Rejunte de preguntas teóricas

1.- Mencione al menos 3 ejemplos donde pueda encontrarse concurrencia.

- Reproductor de DVD. Mientras lee del dispositivo, decodifica audio y video, controla temporización de subtítulos, envía señales de salida.
- Videojuego de estrategia. Mientras mantiene actualizado el mapa en pantalla, calcula movimientos de enemigos, captura órdenes de usuario, controla el desarrollo del juego.
- Procesador de textos. Mientras recibe entradas del usuario, controla ortografia, realiza sugerencias, guardados automáticos.
- Una ruta donde cada conductor está conduciendo un vehículo, además de prestar atención al resto de los vehículos, señales de tránsito y demás factores que alteren su camino.
- En un cine donde las personas miran la película y están atentos al contexto.

2.- Escriba una definición de concurrencia. Diferencie procesamiento secuencial, concurrente y paralelo.

- **Concurrencia** es la capacidad de ejecutar multiples actividades simultaneamente o en paralelo. En el **procesamiento concurrente**, si bien en un mismo instante de tiempo hay una tarea en ejecución, existe más de una tarea activa a la vez. De esta forma todos los trabajos están “en la mano” al mismo tiempo pero no todos necesitan estar operando en dicho momento.
- En el **procesamiento secuencial** estamos obligados a establecer un estricto orden temporal en los programas, dado que se cuenta con un único flujo de control que ejecuta una instrucción, y al finalizar, ejecuta la siguiente. Estos programas son totalmente ordenados y determinísticos, implicando que para los mismos datos de entrada ejecuta la misma secuencia de instrucciones y obtiene la misma salida.
- En el **procesamiento paralelo** se tienen varios procesos ejecutando acciones al mismo tiempo, en una arquitectura multiprocesador, para resolver un problema. El requerimiento fundamental para esto es que el programa tenga partes independientes. Dos partes son dependientes si una produce resultados que la otra necesita (esto sucede si leen y escriben variables compartidas). Por lo tanto dos partes son independientes si no leen y escriben las mismas variables.

3.- Cuáles son las 3 grandes clases de aplicaciones concurrentes que podemos encontrar?

- **Sistemas multithreading** (cantidad de procesos mayor a la cantidad de procesadores). Estos sistemas manejan simultáneamente tareas independientes, asignando (por ejemplo por tiempos) los procesadores de que dispone. Ejemplos típicos son los sistemas de ventanas en Pcs.
- **Sistemas de cómputo distribuido** (equipos mono/multiprocesador comunicados por red). Los procesos se comunican esencialmente por mensajes a traves de la red. Cada componente del sistema puede hacer a su vez multithreading. Ejemplo típico son sistemas de base de datos distribuidas (bancos, reservas de vuelos).
- **Sistemas de procesamiento paralelo** (arquitectura multiprocesador). Utilizando esta arquitectura se trata de distribuir la tarea global en tareas que puedan ejecutarse en diferentes procesadores. Paralelismo de datos y paralelismo de procesos. Ejemplo típico son los sistemas de calculo cientifico.

4.- Describa el concepto de deadlock y qué condiciones deben darse para que ocurra.

- Un conjunto de procesos entran en estado de **deadlock**, si por error de programación, se quedan esperando que otro libere un recurso compartido sin liberar los que posee bajo control. La ausencia de deadlock es una propiedad (de **seguridad**) necesaria en los programas concurrentes. Puede producirse entre dos procesos o entre varios de forma circular.
4 propiedades necesarias y suficientes para que exista deadlock:
- **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 expropiación:** un recurso asignado a un proceso no puede ser expropiado por otro proceso, sino que deben ser liberados voluntariamente.
- **Espera cíclica:** existe una cadena circular de procesos tal que cada uno tiene un recurso que su sucesor, en el ciclo, está esperando adquirir.

5.- Defina inanición. Ejemplifique.

- La inanición se da cuando un proceso no puede acceder a los recursos compartidos necesarios para su ejecución, o cuando el procesador nunca es alocado al proceso, pudiendo no terminar nunca. Esta es una falla en la propiedad de **vida** de los procesos.
Un ejemplo de inanición se da en el problema de los filósofos planteado por Dijkstra, de forma que si dos filósofos adyacentes intentan tomar el mismo tenedor a la vez, se produce una situación de carrera donde ambos compiten por tomar el mismo tenedor, y uno de ellos se queda sin comer.

6.- Qué entiende por no determinismo? Cómo se aplica este concepto a la ejecución concurrente?

- El **no determinismo** es la producción de resultados impredecibles, incluso para los mismos valores y condiciones iniciales.
En un programa concurrente compuesto por *n* procesos, en un momento dado habrá *n* operaciones atómicas elegible (asumiendo que todos están en estado *ready*). Así, pueden obtenerse distintas **historias** de la ejecución de un programa incluso en corridas con los mismos valores de entradas. La historia específica de una corrida/ejecución será decidida no por el programa concurrente, sino por la política de scheduling del SO.
Por este motivo es necesario utilizar mecanismos de sincronización si se desea obtener un resultado idéntico (para los mismos valores y condiciones de entrada) en cada corrida.

7.- Defina comunicación. Explique los mecanismos de comunicación que conozca.

- La comunicación entre procesos concurrentes específica el modo en que se organizan y transmiten datos entre tareas concurrentes. Esta organización requiere especificar **protocolos** para controlar el progreso y la corrección. Los procesos se comunican por **memoria compartida** y por **pasaje de mensajes**.

Comunicación por memoria compartida:

- Los procesos intercambian información sobre la memoria compartida, o actúan coordinadamente sobre datos residentes en ella.
- Lógicamente no pueden operar simultáneamente sobre la memoria compartida, lo que obliga a **bloquear** y **liberar** el acceso a ésta.
- La solución más elemental es la utilización de una variable de control tipo “semáforo” que habilite o no el acceso de un proceso a la memoria compartida (exclusión mutua).

Comunicación por pasaje de mensajes:
- Es necesario establecer un **canal** (lógico o físico) para transmitir información entre procesos.
- También el lenguaje debe proveer un protocolo adecuado.
- Para que la comunicación sea efectiva los procesos deben “saber” cuándo tienen mensajes que leer y cuándo deben transmitir mensajes.

8.- a) Defina sincronización. Explique los mecanismos de sincronización que conozca.

- La sincronización es la posesión de información (por parte de un proceso) acerca de otro proceso para coordinar actividades.

Una **acción atómica** hace una transformación de estado indivisible.
Cada proceso ejecuta un conjunto de sentencias, cada una implementada por una o más acciones atómicas.
La **historia** de un programa concurrente está dada por el intercalado de acciones atómicas ejecutadas por procesos individuales.
El objetivo de la sincronización es restringir las historias de un programa concurrente sólo a las permitidas.

Los mecanismos de sincronización son **por exclusión mutua** y **por condición**.

Sincronización por exclusión mutua:
- Asegura que sólo un proceso tenga acceso a un recurso compartido en un instante de tiempo.
- Si el programa tiene secciones críticas que puede compartir más de un proceso, esta sincronización evita que dos o más procesos puedan encontrarse en la misma sección crítica al mismo tiempo.

Sincronización por condición:
- Permite bloquear la ejecución de un proceso hasta que se cumpla una condición dada.

8.- b) En un programa concurrente pueden estar presentes más de un mecanismo de sincronización?

- Si. Por ejemplo, sea un programa con dos procesos, uno productor y otro consumidor, que comparten un sector de memoria para el pasaje de los datos. Se usaría exclusión mutua para que no accedan a la memoria compartida al mismo tiempo; y sincronización por condición para que un mensaje no sea recibido antes que enviado, y para que un mensaje no sobrescriba a uno que todavía no fue recibido.

9.- 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.

Granularidad de arquitectura: puede verse como la relación entre el número de procesadores y la cantidad de memoria, o la velocidad de procesamiento y la comunicación.

- **Grano grueso:** pocos procesadores muy potentes.
- **Grano fino:** muchos procesadores no muy potentes.

Granularidad de programa: puede verse como la relación entre la cantidad de cómputo y la comunicación necesaria.

- **Grano grueso:** mucho cómputo con relativamente poca comunicación.
- **Grano fino:** poco cómputo con mucha comunicación.

Lo ideal es hacer corresponder las granularidades de la arquitectura y de la aplicación para obtener la mejor performance posible.
Las aplicaciones con concurrencia limitada funcionan mejor en arquitecturas de grano grueso, ya que realizan mucho cómputo. Mientras que las aplicaciones altamente concurrentes funcionan mejor en una arquitectura de grano fino ya que requieren mayor comunicación y sincronización.

10.- 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 suponsicion hecha por otro proceso, ya que las acciones de los procesos de los programas concurrentes pueden ser intercalas. De esta forma las asignaciones de un proceso a variables compartidas pueden afectar el comportamiento o invalidar un supuesto de otro proceso.
La misma puede evitarse utilizando los mecanismos de sincronización (por condición y/o exclusión mutua) para restringir el número de historias (intercalado/interleaving de sentencias) a las permitidas.

11.- Defina y diferencie programa concurrente, programa distribuido y programa paralelo.

(La **concurrencia** es la capacidad de ejecutar multiples actividades en paralelo o en forma simultánea).
El **programa concurrente** especifica dos o más programas secuenciales que pueden ejecutarse concurrentemente en el tiempo (más de un hilo de ejecución), y no necesariamente implica ninguna arquitectura específica.
El **paralelismo** se asocia con la ejecución concurrente en múltiples procesadores que pueden tener memoria compartida, y generalmente con el objetivo de incrementar performance, velocidad o precisión en los resultados que los secuenciales. Ejecuta cada uno un hilo diferente en un momento dado.
El **procesamiento distribuido** es un "caso" de concurrencia con múltiples procesadores y sin memoria compartida, en el cual los procesos se comunican por pasaje de mensajes, RPC o rendezvous. Supone una arquitectura de memoria distribuida aunque esto no es limitante.

12.- Desventajas de la programación concurrente.

- En la 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.**

- Hay un no determinismo implícito en el interleaving de los procesos concurrentes. Esto significa que dos ejecuciones de un 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 producción y puesta a punto respecto de los programas secuenciales. También puede aumentar el costo de los errores → **Mayor costo en 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.
- 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.

13.- Cuál es el objetivo de la programación paralela?

- El objetivo de la programación paralela es, generalmente, reducir el tiempo de ejecución (incrementar la performance) de un problema dado o resolver problemas más grandes o con mayor presición en el mismo tiempo. Esta técnica enfatiza la verdadera simultaneidad en el tiempo de la ejecución de tareas, utilizando una arquitectura multiprocesador en la que se pueda distribuir la tarea global en tareas independientes o interdependientes que puedan ejecutarse en diferentes procesadores.

14.- a) Analice en qué tipo de problemas son más adecuados cada uno de los 5 paradigmas de resolución de problemas concurrentes descriptos en clase.

1- En el **Paralelistmo Iterativo** un programa tiene un conjunto de procesos (posiblemente idénticos) cada uno de los cuales tiene uno o más loops, es decir cada proceso es un programa iterativo. La idea es que si estos procesos cooperan para resolver un único problema (como ser un sistema de ecuaciones) pueden trabajar independientemente, y comunicarse y sincronizar por memoria compartida o envío de mensajes. Ejemplo clásico es la multiplicación de matrices.
2- En el **Paralelismo Recursivo** el problema general (programa) puede descomponerse en procesos recursivos que realizan el mismo procesamiento sobre partes del conjunto total de datos, y el resultado de este, con más de una recursión.
Ejemplos clásicos son sorting by merging o el cálculo de raices en funciones continuas.
3- El esquema **Productor-Consumidor** divide el procesamiento complejo en un conjunto independiente de tareas secuenciales, donde cada tarea genera la entrada de la siguiente y pueden ejecutarse en paralelo generando un flujo unidireccional de datos (pipe). Ejemplo son distintos niveles de SO, secuencia de filtros sobre imágenes, incluso ordenación de un vector.
4- El esquema **Cliente-Servidor** es el dominante en las aplicaciones de procesamiento distribuido. Los servidores son procesos que esperan pedidos de multiples clientes, procesan la información y devuelve los resultados, generando un flujo bidireccional de datos. Unos y otros pueden ejecutarse en procesadores diferentes. Los mecanismos de invocación son variados (Rendezvous, RPC, monitores). El soporte distribuido puede ser simple (LAN) o extendido por toda la web.

5- En el esquema **Pares que interactúan** los procesos (que forman parte de un programa distribuido) resuelven partes del problema (normalmente mediante código idéntico) e intercambian mensajes para avanzar en la tarea y completar el objetivo. El esquema permite mayor grado de asincronismo que Cliente-Servidor. Ejemplo producto de matrices distribuidas.

14.- b) 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 (por ejemplo usando un algoritmo de tipo "quicksort" o uno de tipo "sorting by merging")?

- En el paralelismo recursivo se puede dividir el problema general (programa) en procesos recursivos que trabajan sobre partes del conjunto total de datos al igual que la estrategia de divir y conquistar.
Para aplicar este concepto en la ordenación de un arreglo se lo puede dividir a la mitad, ordenar cada mitad recursivamente utilizando el "ordenamiento por mezcla", y posteriormente mezclar las dos sublistas en una sola ordenada.

14.- c) Mencione algún sistema de tipo cliente/servidor que conozca.

- Casi cualquier sistema web que utilizamos cotidiamente, como puede ser HomeBanking, redes sociales, servidores de mail, ya que existen servidores esperando pedidos por parte de los usuarios, para resolverlos y devolver el resultado, consiguiendo así el flujo bidireccional de datos.

15.- a) Analizando el código de multiplicación de matrices en paralelo planteado en la teoría, y suponiendo que N=256 y P=8 indique cuántas asignaciones, cuántas sumas y cuántos productos realiza cada proceso. Cuál sería la cantidad de cada operación en la solución secuencial realizada por un único proceso?

El codigo para multiplicacion de matrices paralelo es:

```
double a[n,n], b[n,n], c[n,n];
process worker [w = 1 to P]{
        int primera = (w-1)*(n/P) + 1;
        int ultima = primera + (n/P) – 1;
        for [i = primera to ultima]{
                for [j = 1 to n]{
                        c[i,j] = 0; //32 * 256 "="
                        for [k = 1 to n]
                                c[i,j] = c[i,j] + (a[i,k]*b[k,j]); //256^2 * 32 "=" "+" y "*"
                }
        }
}
```

Se consideran sólo las operaciones dentro de los bucles, no las de variables de control de los mismos.
Cada proceso trabaja sobre 32 (256/8) filas.
Asignaciones: 32 * 256 + 32 * 256^2 = 2105344.
Sumas: 32 * 256^2 = 2097152.
Multiplicaciones: 32 * 256^2 = 2097152.

La solución secuencial quedaría:

```
double a[n,n], b[n,n], c[n,n];
for [i = 1 to n] {
        for [j = 1 to n] {
                c[i,j] = 0.0; //256^2 "="
                for [k = 1 to n]
                        c[i,j] = c[i,j] + a[i,k]*b[k,j]; //256^3 "=" "+" y "*"
        }
}
```

Con lo que obtenemos:
Asignaciones: $256^2 + 256^3 = 16842752$.
Sumas: $256^3 = 16777216$.
Multiplicaciones: $256^3 = 16777216$.

Como conclusión puede observarse una mejora en la distribución del trabajo, mientras que la solución secuencial implica aprox. 16 millones de sumas, en la solución paralela cada uno de los 8 procesos efectúa alrededor de 2 millones de operaciones reduciendo el tiempo total. La cantidad total no se modifica, sólo se distribuye.

15.- b) Si los procesadores P1 a P7 son iguales, y sus tiempos de asignación son 1, de suma 2 y de producto 3, y si el procesador P8 es 4 veces más lento, cuánto tarda el proceso total concurrente? Cuál es el valor del speedup?

Para el **cálculo en paralelo** se considera sólo el procesador más lento (el #8):
Asignaciones: 2105344 * 1 * 4 = 8421376.
Sumas: 2097152 * 2 * 4 = 16777216.
Multiplicaciones: 2097152 * 3 * 4 = 25165824.
Total: 50364416.

Para el cálculo en secuencial se consideran los procesadores más rápidos:
Asignaciones: 16842752 * 1 = 16842752.
Sumas: 16777216 * 2 = 33554432.
Multiplicaciones: 16777216 * 3 = 50331648.
Total: 100728832.

SpeedUp = 100728832 / 50364416 = **2** → Se redujo a la mitad el tiempo agregando 7 procesadores.

Suponiendo que se asigna el octavo proceso al octavo procesador (el más lento) podrían utilizarse bandas asimétricas, dándole menor cantidad de trabajo, ya que actualmente, el resto de los procesadores tardan unas 12 millones de unidades de tiempo, permaneciendo ociosos hasta que termine el octavo.
Asignando menor cantidad de trabajo a P8:
P8 → 11 filas → 723712 * 4 + 720896 * 8 + 720896 * 12 = 17312768 ut.
P1 a P7 → 35 filas (+3 c/u) → 2302720 * 1 + 2293760 * 2 + 2293760 * 3 = 13771520 ut.

P8 → 4 filas → 263168 * 4 + 262144 * 8 + 262144 * 12 = 6295552 ut.
P1 a P7 → 36 filas (+4 c/u) → 2368512 * 1 + 2359296 * 2 + 2359296 * 3 = 14164992 ut.

De esta manera, sería conveniente asignar sólo 4 filas al procesador lento y 36 al resto, dado que así logramos un **SpeedUp** de **100728832 / 14164992** = **7,11** (sin llegar a 8 ya que

el procesador más lento permanece ocioso aprox. un 55,6% del tiempo con esta solución, pero sin retrasar al resto de los procesadores).

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

En la resolución de problemas con **memoria compartida**, se tienen N procesadores, M procesos (generalmente N<M) y una memoria que comparten dichos procesos. La comunicación entre ellos se realiza mediante variables compartidas que residen en la memoria. Los procesos no pueden acceder simultáneamente a estas variables, por lo que es necesario utilizar exclusión mutua, lo cual significa bloquear procesos cuando uno accede a dicha memoria compartida. Esto puede generar un problema de bloqueo indefinido. La exclusión mutua se puede llevar a cabo utilizando variables tipo semaforos para bloquear y liberar el acceso a las variables compartidas. También puede ser necesario utilizar sincronización por condición para evitar interferencias.
En la resolución de problemas con **memoria distribuida**, los procesos no comparten variables. No tienen que preocuparse por la exclusión mutua ya que cada procesador cuenta con su propia memoria, inaccesible para el resto, en este aspecto son más simples y faciles. En un programa distribuido, los canales son lo único que los procesos comparten y utilizan para comunicarse. La complejidad es grande a la hora de mantener actualizada la memoria global en todos los procesos.
También podemos decir que usar memoria compartida es siempre menos caro que memoria distribuida porque toma menos tiempo escribir en variables compartidas, que alocar en buffer de mensajes, llevarlo y pasarlo a otro proceso, además estos deben saber cuándo tienen mensajes para leer y cuándo deben transmitir, para lo que el lenguaje debe proveer un protocolo adecuado para el envío y recepción de dichos mensajes.

17.- En qué consisten las arquitecturas SIMD, SISD, MISD y MIMD? Para qué tipo de aplicaciones es más adecuada cada una?

- Se refiere a la manera en que las instrucciones son ejecutadas sobre los datos.
SISD (Single Instruction Single Data): cada instrucción es ejecutada en su único procesador con su única memoria de datos (monoprocesador). La ejecución es determinística.
SIMD (Single Instruction Multiple Data): la unidad de control hace broadcast de la instrucción sobre un conjunto de procesadores idénticos con sus propias memorias, sobre datos distintos. Esto las hace adecuadas para el procesamiento de imágenes.
MISD (Multiple Instruction Single Data): varios procesadores, cada uno con su propio flujo de instrucción y ejecutando los mismos datos. Adecuado para multiples altoritmos de critografía.
MIMD (Multiple Instruction Multiple Data): cada procesador tiene su propio flujo de instrucción y sus propios datos, ejecutando su propio programa en un paralelismo total. Pueden utilizar memoria compartida o distribuida. Ideal para simulaciones, comunicación y diseño.

18.- Que se entiende por arquitectura de grano grueso? Es más adecuada para programas con mucha o poca comunicación?

- Las arquitecturas de grano grueso se conforman con pocos procesadores muy potentes, y se adecúan mejor a programas con poca comunicación y mucho cómputo.

19.- Cómo puede influir la topología de conexión de los procesadores en el diseño de aplicaciones concurrentes/paralelas/distribuidas? Ejemplifique.

- Tanto en pasaje de mensajes como en memoria compartidas, las máquinas pueden armarse conectando procesadores y memorias utilizando diversas redes de interconexión:

Las **redes estáticas** constan de links punto a punto. Típicamente se utilizan para máquinas de pasaje de mensajes.

Las **redes dinámicas** están construidas usando switches y enlaces de comunicación. Normalmente para máquinas de memoria compartida.

El diseño de la red de interconexión depende de una serie de factores (ancho de banda, tiempo de starup, paths estáticos o dinámicos, operación sincrónica o asincrónica, topología, costo, etc.).

20.- Diferencie acciones atómicas condicionales e incondicionales.

- Una **acción atómica** realiza una transformación de estado indivisible, o sea que mientras está siendo ejecutada ningún estado intermedio debe ser visible para los otros procesos.

Una **acción atómica de grano fino** es implementada directamente sobre el hardware que ejecuta el programa concurrente.

En los programas secuenciales, las asignaciones aparecen como atómicas ya que ningún estado intermedio es visible al programa. Esto no sucede en los programas concurrentes, ya que una asignación con frecuencia es implementada por varias instruciones de máquina de grano fino.

Una **acción atómica incondicional** no contiene una condición de demora B, por lo que puede ejecutarse inmediatamente.

Una **acción atómica condicional** es una sentencia AWAIT con una guarda B. Estas son condiciones de demora que hasta no ser true no puede ejecutarse la acción. Si B es false, sólo puede volverse true como resultado de acciones tomadas por otros procesos.

22.- 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.

Una sentencia **x = e** satisface la propiedad de "A lo sumo una vez" si:

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

Una expresión **e** satisface ASV si no contiene más de una referencia crítica.

Las sentencias de un programa concurrente pueden referenciar una variable compartida a lo sumo una vez para evitar interferencias. De esta manera su ejecución parecerá atómica a los otros procesos.

Ejemplos:

```
int x=0, j=0;
co
        x = 1 + j; // j = 1;
oc;
```

-Cumple ASV ya que x tiene una sola referencia critica, j no tiene ninguna, y el resultado final de j es siempre 1, el de x puede ser 1 o 2.

```
int x=0, j=0;
co
        x = 1 + j; // j = 1 + x;
oc;
```
- No cumple ASV ya que ambos procesos poseen referencias críticas, y los resultados finales pueden ser x = 1 j= 2, o x = 2 j = 1.

23.- Dado el siguiente programa concurrente:
```
x = 2; y = 4; z = 3;
co
x = y - z // z = x * 2 // y = y - 1
oc
```
a) Cuáles de las asignaciones dentro de la sentencia co cumplen con ASV?. Justifique claramente.
b) Indique los resultados posibles de la ejecución
Nota 1: las instrucciones NO SON atómicas.
Nota 2: no es necesario que liste TODOS los resultados pero si los que sean representativos de las diferentes situaciones que pueden darse.

- Cumple ASV la asignacion y= y-1, ya que no posee referencia crítica y puede ser leído por otro proceso. Su resultado final siempre será 3.
- No cumple ASV la asignación z= x*2, ya que posee una única referencia crítica, pero es leído por otro proceso. Su resultado puede ser z = 4, 2, 0.
- No cumple ASV la asignación x = y-z, ya que posee 2 referencias críticas y además es leída en otro proceso. Sus posibles resultados son x= 1, 0, -1.

24.- Qué propiedades que deben garantizarse en la administración de una sección crítica en procesos concurrentes?
Cuáles de ellas son propiedades de seguridad y cuáles de vida?
En el caso de las propiedades de seguridad, cuál es en cada caso el estado "malo" que se debe evitar?

Las propiedades que deben garantizarse en la administración de la sección crítica son:
- **Exclusión Mutua:** propiedad de seguridad. Establece que sólo un proceso debe estar en su sección crítica en un momento dado. El caso malo es que más de un proceso se encuentren en la misma sección crítica.
- **Ausencia de Deadlock:** propiedad de seguridad. Establece que si dos o más procesos quieren entrar en su sección crítica, al menos uno tendrá éxito. El caso malo es que ninguno de los procesos que intentan acceder a su sección crítica lo haga, quedando todos bloqueados.
- **Ausencia de demora innecesaria:** propiedad de seguridad. Establece que si un proceso trata de entrar a su sección crítica, y el resto está en su sección no crítica o terminaron, aquel no está impedido de entrar. El caso malo es que un proceso intenta acceder a su sección crítica y es bloqueado aún siendo el único que intenta acceder.
- **Eventual entrada:** propiedad de vida. Establece que un proceso intenta acceder a su sección crítica tiene posibilidades de hacer, es decir eventualmente lo hará.

25.- A que 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 cualquiera de sus posibles historias (intercalado de acciones atómicas), por lo tanto en todas sus ejecuciones. Una **propiedad de seguridad** establece que el programa nunca entra en estado malo (en el cual alguna variable toma valores indeseables). **Ejemplos** de esta propiedad son la ausencia de deadlock, o sea que un proceso no se bloquee esperando una condición que nunca ocurrirá, y la exclusión mutua, o sea que no haya más de un proceso en la misma sección crítica al mismo tiempo.
Una **propiedad de vida** establece que algo bueno eventualmente ocurrirá en la ejecución del programa. **Ejemplos** de estas propiedades son eventual entrada a la sección crítica, terminación del programa o que un mensaje llegue a destino. Estas propiedades depende de la política de scheduling.

26.- Defina fairness. Cuál es la relación con las políticas de scheduling?

- El **fairness** es una caracteristica de las politicas de scheduling, que trata de garantizar que los procesos tengan chance de avanzar sin importar lo que hagan los demás. Existen distintos tipos de políticas de scheduling: incondicionalmente fair, debilmente fair, fuertemente fair.
Incondicionalmente fair: si toda acción atómica incondicional que es elegible eventualmente es ejecutada.
Debilmente fair: si es incondicionalmente fair, y toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada, asumiendo que su condición se vuelve true y permanece así hasta que es vista por el proceso que ejecuta la acción atómica condicional.
Fuertemente fair: si es incondicionalmente fair, y toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada pues su guarda se convierte en true con infinita frecuencia.

Por qué las propiedades de vida dependen de la política de scheduling?

- El hecho de que un proceso logre un estado bueno (por ejemplo la eventual terminación del programa) depende de que la política de scheduling le permita ejecutarse cuando las condiciones sean las adecuadas. Por ejemplo, una política incondicionalmente fair no asegurara la terminación de un programa que tiene acciones atómicas condicionales para todas sus historias posibles, pudiendo éste sufrir de livelock que es la analogía de deadlock.

Cómo aplicaría el concepto de fairness al acceso a una base de datos compartida por n procesos concurrentes?

-Se puede aplicar el concepto de fairness a la política de scheduling que se utilice para acceder a dicha BD.
Si el acceso de un lector permite el acceso de otros, aunque haya escritores esperando → **debilmente fair,** un ingreso permanente de lectores podría causar inanición en los escritores.
Si la llegada de un escritor impide el ingreso de otros y es atendida en primer lugar al liberarse la BD → **debilmente fair,** el ingreso constante de escritores puede causar inanición de lectores.
Si los procesos son atendidos en orden FIFO con acceso concurrente de lectores → **fair,** ya que todos serán atendidos.
Esta última opción podría producir una demora innecesaria en los lectores que lleguen tras un escritor que espera, pero para evitar esto se caería en un esquema de prioridad con riesgo de inanición de escritores.

27.- Dado el siguiente programa concurrente, indique cuál es la respuesta correcta.
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

- La respuesta correcta es la c, ya que existen ambos casos. Podría terminar cuando se ejecuta la primer asignación del while, el procesador se asigna a la ejecución de la sentencia await, la cual termina correctamente dejando la variable a = 0, permitiendo que el while corte el bucle. La otra posibilidad es que se ejecute completo el while siempre, impidiendo la ejecución de la sentencia await, haciendo que el programa no termine nunca ya que nunca sucedería que a=0.

28.- a. Qué significa que un problema sea de "exclusión mutua selectiva"?

- Un problema es de exclusión mútua selectiva cuando los procesos deben competir por los recursos no contra todos sino contra otro subconjunto de ellos. Típicamente se da el caso de que los procesos compiten por los recursos según su clase o tipo.
Por ejemplo, sean dos clases de procesos **A** y **B**, primero llega un proceso de clase **A**, gana el acceso al recurso, luego llega un proceso de clase **B**, pero es excluído y debe esperar que liberen el recurso, posteriormente llega otro proceso de clase **A** el cual no es impedido de entrar ya que el acceso fue ganado por otro de su misma clase; así seguirían bloqueados todos los procesos de clase **B** que puedan llegar, hasta que todos los procesos de clase **A** hayan liberado el recurso en cuestión.

28.- b. El problema de los lectores y escritores es de exclusión mutua selectiva? Por qué?

- En este problema se presentan ambos casos de exclusión mutua para el recurso compartido por todos, que es la base de datos.
Los escritores compiten contra todos los procesos independientemente de su clase (exclusión mutua general), ya que puede acceder uno a la vez a la bd. Pero los lectores compiten como una clase contra los escritores (exclusión mutua selectiva entre clases de procesos) dado que no hay un número límite de lectores que puede acceder simultáneamente, pero no se permiten escritores mientras haya al menos un lector.

29.- c. Si en el problema de los lectores y escritores se acepta solo 1 escritor y 1 lector en la BD, tenemos un problema de exclusión mutua selectiva? Por qué?

-No, en ese caso la exclusión mutua es general, ya que el recurso puede ser utilizado por un único proceso por vez, haciendo que todos los procesos compitan entre sí sin importar la clase de los mismos.

29.- d. De los problemas de los baños planteados en la teoría, Cuáles podría ser de exclusión mutua selectiva?

- De los problemas mencionados podrían ser de exclusión mútua selectiva el de un único baño para varones y mujeres (excluyente) sin límite de espacio; y también sería de exclusión mutua selectiva el de un único baño para varones y mujeres (excluyente) con

límite de espacio, donde al completarse el baño, dejaría de ser selectiva y pasaría a ser exclusión mutua general dado que ningún otro proceso, sin importar su clase, podría hacer uso del recurso hasta ser liberado por algun otro.

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

- Una instrucción de grano grueso es aquella que aparece como una secuencia indivisible de acciones atómicas de grano fino. La sentencia await es muy poderosa para especificar acciones atomicas de grano grueso arbitrariamente. Es muy conveniente para expresar sincronización pero es muy costosa de implementar en su forma generica.
- Tiene la sintaxis: < AWAIT (B) → S >.
- B es una expresión booleana que especifica una condición de demora, y S es una secuencia de acciones que se asegura que terminan. Se garantiza que B es true al momento de ejecutar S, y ningún estado intermedio de S es visible para otros procesos.
- Su relación con las instrucciones T&S y F&A, presentes en casi todos los procesadores, es que son utilizadas para hacer atómico el AWAIT de grano grueso implementando los protocolos de E/S de la sección crítica.

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

- La sincronización por busy waiting presenta varias problemáticas:
1) Es **ineficiente** empleada en mutiprogramación, ya que el “spinning” mantiene al procesador ocupado, mientras que este tiempo podría ser usado por otro proceso productivamente.
2) Son **complejos** y **sin clara separación** entre variables de sincronización y las usadas para computar resultados.
3) Es **dificil** diseñar para **probar corrección**. Y la verificación es compleja cuando se incrementa el número de procesos.

- Los semáforos son una herramienta para implementar sincronización que evita el busy waiting. Al bloquearse en un semáforo un proceso no consume tiempo de procesamiento, hasta que tenga la posibilidad de ejecutarse, en cuyo caso pasaría a competir por el uso del procesador.

33.- Qué mejoras introducen los algoritmos Tie-breaker, Ticket o Bakery en relación a las soluciones de tipo spin- locks?

- El problema de los algoritmos **spin-locks** es que no mantienen un orden de los procesos demorados, pudiendo alguno de ellos no entrar nunca **si** el **scheduling no** es **fuertemente fair**.
- En el algotimo **Tie-Breaker** se decrementa prioridad al último en ingresar al protocolo de entrada. Requiere scheduling debilmente fair y no usa instrucciones especiales (es complejo y costoso en tiempo). Una variable por cada proceso para indicar que el proceso comenzó a ejecutar su protocolo de entrada a la SC, y una variable adicional para romper empates, indicando qué proceso fue el último en comenzar dicha entrada.
- En el algoritmo **Ticket** se reparten números y se espera que sea el turno (potencial problema → valores de turno ilimitados). Los procesos toman un número mayor que el de cualquier otro que espera ser atendido; luego espera que todos los que tienen números más chicos hayan sido atentidos.

- En el algoritmo **Bakery** cada proceso que intenta entrar recorre los números de los demás y se auto asigna uno mayor. Luego espera que sea el menor de los que esperan (esta solución no es implementable directamente). Los procesos se chequean entre ellos y no contra un global. Este algoritmo es más complejo, pero es fair y no requiere instrucciones especiales.

34.- En qué consiste la sincronización barrier? Mencione alguna de las soluciones posibles usando variables compartidas.

- Consiste en un punto de encuentro, al cual deben llegar todos los procesos para recién en ese momento poder continuar cada uno con su ejecución. Típicamente se utiliza para proveer sincronización de los procesos al final de cada iteración.
Ejemplo:

```
Int cantidad=0;
Process Worker[i=1 to n]{
        While(true){
                #Código para implementar la tarea i;
                <cantidad=cantidad+1> #Puede implementarse con un FA
                <await(cantidad==n);> #Puede implementarse como while(cantidad!=n) skip;
        }
}
```

Otra solución consiste en distribuir la variable cantidad usando n variables (arreglo llegada(1..n)). El await pasaría a ser < await(llegada(1)+..+llegada(n) == n); >

35.- a) Explique la semántica de un semáforo.

- Un semáforo es una herramienta utilizada para implementar exclusión mutua y sincronización por condicición. Es una instancia de un tipo abstracto de datos con un valor entero y sólo dos operaciones para manejarlo.

P() → en caso de que el valor entero sea positivo se decrementa; en caso contrario se bloquea hasta que dicho valor sea positivo, luego se decrementa y continúa su ejecución. Equivale a: <await (valor>0) valor = valor – 1>

V() → incrementa el valor del semáforo, lo cual “despertaría” a alguno que esté bloqueado en un P(). Equivale a <valor = valor + 1>

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

```
int x = 4; sem s1 = 1, s2 = 0;
co
        P(s1); x = x * x ; V(s1);       // P(s2); P(s1); x = x * 3; V(s1);
// P(s1); x = x - 2; V(s2); V(s1);
oc
```

- Los posibles valores de x son 42, 12, 36. Al utilizar semáforos (correctamente) no se presenta problemas de consistencia en los valores.
El resultado final 42 se da de la ejecución de los procesos de la siguiente manera: 1er co, 3er co, 2^{o} co; el resultado 12 se da por la siguiente ejecución: 3er co, 1er co, 2^{o} co; y el número 36 se obtiene de la ejecución de 3er co, 2^{o} co, 1er co.

36.- a) Describa la técnica de Passing the Baton.

- Passing the Baton es una técnica general para implementar sentencias AWAIT.
Cuando un proceso está dentro de la SC, éste mantiene el baton (o token) que equivale al permiso para ejecutar.
Cuando el proceso llega a un SIGNAL, pasa el baton (control) a otro proceso. Si no hay procesos esperando el baton (o sea entrar a la SC), entonces se libera para que lo tome el próximo que trata de entrar.

36.- b) Cuál es su utilidad en la resolución de problemas mediante semáforos?

- Presenta una forma sencilla y sistemática de implementar cualquier tipo de sincronización entre proceso, con o sin condiciones, utilizando semáforos para una solución de grano fino. Los programas resultantes son faciles de modificar y extender.

36.- c) En qué consiste la técnica de Passing the Condition y cuál es su utilidad en la resolución de problemas con monitores?

- Passing the condition trata de señalar (signal) una variable condición que tenga procesos esperando para ejecutar pero sin volver verdadera la condición asociada; o cambiando a verdadera la condición asociada, pero sin señalar (signal) la variable condición, si no hay procesos esperando para ejecutar.
De esta manera se asegura que el siguiente proceso en ejecutar su SC es el despertado de la lista, y no otro que gane el acceso al monitor, ya que la condición será falsa y pasará a la cola de espera.

36.- d) Qué relación encuentra entre passing the condition y passing the baton?

- Ambas técnicas pasan el control a un proceso demorado que será despertado para continuar su ejecución de la SC. La diferencia está en que passing the baton pasa el control a otro proceso para que sea el único que pueda ejecutar SC; mientras que passing the condition le avisa a un proceso que están dadas las condiciones para que pueda seguir ejecutando y este sea el único que pueda continuar de todos los que esperan dicha condición.

37.- Mencione qué inconvenientes presentan los semáforos como herramienta de sincronización para la resolución de problemas concurrentes.

Presentan los siguientes inconvenientes:
- Variables compartidas globales a los procesos.
- Sentencias de control de acceso a la seccion crítica dispersas en el código.
- Al agregar procesos, se debe verificar el correcto acceso a las variables compartidas.
- Aunque sincronización por condición y exclusión mutua son conceptos diferentes, se programan de forma similar.

38.- Defina el problema general de alocación de recursos y su resolución mediante una política SJN. Minimiza el tempo promedio de espera? Es fair? Si no lo es, plantee una alternativa que lo sea.

- El problema general de alocación de recursos se basa en decidir cuándo permitir el acceso de un proceso a un recurso, habiendo varios procesos que quieren acceder al mismo.
Un proceso ejecuta un request con la cantidad de unidades del recurso que necesita y su identificación. De haber disponibles las unidades especificadas se asignan al proceso, en

caso contario se lo demora hasta que haya suficientes unidades disponibles. Luego de utilizar el recurso se lo libera ejecutando un release.

La resolución mediante una política SJN se basa en la existencia de un único recurso compartido por varios procesos, los cuales deben especificar, en el request, su id y la cantidad de tiempo que van a utilizar dicho recurso. Entonces, al ejecutar un request, si el recurso se encuentra disponible se lo asigna al proceso que lo pide, en caso contrario se demora al proceso y se almacena su id y el tiempo que requiere el recurso. Una vez liberado dicho recurso, el mismo es asignado al proceso demorado (si lo hay) con menor cantidad de tiempo especificado. En caso que dos procesos tengan la misma cantidad de tiempo, el recurso es asignado al que tiene mayor tiempo de demora.

La política SJN mejora el tiempo promedio de ejecución, pero es unfair dado que podría ocurrir que un proceso quede demorado para siempre si existe una corriente de peticiones de que especifican menor tiempo que el proceso mencionado. Esta problemática se puede resolver mediante la técnica **aging**, que asigna preferencia a aquel proceso que haya estado demorado demasiado tiempo, de esta manera la política sería fair.

39.- Explique brevemente los 7 paradigmas de interacción entre procesos en programación distribuida vistos en teoría. En cada caso ejemplifique, 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. **Permite incrementar la accesibilidad** a los datos o servicios, donde cada servidor interactúa con los demás para darle al cliente la sensación de que es un único servidor. Ejemplo: la resolución descentralizada de los filósofos, donde cada proceso mozo interactúa con otro para conseguir los tenedores de forma transparente a los filósofos.
- **Algoritmos Heartbeat:** Los procesos periodicamente deben intercambiar información con mecanismos tipo send/receive. Se realiza en dos etapas, en la primera se expande, enviando información (SEND a todos) y en la segunda se contrae, adquiriendo información (RECEIVE de todos). Su uso más importante es paralelizar soluciones iterativas. Ejemplo: modelo biológico, problema de los N cuerpos.
- **Algoritmos Pipeline:** La información recorre una serie de procesos utilizando alguna forma de send/receive, donde **la salida de un proceso es la entrada de otro**. Ejemplo: redes de filtro.
- **Probes (send) y Echoes (receive):** La interacción entre procesos permite recorrer arboles y grafos (o estructuras dinámicas) juntando y diseminando información. Ejemplo: conocer la topología de una red cuando no se conoce de antemano el número de nodos activos.
- **Algoritmos Broadcast:** Permiten alcanzar una información global en una arquitectura distribuida. Sirven para toma de decisiones descentralizadas. Ejemplo: sincronización de relojes en un Sistema Distribuido de Tiempo Real.
- **Token Passing:** En muchos casos la arquitectura distribuida recibe una información global a través del viaje de tokens de control o datos. También permite la toma de decisiones distribuidas. Ejemplo: determinar la terminación de un proceso en una arquitectura distribuida que no puede decidirse localmente.
- **Manager/Workers:** Implementación distribuida del modelo bag of tasks, que consiste en un proceso, controlador de datos y/o procesos, y múltiples workers que acceden a él para obtener datos y/o tareas a ejecutar en forma distribuida. Ejemplo: algoritmos de paralelismo recursivo en entornos distribuidos.

40.- Marque al menos 2 similitudes y 2 diferencias entre los pasajes de mensajes sincrónicos y asincrónicos.

- Similitudes:
    - En ambos casos la sentencia receive es bloqueante, o sea el proceso no hace nada hasta recibir el mensaje. En PMS permanece bloqueado hasta sincronizar con el otro proceso; en PMA permanece bloqueado hasta que la cola sea no vacía.
    - En ambos casos el acceso al contenido del canal es atómico.
    - En ambos casos se cuentan con variables locales al proceso, lo único que utilizan para compartir datos son los canales.

- Diferencias:
    - En PMA el send es no bloqueante, mientras que en PMS el send sí bloquea el proceso hasta que otro esté esperando recibir por ese canal.
    - En PMS la cola de mensaje se reduce a un sólo mensaje ya que los procesos emisor y receptor se sincronizan en todo punto de comunicación, mientras que en PMA la cola de mensajes es ilimitada (conceptualmente) por no requerir la sincronización mencionada.
    - En PMS el grado de concurrencia se reduce respecto de la sincronización por PMA (dado que siempre un proceso se bloquea).
    - En PMS hay mayor probabilidad de deadlock, por lo que el programador debe ser cuidadoso de que las sentencias de send y receive hagan matching.
    - En PMS los canales de mensajes son punto a punto (1 emisor 1 receptor) mientras que en PMA pueden existir muchos emisores y/o muchos receptores sobre el mismo canal.

41.- Defina el concepto de “continuidad conversacional” entre procesos.

- Dado un escenario donde procesos “cliente” acceden a archivos externos almacenados en disco. Interactuan con 1 File Server por archivo. Estos procesos servidores son idénticos, y cualquiera que esté libre puede responder la petición abrir.
Esta interacción entre cliente y servidores se denomina continuidad conversacional (del abrir al cerrar, es decir, desde la petición del archivo hasta que éste es entregado al cliente). Siendo abrir un canal compartido por el que cualquier FS puede recibir, pero cada canal tiene un único receptor, necesitamos un **alocador de archivos** separado **para asegurar** la **continuidad conversacional**, que reciba los pedidos de abrir y asigne un FS libre a un cliente; y al que los FS le avisen que están libres.
Si el lenguaje soporta creación dinámica de procesos y canales, el número n de FS puede adaptarse a los requerimientos del sistema.

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

- Existe una dualidad entre monitores y pasaje de mensajes ya que cada uno puede simular al otro.

Monitores	**Pasaje de mensajes**

Variables permanentes.	Variables locales del servidor.
Identificadores de procedures.	Canal request y tipo de operación
Llamado a procedure	Send request(); receive respuesta();
Entry del monitor	Receive request();
Retorno del procedure	Send respuesta();
Sentencia wait	Salvar pedido pendiente.
Sentencia signal	Recuperar / procesar pedido pendiente.
Cuerpos de los procedure	Sentencia del “case” de acuerdo a la clase de operación.

Por ejemplo podríamos simular monitores utilizando procesos servidores y PM, como procesos activos en lugar de como conjuntos pasivos de procedures.

43.- En qué consiste la comunicación guardada (introducida por CSP) y cuál es su utilidad? Describa cómo es la ejecución de sentencias de alternativa e iteración que contienen comunicaciones guardadas.

- Las ideas básica de Hoare (creador de CSP) fueron PMS y la comunicación guardada, o sea el **pasaje de mensajes con waiting selectivo**. Consiste en la idea de **tener canales simples como links directos entre dos procesos** (en vez de mailbox global) **y sentencias de entrada/salida bloqueantes** (?, !) **como único medio por el cual se**

comunican los procesos. Es útil para problemas como el de copiar con un buffer limitado, el de asignación de recursos y el de intercambio de valores.
Las sentencias que contiene comunicación guardada son las siguientes:
- If, de alternativa:

```
if B1; comunicacion1 → S1;
 B2; comunicación2 → S2;
fi
```

- Do, de iteración:

```
do B1; comunicacion1 → S1;
 B2; comunicacion2 → S2;
od
```

Funcionan de la siguiente manera:
1) Se evalúan las expresiones booleanas:
I. Si todas las guardas fallan, el if termina sin efecto.
II. Si al menos una guarda tiene éxito, se elige una de ellas no determinísticamente.
III. Si alguna/s guarda/s se bloquea/n, se espera hasta que alguna de ellas tenga éxito.

2) Luego de elegir una guarda exitosa, se ejecuta la sentencia de comunicación en la guarda elegida.

3) Se ejecuta la sentencia S1..n según corresponda.

La ejecución del do es similar, sólo que se repite hasta que todas las guardas fallen.

44.- Describa el paradigma "bag of tasks". Cuáles son las principales ventajas del mismo?

- Este paradigma indica un **modo de organizar un programa paralelo**, donde se c**uenta con procesos idénticos** llamados workers **que comparten una bolsa de tareas independientes** a realizar. **Cada worker** se mantiene en un ciclo en el que **toma una tarea**, **la resuelve y de ser necesario crea nuevas** tareas. El problema general se resuelve cuando la bolsa está vacía y todos los workers están a la espera de una tarea.
El número de tareas puede fijarse inicialmente (propio del paralelismo iterativo) o puede ser sólo una inicial y crearse dinámicamente (propio del paralelismo recursivo).
Este **esquema es sencillo**; **favorece** el **balance de carga** entre procesos, si un proceso demora en realizar una tarea probablemente termine resolviendo menos cantidad de tareas sin necesariamente retrasar al resto; **escalable** permitiendo que **el programa pueda aprovechar mayor cantidad de procesadores** sin mayores cambios en el código, sólo **creando nuevos workers**.

45.- Describa sintéticamente las características de sincronización y comunicación de MPI

- El uso de bibliotecas de funciones es una técnica muy utilizada para comunicar/sincronizar procesos no dependiente de un lenguaje. Aunque pueden ser menos eficientes comparadas a lenguajes "reales" concurrentes, son "agregables" al código secuencial con bajo costo de desarrollo.
MPI (Interfaz de Paso de Mensajes) es una biblioteca de comunicaciones, que usa un estilo SPMD (simple program multiple data), donde cada proceso ejecuta una copia del mismo

programa y toma decisiones basado en su “identidad”. Las instancias interactúan llamando a funciones MPI, que soportan comunicación proceso a proceso y grupales.
Esta biblioteca debe ser incializada y finalizada explícitamente, provee primitivas de envío sincrónico y asincrónico, recepción de mensajes sincrónico y de tipo polling o no bloqueante, desde un proceso particular o desde cualquiera dentro de un grupo. Los grupos de procesos se pueden crear y manejar dinámicamente, pudiendo utilizar primitivas de sincronización en barrera, comunicación broadcast, distribuir elementos de un array entre procesos, o esperar mensajes desde los procesos del grupo y reunirlos en un array, realizar operaciones entre los valores recibidos de otros procesos, etc.

46.- Describa brevemente en que consisten los mecanismos de RPC y Rendezvous. Para qué tipo de problemas son más adecuados?

- RPC (Remote Procedure Call) sólo provee un mecanismo de comunicación, la sincronización debe ser implementada por el programador utilizando algún método adicional. En cambio, Rendezvous es un mecanismo tanto de comunicación como de sincronización.
En RPC la comunicación se efectúa mediante la ejecución de un CALL a un procedimiento, el cual crea un proceso, y demora al llamador hasta que dicho proceso realice la tarea, devuelva los resultados y termine. De esta manera con RPC se puede atender varios pedidos al mismo tiempo.
En Rendezvous el CALL es atendido por un proceso ya existente, que está continuamente activo. Así con Rendezvous los pedidos se atienden de a uno por vez (sería necesario un pool de procesos aceptando el mismo CALL para poder atender a más de uno simultaneamente).
Estos mecanismos son más adecuados para la interacción cliente/servidor, donde la comunicación es bidireccional y sincrónica. La ejecución de un CALL demora al llamador (cliente) y este no debe realizar ninguna otra tarea hasta tener los resultados.

47.- Por qué es necesario proveer sincronización dentro de los módulos de RPC? Cómo puede realizarse esta sincronización?

- Para el caso de los módulos, es necesario proveer sincronización porque los procesos pueden ejecutar concurrentemente. En cambio si fuese un proceso server para un proceso llamador la sincronización sería implícita.
La sincronización puede usarse tanto para acceder a variables compartidas como para sincronizar interacciones entre los procesos.
En RPC existen dos modos de proveer sincronización y depende del modo en que se ejecutan los procesos dentro del módulo:
- Si los procesos se **ejecutan con exclusión mutua**, sólo hay un único proceso activo a la vez dentro del módulo, el acceso a las variables compartidas tienen exclusión mutua implícita, pero la sincronización por condición debe programarse. Pueden usarse sentencia wait y variables condición.
- Si los procesos **ejecutan concurrentemente** dentro del módulo, es necesario implementarse tanto la exclusión mutua como la sincronización por condición. Para esto pueden utilizarse cualquiera de los mecanismos existentes, ya sea semáforos, monitores, PM o rendezvous.

48.- Cuál es a su criterio la mayor utilidad de la notación de primitivas múltiples?

- Es una notación que combina RPC, rendezvous y PMA en un paquete coherente. Provee gran poder expresivo combinando ventajas de los 3 componentes, y poder adicional.? Como con RPC los programas son estructurados como 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 el proceso o procedure del módulo que la declara. También se usan operaciones locales, que son declaradas, invocadas y servidas sólo por el módulo que la declara.

Una operación puede ser invocada por **call** sincrónico o por **send** asincrónico y las sentencias de invocación son:
call Mname.op(argumentos)
send Mname.op(argumentos)

Como con RPC y rendezvous, el call termina cuando la operación fue servida y los resultados fueron retornados. Como con PMA 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). Es elección del programador.
Esto depende si el programador desea que cada petición sea servida por un proceso diferente, o si es más apropiado rendezvous con un proceso existente.

Finalmente podemos decir que hay dos maneras de invocar una operación (call o send) y dos maneras de servir una invocación (proc o in).

50.- Cuál es el significado de las métricas de speedup y eficiencia? Cuáles son los rangos de valores en cada caso?

- **Speedup**: es una métrica entre la ejecución secuencial y la ejecución paralela de un programa. Mide la ganancia en términos de velocidad de la paralela respecto a la secuencial. El rango general de valores se encuentra entre 0 y p, siendo p la cantidad de procesadores, aunque hay casos en los cuales puede superar a p, siendo esto una ganancia sueperlineal.
La fórmula se define de la siguiente manera: S = Ts / Tp , donde:
    - p es el número de procesadores.
    - Ts es el tiempo de ejecución del algoritmo secuencial.
    - Tp es el tiempo de ejecución del algoritmo paralelo con p procesadores.

Por ejemplo si obtenemos un speedup S=p estamos frente a un speedup lineal, lo cual señala que duplicando el número de procesadores duplicaríamos la velocidad del programa.
- **Eficiencia:** esta métrica mide la fracción de tiempo que los procesadores son útiles para el cómputo. Indica que tan bien se aprovechan los procesadores en la solución paralela, relacionando el speedup con la cantidad de procesadores empleados para conseguir dicho speedup. Su rango varía entre 0 y 1.
Por ejemplo: algoritmos con speedup lineal tienen eficiencia 1. Hay casos de algoritmos dificiles de paralelizar cuya eficiencia es cercana a 1/log p, que se aproxima a cero a medida que se incremental la cantidad de procesadores.

51.- En qué consiste la “ley de Amdahl”?

- Esta ley enuncia que para cualquier tipo de problema existe un speedup máximo alcanzable y este no varía por la cantidad de procesadores que se empleen para su resolución.
En general, un algoritmo cualquiera realiza una entrada, procesamiento y salida. Siendo que la entrada (inicializaciones) y la salida generalmente mantienen componentes secuenciales, sólo el procesamiento puede ser paralelizable, y por consiguiente su mejora total queda limitada por sus componentes no paralelizables.
En conclusión, la ley de Amdahl indica que la mejora obtenida al alterar una porción de un programa está limitada por la cantidad de tiempo que se utiliza dicha porción. De esta manera es el algoritmo y no la cantidad de procesadores el factor que limita el speedup. O sea que agregando procesadores puede mejorarse el speedup hasta cierto punto límite.

52.- Mencione 3 técnicas fundamentales de la computación científica. Ejemplifique.

- Entre las diferentes aplicaciones de cómputo científico y modelos computacionales, existen las siguientes 3 técnicas fundamentales:
Computación de grillas: dividen una región espacial en un conjunto de puntos. Soluciones numéricas a PDE e imágenes.
Computación de particulas: modelos que simulan interacciones de partículas individuales como moléculas u objetos estelares. Ejemplo: partículas cargadas que interactúan debido a las fuerzas electromagnéticas, o moléculas que interactúan debido a la unión química.
Computación de matrices: sistemas de ecuaciones simultáneas. Ejemplo: aplicaciones científicas y de ingeniería, así como también modelos económicos

53.- 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, el mismo mensaje a varios procesos.
En un ambiente con variables compartidas es sencillo dado que bastaría con escribir dicha variable y que todos los procesos receptores la lean.
En un ambiente distribuido no se cuentan con variables compartidas por lo que es necesario emplear si o si pasaje de mensajes.
De esta manera se cuenta con dos mecanismo, PMA y PMS.
Con PMA existen dos posibilidades:
1) Utilizar un único canal donde todos los procesos receptores lean, por lo que el emisor debe enviar tantos mensajes como procesos están esperando recibirlo. Debe contemplarse que cada proceso tome sólo un mensaje.
2) Utilizar un canal individual por cada proceso receptor.

La opcion primera es más eficiente dado que cuando un receptor está listo simplemente toma un mensaje del canal y continua ejecutando. En cambio la segunda opción es más ineficiente ya que cuando un proceso está listo para recibir debe bloquearse hasta que el emisor deposite un mensaje en su canal privado.

Con PMS no se puede utilizar un canal compartido porque todos son punto a punto. Dada su naturaleza bloqueante de las sentencias de salida el emisor es bloqueado hasta que los procesos receptores reciben el mensaje.
En este caso existen dos posibilidades para obtener el efecto broadcast:
1) Utilizar un proceso separado que haga de buffer con el cual los clientes se comunican para depositar un mensaje broadcast y del cual lo reciben. O bien podría pensarse en

utilizar un proceso buffer por cada proceso receptor, de manera que el emisor se comunique con dicho buffer por si el proceso en cuestión está haciendo otra cosa.
2) Utilizar comunicación guardada con dos guardas. Cada guarda tiene un cuantificador para enumerar a los procesos partners. De forma que una guarda saca el mensaje hacia los otros partners; la otra guarda ingresa el mensaje desde los otros partners.

54.- Mencione al menos 4 problemas en los cuales Ud. Entiende que es conveniente el uso de técnicas de programación paralela.

- Procesamiento de imágenes.
- Mutiplicación de matrices.
- Simulación de circulación oceánica.
- Generación de números primos.
- Ataques por fuerza bruta.
- Uso de técnicas criptográficas.

----------RAMIRO----------

(13-11-2014)

1(a) Que significa el problema de "interferencia" en programacion concurrente? Como puede evitarse?

La interferencia se da cuando un proceso toma una acción que *invalida alguna suposición hecha por otro proceso* y esto se debe a que las acciones de los procesos en un programa concurrente pueden ser intercaladas. La interferencia se da por la realización de asignaciones en un proceso a variables compartidas que pueden afectar el comportamiento o invalidar un supuesto realizado por otro proceso. Por lo tanto, para evitar la interferencia entre procesos se usa la sincronización cuyo rol es restringir el número de historias (interleaving de sentencias) posibles sólo a las deseables y para hacerlo existen dos mecanismos de sincronización: exclusión mutua o por condición.

1(b) En que consiste la propiedad de "A lo sumo una vez" y que efecto tiene sobre las sentencias de un programa concurrente? De ejemplos de sentencias que cumplan y de sentencias que no cumplan ASV.

Una sentencia de asignación $x = e$ satisface la propiedad de "*a lo sumo una vez*" sí:
• e contiene a lo sumo una referencia crítica y x no es referenciada por otro proceso, o
• e no contiene referencias críticas, en cuyo caso x puede ser leída por otro proceso

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

x = y – z NO CUMPLE. Dos referencias críticas, y *x* es leída por otros procesos.
z = x * 3 NO CUMPLE. Una referencia crítica, pero *z* es leída por otros procesos.
y = y + 3 CUMPLE CON ASV. No posee referencias críticas al lado derecho; *y* puede ser leída por otros procesos

1(c) Dado el siguiente fragmento de programa concurrente con variables compartidas:
int s=6, t=4;
co <s = s + t> // <t=s * t> oc

-Cuales son los posibles valores finales de s y t ?
-Si se eliminan los corchetes angulares y las asignaciones se implementan con 3 acciones atomicas (leer, sumar o multiplicar y escribir en una variable) cuales son los valores posibles de s y t? (Nota no es necesario que liste todos los resultados pero si los casos que resulten significativos)

(13-11-2014) (22-12-2014)
2)Resuelva el problema de acceso a Sección Critica usando un proceso coordinador. En este caso cuando proceso SC[I] quiere entrar en su sección critica le avisa al coordinador, y espera que este le de permiso. Al Terminar de ejecutar su sección critica el proceso SC[I] le avisa al coordinador. Desarrolle una solución de grano fino usando solo variables compartidas (no semáforos, ni monitores).

```
int permiso[1:N] = ([N] 0), aviso[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;
}
}
```

(13-11-2014)
3(a) Defina que son los paradigmas master/worker, token passing y prueba/eco. Marque claramente ventajas y desventajas para el caso de mensajes sincrónicos y asincrónicos.

Algoritmos probe/echo: La interacción entre los procesos permite recorrer grafos o árboles (o estructuras dinámicas) diseminando y juntando información. Ejemplo: realizar un broadcast (sin *spanning tree*) o conocer la topología de una red cuando no se conocen de antemano la cantidad de nodos activos.
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. Ejemplo: determinar la terminación de un proceso en una arquitectura distribuida cuando no puede decidirse localmente.
Manager/workers: Implementación distribuida del modelo de *bag of tasks* que consiste en un proceso *controlador* de datos y/o procesos y múltiples *workers* que acceden a él para poder obtener datos y/o tareas a ejecutar en forma distribuida. Ejemplo: algoritmos de paralelismo recursivo en entornos distribuidos.

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

```
Chan token[1:n](), enter[1:n](), go[1:n](), exit[1:n]()
Helper[i: 1..n]:: do true -> {DMUTEX}
receive token[i]() #adquiere el token
if not(empty(enter[i]))-> #P[i] quiere entrar
receive enter[i]()
send go[i]()
receive exit[i]()
fi
send token[(i mod n) + 1]() #pasa el token
od
P[i:1..n]::do true ->
send enter[i]() #entry protocol
receive go[i]()
SC
send exit[i]() #exit protocol
SNC
od
```

(13-11-2014)
4(a) En que consiste la comunicación guardada y cual es su utilidad? Ejemplifique.
Con frecuencia un proceso se quiere comunicar con varios otros, 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 quedar bloqueado si en alguno 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 condición lógica que habilita la comunicación C (sea C un *send* o *receive*). Puede omitirse, asumiéndose *true*.
-B y C forman la **guarda**. La misma:
– tiene *éxito* si B es *true* y ejecutar C no causa demora.
– *falla* si B es *false*.
– se *bloquea* si B es *true* pero C no puede ejecutarse inmediatamente.
Puede indicarse varias sentencias de comunicación guardada en un if o un do. Si todas fallan se continúa la ejecución con la instrucción siguiente; si hay guardas con éxito se ejecuta una no determinísticamente, y si algunas se bloquean se demora al proceso hasta que pueda ejecutarse con éxito a alguna de ellas. Si se trata de un ciclo do, se itera hasta que todas las guardas fallen.

4(b) Describa como es la ejecución de sentencias de alternativa e iteración que contienen comunicaciones guardadas.

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

if B1; comunicacion1 → S1
[] B2; comunicacion2 → S2
fi

Se evalúan las expresiones booleanas B_i y la sentencia de comunicación.
-Si todas las guardas fallan (todos los B_i son *false*), el *if* termina sin efecto.
-Si al menos una guarda tiene éxito (su B_i es verdadero y la sentencia de comunicación puede realizarse sin demora), se elige una de ellas no determinísticamente.
-Si algunas guardas se bloquean (su B_i es verdadero pero no puede realizarse la comunicación en ese instante), se espera hasta que alguna tenga éxito. Luego de elegir una guarda exitosa se ejecuta la sentencia de comunicación asociada y, por último, se ejecutan las sentencias S_i relacionadas.
La ejecución de la iteración *do..od* es similar, iterando los pasos anteriores hasta que todas las guardas fallen.

4(c) 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
```

1.
2. **Si el valor de A=1 y B=2 antes del IF y solo P2 envía el valor 6**

En este caso se queda bloqueado, ya que la guarda (A==1) es true, pero no ha recibido mensaje de P3

1.
2. **Si el valor de A=0 y B=2 antes del IF y solo P2 envía el valor 8**

En este caso la guarda (A==0) es true y recibe mensaje de P2, aux=10

1.
2. **Si el valor de A=2 y B=0 antes del IF y solo P3 envía el valor 8**

En este caso la guarda (B==0) es true y recibe mensaje de P3, aux=15

1.
2. **Si el valor de A=2 y B=1 antes del IF y solo P3 envía el valor 9**

En este caso todas las guardas fallan, sale del IF y aux=1

1.
2. **Si el valor de A=1 y B=0 antes del IF y solo P3 envía el valor 14**

Existen dos guardas exitosas y son elegidas en forma no determinística aux=19 (A==1) o aux=21 (B==0).

1.
2. **Si el valor de A=0 y B=0 antes del IF y P3 envía el valor 9 y P2 el valor 5**

Existen dos guardas exitosas y son elegidas en forma no determinística aux=7 (A==0) o aux=16 (B==0).

(13-11-2014)

5(a) Cual es el objetivo de la programación paralela?

-El sentido de la programación paralela es ganar velocidad, permitiendo resolver un problema en menos tiempo que una solución secuencial, o resolver un problema más grande en el mismo tiempo.

5(b) Defina las métricas de speedup y eficiencia. Cual es el significado de cada una de ellas (que miden)? Cual es el rango de valores posibles de cada uno? Ejemplifique.

-SpeedUp: Indica la relación entre el tiempo de ejecución secuencial y paralelo de un programa. A menor tiempo paralelo, mayor será el speedup. Indica qué tan rápida es la solución paralela respecto de la secuencial. Su rango de valores varía entre [1; p) siendo p la cantidad de procesadores empleados en paralelo. **Eficiencia:** Indica qué tan bien se aprovechan los procesadores en la solución paralela, relacionando el speedup con la cantidad de procesadores empleados para conseguirla. Su rango de valores varía entre (0; 1).

5(c) Defina escalabilidad de un sistema paralelo.

5(d) 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-2 y en el otro por la función S=p/2 (para p>2) Cual de las dos soluciones se comportara mas eficientemente al crecer la cantidad de procesadores?

-El que mejor se comporta con mayor número de procesadores es aquel cuya función de speedup es p-2 por definición de speedup su rango esta entre 0 y p entonces con esta función el speedup es más cercano a p. Si comparamos la eficiencia E=S/p entonces tenemos (p-2)/p = 1-2/p en el primer caso, y (p/2)/p = 1/2 en el segundo, 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.

5(e)Describa conceptualmente que dice la ley de Amdahl (no es necesaria la formula)

-Permite calcular el límite superior para el speedup de un algoritmo en particular. En general, un algoritmo cualquiera realiza una entrada, procesamiento y salida. No todo el algoritmo puede mejorarse en una versión paralela, ya que normalmente la etapa de entrada -o inicializaciones- y salida mantendrán componentes secuenciales: sólo el procesamiento será apto de optimización. La mejora total alcanzable en un programa estará limitada entonces por su componente no paralelizable
El SpeedUp máximo alcanzable se obtiene considerando la valor óptimo de un speedup lineal para la parte paralelizable (SpeedUp Parte Paralelizable = P, la cantidad de procesadores). Como muchas veces la E/S llevan un tiempo mucho menor al procesamiento la limitación puede no ser tan fuerte. La ley de Amdahl indica que la mejora obtenida al alterar una porción del sistema estará limitada por la fracción del tiempo que dicha porción se utiliza. Así, es el algoritmo y no la cantidad de procesadores el factor que limita el speedup; agregar procesadores puede aumentar el speedup pero sólo hasta cierto punto.

5(f) Suponga que el tiempo de ejecución de un algoritmo secuencial es de 10000 unidades de tiempo de las cuales solo el 90% corresponde a código paralelizable. Cual es el limite en la mejora que puede obtenerse paralelizando el algoritmo?

El límite de mejora se da teniendo 9000 procesadores los cuales ejecutan una unidad de tiempo obteniendo un tiempo paralelo de 1001 unidades de tiempo. El *speedup* mide la mejora de tiempo obtenida con un algoritmo paralelo comparándola con el tiempo secuencial. S=Ts/Tp=10000/1001=

9,99001~10 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.

(22-12-2014)
6- 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

(22-12-2014)
7-Suponga que una imagen se encuentra representada por una matriz a (nxn), y que el valor de cada pixel es un numero 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) .

1.
    1.
    2. 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.
    3.
    4. Analice la solución de desde el punto de vista del numero de mensajes.
    5.

4*n^2*(n-1) (4 por N al cuadrado por (n-1) mensajes)

1.
    1.
    2. Puede realizar alguna mejora para reducir el numero de mensajes.
    3.

```
Chan valores [1..n,1..n](int, int);
Process grilla[i:1..n,j:1..n]{
Int v;
Int maxL=v; minL=v; cantv=0; x; y; max; min;
Cola vecinos(int, int)=null;
```

```
If (i !=1) push(vecinos, (i-1,j)); cantv++;
If (i !=n) push(vecinos, (i+1,j)); cantv++;
If (j !=1) push(vecinos, (i,j-1)); cantv++;
If (j !=n) push(vecinos, (i, j+1)); cantv++;

For(p =1 to n){
//envio mi info a mis vecinos
For(q=1 to cantv){
Pop(vecinos, (x,y));
Send valores[x,y](maxL,minL);
Push(vecinos,(x,y));
}
//recojo info de mis vecinos y actualizo
For (s =1 to cantv){
Receive valores [i,j](max, min);
If (max>maxL)
maxL=max;
if(min<minL)
minL=min;
}
}
}
```

(22-12-2014)
8-Sea la siguiente solución al problema del producto de matrices de nxn con p procesos en paralelo con variables compartidas:

```
Process worker[w=1 to p] {#strips en paralelo (p strips de n/p filas)}
Int first= (w-1) * n/p + 1
Int last= first + n/p – 1
For [j=1 to last]
{
        For [j= 1 to n]
        {
                C[i, j] = 0.0; //seq.128^2 “=”; para.128*32 “=”
```

For [k= 1 to N] {
C[I,J] = C[I,J] + a [I,K] * b[K,j];//seq. 128^3 “=” “+” “*”; para.128^2*32

}

}}

1.
2. Suponga n= 128 y cada procesador capaz de ejecutar un proceso. Cuantas asignaciones, sumas y productos se hacen secuencialmente (caso en que p=1)?

Asignaciones=128^2 + 128^3=2113536
Sumas=128^3=2097152
Productos=128^3=2097152
Cuantos se realizan en cada procesador en la solución paralela con p= 4?
Asignaciones=128*32 + 128^2 * 32=528384
Sumas=128^2 * 32=524288
Productos=128^2 * 32=524288

1.
2. si p1=p2=…=p4 y los tiempos de asignación son 1, de suma 2 y de producto 3, y si p4 es 4 veces mas lento, cuanto tarda el proceso total concurrente? Cual es el valor del speedup?

-Por ser el procesador #4 **el más lento, es el que se tiene en cuenta para el cálculo en paralelo**:
Asignaciones=528384 * 1 * 4 = 2113536
Sumas=524288 * 2 * 4 = 4194304
Productos=524288 * 3 * 4 = 6291456
TOTAL = 12599296
-Para el **procesamiento secuencial, se consideran los mejores procesadores**:
Asignaciones=2113536 * 1 = 2113536
Sumas=2097152 * 2 = 4194304
Productos=2097152 * 3 = 6291456
TOTAL = 12599296
-SpeedUp = 12599296 / 12599296 = 1 → Por ser uno de los procesadores 4 veces mas lento, no se redujo agregando 4 procesadores
Modifique el código para lograr un mayor speedup.

----------OTRAS----------

15-09-2010

6) Resuelva con monitores el siguiente problema: Tres clases de procesos comparten el acceso a una lista enlazada: searchers, inserters y deleters. Los searchers solo 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 mas searchers. Por ultimo, los deleters remueven items de cualquier

lugar de la lista. A lo sumo un deleter puede acceder a la lista a la vez, y el borrado tambien debe ser mutuamente exclusivo con searchers e inserciones.

```
monitor Acceso_Lista {

  cond okSearch;
  cond okInsert;
  cond okDelete;
  int nS=0, nI=0, nD=0;

procedure pedidoSearch() {
        while(nD>0) wait(okSearch);
                nS = nS + 1;
}

procedure liberaSearch() {
        nS = nS – 1;
        if(nS==0 AND nI==0 AND nD==0)
                signal(okDelete);
}

procedure pedidoInsert() {
        while(nI==1 or nD>0) wait(okInsert);
        nI = nI + 1;
}

procedure liberaInsert() {
        nI = nI – 1;
        # no puede haber mas de un insert en paralelo, asi que nI == 0
        if(nD==0){
                signal(okInsert);
                if(nS==0)
                        signal(okDelete);
        }
}
procedure pedidoDelete() {
        while(nD==1 or nS>0 or nI>0)
                wait(okDelete);
        nD = nD + 1;
}

procedure liberaDelete() {
        nD = nD – 1;
        # no puede haber mas de un delete en paralelo, asi que nD == 0
        signal_all(okSearch);
```

```
        if(nI==0) signal(okInsert);
                if(nS==0) signal (okDelete);
}
}
```

13-07-2011

3) Dada la siguiente solucion con monitores al problema de alocacion de un recurso con multiples unidades, transforme la misma en una solucion utilizando mensajes asincronicos.

```
        Monitor Alocador_Resurso {
                INT disponible = MAXUNIDADES;
                SET unidades = valores iniciales;
                COND libre;    #True cuando hay recursos

                procedure adquirir(INT id){
                        if(disponible==0) wait(libre)
                        else {
                                diponible = 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, clase_op oper, INT id_unidad);
chan respuesta[n] (INT id_unidad);

Process Alocador {
        INT disponible = MAXUNIDADES;
        SET unidades = valor inicial disponible;
        QUEUE pendientes;                       #inicialmente vacia
        #declaracion de otras variables

        WHILE(true){
                receive request(idCliente, oper, id_unidad);
                if(oper==adquirir){
```

```
                        if(disponible>0){               # puede atender el pedido ahora
                                disponible := disponible – 1;
                                remove(unidades, id_unidad);
                                send respuesta[idCliente](idUnidad);
                        } else {
                                insert(pendientes, idCliente);
                        }
                } else {                                #significa que el pedido es un liberar
                        if empty(pendientes) {
                                disponible := disponible + 1;
                                insert(unidades, idUnidad);
                        } else {                        #darle un id unidad a un cliente
esperando
                                remove(pendientes, idCliente);
                                send respuesta[idCliente](idUnidad);
                        }
                }
        }
}
```

13-11-2014

3)

b) Implemente una solución al problema de EM distribuida entre N procesos utilizando un algoritmo de tipo Token Passing con PMA.

Los procesos necesitan realizar su ejecución normal, y a la vez estar pendientes por si reciben un token para pasarlo al siguiente proceso en caso que no lo precise. Se utiliza un proceso helper que lo libere de esta tarea:

```
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 envía al siguiente cíclicamente
        }
}
```

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

Final Septiembre – 2014

1 y 3) Describa brevemente en qué consisten los mecanismos de RPC y Rendevouz. ¿Para que tipo de problemas son mas adecuados? ¿Por que es necesario proveer sincronización dentro de los módulos en RPC? Cómo puede realizarse la sincronización dentro de los módulos en RPC. Qué elementos de la forma general de rendevouz no se encuentran en el lenguaje ADA.

RPC

Los programas se descomponen en módulos que contienen procesos y procedimientos. Cada uno reside en espacios de memoria diferentes. Los procesos de un módulo pueden compartir variables y llamar a procedures declarados en ese módulo. Un proceso en un módulo puede comunicarse con procesos de otro módulo sólo invocando procedimientos exportados por éste.

Un modulo tiene dos partes: la especificacion, que contiene los headers de las operaciones que pueden ser llamadas por otros; y el cuerpo, que contiene la implementacion de los procedimientos, variables y procesos locales.

Forma general de un modulo:

```
modulo unNombre
        headers de operaciones exportadas;
cuerpo
        declaracion de variables;
        codigo de inicializacion;
        procedimientos de operaciones exportadas;
        procedimientos locales y procesos;
fin unNombre
```

El header de un procedure visible tiene la forma:

```
        op opname (formales) [returns result]
```

El cuerpo de un procedure visible es contenido en una declaración proc:

```
proc opname (identif. formales) returns identificador resultado
declaración de variables locales
        sentencias
end
```

Un proceso (o procedure) en un módulo llama a un procedure en otro ejecutando:

```
call Mname.opname (argumentos)
```

Cuando un modulo quiere llamar a otro, un nuevo proceso sirve el llamado y los argumentos son pasados como mensajes entre el llamador y el proceso server. Mientras se queda esperando hasta que el server finalice la ejecucion de la operación solicitada. Una vez que termina, envia los resultados al proceso que lo llamo. Una vez que el primer proceso los recibe, puede continuar con su ejecucion.

Si el proceso llamador y el procedure estan en el mismo espacio de direcciones, es posible evitar crear un nuevo proceso pero en general esto no sucede ya que el llamado sera remoto.

Rendezvous

Rendezvous combina comunicación y sincronización. Un proceso cliente invoca una operación (call). Esta es servida por un proceso existente. Un proceso servidor usa una sentencia de entrada para esperar por un call y luego actuar. Las operaciones son servidas de a una por vez.

Utiliza un modulo que se espcifica de forma similar a RPC pero que varia en el body: proceso unico que da servicio a las operaciones.

Si un módulo exporta opname, el proceso server en el módulo realiza rendezvous con un llamador de opname ejecutando una sentencia de entrada:

```
in opname (identif. formales) → S; ni
```

Las partes entre in y ni se llaman operación guardada.

Una sentencia de E/ demora al proceso server hasta que haya al menos un llamado pendiente de opname; luego elige el llamado pendiente más viejo, copia los argumentos en los parámetros formales, ejecuta S (lista de sentencias que dan servicio a la invocacion de la operacion) y finalmente retorna los parámetros de resultado al llamador. Luego, ambos procesos pueden continuar.

Combinando comunicación guardada con rendezvous:

```
in op1 (formales1) and B1 by e1 → S1;
...
opn (formalesn) and Bn by en → Sn;
ni
```

Ambos son tecnicas de comunicación y sincronizacion entre procesos que suponen un canal bidireccional, por lo tanto, son ideales para programar aplicaciones cliente/servidor.

RPC es solo un mecanismo de comunicación, por lo que necesitamos proveer sincronización. Existen dos enfoques:

Asumir que todos los procesos en el mismo modulo se ejecutan con exclusion mutua. Los procesos, ademas, deber poder programar sincronizacion por condicion. Para esto se pueden utilizar variables condicion como con los monitores, o utilizar semaforos.
Asumir que todos los procesos se ejecutan concurrentemente por lo tanto hay que programar explicitamente la exclusion mutua y la sincronizacion por condicion. Para esto se pueden utilizar mecanismos como semaforos, monitores, rendezvous o incluso pasaje de mensajes.

Ada soporta rendezvous mediante sentencias accept (similares a la forma simple de in) y comunicación guardada mediante sentencias select (similar a la forma general de in). Esta ultima es menos poderosa a la forma general in ya que select no puede referenciar argumentos a operaciones o contener expresiones de scheduling. Esto hace dificil resolver problemas de sincronizacion y scheduling.

2) Hacer una sección critica con variables compartidas

4) Dados los siguientes dos segmentos de código, indicar para cada uno de los ítems si son equivalentes o no. Justificar

En el if las guardas se evaluan en algun orden arbitrario (eleccion no deterministica). Si ninguna guarda es verdadera, el if no tiene efecto.
En el do las entencias guardadas son evaluadas y ejecutadas hasta que todas las guardas sean falsas. La eleccion es no deterministica si mas de una guarda es verdadera.

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

En el seegmento 1, para valores de cant que esten entre -9 y 10, excepto cuando cant=0, todas las condiciones serian falsas, por lo tanto la ejecucion del do terminaria. Esto no se da en el segmento 2: si bien ante las mismas condiciones el if terminaria su ejecucion, gracias al while(true), la iteracion se seguiria realIzando.

b) INCOGNITA equivale a: (cant > -100)

En este caso si son equivalentes ya que las condiciones abarcan todos los numeros enteros por los que puede pasar cant, entonces, siempre habria una condicion verdadera.

c) INCOGNITA equivale a: ((cant > 0) or (cant < 0))

No son equivalentes ya que si se da que cant = 0, ninguna de las guardas seria verdadera, por lo tanto, la ejecucion del do terminaria, la ejecucion del if tambine pero como esta encerrado en un while(true), la iteracion continua.

d) INCOGNITA equivale a: ((cant > -10) or (cant < 10))

No son equivalentes ya que para cant = 10 o cant = -10 la ejecucion tanto del do como del if terminarian (todas las condiciones son falsas), pero debido al while(true) el if vuelve a ejecutarse.

e) INCOGNITA equivale a: ((cant >= -10) or (cant <= 10))

Son equivalentes. Evita lo que sucede en el punto d).

5) 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;# Ultima 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)? Cuántas se realizan en cada procesador en la solución paralela con P=8?

Solucion secuencial

Sea n=128, P=1, w=1 → first = (1 – 1) * 128/1 = 0
last = 0 + 128/1 – 1 = 127

Asignaciones = $128^2 + 128^3 = 16384 + 2097152 = 2113536$
Sumas = $128^3 = 2097152$
Productos = $128^3 = 2097152$

Solucion paralela

Sea n=128, P=8, w=8 → first = (8 – 1) * 128/8 = 112
last = 112 + 128/8 – 1 = 127

Primer for: desde 112 hasta 127 → itera 16 veces

Asignaciones = 16 * 128 + 16 * 128² = 2048 + 262144 = 264192
Sumas = 16 * 128² = 262144
Productos = 16 * 128² = 262144

b) 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.

Tiempo proceso total concurrente

Procesadores P1-P7 (cada uno):

Asignaciones = 264192 * 1 = 264192
Sumas = 262144 * 2 = 524288
Productos = 262144 * 3 = 786432

Total = Asignaciones + Sumas + Productos = 1574912

Procesador P8 (4 veces mas lento):

Total = 1574912 * 4 = 6299648

Tiempo secuencial

Asignaciones = 2113536 * 1 = 2113536
Sumas = 2097152 * 2 = 4194304
Productos = 2097152 * 3 = 6291456

Total = Asignaciones + Sumas + Productos = 12599296

Speedup

Tomo el valor del procesador 8, por ser el mas lento, para hacer el calculo del speedup (tener en cuenta que los procesadores se ejecutan en paralelo).

12599296 / 6299648 = 2

Para mejorar el speedup se podria reducir la cantidad de filas asignadas a P8, agregandoselas al resto de los procesadores (bandas asimetricas), ya que son mas rapidos. Esto asegura que los procesadores no queden ociosos mientras esperan que P8 finalice.

Por ejemplo: a P8 se le asignan 2 filas, y a P1 ... P7 18. De esta forma se corrigen los tiempos:

Asignaciones = 18 * 128 + 18 * 128^2 = 2304 + 294912 = 297216
Sumas = 18 * 128^2 = 294912
Productos = 18 * 128^2 = 294912

P1 = … = P7 = (297216 asignaciones * 1) + (294912 sumas * 2) + (294912 productos * 3)
P1 = … = P7 = 297216 asignaciones + 589824 sumas + 884736 productos
P1 con 18 filas = … = P7 con 18 filas = 1771776 unidades de tiempo

Asignaciones = 2 * 128 + 2 * 128^2= 256 + 32768 = 33024
Sumas = 2 * 128^2 = 2 * 16384 = 32768
Productos = 2 * 128^2 = 2 * 16384 = 32768

P8 = (33024 asignaciones * 1) + (32768 sumas * 2) + (32768 productos * 3)
P8 = 33024 asignaciones + 65536 sumas + 98304 productos
P8 con dos filas = 196864
P8 con dos filas = 196864 * 4
P8 con dos filas = 787456 unidades de tiempo

Speedup = 12599296 / 787456 = 7.1

Final 16 – 12 - 2015

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

Problema de la seccion critica

Una seccion critica es una secuencia de sentencias que referencian a variables compartidas entre distintos procesos. Cada CS esta prcedida por un protocolo de entrada, seguida por un protocolo de salida. Estos protocolos permiten implementar acciones atomicas de grano grueso, ademas se deben disenar de manera tal que se garanticen las siguientes propiedades:

Exclusion mutua: Propiedad de seguridad. Solo un proceso esta ejecutando su SC en un momento determinado.
Ausencia de deadlock: Propiedad de seguridad. Si dos o mas procesos estan intentando acceder a su SC, uno lo lograra.
Ausencia de espera innecesaria: Propiedad de seguridad. Si un proceso esta intentando ingresar a su SC y el resto ya termino o esta en su SNC, el primero no esta impedido de ingresar.
Eventual entrada: Propiedad de vida. Un proceso que esta intentando ingresar a su SC, eventualmente lo hara.

Forma general:

```
process SC [i= 1 to n]{
        while(true){
                …
                protocolo de entrada;
                SC;
                protocolo de salida;
                SNC;
                ...
        }
}
```

Spin locks

Las soluciones de este tipo se denominan asi ya que los procesos se quedan iterando (spinning) mientras esperan que se limpie el lock. Un lock es una variable booleana que indica si algun proceso esta en su SC o no.

Ej protocolo de entrada y salida grano grueso: <await (not lock) lock = true;>; lock = false;

Test-and-set: Se utiliza la instrucción especial TS(lock) disonible en los procesadores. TS sobreescribe el valor de lock y devuelve su valor inicial. Protocolo de entrada:
while(TS(lock)) skip;
Desventajas:
Memory contention: Lock es una variable compartida que cada proceso demorado la referencia continuamente → baja performance.
Overhead por cache invalida: TS() sobreescribe el valor de lock por mas que este no cambie. Esto es costoso ya que, en multiprocesadores, cuando una variable se escribe, las caches del resto se invalidan si tienen una copia de la variable.
Test-test-and-set: Mejora, aunque no elimina, memory contention ya que agrega una instancia de testeo: while (lock == false) skip;. De esta manera se asegura que solo se ejecutara TS(lock) cuando su valor sea verdadero.
Sentencias await:

SCEnter
while(not B){SCExit; Delay; SCEnter}
S
SCExit
Esto permite reducir la contencion de memoria ya que los procesos pueden demorarse antes de volver a intentar ganar el acceso a la SC.

Desventajas generales de soluciones de tipo Spin locks: no controla el orden de los procesos demorados. Esto genera la posibilidad de que alguno no entre nunca a su SC si la politica de scheduling no es fuertemente fair.

Tie breaker

Rompe el empate cunado varios procesos intentan entrar a la SC utilizando una variable adicional que indica cual fue el ultimo proceso en entrar a su SC. Utiliza una variable para indicar cual fue el ultimo proceso en entrar a su SC. Se demora al ultimo proceso en comenzar su protocolo de entrada.

Desventajas: complejo y costoso en tiempo.

Ticket

Se reparten numeros (tickets) y se espera turno. Cuando un cliente llega, se le asigna un numero mayor al repartido entre los clinetes anteriores. Luego este valor se incrementa de manera tal que cuando llegue un nuevo proceso, siempre tenga el ultimo turno.

El proceso se queda esperando a su turno para entrar a su SC. Una vez que lo logra, incrementa el valor del proximo turno a ser atendido.

Su principal desventaja es que incrementar indefinidamente el contador puede causar desbordamiento aritmetico. Ademas, si el procesador no posee la instrucción Fetch-and-add la solucion puede no ser fair.

Bakery

Es fair y no requiere instrucciones especiales pero es mas complejo. Cuando un proceso “llega”, observa los turnos asignados al resto de los procesos y se asigna uno mayor. Luego espera a ser atendido (el proceso con el menor nro. sera el siguiente). Este algoritmo, en lugar de utilizar un contador central, chequea el turno entre los demas procesos.

2)

a. ¿En qué consiste la comunicación guardada y cual es su utilidad? Ejemplifique.

Las sentencias de comunicación guardada soportan comunicación no deterministica. Tiene la forma:

$$B; C \rightarrow S$$

B es una expresion booleana, C una sentencia de comunicación y S una lista de sentencias. B puede ser omitida, en cuyo caso su valor implicito es true.

B y C forman la guarda. Una guarda tiene éxito si B es true y ejecutar C no causa delay. Una guarda falla si B es false. Una guarda se bloquea si B es true y C no puede ejecutarse inmediatamente.

Aparecen en las sentencias IF y DO.

Ej:

if $B^1 ; C^1 \rightarrow S^1 ;$
$B^2 ; C^2 \rightarrow s^2 ;$
fi

do $B^1 ; C^1 \rightarrow S^1 ;$
$B^2 ; C^2 \rightarrow s^2 ;$
od

b. Describa cómo es la ejecución de sentencias de alternativa e iteración que contienen comunicación guardadas.

IF: Primero se ejecutan las expresiones booleanas de las guardas. Si ninguna es verdadera, el IF termina; si a lo sumo alguna es verdadera, se elige de forma no deterministica; si todas las guardas se bloquean, se espera hasta que una tenga éxito. Luego de elegir una guarda exitosa, se ejecuta C y luego S. Luego el IF termina.

El DO se ejecuta de una forma similar: se repite hasta que todas las guardas fallen. Por ejemplo, si ninguna guarda tiene una expresion booleana, loopeara para siempre.

c. Dado el siguiente bloque de código, indique para cada inciso que valor quedó en Aux, o si el codigo quedo bloqueado.

```
Aux = 1;
....
if      (A==0); P2?(Aux) > Aux = Aux +2;
        (A==1); P3?(Aux) > Aux = Aux +5;
        (B==0); P3?(Aux) > Aux = Aux +7;
endif;
...
```

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

Se queda bloquado en la guarda A==1 ya que la condicion es la unica verdadera pero P3 no se ejecuto.

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 (unica con condicion verdadera) y se ejecuta P2. Luego el valor de aux sera 10.

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

Entra en la tercer guarda (unica con condicion verdadera) y se ejecuta P3. Luego el valor de aux sera 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 condicion 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 tercer guarda (eleccion 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 condicion de la primer guarda y la condicion de la ultima son verdaderas; ademas P3 y P2 se ejecutaron, por lo tanto no hay bloqueo. Como la eleccion 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.

3) 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.

a. 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 Arbol.

Arquitectura en estrella (centralizada)

En este tipo de arquitectura todos los procesos (workers) envian su valor local al procesador central (coordinador), este suma los N datos y reenvia la informacion 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 envian casi al mismo tiempo. Estos se quedaran esperando hasta que el coordinador termine de computar la suma y envie el resultado a todos. Por otro lado, el uso de la memoria es ineficiente: cada worker tendra una copia de la suma final.

Anillo circular

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

Este esquema consta de dos etapas:

1. Cada proceso recibe un valor y lo suma con su valor locar, transmitiendo la suma local a su sucesor.
2. Todos deciben la suma global.

P[0] debe ser algo diferente para poder “arrancar” el procesamiento: debe envia su valor local ya que es lo unico que conoce.. Se requeriran 2(n-1) mensajes.

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

Totalmente conectada (simetrica)

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 esta calculando la suma total y tiene acceso a los n datos.

Se ejecutan n(n-1) mensajes. Si se dispone de una primitiva de broadcast, seran n mensajes. Es la solucion mas corta y sencilla de programar, pero utiliza el mayor numero 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, deberia construir un arbol de expansion de la misma, poniendose a el mismo como raiz.

El nodo raiz envia un mensaje por broadcast a todos los hijos, junto con el arbol construido. Cada nodo examina el arbol recibido para determinar los hijos a los cuales deben reenviar el mensaje, y asi sucesivamente.

Se envian n-1 mensajes, uno por cada padre/hijo del arbol.

b. Escriba las soluciones de al menos dos de las arquitecturas mencionadas.

Arquitectura en estrella (centralizada)

```
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);
}
```

Anillo circular

```
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 (simetrica)

```
chan valor[n](INT);
process p[ i=0 to n-1]{
        INT v;
        INT nuevo, suma = v;

        for [k=0 to n-1 st k <> i]
                send valor[k](v);
        for [k=0 to n-1 st k <> i]{
                receive valor[i](nuevo);
                suma = suma+nuevo;
        }
}
```

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

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

Sean dos procesos P1 y P2, cada uno inicialmente con n/2 valores (arreglos a1 y a2 respectivamente). Los n/2 valores de cada proceso se encuentran ordenados inicialmente. La idea es realizar una serie de intercambios: en cada uno P1 y P2 intercambian a1[mayor] y a2[menor], hasta que a1[mayor]<a2[menor].

```
Process P1{
        INT a1[1:n/2]                   #inicializado con n/2 valores
        const mayor:=n/2; INT nuevo
        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
viejor a1[mayor];
          P2 ! a1[mayor];
          P2 ? nuevo;
        od
```

```
}

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

Sigo la ejecucion con un ejemplo:

Inicialmente:

a1---> 1-3-6-9-10
a2---> 2-4-5-7-8

Ejecucion P1
mayor:=5
envia a1[mayor] a P2, le mando el valor 10
recibe el valor 2 de P2, nuevo=2
***do
///Primera vuelta
si a1[mayor]>nuevo, ordeno; 10>2 VERDADERO, entonces ordeno alterando a1
a1---> 1-2-3-6-9
envia a1[5] a P2, le mando el valor 9
recibe el valor 4 de P2, nuevo=4
///Segunda vuelta
si a1[5]>nuevo, ordeno; 9>4 VERDADERO, entoncs ordeno alterando a1
a1---> 1-2-3-4-6
envia a1[5] a P2, le mando el valor 6
recibe el valor 5 de P2 nuevo=5
//Tercera vuelta
si a1[5]>nuevo, ordeno; 6>5 VERDADERO, entoncs ordeno alterando a1
a1---> 1-2-3-4-5
envia a1[5] a P2, le mando el valor 5
recibe el valor 6 de P2 nuevo=6
///Cuarta vuelta
si a1[5]>nuevo, ordeno; 5>6 FALSO

Ejecucion P2
menor:=1
recibe el valor 10 de P1, nuevo=10
envia a2[menor] a P2, le mando el valor 2
***do
///Primera vuelta
si a2[menor]<10, ordeno; 2<10 VERDADERO, entonces ordeno alterando a2
a2---> 4-5-7-8-10
recibe el valor 9 de P1, nuevo=9
envia a2[1] a P2, le mando el valor 4
///Segunda vuelta
si a2[1]<9, ordeno; 4<9 VERDADERO, entonces ordeno alterando a2
a2---> 5-7-8-9-10
recibe el valor 6 de P1, nuevo=6
envia a2[1] a P2, le mando el valor 5
//Tercera vuelta
si a2[1]<6, ordeno; 5<6 VERDADERO, entonces ordeno alternando a2
a2---> 6-7-8-9-10
recibe el valor 5 de P1, nuevo=5
envia a2[1] a P2, le mando el valor 6
///Cuarta vuelta
si a2[1]<5, ordeno; 6<5 FALSO

b. ¿Cuántos mensajes intercambian en el mejor caso? ¿Y en el peor caso?

En el mejor caso los procesos intercambian solo un par de valores. En el peor caso intercambian n/2 + 1 valores: n/2 para tener cada valor en el proceso correcto y uno para detectar terminacion.

c. Utilizando la idea de a), 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[odd] intercambian valores con el siguiente proceso impar P[odd+1] si el valor esta fuera de orden. En rondas pares, los procesos pares P[even] intercambia valores con el siguiente proceso par P[even+1] si los valores estan fuera de orden. P[1] y P[n] no hacen nada en las rondas pares.

```
process Proc[i:1..k] {
int largest = n/k, smallest = 1, a[1..k], dato;
  ... inicializar y ordenar 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) {
      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;
        }
      }
    } elseif(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];
      }
    }
  }
}
```

d. ¿Cuántos mensajes intercambian en c) en el mejor caso? ¿Y en el peor caso?
Nota: Utilice un mecanismo de pasaje de mensajes, justificando la elección del mismo.

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

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.

5) Suponga que una imagen se encuentra representada por una matriz A(n x 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 está en el proceso P(i,j)). Cada proceso puede comunicarse sólo con sus vecinos izquierdo, derecho, arriba y abajo (los proceso de las esquinas tienen solo 2 vecinos y los otros en los borde de la grilla tienen 3
vecinos).

a. Escriba un algoritmo HeartBeat que calcule el máximo y el mínimo 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 nuevatop[1:n,1:n];
        int r = 0; bool listo = false;
        int emisor; bool qlisto;
        int miValor, max, min;              # miValor inicializado con el valor del pizel
        top[p,1..n] = vecinos;              # llena la fila para los vecinos
        max := miValor; min:= miValor;
        while(not listo) {                          # envía 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,nuevatop, nuevoMax, nuevoMin);

                        # recibe las topologías y hace OR con su top juntando la informacion
                        top = top or nuevatop;
                        # actualiza los maximos y minimos
                        if(nuevoMax>max) nuevoMax := max;
                        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,nuevatop, nuevoMax, nuevoMin);
}
```

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

Si M es el numero maximo de vecinos que puede tener un nodo, y D es el diametro de la red, el numero de mensajes maximo que pueden intercambiar es de 2n * m * (D+1). Esto es porque cada nodo ejecuta a lo 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.

Final 22/12/2014

2) Resuelva el problema de acceso a Seccion Critica usando un proceso coordinador. En este caso cuando proceso SC[I] quiere entrar en su sección critica le avisa al coordinador, y espera que este le de permiso. Al Terminar de ejecutar su sección critica el proceso SC[I] le avisa al coordinador. Desarrolle una solución de grano fino usando solo variables compartidas (no semáforos, ni monitores).

```
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;
        }
}
```

4) Cual es el objetivo de la programación paralela?

Su principal objetivo es reducir los tiempos de ejecucion. Tambien se preocupa por resolver varias instancias de un mismo problema en la misma cantidad de tiempo.

a) Defina las métricas de speedup y eficiencia. Cual es el significado de cada una de ellas (que miden)? Cual es el rango de valores posibles de cada una? Ejemplifique.

La performance de un programa es el tiempo de ejecucion total. Sea TS el tiempo de ejecucion para resolver un problema de forma secuencial que se ejecuta en un unico procesador y TP el tiempo de ejecucion para resolver el mismo problema usando programacion paralela que se ejecuta en p procesadores:

Speedup = TS / TP

Su rango de valores varía entre [1; p) siendo p la cantidad de procesadores empleados en paralelo.

La eficiencia es una medida que permite determinar cuan bien un programa paralelo utiliza procesadores extra.

Eficiencia = Speedup / p = TS / (p * TP)

Si la eficiencia es 1 entonces el speedup es perfecto.

b) Defina escalabilidad de un sistema paralelo.

Da una medida de usar eficientemente un numero creciente de procesadores.

c) Suponga que la solución a un problema es paralelizada sobre p procesadores de dos maneras diferentes. En un caso, el speedup(5) esta regido por la funcion S= p-4 y el otro por la funcion S= p/3 para p > 4. Cual de las dos soluciones se comportara mas eficientemente al crecer la cantidad de procesadores?

Si la cantidad de procesadores crece, la segunda solucion se comportara mas eficientemente (S = p/3 para p > 4). Ya que si p es muy grande y se divide en 3 partes, estas no seran tan grandes. En cambio, si p es muy grande y se le resta 4, el resultado seguiria siendo muy grande.

d) Describa conceptualmente que dice la ley de Amdahl (no es necesaria la formula).

Esta relacionada con el limite del speedup. Dice que la mejora de un programa al mejorar el speedup de una parte esta limitada por la fraccion de tiempo que toma realizar esa parte. En otras palabras, para un problema dado existe un maximo speedup alcanzable independientemente del numero de proesadores.

e) Suponga que el tiempo de ejecución de un algoritmo secuencial es de 10000 unidades de tiempo, de las cuales 95% corresponden a código paralelizable . Cual es el límite en la mejora que puede obtenerse paralelinzado el algoritmo.

El limite de mejora seria utilizando 9500 procesadores los cuales ejecutan una unidad de tiempo objetiendo un tiempo paralelo de 501 unidades de tiempo. El speedup mide la mejora del tiempo objetida con un algoritmo paralelo comparandola con el

secuencial. Speedup = TS/TP = 10000/501 = 19,9 ~ 20. Aunque se ejecuten mas procesadores el mayor speedup alcanzable es este, cumpliendose asi la Ley de Amdahl diciendo para todo un problema hay un limite de paralelizacion, dependiendo el mismo no de la cantidad de procesadores, sino de la cantidad de codigo secuencial.