en los que tiene datos sin necesidad de demorarse.

60. ¿Cuál (o cuáles) es el paradigma de interacción entre procesos más adecuado para resolver problemas del tipo "Juego de la vida"?

El paradigma que mejor se adecua es el heartbeat ya que permite enviar información a todos los vecinos y luego recopilar la información de todos ellos. Entonces una célula recopila la información de sus vecinos en la cual se basa su próximo cambio de estado.

a) ¿Considera que es conveniente utilizar mensajes sincrónicos o asincrónicos?

Es mejor la utilización de mensajes asincrónicos ya que evitan problemas de deadlock cuando se debe decidir qué proceso envía primero su información. Como en el algoritmo general de heartbeat todos los procesos primero envían su información y luego recopilan la información de sus vecinos se hace intuitivo el uso de PMA para su resolución evitando tener que realizar procesos asimétricos con demora innecesaria aunque la performance podría mejorar si se utiliza comunicación guardada para las sentencias de envío y recepción.

b) ¿Cuál es la arquitectura de hardware que se ajusta mejor? Justifique claramente sus respuestas.

Es conveniente una arquitectura de grano fino, con mucha capacidad para la comunicación y poca capacidad para el cómputo ya que este tipo de programas es de grano fino muchos procesos o tareas con poco computo que requieren de mucha comunicación. Una arquitectura en forma de grilla es una forma óptima para este tipo de problemas.

61. ¿En qué consiste la utilización de relojes lógicos para resolver problemas de sincronización distribuida? Ejemplifíquelo.

Las acciones de comunicación, tanto send como receive, afectan la ejecución de otros procesos por lo tanto son eventos relevantes dentro de un programa distribuido y por su semántica debe existir un orden entre las acciones de comunicación. Básicamente cuando se envía un mensaje y luego este es recibido existe un orden entre estos eventos ya que el send se ejecuta antes que el receive por lo que puede imponerse un orden entre los eventos de todos los procesos. Sea un ejemplo el caso en que procesos solicitan acceso a un recurso, muchos de ellos podrían realizar la solicitud casi al mismo tiempo y el servidor podría recibir las solicitudes desordenadas por lo que no sabría quién fue el primero en solicitar el acceso y no tendría forma de responder a los pedidos en orden. Para solucionar este problema se podrían usar los relojes lógicos.

Para establecer un orden entre eventos es que se utilizan los relojes lógicos que se asocian mediante un timestamps a cada evento. Entonces, un reloj lógico es un contador que es incrementado cuando ocurre un evento dentro de un proceso ya que el reloj es local a cada proceso y se actualiza con la información distribuida del tiempo que va obteniendo en la recepción de mensajes. Este reloj lógico (rl) será actualizado de la siguiente forma dependiendo del evento que suceda:

Cuando el proceso realiza un SEND, setea el timestamp del mensaje al valor actual de rl y luego lo incrementa en 1. Cuando el proceso realiza un RECEIVE con un timestamp (ts), setea rl como max (rl, ts+1) y luego incrementa rl.

Con los relojes lógicos se puede imponer un orden parcial ya que podría haber dos mensajes con el mismo timestamp pero se puede obtener un ordenamiento total si existe una forma de identificar únicamente a un proceso de forma tal que si ocurre un empate entre los timestamp primero ocurre el que proviene de un proceso con menor identificador.

66. ¿Qué condiciones deben darse para que ocurra deadlock?

Puede ocurrir de deadlock si ocurre:

Recursos reusables serialmente: los procesos comparten recursos que pueden usar con exclusión mutua.

Adquisición incremental: los procesos mantienen los recursos que poseen mientras esperan adquirir recursos adicionales.

No-preemption: una vez que son adquiridos por un proceso, los recursos no pueden quitarse de manera forzada sino que sólo son liberados voluntariamente.

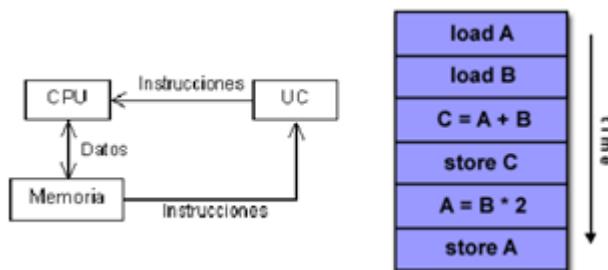
Espera cíclica: existe una cadena circular (ciclo) de procesos tal que cada uno tiene un recurso que su sucesor en el ciclo está esperando adquirir.

67. Cómo pueden clasificarse las arquitecturas multiprocesador.

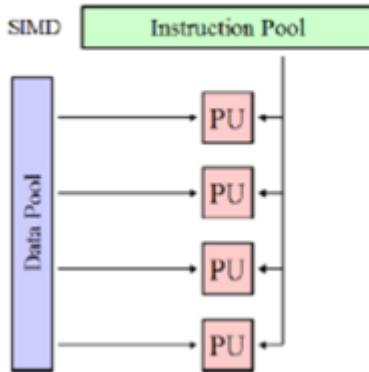
¿Según el mecanismo de control? Describa.

Esta clasificación se basa en el modo en que las instrucciones son ejecutadas sobre los datos y hay 4 variantes.

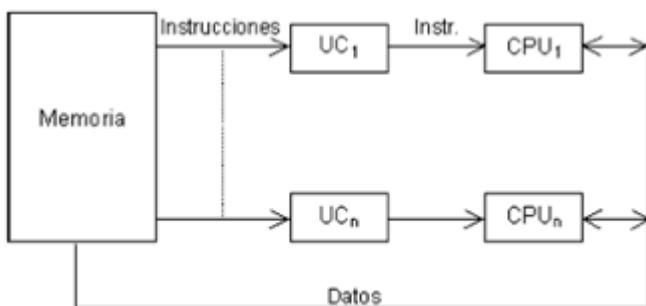
SISD, Single instruction single data: Las instrucciones son ejecutadas en secuencia, de a una por ciclo de instrucción por lo tanto su ejecución es determinística. Básicamente la CPU ejecuta instrucciones sobre los datos, la memoria recibe los datos, los almacena y también brinda los datos. Usada en uniprocesadores.



SIMD, Single instruction multiple data: Hay un conjunto de procesadores idénticos, con su memoria, que ejecutan la misma instrucción pero sobre diferentes datos. . El host hace broadcast de la instrucción. Ejecución sincrónica y determinística. Pueden deshabilitarse y habilitarse selectivamente procesadores para que ejecuten o no instrucciones. Los procesadores en general son muy simples. Adecuados para aplicaciones con alto grado de regularidad, (por ej. procesamiento de imágenes).



MISD, multiple instruction single data: los procesadores ejecutan un flujo de instrucciones distinto pero comparten datos comunes. Operación sincrónica (en lockstep). No son máquinas de propósito general ("hipotéticas", Duncan). Ejemplos posibles: múltiples filtros de frecuencia operando sobre una única señal, múltiples algoritmos de criptografía intentando crackear un único mensaje codificado.



MIMD, multiple instruction multiple data: Cada procesador tiene su propio flujo de instrucciones y su de datos, es decir, cada uno ejecuta su propio programa. Pueden utilizar memoria compartida o memoria distribuida. Además pueden subclasicificarse en MPMD (multiple program multiple data) donde cada procesador ejecuta su propio programa y SPMD (single program multiple data) hay un único programa fuente y cada procesador ejecuta su copia independientemente.

¿Según la organización del espacio de direcciones? Describa

Se clasifican en memoria compartida y memoria distribuida.

Con memoria compartida la interacción se realiza modificando los datos compartidos. Pueden existir problemas de consistencia. Dentro de esta clasificación se pueden usar dos esquemas UMA y NUMA con el primero la memoria física es compartida uniformemente por todo procesadores mientras que con el esquema NUMA cada procesador tiene su propia memoria local pero esta puede ser accedida por los demás ya que el conjunto de todas las memorias locales forman una única memoria compartida.

Con memoria distribuida cada proceso tiene su propia memoria local que es usada únicamente por él y la interacción y el intercambio de información se realiza a través de pasaje de mensajes.

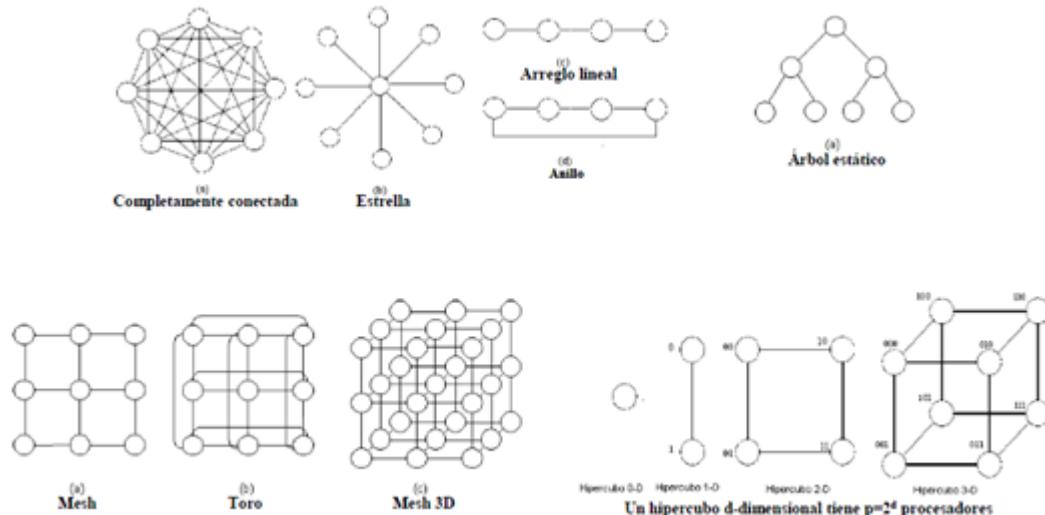
¿Según la red de interconexión? Describa.

Tanto en memoria compartida como en pasaje de mensajes las máquinas pueden construirse conectando procesadores y memorias usando diversas redes de interconexión:

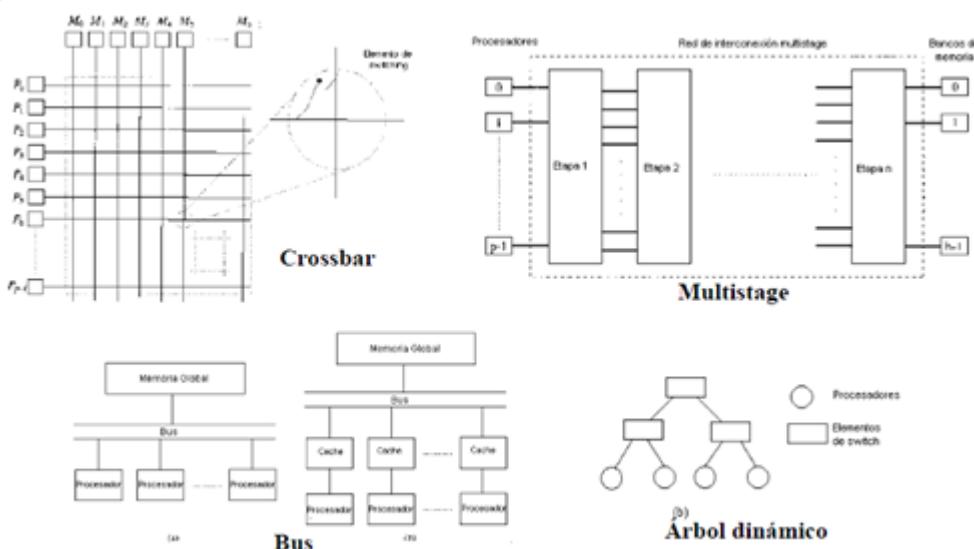
Las redes estáticas: son enlaces punto a punto, típicamente se utilizan en máquinas de pasaje de mensajes .

Las redes dinámicas: están construidas con switches y enlaces de comunicación, normalmente son utilizadas para memoria compartida.

Dentro de las redes estáticas tenemos las diferentes topologías de red como anillo, estrella, árbol, hipercubo, etc.



Y dentro de las redes dinámicas tenemos las redes multistage, árbol dinámico, bus, etc.



PRÁCTICA

Sea la siguiente solución al problema del producto de matrices de nxn con P procesos trabajando en paralelo.

```
process worker[w = 1 to P] {           # strips en paralelo (p strips de n/P filas ) }
    int first = (w-1) * n/P;          # Primera fila del strip
    int last = first + n/P - 1;       # Última fila del strip
    for [i = first to last] {
        for [j = 0 to n-1] {
            c[i,j] = 0.0;
            for [k = 0 to n-1]
                c[i,j] = c[i,j] + a[i,k]*b[k,j];
        }
    }
}
```

- a) Suponga que $n=128$ y cada procesador es capaz de ejecutar un proceso. ¿Cuántas asignaciones, sumas y productos se hacen secuencialmente (caso en que $P=1$)?

Si el algoritmo se ejecuta secuencialmente se tienen:

$$\text{Asignaciones: } 128^3 + 128^2 = 2097152 + 16384 = 2113536$$

$$\text{Sumas: } 128^3 = 2097152.$$

$$\text{Productos: } 128^3 = 2097152.$$

- b) ¿Cuántas se realizan en cada procesador en la solución paralela con $P=8$?

Si tenemos 8 procesos cada uno con un strip de 16 ($128/8$) los cálculos de tiempo quedarían para cada proceso como:

$$\text{Asignaciones: } 128^2 * 16 + 128 * 16 = 262144 + 2048 = 264192.$$

$$\text{Sumas: } 128^2 * 16 = 262144.$$

$$\text{Productos: } 128^2 * 16 = 262144$$

- c) Si los procesadores P1 a P7 son iguales, y sus tiempos de asignación son 1, de suma 2 y de producto 3, y si P8 es 4 veces más lento, ¿Cuánto tarda el proceso total concurrente? ¿Cuál es el valor del speedup (Tiempo secuencial/Tiempo paralelo)? Modifique el código para lograr un mejor speedup.

p1 a p8 tienen igual número de operaciones.

Asignaciones =264192, Sumas = 262144 y Productos = 262144

unidades de tiempo de p1 a p7 (Asignaciones =1t, Sumas =2t y Productos =3t)

$$\text{Total de tiempo por proceso} = (264192 * 1) + (262144 * 2) + (262144 * 3) = 1574912$$

Pero P8 es 4 veces más lento

$$\text{total de tiempo de P8} = (1574912 * 4) = 6299648$$

Tiempo de ejecución total paralelo es 6299648.

Usando los cálculos de a) obtenemos *el tiempo secuencial* con los tiempos de las operaciones del procesador más eficiente, es decir p1 a p7

$$\text{Asignaciones: } 128^3 + 128^2 = 2113536$$

$$\text{Sumas: } 128^3 * 2 = 2097152 * 2 = 4194304$$

$$\text{Productos: } 128^3 * 3 = 2097152 * 3 = 6291456$$

Entonces el tiempo que requiere la ejecución secuencial es de 12599296 unidades de tiempo.

$$\text{Speedup de } 12599296 / 6299648 = 2$$

Esto puede ser mejorado si realizamos un mejor balance de carga haciendo que P8 trabaje sobre un strip más pequeño. Dándole a P8 solo dos filas y los demás procesando 18 filas obtenemos los siguientes cálculos:

p1 a p7

Asignaciones: $128^2 * 18 + 128 * 18 = 294912 + 2304 = 297216$.

Sumas: $(128^2 * 18) * 2 = 294912 * 2 = 589824$.

Productos: $(128^2 * 18) * 3 = 294912 * 3 = 884736$.

Entonces, P1...P7 ejecutan en 1771776 unidades de tiempo.

p8

Asignaciones: $128^2 * 2 + 128 * 2 = 32768 + 256 = 33024$.

Sumas: $(128^2 * 2) * 2 = 32768 * 2 = 65536$.

Productos: $(128^2 * 2) * 3 = 32768 * 3 = 98304$.

P8 consume $196864 * 4 = 787456$.

Tiempo paralelo de ejecución pasa a ser 1771776.

Speedup = $12599296/1771776 = 7,1$ logrando una gran mejora con respecto al algoritmo secuencial gracias al balance de carga realizado para evitar retrasos por parte de P8.

Dado el siguiente programa concurrente con memoria compartida:

X:=4; y:=2; z:=3;

Co

x:=x-z//z:=z*2//y:=z+4

Oc

a) Cuáles de las asignaciones dentro de la sentencia co cumplen la propiedad de ASV. Justifique claramente.

Una sentencia de asignación x = e satisface la propiedad de A lo sumo una vez sí:

(1) e contiene a lo sumo una referencia crítica y x no es referenciada por otro proceso, o

(2) e no contiene referencias críticas, en cuyo caso x puede ser leída por otro proceso

- X:=X-Z Cumple la propiedad de a lo sumo una vez ya que posee una única referencia crítica (Z) y X no es referenciada por otro proceso.
- Z:=Z*2 Cumple la propiedad de a lo sumo una vez porque Z no es una referencias críticas por lo tanto, Z puede ser leída por otro proceso.
- Y:=Z+4 Cumple la propiedad de a lo sumo una vez ya que posee una referencia crítica (Z) e Y no es referenciada por ningún otro proceso.

b) Indique los resultados posibles de la ejecución. Justifique.

Cada tarea se ejecuta sin ninguna interrupción hasta que termina (Si una sentencia no es atómica se puede cortar su ejecución pero al cumplir ASV la ejecución no se ve afectada) y las llamamos T1, T2 y T3 respectivamente obtenemos el siguiente subconjunto de historias:

T1, T2, T3 => X=1, Z=6, y=10

T1, T3, T2 => X=1, Z=6, y=7

T2, T1, T3 => X=-2, Z=6, y=10

T2, T3, T1 => X=-2, Z=6, y=10

T3, T1, T2 => X=1, Z=6, y=7

T3, T2, T1 => X=-2, Z=6, y=7

El valor de Z es siempre el mismo ya que no posee ninguna referencia crítica. Los valores de X e Y se ven afectados por la ejecución de T2 ya que sus resultados dependen de la referencia que hacen a la variable Z que es modificada. Entonces, si T1 y T3 se ejecutan antes que T2 ambas usarán el valor inicial de Z que es 3

obteniendo los resultados X=1 e Y=7; ahora si T2 se ejecuta antes que las demás los resultados serán X=-2 e Y=10 y por último, tenemos los casos en que T2 se ejecuta en medio con T1 antes y T3 después o con T3 antes y T1 después.

Nota 1: las instrucciones NO SON atómicas.

Nota 2: no es necesario que liste TODOS los resultados.

Dado el siguiente programa concurrente con memoria compartida:

```
x = 3; y = 2; z = 5;  
co  
x = y * z // z = z * 2 // y = y + 2x  
oc
```

a) Cuáles de las asignaciones dentro de la sentencia co cumplen la propiedad de "A lo sumo una vez". Justifique claramente.

Siendo:

- A: $x = y * z$
- B: $z = z * 2$
- C: $y = y + 2x$

A tiene 2 referencias críticas (a y, a z), por lo tanto no cumple ASV. (además x es leída en C).

B no tiene referencia crítica y es leída por otro (en A se lee z), por lo tanto cumple ASV.

C tiene 1 referencia crítica (a x) y además es leída por otro proceso (en A se lee y), por lo tanto no cumple ASV.

b) Indique los resultados posibles de la ejecución. Justifique.

Si se ejecutan en el orden A, B y C o A, C y B -> $x = 10; z = 10; y = 22$

Si se ejecutan en el orden C, B y A o B, C y A -> $x = 80; z = 10; y = 8$

Si se ejecutan en el orden C, A y B -> $x = 40; z = 10; y = 8$

Si se ejecutan en el orden B, A y C-> $x = 20; z = 10; y = 42$

Si se empieza a ejecutar A leyendo a $y = 2$, y en ese momento se ejecuta C leyendo a $x = 3$ (porque no terminó la asignación de A), y luego termina lo que falta de A y se ejecuta B: $x = 10; z = 10; y = 8$

Si se empieza a ejecutar A leyendo a $y = 2$, y en ese momento se ejecuta C leyendo a $x = 3$ (porque no terminó la asignación de A), y luego se ejecuta B y lo que falta de A: $x = 20; z = 10; y = 8$

Sea la siguiente solución propuesta al problema de alocación SJN (short Job Next):

```
Monitor SJN {  
    Bool libre=true;  
    Cond turno;  
    Procedure request {  
        If (not libre) wait (turno, tiempo);  
        Libre = false;  
    }  
    Procedure release {  
        Libre = true;  
        Signal (turno);  
    }  
}
```

a) ¿Funciona correctamente con disciplina de señalización Signal and continue? Justifique.

Con S&C un proceso que es despertado para poder seguir ejecutando es pasado a la cola de ready en cuyo caso su orden de ejecución depende de la política que se utilice para ordenar los procesos en dicha cola. Puede ser que sea retrasado en esa cola permitiendo que otro proceso ejecute en el monitor antes que el por lo que podría no cumplirse el objetivo del SJN.

b) ¿Funciona correctamente con disciplina de señalización signal and wait? Justifique.

En cambio, con S&W se asegura que el proceso despertado es el próximo en ejecutar después de que el señalador ejecuta signal. Por lo tanto, SJN funcionaría correctamente de esta forma evitando que cualquier otro proceso listo para ejecutar le robe el acceso al proceso despertado.

Definiciones:

Signal and Continued -> el proceso que hace el signal continua usando el monitor, y el proceso despertado pasa

a competir por acceder nuevamente al monitor para continuar con su ejecución (en la instrucción que lógicamente le sigue al wait).

Signal and Wait -> el proceso que hace el signal pasa a competir por acceder nuevamente al monitor, mientras que el proceso despertado pasa a ejecutar dentro del monitor a partir de instrucción que lógicamente le sigue al wait.

Utilice la técnica de “passing the condition” para implementar un semáforo fair usando monitores.

```
monitor Semaforo
{ int s = 1, espera = 0; cond pos;

procedure P ()
{ if (s == 0) { espera ++; wait(pos);}
else s = s-1;
};

procedure V ()
{ if (espera == 0 ) s = s+1
else { espera --; signal(pos);}
};
};
```

Resuelva con monitores el siguiente problema:

Tres clases de procesos comparten el acceso a una lista enlazada: **searchers, inserters y deleters**. Los searchers sólo examinan la lista, y por lo tanto pueden ejecutar concurrentemente unos con otros. Los inserters agregan nuevos items al final de la lista; las inserciones deben ser mutuamente exclusivas para evitar insertar dos items casi al mismo tiempo. Sin embargo un insert puede hacerse en paralelo con uno o más searchers. Por último, los deleters remueven items de cualquier lugar de la lista. A lo sumo un deleter puede acceder la lista a la vez, y el borrado también debe ser mutuamente exclusivo con searchers e inserciones.

```
monitor Controlador_ListaEnlazada{
int numSearchers=0, numInserters=0, numDeleters =0;
cond searchers, inserters, deleters;

procedure pedir_Deleter(){
    while (numSearchers>0 OR numInserters>0 OR numDeleters >0){
wait(deleters);
    numDeleters=numDeleters+1;
}

procedure liberar_Deleter(){
```

```

numDeleters=numDeleters-1;
signal(inserters);
signal(deleters);
signal_all(searchers);
}

procedure pedir_Sercher(){
    while(numDeleters >0){ wait(searchers);}
    numSearchers=numSearchers+1;
}

procedure liberar_Sercher(){
    numSearchers=numSearchers-1;
    if(numSearchers==0 AND numInserters==0 ){
        signal(inserters);
        signal(deleters);
    }
}

procedure pedir_Inserter(){
    while(numDeleters >0 OR numInserters>0 ){ wait(inserters);}
    numInserters=numInserters+1;
}

procedure liberar_Inserter(){
    numInserters=numInserters-1;
    if(numSearchers==0 ){
        signal(inserters);
        signal(deleters);
    }
}
}

process Searchers[i=1..S] {
    Controlador_ListaEnlazada.pedir_Searcher();
    <Realiza búsqueda en la lista>
    Controlador_ListaEnlazada.liberar_Searcher();
}

process Inserters[j=1..I] {
    Controlador_ListaEnlazada.pedir_Inserter();
    <Inserta en la lista>
    Controlador_ListaEnlazada.liberar_Inserter();
}

process Deleters[k=1..D] {
    Controlador_ListaEnlazada.pedir_Delete();
    <Borra en la lista>
    Controlador_ListaEnlazada.liberar_Delete();
}

```

En los protocolos de acceso a sección crítica vistos en clase, cada proceso ejecuta el mismo algoritmo. Una manera alternativa de resolver el problema es usando un proceso coordinador. En este caso, cuando cada proceso SC[i] quiere entrar a su sección crítica le avisa al coordinador, y espera a que éste le dé permiso. Al terminar de ejecutar su sección crítica, el proceso SC[i] le avisa al coordinador.

Desarrolle protocolos para los procesos SC[i] y el coordinador usando sólo variables compartidas (no tenga en cuenta la propiedad de eventual entrada).

```

int aviso [1:n]=([n]0), permiso [1:n]=([n]0);

process SC[i = 1 to N] {
    SNC;
    permiso[i] = 1; # Protocolo
    while (aviso[i]==0) skip; # de entrada
    SC;
    aviso[i] = 0; # Protocolo de salida
    SNC;
}

process Coordinador {
    int i = 1;
    while (true) {
        while (permiso[i]==0) i = i mod N +1;
        permiso[i] = 0;
        aviso[i] = 1;
        while (aviso[i]==1) skip;
    }
}

```

8. Describa la solución utilizando la criba de Eratóstenes al problema de hallar los primos entre 2 y n. Cómo termina el algoritmo? Qué modificaría para que no termine de esa manera?

La criba de Eratóstenes es un algoritmo clásico para determinar cuáles números en un rango son primos. Supongamos que queremos generar todos los primos entre 2 y n. Primero, escribimos una lista con todos los números:

2 3 4 5 6 7 ... n

Comenzando con el primer número no tachado en la lista, 2, recorremos la lista y borramos los múltiplos de ese número. Si n es impar, obtenemos la lista:

2 3 5 7 ... n

En este momento, los números borrados no son primos; los números que quedan todavía son candidatos a ser primos. Pasamos al próximo número, 3, y repetimos el anterior proceso borrando los múltiplos de 3. Si seguimos este proceso hasta que todo número fue considerado, los números que quedan en la lista final serán todos los primos entre 2 y n.

Para solucionar este problema de forma paralelizar podemos emplear un pipeline de procesos filtro. Cada filtro recibe una serie de números de su predecesor y envía una serie de números a su sucesor. El primer número que recibe un filtro es el próximo primo más grande; le pasa a su sucesor todos los números que no son múltiplos del primero.

El siguiente es el algoritmo pipeline para la generación de números primos. Por cada canal, el primer número es primo y todos los otros números no son múltiplo de ningún primo menor que el primer número:

```

Process Criba[1]
{   int p = 2;

    for [i = 3 to n by 2] Criba[2] ! (i);
}

Process Criba[i = 2 to L]
{   int p, proximo;

    Criba[i-1] ? P;
    do Criba[i-1] ? (Proximo) →
        if ((proximo MOD p) <> 0 ) → Criba[i+1] ! (proximo); fi
    od
}

```

El primer proceso, Criba[1], envía todos los números impares desde 3 a n a Criba[2]. Cada uno de los otros procesos recibe una serie de números de su predecesor. El primer número p que recibe el proceso Criba[i] es el i-ésimo primo. Cada Criba[i] subsecuentemente pasa todos los otros números que recibe que no son múltiplos de su primo p. El número total L de procesos Criba debe ser lo suficientemente grande para garantizar que todos los primos hasta n son generados. Por ejemplo, hay 25 primos menores que 100; el porcentaje decrece para valores crecientes de n.

El programa anterior termina en deadlock, ya que no hay forma de saber cuál es el último número de la secuencia y cada proceso queda esperando un próximo número que no llega. Podemos fácilmente modificarlo para que termine normalmente usando centinelas, es decir que al final de los streams de entrada son marcados por un centinela EOS (End Of Stream).

```

Process Criba[1] {
    INT p=2;
    for [i = 3 to n by 2] Criba[2] ! i # pasa impares a Criba[2]
    Criba[2] ! -1;
}

Process Criba[i = 2 TO L] {
    INT p, proximo;
    boolean seguir = true;
    Criba[i-1] ? p      # p es primo
    do (seguir); Criba[i-1] ? proximo -> # recibe próximo candidato
        if (proximo = -1) {
            seguir = false;
            Criba[i+1] ! -1;
        }
        else if ((proximo MOD p) <> 0) # si es primo
            Criba[i+1] ! proximo;    # entonces lo pasa
        fi
    od
}

```

Suponga los siguientes programas concurrentes. Asuma que “función” existe, y que los procesos son iniciados desde el programa principal.

P1	<pre> chan canal (double) process grano1 { int veces, i; double sum; for [veces= 1 to 10] { for [i = 1 to 10000] sum=sum+funcion(i); send canal (sum); } } </pre>	<pre> process grano2 { int veces; double sum; for [veces= 1 to 10] { receive canal (sum); printf (sum); } } </pre>	P2	<pre> chan canal (double) process grano1 { int veces, i; double sum; for [veces= 1 to 10000] { for [i = 1 to 10] sum=sum+i; send canal (sum); } } </pre>	<pre> process grano2 { int veces; double sum; for [veces= 1 to 10000] { receive canal (sum); printf (sum); } } </pre>
----	---	--	----	--	---

a) Analice desde el punto de vista del número de mensajes.

En la P1, sólo se envían 10 mensajes al proceso grano2. En P2 se envían 10000 mensajes.

b) Analice desde el punto de vista de la granularidad de los procesos.

En P1, el envío de los mensajes se realiza después de largos períodos de ejecución ya que entre cada send se ejecuta una iteración de 10000 unidades de tiempo, esto nos asegura que la comunicación entre los dos procesos es poco frecuente. Dadas dichas características podemos decir, que desde el punto de vista de la granularidad, P1 es de granularidad gruesa ya que la comunicación entre los procesos no es de manera reiterada. Al tener mayor granularidad disminuye la concurrencia y la sobrecarga de bloqueos.

En P2, el envío de mensajes se realiza en intervalos cortos de tiempo (entre la ejecución de cada send sólo se ejecuta un for de 1 a 10), aumentando considerablemente la comunicación respecto de P1. Por lo tanto, podemos decir que P2 es de granularidad fina, ya que en cada iteración el volumen de comunicación aumenta, por lo tanto la relación cálculo / comunicación disminuye. Al disminuir la granularidad aumenta la concurrencia pero también aumenta la sobrecarga de bloqueos.

c) Cuál de los programas le parece más adecuado para ejecutar sobre una arquitectura de tipo cluster de PCs? Justifique.

La implementación más adecuada para este tipo de arquitecturas es P1, por ser de granularidad gruesa. Al tratarse de una arquitectura con memoria distribuida la comunicación entre los procesos es más costosa ya que cada proceso puede ejecutarse en computadores diferentes, por lo tanto sería más eficiente que la sobrecarga de comunicación sea lo más baja posible, y dicha característica la brinda la granularidad gruesa.

Suponga los siguientes programas concurrentes. Asuma que EOS es un valor especial que indica el fin de la secuencia de mensajes, y que los procesos son iniciados desde el programa principal.

P1	<pre> chan canal (double) process Genera { int fila, col; double sum; for [fila= 1 to 10000] for [col = 1 to 10000] send canal (a(fila,col)); send canal (EOS) } </pre>	<pre> process Acumula { double valor, sumT; sumT=0; receive canal (valor); while valor<>EOS { sumT = sumT + valor receive canal (valor); } printf (sumT); } </pre>	P2	<pre> chan canal (double) process Genera { int fila, col; double sum; for [fila= 1 to 10000] { sum=0; for [col = 1 to 10000] sum=sum+a(fila,col); send canal (sum); } send canal (EOS) } </pre>	<pre> process Acumula { double valor, sumT; sumT=0; receive canal (valor); while valor<>EOS { sumT = sumT + valor receive canal (valor); } printf (sumT); } </pre>
----	---	--	----	---	--

a) ¿Qué hacen los programas?

Ambos programas realizan la suma de todos los elementos de una matriz pero difieren en la forma de llegar al resultado. P1 Genera le envía los 10000^2 valores a acumula para que este los sume mientras que en P2 Genera solo le envía 10000 valores ya que el mismo se encarga de sumar toda una fila. Es decir, le envía a acumula la suma de todos los elemento de cada fila.

b) Analice desde el punto de vista del número de mensajes.

P1 realiza 10000^2 envíos de mensajes (uno por cada elemento de la matriz) mientras que P2 solo realiza 10000 (uno por cada fila de la matriz).

c) Analice desde el punto de vista de la granularidad de los procesos.

Se puede decir que el proceso P2 es de grano más grueso que el proceso P1 ya que realiza más procesamiento en Genera y esto hace que se necesite un menor número de comunicaciones con Acumula para llegar al resultado.

d) ¿Cuál de los programas le parece más adecuado para ejecutar sobre una arquitectura de tipo cluster de PCs? Justifique.

Las arquitecturas del tipo cluster pueden considerarse arquitecturas de grano grueso ya que poseen muchos procesadores con mucha capacidad de cómputo pero con poca capacidad para la comunicación haciéndolos más apropiados para ejecutar programas de grano grueso que requieren de menos comunicación que capacidad de computo. Por lo tanto, P2 sería más apropiado para ejecutarse sobre este tipo de arquitecturas.

Dada la siguiente solución con monitores al problema de alocación de un recurso con múltiples unidades, transforme la misma en una solución utilizando mensajes asincrónicos.

```
Monitor Alocador_Recurso {
    INT disponible = MAXUNIDADES;
    SET unidades = valores iniciales;
    COND libre;          # TRUE cuando hay recursos
    procedure adquirir( INT Id ) {
        if (disponible == 0) wait(libre)
        else disponible = disponible - 1; remove(unidades, id); }
    procedure liberar( INT id ) {
        insert(unidades, id);
        if (empty(libre)) disponible := disponible + 1
        else signal(libre); }
}
```

```

type clase_op = enum(adquirir, liberar);
chan request(int idCliente, claseOp oper, int idUnidad );
chan respuesta[n] (int id_unidad);

Process Administrador Recurso
{
    int disponible = MAXUNIDADES;
    set unidades = valor inicial disponible;
    queue pendientes;
    while (true)
    {
        { receive request (IdCliente, oper, id_unidad);
            if (oper == adquirir)
                { if(disponible > 0)
                    { disponible = disponible - 1;
                      remove (unidades, id_unidad);
                      send respuesta[IdCliente] (id_unidad);
                    }
                else insert (pendientes, IdCliente);
            }
        }
        { if empty (pendientes)
            { disponible= disponible + 1;
              insert(unidades, id_unidad);
            }
        }
        else
            { remove(pendientes, IdCliente);
              send respuesta[IdCliente](id_unidad);
            }
    }
} //while
} //process Administrador_Recurso

```

```

Process Cliente[i = 1 to n]
{
    int id_unidad;
    send request(i, adquirir, 0);
    receive respuesta[i](id_unidad);
    //Usa la unidad
    send request(i, liberar, id_unidad);
}

```

Dados los siguientes dos segmentos de código, indicar para cada uno de los ítems si son equivalentes o no. Justificar cada caso (de ser necesario dar ejemplos).

Segmento 1	Segmento 2
<pre> ... int cant=1000; <i>new</i> DO (cant < -10); datos?(cant) → Sentencias1 □ (cant > 10); datos?(cant) → Sentencias2 □ (<u>INCognita</u>); datos?(cant) → Sentencias3 END DO ... </pre>	<pre> ... int cant=1000; While (true) { IF (cant < -10); datos?(cant) → Sentencias1 □ (cant > 10); datos?(cant) → Sentencias2 □ (<u>INCognita</u>); datos?(cant) → Sentencias3 END IF } ... </pre>

En todos los casos, en primera instancia el valor de cant va a ser 1000, por lo tanto entra por la guarda (cant>10). Si en esa guarda el valor de cant se modifica (porque llega por el canal, o porque lo modifica Sentencia 2), las guardas deben cubrir todos los posibles valores que toma la variable, para que ambos segmentos sean equivalentes. De no ser así, cuando en el segmento 1 todas las guardas son falsas, el do termina y sigue la ejecución del programa, mientras que el segmento 2, se queda en un bucle hasta que la variable sea modificada por otro proceso.

a) INCognITA equivale a: (cant = 0),

No son equivalentes ya que el DO en segmento 1 terminaría para valores de cant (1...10) (-1...-10) mientras que en segmento 2 el if fallaría pero la iteración se seguiría realizando.

b) INCognITA equivale a: ($\text{cant} > -100$)

Son equivalentes con estas condiciones no hay ningún caso en el que todas las guardas fallen por lo tanto el DO no termina nunca, obteniendo el mismo comportamiento que el while true con el IF.

c) INCognITA equivale a: ($(\text{cant} > 0)$ or $(\text{cant} < 0)$)

No son equivalentes ya que el DO termina si $\text{cant}=0$ porque todas las guardas son false.

d) INCognITA equivale a: ($(\text{cant} > -10)$ or $(\text{cant} < 10)$)

No son equivalentes ya que para $\text{cant} = 10$ o $\text{cant}=-10$ segmento 1 terminaría su ejecución del DO porque todas las guardas serían falsas.

e) INCognITA equivale a: ($(\text{cant} \geq -10)$ or $(\text{cant} \leq 10)$)

Serian equivalentes ya que se estarían eliminando en segmento 1 los valores puestos en d) que provocarían que el DO terminase.

Sea “ocupados” una variable entera inicializada en N que representa la cantidad de slots ocupados de un buffer, y sean P1 y P2 dos programas que se ejecutan de manera concurrente, donde cada una de las instrucciones que los componen son atómicas.

P1:: if ($\text{ocupados} < N$) then begin buffer := elemento_a_agregar; ocupados := ocupados + 1; end;	P2:: if ($\text{ocupados} > 0$) then begin ocupados := ocupados - 1; elemento_a_sacar := buffer; end;
---	---

El programa funciona correctamente para asegurar el manejo del buffer? Si su respuesta es afirmativa justifique. Sino, encuentre una secuencia de ejecución que lo verifique y escríbala, y además modifique la solución para que funcione correctamente (Suponga buffer, elemento_a_agregar y elemento_a_sacar variables declaradas).

El programa no funciona correctamente, observemos la siguiente situación para comprobarlo:

El buffer está lleno. P2 ejecuta el if($\text{ocupados} > 0$) que resulta true, luego decrementa la variable ocupados. En ese momento el P1 ejecuta su respectivo if, que en ese momento es verdadero, y procede a agregar al buffer un nuevo elemento descartando al elemento que P2 todavía no sacó del buffer.

Esta situación es fácil de solucionar, simplemente se posterga el decremento de ocupados de P2 hasta después de obtener el elemento del buffer.

```

P2::  

If(ocupado>0)then  

Begin  

  Elemento_a_sacar:=buffer;  

  Ocupado:=ocupado-1;  

End;

```

Sea "cantidad" una variable entera inicializada en 0 que representa la cantidad de elementos de un buffer, y sean P1 y P2 dos programas que se ejecutan de manera concurrente, donde cada una de las instrucciones que los componen son atómicas.

P1:: <pre> if (cantidad = 0) then begin cantidad:= cantidad + 1; buffer := elemento_a_agregar; end; </pre>	P2:: <pre> if (cantidad > 0) then begin elemento_a_sacar:= buffer; cantidad:= cantidad - 1; end; </pre>
--	--

Además existen dos alumnos de concurrente que analizan el programa y opinan lo siguiente:

"Pepe: este programa funciona correctamente ya que las instrucciones son atómicas".

"José: no Pepe estás equivocado, hay por lo menos una secuencia de ejecución en la cual funciona erróneamente"

¿Con cuál de los dos alumnos está de acuerdo? Si está de acuerdo con Pepe justifique su respuesta. Si está de acuerdo con José encuentre una secuencia de ejecución que verifique lo que José opina y escríbala, y modifique la solución para que funcione correctamente (Suponga buffer y elemento variables declaradas). (22-04-2009)

Estoy de acuerdo con José. Ya que existen secuencias de ejecución en las cuales no funciona correctamente.

Por ejemplo si P1 ejecuta su primera linea y suma cantidad:=cantidad + 1, y corta su ejecución, se ejecuta P2, la condición de cantidad>0 da verdadera pero no existe nada en el buffer y P2 en su primera linea ejecuta elemento_a_sacar := buffer.

4. Dado el siguiente bloque de código, indique para cada inciso que valor queda en aux, o si el código queda bloqueado. Justifique sus respuestas.

```

aux = -1;
...
if (A == 0); P2?(aux) -> aux = aux + 2;
□ (A == 1); P3?(aux) -> aux = aux + 5;
□ (B == 0); P3?(aux) -> aux = aux + 7;
end if;
...

```

- a) Si el valor de A=1 y B=2 antes del IF, y sólo P2 envía el valor 6.
- b) Si el valor de A=0 y B=2 antes del IF, y sólo P2 envía el valor 8.
- c) Si el valor de A=2 y B=0 antes del IF, y sólo P3 envía el valor 6.
- d) Si el valor de A=2 y B=1 antes del IF, y sólo P3 envía el valor 9.
- e) Si el valor de A=1 y B=0 antes del IF, y sólo P3 envía el valor 14.
- f) Si el valor de A=0 y B=0 antes del IF, y P3 envía el valor 9 y P2 el valor 5.

i. Si el valor de A = 1 y B = 2 antes del if, y solo P2 envia el valor 6.

Se queda bloqueado en la guarda A==1 ya que la condición es la única verdadera pero P3 no se ejecuta.

ii. Si el valor de A = 0 y B = 2 antes del if, y solo P2 envia el valor 8.

Entra en la primer guarda (única con condición verdadera) y se ejecuta P2. Luego el valor de aux será 10.

iii. Si el valor de A = 2 y B = 0 antes del if, y solo P3 envia el valor 6.

Entra en la tercera guarda (única con condición verdadera) y se ejecuta P3. Luego el valor de aux será 13.

iv. Si el valor de A = 2 y B = 1 antes del if, y solo P3 envia el valor 9.

El if no tiene efecto ya que ninguna condición es verdadera. Aux mantiene el valor (-1).

v. Si el valor de A = 1 y B = 0 antes del if, y solo P3 envia el valor 14.

Puede entrar en la segunda o tercera guarda (elección no determinista) ya que ambas son

verdaderas y la sentencia de comunicación puede ejecutarse inmediatamente (no hay bloqueo).

Caso A==1: el valor de aux sera 19.

Caso B==0: el valor de aux sera 21.

vi. Si el valor de A = 0 y B = 0 antes del if, P3 envia el valor 9 y P2 el valor 5.

Tanto la condición de la primer guarda y la condición de la ultima son verdaderas; ademas P3 y P2 se ejecutaron, por lo tanto no hay bloqueo. Como la elección es no determinista, pueden suceder dos casos:

Caso A==0: el valor de aux sera 7.

Caso B==0: el valor de aux sera 16.

- 2) Dado el siguiente programa concurrente con memoria compartida, y suponiendo que todas las variables están inicializadas en 0 al empezar el programa y las instrucciones NO son atómicas. Para cada una de las opciones indique verdadero o falso. En caso de ser verdadero indique el camino de ejecución para llegar a ese valor, y en caso de ser falso justifique claramente su respuesta.

P1::
if(x = 0) then
y:= 4*x + 2;
x:= y + 2 + x;

P2::
if(x > 0) then
x:= x + 1;

P3::
x:= x*8 + x*2 + 1;

- a) El valor de x al terminar el programa es 9.
b) El valor de x al terminar el programa es 6.
c) El valor de x al terminar el programa es 11.
d) Y siempre termina con alguno de los siguientes valores: 10 o 6 o 2 o 0.

- a) Verdadero: p2->p1(solo if)->p3->p1(asignación de x e y)
b) Verdadero, caso1: p1(solo if y asignación de y) ->p3->p2->p1(resto)
caso 2: p1(solo if y asignación de y) ->p3->p1(resto) ->p2
c) Falso: la única forma de que el valor de X sea 11 es que al ejecutar P3, X=1 y esto no puede darse porque P2 no puede alterar X porque X=0 y si P1 modificara el valor de X este seria 4.
d) Verdadero.

Y=0: El único caso en el que Y sería 0, es que se ejecute P3 primero alterando el valor de X por lo que al ejecutarse P1 no realizaría ningún calculo.

Y=2: Siempre que asignación se realice antes de que los demás procesos alteren X (X=0). Si primero se ejecuta P1 completando la asignación de Y, el valor final de Y es 2 ya que ningún proceso vuelve a modificarla o modifíco el valor de X.

Y=6: Si primero se ejecuta P2 terminando su ejecución porque x=0 y luego P1 evalúa el if y el control se pasa a P3 en ese instante, X=1 y al volver a realizar la asignación de Y en P1 el valor de Y=6.

Y=10: Se ejecuta P3 habilitando la asignación de P2 por lo que el valor de X=2 y luego se realiza la asignación de Y.

Sea el problema de ordenar de menor a mayor un arreglo de A[1..n]

1. Escriba un programa donde dos procesos (cada uno con n/2 valores) realicen la operación en paralelo mediante una serie de intercambios.

```

Process P1
{ int nuevo, a1[1:n/2]; const mayor = n/2;
  ordenar a1 en orden no decreciente
  P2 ! (a1[mayor]);
  P2 ? (nuevo)
  do a1[mayor] > nuevo →
    poner nuevo en el lugar correcto en a1, descartando el viejo a1[mayor]
    P2 ! (a1[mayor]);
    P2 ? (nuevo);
  od
}

Process P2
{ int nuevo, a2[1:n/2]; const menor = 1;
  ordenar a2 en orden no decreciente
  P1 ? (nuevo);
  P1 ! (a2[menor]);
  do a2[menor] < nuevo →
    poner nuevo en el lugar correcto en a2, descartando el viejo a2[menor]
    P1 ? (nuevo);
    P1 ? (a2[menor]);
  od
}

```

2. ¿Cuántos mensajes intercambian en el mejor de los casos? ¿Y en el peor de los casos?

En el mejor caso, los procesos en el programa necesitan intercambiar solo un par de valores. Solo intercambia dos valores, para comprobar que ya están ordenados. Esto ocurrirá si los $n/2$ valores menores están inicialmente en P1, y los $n/2$ mayores en P2. Así que solo requieren 2 mensajes, uno por cada valor intercambiado.

En el peor caso (que ocurrirá si todo valor está inicialmente en el proceso equivocado) los procesos tendrán que intercambiar $n/2$ valores para tener cada valor en el proceso correcto. Esto equivale a n mensajes, uno por cada valor intercambiado ($n/2$ valores de p1 a p2 y $n/2$ valores de p2 a p1 = $(n/2)^2=n$)

3. Utilice la idea de 1), extienda la solución a K procesos, con n/k valores c/u (“odd-even-exchange sort”).

Asumimos que existen n procesos $P[1:n]$ y que n es par. Cada proceso ejecuta una serie de rondas. En las rondas impares, los procesos impares $P[\text{odd}]$ intercambian valores con el siguiente proceso impar $P[\text{odd}+1]$ si el valor esta fuera de orden. En rondas pares, los procesos pares $P[\text{even}]$ intercambia valores con el siguiente proceso par $P[\text{even}+1]$ si los valores estan fuera de orden. $P[1]$ y $P[n]$ no hacen nada en las rondas pares.

```

//k es el numero de procesos
//n cantidad de numeros a ordenar

process Proc[i:1..k] {
  int largest = n/k;
  int smallest = 1;
  int a[1..k];
  int dato;

  //se ordena el arreglo a de menor a mayor

  for(ronda=1;ronda<=k;ronda++) {

    # si el proceso tiene = paridad que la ronda, pasa valores para adelante
    if(i mod 2 == ronda mod 2) {ronda par y proceso par o ronda impar y
    proceso impar
      if(i!=k) {
        proc[i+1]!a[largest];
        proc[i+1]?dato;
        while(a[largest]>dato) {

          //inserto dato en a ordenado, pisando a[largest]

```

```

        proc[i+1]!a[largest];
        proc[i+1]?dato;
    }
}
else{
    if(i!=1) { # si tiene distinta paridad, recibe valores desde atrás

        proc[i-1]?dato;
        proc[i-1]!a[smallest];
        while(a[smallest]<dato) {

            //inserto dato en a ordenado, pisando a[smallest]

            proc[i-1]?dato;
            proc[i-1]!a[smallest];
        }
    }
}
}

```

¿Cuántos mensajes intercambian en 3) en el mejor caso? ¿Y en el peor de los casos?

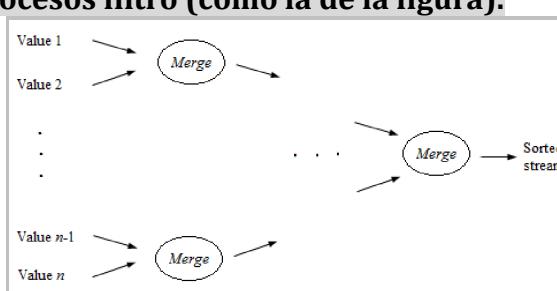
Si cada proceso ejecuta suficientes rondas para garantizar que la lista estará ordenada (en general, al menos k rondas), en el k-proceso, cada uno intercambia hasta $n/k+1$ mensajes por ronda. El algoritmo requiere hasta $k^2 * (n/k+1)$. Se puede usar un proceso coordinador al cual todos los procesos le envían en cada ronda si realizaron algún cambio o no. Si al recibir todos los mensajes el coordinador detecta que ninguno cambió nada les comunica que terminaron. Esto agrega overhead de mensajes ya que se envían mensajes al coordinador y desde el coordinador. Con n procesos tenemos un overhead de $2*k$ mensajes en cada ronda.

Nota: Utilice un mecanismo de pasaje de mensajes, justifique la elección del mismo.

PMS es más adecuado en este caso porque los procesos deben sincronizar de a pares en cada ronda por lo que PMA no sería tan útil para la resolución de este problema ya que se necesitaría implementar una barrera simétrica para sincronizar los procesos de cada etapa.

Suponga los siguientes métodos de ordenación de menor a mayor para n valores (n par y potencia de 2), utilizando pasaje de mensajes:

1. Un pipeline de filtros. El primero hace input de los valores de a uno por vez, mantiene el mínimo y le pasa los otros al siguiente. Cada filtro hace lo mismo: recibe un stream de valores desde el predecesor, mantiene el más chico y pasa los otros al sucesor.
2. Una red de procesos filtro (como la de la figura).



3. Odd/even exchange sort. Hay n procesos $P[1:n]$. Cada uno ejecuta una serie de rondas. En las rondas "impares", los procesos con número impar $P[\text{impar}]$ intercambian valores con $P[\text{impar}+1]$. En las rondas "pares", los procesos con número par $P[\text{par}]$ intercambian valores con $P[\text{par}+1]$ ($P[1]$ y $P[n]$ no hacen nada en las rondas "pares"). En cada caso, si los números están desordenados actualizan su valor con el recibido.

Asuma que cada proceso tiene almacenamiento local sólo para dos valores (el próximo y el mantenido hasta ese momento).

a) ¿Cuántos procesos son necesarios en 1 y 2? Justifique.

Para la alternativa del pipeline de filtros se necesitan n procesos ya que cada uno de ellos calculará un mínimo y pasará el resto de los valores. Por lo tanto, para terminar de procesar todos los números hacen falta tantos procesos como números se quieran ordenar.

Para una red de procesos filtro se necesitan $n-1$ procesos, ya que al ser un árbol binario con $\log_2 n$ niveles tenemos $2^n - 1$ nodos o procesos.

b) ¿Cuántos mensajes envía cada algoritmo para ordenar los valores? Justifique.

Pipeline: Asignándole a cada proceso un número entero consecutivo, siendo el primer proceso el 1, el segundo el 2, así sucesivamente hasta el último proceso n , tenemos que:

Al proceso 1 le llegan n mensajes, y envía $n-1$ mensajes, porque se queda con el valor más chico de todos los que le llegan y va enviando los demás.

Al proceso 2 le llegan $n-1$ mensajes y envía $n-2$ mensajes, por la misma razón del proceso anterior.

Y así sucesivamente, hasta el proceso n , el cual recibe 1 ($n-(n-1)$) mensaje y no manda ningún ($n-n$) mensajes.

En conclusión se envían $(n(n+1))/2$ mensajes

A este resultado se le pueden sumar los mensajes de EOS, cada proceso envía un EOS teniendo n mensajes EOS.

En la red de procesos filtro: siendo n la cantidad de números a ordenar y $\log_2 n$ el total de niveles de la red tenemos que la cantidad de mensajes es $(n * \log_2 n)$.

A lo que podemos sumarle nuevamente los mensajes de EOS, que son los n mensajes que llegan con los números a ordenar (primeros nodos) y el que envía cada nodo ($n-1$) es decir que tenemos $(n-1) + n = 2n - 1$ mensajes extra.

Odd/even Exchange sort : Si cada proceso ejecuta suficientes rondas para garantizar que la lista estará ordenada (en general, al menos k rondas), en el k -proceso, cada uno intercambia hasta $(n/k+1)$ mensajes por ronda. El algoritmo requiere hasta $k^2 * (n/k+1)$.

c) ¿En cada caso, cuáles mensajes pueden ser enviados en paralelo (asumiendo que existe el hardware apropiado) y cuáles son enviados secuencialmente? Justifique.

En el pipeline de filtros en un instante determinado (pipeline lleno) se podrán enviar en paralelo tantos mensajes como procesos tenga el pipeline ya que como cada uno de los procesos recibe, procesa y envía pueden todos estarse enviando los valores en un momento determinado.

En una red de filtros cada proceso envía de a un mensaje por vez ya que el funcionamiento interno de cada uno es igual a la de los filtros en un pipeline pero la diferencia está en la distribución e interacción entre ellos. Por lo tanto, se pueden enviar en paralelo tantos mensajes como procesos haya en el nivel en el que se esté dentro de la red.

Por último, en odd/even Exchange sort pueden enviarse en paralelo tantos mensajes como procesos se encuentren involucrados en una ronda ya que todos enviaran un valor.

d) ¿Cuál es el tiempo total de ejecución de cada algoritmo? Asuma que cada operación de comparación o de envío de mensaje toma una unidad de tiempo. Justifique.

Algoritmo i: Utilizando el cálculo de la cantidad de mensajes del punto a) y razonando que cada vez que se envía un mensaje se realiza una comparación antes, tenemos que el tiempo de ejecución será igual a la cantidad de mensajes multiplicada por 2 unidades de tiempo (1 comparación + 1 mensaje): $2*((n(n+1))/2) = 2 n (n + 1) = n (n + 1) \Rightarrow O(n^2)$

Algoritmo ii: Por la misma razón que el algoritmo anterior tenemos:

$$2 (n * \log_2 n) \Rightarrow O(n)$$

Algoritmo iii: Cada proceso en cada ronda hace una comparación, envía, recibe y hace una asignación, por eso tenemos que por cada proceso se tarda 4 unidades. Como se necesitan n rondas en el peor de los casos, se realizará $4n$ unidades de tiempo en total, por lo tanto es de $O(n)$.

"ESTE PUNTO MUY DUDOSO"

Suponga que la solución a un problema es paralelizada sobre p procesadores de dos maneras diferentes. En un caso, el speedup (S) está regido por la función $S=p-1$ y en el otro por la función $S=p/2$. ¿Cuál de las dos soluciones se comportará más eficientemente al crecer la cantidad de procesadores? Justifique claramente.

El que mejor se comportara con mayor número de procesadores es aquel cuya función de speedup es $p-1$ por definición de speedup su rango esta entre 0 y p entonces con esta función el speedup es más cercano a p . Si comparamos las eficiencias tenemos que:

$E=S/p$ entonces tenemos $p-1/p$ y $(p/2)/p$ cuanto más grande sea p mayor eficiencia tendrá la primer solución ya que su valor será casi uno y la otra solución no alcanzara nunca una mayor eficiencia que la mitad.

Ahora suponga $S=1/p$ y $S=1/p^2$.

Es más eficiente la solución con speedup $1/p$ su speedup siempre será mayor. Además su eficiencia será siempre mayor que la de la segunda solución, ambas a medida que crecen los procesadores van disminuyendo su eficiencia pero la función $1/p$ decrece más lentamente.

Suponga que la solución a un problema es paralelizada sobre p procesadores de dos maneras diferentes. En un caso, el speedup (S) está regido por la función $S=p/3$ y en el otro por la función $S=p-3$. ¿Cuál de las dos soluciones se comportará más eficientemente al crecer la cantidad de procesadores? Justifique claramente.

Suponiendo el uso de 5 procesadores:

$$\text{Solución 1} \Rightarrow S=5/3=1,66$$

$$\text{Solución 2} \Rightarrow S=5-3=2$$

Ahora, incrementamos la cantidad de procesadores suponemos 100 procesadores:

$$\text{Solución 1} \Rightarrow S=100/3=33,33$$

$$\text{Solución 2} \Rightarrow S=100-3=97$$

Podemos decir, que a medida que p tiende a infinito, para la solución 1 siempre el Speedup será la tercera parte en cambio para la solución 2 el valor "-3" se vuelve despreciable. Por lo tanto la solución 2 es la que se comporta más eficientemente al crecer la cantidad de procesadores.

Suponga que la solución a un problema es paralelizada sobre p procesadores de dos maneras diferentes. En un caso, el speedup(s) esta regido por la función $S= p-4$ y el otro por la función $S= p/3$ para $p > 4$. ¿Cuál de las dos soluciones se comportara más eficientemente al crecer la cantidad de procesadores?

Si la cantidad de procesadores crece, la primera solución se comportara más eficientemente. Ya que si p es muy grande y se divide en 3 partes, estas no serán tan grandes. En cambio, si p es muy grande y se le resta 4, el resultado seguiría siendo muy grande.

Suponga que la solución a un problema es paralelizada sobre p procesadores de dos maneras diferentes. En un caso, la eficiencia está regido por la función $E=1/p$ y en el otro por la función $E= 1/p^2$. ¿Cuál de las dos soluciones se comportará más eficientemente al crecer la cantidad de procesadores? Justifique.

Claramente se observa que a partir de $p=2$, $E_1=1/p$ será mayor que $E_2= 1/p^2$, observemos los siguientes casos:

- I. $p = 1$, tenemos $E_1 = 1/1 = 1$ y $E_2 = 1/1 = 1$
- II. $p = 2$, tenemos $E_1 = 1/2 = 0,5$ y $E_2 = 1/4 = 0,25$
- III. $p = 3$, tenemos $E_1 = 1/3 = 0,33$ y $E_2 = 1/9 = 0,11$

Como se puede apreciar, a partir de $p=2$ E_1 se mantiene más eficiente que E_2 , pero sin embargo ambos van a decrecer conforme p crezca, por lo tanto ninguno de los dos se comportará más eficiente al crecer la cantidad de procesadores.

Suponga que el tiempo de ejecución de un algoritmo secuencial es de 1000 unidades de tiempo, de las cuales el 80% corresponden a código paralelizable. ¿Cuál es el límite en la mejora que puede obtenerse paralelizando el algoritmo?

El límite de mejora se da teniendo 800 procesadores los cuales ejecutan una unidad de tiempo obteniendo un tiempo paralelo de 201 unidades de tiempo. El speedup mide la mejora de tiempo obtenida con un algoritmo paralelo comparándola con el tiempo secuencial. $S=Ts/T_p=1000/201= 4,97\sim 5$ aunque se utilicen más procesadores el mayor speedup alcanzable es el anterior cumpliéndose así la **ley de Amdahl** que dice que para un problema existe un límite en la paralización del mismo que no depende del número de procesadores sino que depende de la cantidad de código secuencial.

Si el tiempo de ejecución de un algoritmo secuencial es de 10000 unidades y el porcentaje de unidades paralelizables es del 80%, quiere decir que 2000 unidades (el 20%) deben ejecutarse secuencialmente, por lo tanto el tiempo mínimo esperable es de 2000 para un procesador. Por esta razón nos conviene utilizar una cantidad de procesadores tal que los mantenga a todos ocupados esa cantidad de tiempo, es decir en este caso conviene tener 5 procesadores porque si tuviéramos más un procesador estaría trabajando 2000 unidades de tiempo y los demás terminarían de ejecutar las instrucciones antes, y tendrían que esperarlo.

Suponga que el tiempo de ejecución de un algoritmo secuencial es de 8000 unidades de tiempo, de las cuales solo el 90% corresponde a código paralelizable. ¿Cuál es el límite en la mejora que puede obtenerse paralelizando el algoritmo? Justifique.

Si yo tengo 8000 unidades de tiempo de las cuales el 90% de éstas pueden ser paralelizables entonces:

La mejora de la paralelización de un algoritmo se mide con el Speedup.

El tiempo total del algoritmo lo podemos dividir en:

$$\begin{aligned} T &= T_{sec} + T_{par} \\ T &= 800 + 7200 \end{aligned}$$

Ahora suponiendo que el tiempo paralelizable (T_{par}) que es 7200 podemos reducirlo a 1 si pudiéramos utilizar 7200 procesadores, y el tiempo secuencial (T_{sec}) que es 800 no puede disminuir de ese número porque se debe ejecutar solo por un procesador, por lo tanto:

$$T_{mejor} = 800 + 1 = 801$$

Y ahora calculamos el límite de mejora que podemos llegar a obtener:

$S = T / T_{mejor}$ (paralelo)
 $S = 8000 / 801 \rightarrow$ Aproximadamente 10, por lo tanto, ese es el límite de mejora alcanzable.

Suponga que el tiempo de ejecución de un algoritmo secuencial es de 10000 unidades de tiempo, de las cuales 95% corresponden a código paralelizable. ¿Cuál es el límite en la mejora que puede obtenerse paralelizado el algoritmo?

El límite de mejora se consigue utilizando 9500 procesadores los cuales ejecutan una unidad de tiempo obteniendo un tiempo paralelo de 1 unidad de tiempo + las 500 unidades de tiempo del código secuencial, siendo un total de 501 unidades de tiempo. El speedup mide la mejora del tiempo obtenida con un algoritmo paralelo comparándola con el secuencial. $Speedup = TS/TP = 10000/501 = 19,9 \sim 20$. Aunque se utilicen más procesadores el mayor speedup alcanzable es este, cumpliéndose así la Ley de Amdahl que establece que para todo problema hay un límite de paralelización, dependiendo el mismo no de la cantidad de procesadores, sino de la cantidad de código secuencial.

Indique los posibles valores finales de x en el siguiente programa (justifique claramente su respuesta):

```
int x = 3; sem s1 = 1, s2 = 0;
co P(s1); x = x * x; V(s1);
# P1
// P(s2); P(s1); x = x * 3; V(s1);
# P2
// P(s1); x = x - 2; V(s2); V(s1);
# P3
oc
```

s1 → mutex: puede pasar un solo proceso por vez.

s2 → semáforo de señalización: esperan señalización de un evento y pasa sólo uno.

P1 y P3 comienzan esperando a s1. Por ser un mutex, sólo puede continuar uno de ellos y no será interrumpido por el otro hasta liberar a s1.

- Si comienza P1: Asigna x=9, luego incrementa s1 permitiendo que continúe P3. P3 asigna x=7 y señala s2. Esto habilita a P2 que estaba esperando. Si P2 continúa, intentará obtener s1 con lo cual se vuelve a bloquear volviendo el control a P3. En cualquier caso, P3 libera a s1 y termina. P2 es despertado, asigna x = 21 y termina. Valor final **x=21**.

- Si comienza P3: Asigna x = 1 y señala a s2. Esto habilita a P2 que estaba esperando. Si P2 continúa, intentará obtener s1 con lo cual se vuelve a bloquear volviendo el control a P3. Cuando P3 libera a s1, P1 y P2 pueden competir por él:

-Si gana P1: asigna x=1, libera a s1 y termina; finaliza P2 y asigna x = 3. Valor final **x=3**.

-Si gana P2: asigna x=3, libera a s1 y termina; finaliza P1 y asigna x = 9. Valor final **x=9**.

P2 nunca puede comenzar la historia ya que espera un semáforo de señalización que sólo P3 señala. Cualquier historia en la que P2 esté antes de P3 es inválida.

En todas las historias los semáforos terminan con los mismos valores con los que están inicializados.

c) Dado el siguiente programa concurrente indique cuáles valores de K son posibles al finalizar, y describa una secuencia de instrucciones para obtener dicho resultado:

Process P1{ fa i=1 to K → N=N+1 af} Process P2{ fa i=1 to K → N=N+1 af}

- i) 2K
- ii) 2K+2
- iii) K
- iv) 2

- i) 2K -> VALOR POSIBLE (Si ejecuta todo el ForAll P1 y luego todo el ForAll P2)
 ii) 2K+2 -> VALOR IMPOSIBLE
 iii) K -> VALOR POSIBLE (Si ejecutan al mismo tiempo P1 y P2)
 iv) 2 -> VALOR POSIBLE (Si P1 carga la variable, P2 hace ForAll de k-1, luego P1 realiza el incremento de n y luego sucede lo mismo para P2)

Suponga que N procesos poseen inicialmente cada uno un valor. Se debe calcular la suma de todos los valores y al finalizar la computación todos deben conocer dicha suma.

Analice (desde el punto de vista del número de mensajes y la performance global) las soluciones posibles con memoria distribuida para arquitecturas en Estrella (centralizada), Anillo Circular, Totalmente Conectada y Árbol.

Arquitectura en estrella (centralizada)

En este tipo de arquitectura todos los procesos (workers) envían su valor local al procesador central (coordinador), este suma los N datos y reenvía la información de la suma al resto de los procesos. Por lo tanto se ejecutan $2(N-1)$ mensajes. Si el procesador central dispone de una primitiva broadcast se reduce a N mensajes.

En cuanto a la performance global, los mensajes al coordinador se envían casi

```

chan valor(INT),
resultados[n] (INT suma);

Process P[0]{      #coordinador, v esta inicializado
    INT v;  INT sum=0;
    sum = sum+v;
    for [i=1 to n-1]{
        receive valor(v);
        sum=sum+v;
    }
    for [i=1 to n-1]
        send resultado[i] (sum);
}

process P[i=1 to n-1]{ #worker, v esta inicializado
    INT v;  INT sum;
    send valor(v);
    receive resultado[i] (sum);
}
  
```

al mismo tiempo. Estos se quedaran esperando hasta que el coordinador termine de computar la suma y envié el resultado a todos.

Anillo circular

Se tiene un anillo donde $P[i]$ recibe mensajes de $P[i-1]$ y envía mensajes a $P[i+1]$. $P[n-1]$ tiene como sucesor a $P[0]$. El primer proceso envía su valor local ya que es lo único que conoce.

Este esquema consta de dos etapas:

1. Cada proceso recibe un valor y lo suman a su suma local.
2. Todos reciben la suma global.

$P[0]$ debe ser algo diferente para poder enviar su valor local ya que es lo único que conoce.

A diferencia de la solución centralizada, esta reduce los requerimientos de memoria por proceso pero tardara más en ejecutarse, por más que el número de mensajes requeridos sea el mismo. Esto se debe a que cada proceso debe esperar un valor para computar una suma parcial y luego enviársela al siguiente proceso; es decir, un proceso trabaja por vez, se pierde el paralelismo.

```

chan valor[n] (sum);
process p[0]{
    INT v; INT suma = v;
    send valor[1] (suma);
    receive valor[0] (suma);
    send valor[1] (suma);
}

process p[i = 1 to n-1]{
    INT v; INT suma;
    receive valor[i] (suma);
    suma = suma + v;
    send valor[(i+1) mod n] (suma);
    receive valor[i] (suma);
    if (i < n-1)
        send valor[i+1] (suma);
}

```

Totalmente conectada (simétrica)

Todos los procesos ejecutan el mismo algoritmo. Existe un canal entre cada par de procesos.

Cada uno transmite su dato local v a los $n-1$ restantes. Luego recibe y procesa los $n-1$ datos que le faltan, de modo que en paralelo toda la arquitectura está calculando la suma total y tiene acceso a

Se ejecutan $n(n-1)$ mensajes. Si se consideran los canales entre procesos, habrá n^2 canales. Serán n mensajes. Es la solución más económica ya que requiere el menor número de mensajes si no hay broadcast.

Arbol

Se tiene una red de procesadores (nodos) conectados por canales de comunicación bidireccionales. Cada nodo se comunica directamente con sus vecinos. Si un nodo quiere enviar un mensaje a toda la red, debería construir un árbol de expansión de la misma, poniéndose a él mismo como raíz.

El nodo raíz envía un mensaje por broadcast a todos los hijos, junto con el árbol construido. Cada nodo examina el árbol recibido para determinar los hijos a los cuales deben reenviar el mensaje, y así sucesivamente.

Se envían $n-1$ mensajes, uno por cada parente/hijo del árbol.

Implemente una solución al problema de exclusión mutua distribuida entre N procesos utilizando un algoritmo de tipo token passing con mensajes asincrónicos.

El algoritmo de token passing, se basa en un tipo especial de mensaje o “token” que puede utilizarse para otorgar un permiso (control) o para recoger información global de la arquitectura distribuida.

Si User[1:n] son un conjunto de procesos de aplicación que contienen secciones críticas y no críticas. Hay que desarrollar los protocolos de interacción (E/S a las secciones críticas), asegurando exclusión mútua, no deadlock, evitar demoras innecesarias y eventualmente fairness.

Para no ocupar los procesos User en el manejo de los tokens, ideamos un proceso auxiliar (helper) por cada User, de modo de manejar la circulación de los tokens. Cuando helper[i] tiene el token adecuado, significa que User[i] tendrá prioridad para acceder a la sección crítica.

```
chan token[n]();                                # para envio de tokens
chan enter[n](), go[n](), exit[n]();             # para comunicación proceso-helper

process helper[i = 1..N] {
    while(true){
        receive token[i]();                      # recibe el token
        if(!empty(enter[i])){                     # si su proceso quiere usar la SC
            receive enter[i]();
            send go[i]();                         # le da permiso y lo espera a que termine
            receive exit[i]();
        }
        send token[i MOD N +1]();                # y lo envia al siguiente ciclicamente
    }
}

process user[i = 1..N] {
    while(true){
        send enter[i]();
        receive go[i]();
        ... sección critica ...
        send exit[i]();
        ... sección no critica ...
    }
}
```

(Broadcast atómico). Suponga que un proceso productor y n procesos consumidores comparten un buffer unitario. El productor deposita mensajes en el buffer y los consumidores los retiran. Cada mensaje depositado por el productor tiene que ser retirado por los n consumidores antes de que el productor pueda depositar otro mensaje en el buffer.

a) Desarrolle una solución utilizando semáforos.

```

sem depositar:=1;
sem retirar:= 1;
sem consumir[n]:=[[n] 0];
int cant_consumido:=[[n] 0];
T buffer;

process productor{
    while(true){
        P(depositar);
        buffer:= generarDatos(); //Devuelve un entero para el buffer
        cant_consumido:= 0;
        for i to n do
            V(consumir[i])
    }
}

process consumidor[i: 1..n]{
    T dato;
    while(true){
        P(consumir[i])           //Espero que el dato este en el buffer
        P(retirar)               //Espero para tener acceso al buffer
        dato:=buffer;
        cant_consumido++;
        if (cant_consumido == n)
            V(depositar);
        V(retirar);
    }
}

```

b) Suponga que el buffer tiene b slots. El productor puede depositar mensaje sólo en slots vacíos y cada mensaje tiene que ser recibido por los n consumidores antes de que el slot pueda ser reusado. Además, cada consumidor debe recibir los mensajes en el orden en que fueron depositados (note que los distintos consumidores pueden recibir los mensajes en distintos momentos siempre que los reciban en orden). Extienda la respuesta dada en a) para resolver este problema más general.

```

sem retirar[tamBuffer]:= 1;           //semaforo para cada slot del buffer
sem consumir[n]:=[[n] 0];
int cant_consumido[tamBuffer]:=[[n] 0];
T buffer[tamBuffer];

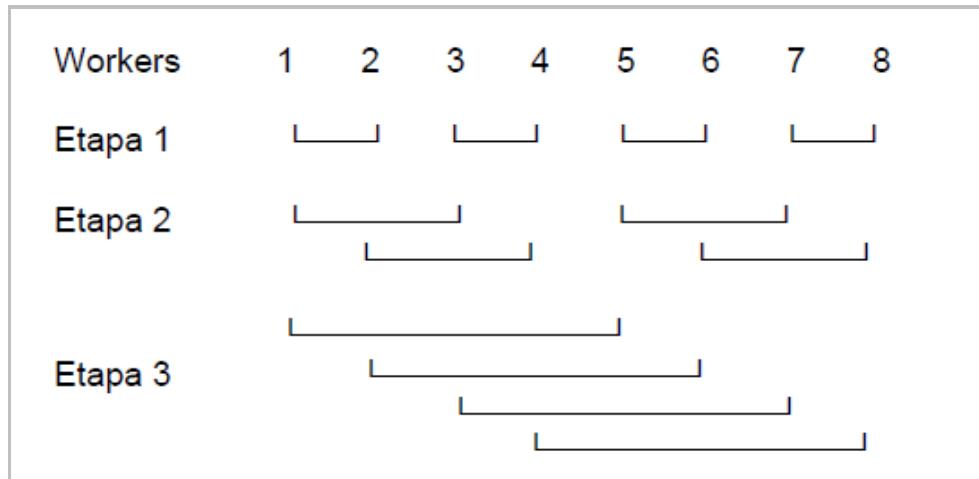
process productor{
    int posLibre:= 0;                //Siguiente posicion libre del buffer (productor)
    while(true){
        P(depositar);
        buffer[posLibre]:=generarDatos(); //Devuelve dato tipo T para el buffer
        cant_consumido[posLibre]:= 0;
        for i to n
            V(consumir[i]);
        posLibre:= (posLibre + 1) mod n;
    }
}

process Consumidor[i: 1..n]{
    int post_actual:=0;              //Slot del que debe consumir
    T dato;
    while(true){
        P(consumir[i]);
        P(retirar[post_actual]);     //Espera por un slot en particular
        dato:=buffer;
        cant_consumido[post_actual]++;
        if (cant_consumido[post_actual] == n)
            V(depositar);
        V(retirar[post_actual]);
        post_actual[i]:= (post_actual[i] + 1) mod n;
    }
}

```

Implemente una butterfly barrier para 8 procesos usando variables compartidas.

Una butterfly barrier tiene $\log_2 n$ etapas. Cada Worker sincroniza con un Worker distinto en cada etapa. En particular, en la etapa s un Worker sincroniza con un Worker a distancia $2^{(s-1)}$. Se usan distintas variables flag para cada barrera de dos procesos. Cuando cada Worker pasó a través de $\log_2 n$ etapas, todos los Workers deben haber arribado a la barrera y por lo tanto todos pueden seguir. Esto es porque cada Worker ha sincronizado directa o indirectamente con cada uno de los otros.



```
int N =8;
int E=log(N);
int arribo[1:N] = ([n] 0);

process Worker[i = 1 to n] {
    int j;
    int resto;
    int distancia;

    while (true) {
        //Sección de código anterior a la barrera.
        //Inicio de la barrera

        for (etapa = 1; etapa<=E; etapa++) {
            distancia = 2^(s-1);
            resto = (i mod 2^s);
            if (resto==0)or(resto>distancia) //Ejemplo, 7 con 8 ronda 1, 1 con 5 ronda 3
                distancia = -distancia;
            j = i + distancia;
            while (arribo[i] == 1) skip;
            arribo[i] = 1;
            while (arribo[j] == 0) skip;
            arribo[j] = 0;
        }
        //Fin de la barrera
        //Sección de código posterior a la barrera.
    }
}
```

Suponga n^2 procesos organizados en forma de grilla cuadrada. Cada proceso puede comunicarse solo con los vecinos izquierdo, derecho, de arriba y de abajo (los procesos de las esquinas tienen solo 2 vecinos, y los otros en los bordes de la grilla tienen 3 vecinos). Cada proceso tiene inicialmente un valor local v .

- a) Escriba un algoritmo heartbeat que calcule el máximo y el mínimo de los n^2 valores. Al terminar el programa, cada proceso debe conocer ambos valores. (Nota: no es necesario que el algoritmo esté optimizado).

```

Chan valores[1:n;1:n] (int);

Process P[i = 1 to n, j = 1 to n]{
    Int v;
    Int Nuevo, minimo=v, maximo=v;
    Int cantVecinos;
    Vecinos[1..cantVecinos]

    For [k = 1 to cantGeneraciones]{
        For [p = 1 to cantVecinos]{
            Send valores[ vecinos[p].fila,vecinos[p].columna ] (v)
        }
        For [p = 1 to cantVecinos]{
            Receive valores[i,j] (nuevo)
            if nuevo < minimo
                minimo= nuevo
            if nuevo> máximo
                Máximo = nuevo
        }
    }
}

```

b) Analice la solución desde el punto de vista del número de mensajes.

4 procesos (las esquinas) envían solo 2 mensajes por ronda. $4*2*(n-1)*2 = (n-1)*16$ mensajes

$(n-2)$ * 4 procesos (los bordes) envían 3 mensajes por ronda.
 $(n-2)*2*3*(n-1)*2 = (n-1)(n-2)*12$ mensajes.

$(n-2)*(n-2)$ procesos envían 4 mensajes por ronda. $(n-2)^2*4*(n-1)*2 = (n-2)^2*(n-1)*8$.

c) ¿Puede realizar alguna mejora para reducir el número de mensajes?

No, no existe una manera de determinar cuándo un proceso obtuvo el mínimo o el máximo por lo que es necesario para asegurarse que todos los procesos tienen los valores globales de mínimo y máximo que se ejecuten las $(n-1)*2$.

Suponga que una imagen se encuentra representada por una matriz a ($n \times n$), y que el valor de cada pixel es un número entero que es mantenido por un proceso distinto(es decir, el valor del pixel I,J , esta en el proceso $P(I,J)$. Cada proceso puede comunicarse solo con su vecinos izquierdo, derecho, arriba y abajo.(los procesos de los esquinas tienen solo 2 vecinos, y los otros bordes de la grilla tienen 3 vecinos) .

- a) **Escriba un algoritmo Herbeat que calcule el maximo y el minimo valor de los pixels de la imagen. Al terminar el programa, cada proceso debe conocer ambos valores.**

```

chan topologia[1:n](emisor : int; listo : bool; top : [1:n,1:n] bool, max : int, min : int);

process nodo[p = 1..n] {
    bool vecinos[1:n];           # inicialmente vecinos[q] true si q es vecino de p
    bool activo[1:n] = vecinos;   # vecinos aún activos
    bool top[1:n,1:n] = ([n*n]false); # vecinos conocidos (matriz de adyacencia)
    bool nuevatom[1:n,1:n];
    int r = 0;
    bool listo = false;
    int emisor;
    bool qlisto;
    int miValor, max, min;       # miValor inicializado con el valor del pixel
    top[p,1..n] = vecinos;       # llena la fila para los vecinos
    max := miValor; min:= miValor;

    while(not listo) {           # envia conocimiento local de la topología a
        for[q = 1 to n st activo[q]] # sus vecinos
            send topologia[q](p, false, top, max, min);
        for [q = 1 to n st activo[q]] {
            receive topologia[p](emisor,qlisto,nuevatom, nuevoMax, nuevoMin);
            top = top or nuevatom;      # recibe las topologías y hace OR con su top juntando la información
            if(nuevoMax>max) nuevoMax := max; # actualiza los máximos y mínimos
            if(nuevoMin<min) nuevoMin := min;
            if(qlisto) activo[emisor] = false;
        }
        if(todas las filas de top tiene 1 entry true) listo = true;
        r := r + 1;
    }
    # envía topología completa a todos sus vecinos aún activos
    for[q = 1 to n st activo[q]]
        send topologia[q](p,listo,top, max, min);
    # recibe un mensaje de cada uno para limpiar el canal
    for [q=1 to n st activo[q]]
        receive topologia[p](emisor,d,nuevatom, nuevoMax, nuevoMin);
    }
}

```

b) Analice la solución de desde el punto de vista del número de mensajes.

Si M es el número máximo de vecinos que puede tener un nodo, y D es el diámetro de la red, el número de mensajes máximo que pueden intercambiar es de $2n * m * (D+1)$. Esto es porque cada nodo ejecuta al sumo $D-1$ rondas, y en cada una de ellas manda 2 mensajes a sus m vecinos.

c) Puede realizar alguna mejora para reducir el número de mensajes.

El algoritmo centralizado requiere el intercambio de $2n$ mensajes, uno desde cada nodo al server central y uno de respuesta. El algoritmo descentralizado requiere el intercambio de más mensajes. Si m y D son relativamente chicos comparados con n , entonces el número de mensajes no es mucho mayor que para el algoritmo centralizado. Además, estos pueden ser intercambiados en paralelo en muchos casos, mientras que un server centralizado espera secuencialmente recibir un mensaje de cada nodo antes de enviar cualquier respuesta.

Dado el siguiente programa concurrente, indique cuál es la respuesta correcta (justifique claramente):

```

int a = 1, b = 0;
co <await (b = 1) a = 0 >
// while (a = 1) { b = 1; b = 0; }
oc

```

- a) Siempre termina
- b) Nunca termina
- c) Puede terminar o no

Desde el punto de vista conceptual con una política fuertemente fair, el programa eventualmente termina, porque b se vuelve infinitamente 1. Sin embargo con una política débilmente fair, el programa puede no terminar, porque b es también infinitamente 0. Desafortunadamente, es imposible idear un procesador con una política de scheduling que sea práctica y fuertemente fair. Por ejemplo, RR y el time slicing son prácticos pero no fuertemente fair, porque, en general, los procesos ejecutan en orden impredecible.

Un scheduler multiprocesador que ejecute los procesos en paralelo también es práctico, pero no es fuertemente fair. Esto se debe a que el segundo proceso siempre puede examinar b cuando es 0.

Resuelva el problema de encontrar la topología de una red utilizando mensajes asincrónicos. Muestre con un ejemplo la evolución de la matriz de adyacencia para una red con al menos 7 nodos y de diámetro al menos 4. Compare conceptualmente con una solución utilizando PMS.

```

chan topologia[1:n] (emisor : int; listo : bool; top : [1:n,1:n] bool)

Process Nodo[p:1..n] {
    bool vecinos[1:n];# inicialmente vecinos[q] true si q es vecino de Nodo[p]
    bool activo[1:n] = vecinos # vecinos aún activos
    bool top[1:n,1:n] = ([n*n]false) # vecinos conocidos
    int r = 0; bool listo = false;
    int emisor; bool qlisto; bool nuevatop[1:n,1:n];
    top[p,1..n] = vecinos; # llena la fila para los vecinos

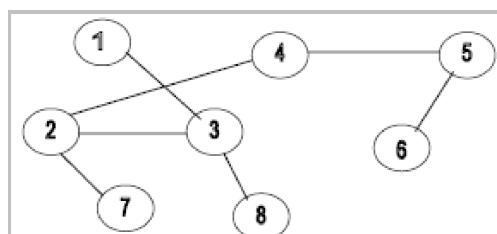
    while (not listo) {
        # envía conocimiento local de la topología a sus vecinos
        for [q = 1 to n st activo[q] ] send topologia[q](p,false,top);
        # recibe las topologías y hace or con su top
        for [q = 1 to n st activo[q] ] {
            receive topologia[p](emisor,qlisto,nuevatop);
            top = top or nuevatop;
            if (qlisto) activo[emisor] = false;
        }
        if (todas las filas de top tiene 1 entry true) listo=true;
        r := r + 1
    }

    # envía topología a todos sus vecinos aún activos
    for [q = 1 to n st activo[q] ] send topologia[q](p,listo,top);
    # recibe un mensaje de cada uno para limpiar el canal
    for [q=1 to n st activo[q]] receive topologia[p](emisor,d,nuevatop)
}

```

Representamos cada nodo por un proceso `Nodo[p:1..n]`. Dentro de cada proceso, podemos representar los vecinos de un nodo por un vector booleano `vecinos[1:n]` con el elemento `vecinos[q]` true en `Nodo[p]` si `q` es vecino de `p`. Estos vectores se asumen inicializados con los valores apropiados. La topología final `top` puede ser representada por una matriz de adyacencia, con `top:[1:n,1:n]` true si `p` y `q` son nodos vecinos.

Después de `r` rondas, el nodo `p` conocerá la topología a distancia `r` de él. En particular, para cada nodo `q` dentro de la distancia `r` de `p`, los vecinos de `q` estarán almacenados en la fila `q` de `top`. Dado que la red es conectada, cada nodo tiene al menos un vecino. Así, el nodo `p` ejecutó las suficientes rondas para conocer la topología tan pronto como cada fila de `top` tiene algún valor true. En ese punto, `p` necesita ejecutar una última ronda en la cual intercambia la topología con sus vecinos; luego `p` puede terminar. Esta última ronda es necesaria pues `p` habrá recibido nueva información en la ronda previa. También evita dejar mensajes no procesados en los canales. Dado que un nodo podría terminar una ronda antes (o después) que un vecino, cada nodo también necesita decirle a sus vecinos cuando termina. Para evitar deadlock, en la última ronda un nodo debería intercambiar mensajes solo con sus vecinos que no terminaron en la ronda previa.



Inicialmente en el nodo 3:

	1	2	3	4	5	6	7	8
1								
2								
3	T	T	T					T
4								
5								
6								
7								
8								

Después de una ronda, en nodo 3:

	1	2	3	4	5	6	7	8
1	T		T					
2		T	T	T			T	
3	T	T	T					T
4								
5								
6								
7								
8			T					T

Después de dos rondas, en nodo 3:

	1	2	3	4	5	6	7	8
1	T		T					
2		T	T	T			T	
3	T	T	T					T
4		T		T	T			
5								
6								
7		T					T	
8			T					T

PARA MI ACA DEBERIA SEGUIR LA RONDA HASTA LLEGAR HASTA 4 QUE ES EL DIÁMETRO.

El proceso del broadcast es un proceso asincrónico, resolverlo con PMS causa que se reduzca la eficiencia.

a. ¿Qué mecanismo de pasaje de mensaje es más adecuado para la resolución? Justifique claramente.

El mecanismo de pasaje de mensajes asincrónico es el más adecuado para la resolución junto con algoritmos heartbeat ya que nos permite una interacción entre procesos de manera que cada procesador es modelizado por un proceso y los links de comunicación con canales compartidos. En particular, cada proceso ejecuta una secuencia de iteraciones. En cada iteración, un proceso envía su conocimiento local de la topología a todos sus vecinos, luego recibe la información de ellos y la combina con la suya. La computación termina cuando todos los procesos aprendieron la topología de la red entera. El criterio de terminación no siempre puede ser decidida localmente.