

¿Qué es la propiedad ASV? De un ejemplo de código

La propiedad “A lo sumo una vez” es lo que debe cumplir las sentencias críticas en programas concurrentes para poder considerar que su ejecución es atomizable sin necesidad de hardware.

Una sentencia cumple ASV cuando accede a lo sumo una variable compartida, y esta es referenciada a lo sumo una vez. Cuando una sentencia cumple la propiedad puede tratarse como si fuera atómica ya que no existe riesgo de interferencia entre distintos programas.

Ejemplo:

```
boolean x = true; → variable compartida
```

```
if (x) { → operación sobre la variable compartida
    print("ejemplo")
}
```

Este ejemplo cumple la definición ya que se declara una única variable compartida, la cual solo se referencia una sola vez en el condicional del if para hacer una operación de lectura.

¿Qué es atomizable?

Una sentencia o bloque de sentencias es atomizable si su comportamiento concurrente equivale al de ejecutarla de una sola vez, de manera indivisible.

Definición de procesamiento concurrente, paralelo y secuencial.

Procesamiento secuencial: implica la ejecución de una tarea o proceso a la vez, siguiendo un orden temporal estricto, existe un único flujo de control, cuando se ejecuta una instrucción y esta finaliza, se ejecuta la siguiente sin superposición temporal.

Procesamiento concurrente: implica la ejecución de múltiples tareas o procesos con el objetivo de resolver un problema de manera conjunta. Múltiples procesos están en progreso al mismo tiempo, aunque no necesariamente se ejecuten simultáneamente.

Procesamiento paralelo: implica la ejecución de dos o más tareas o procesos que se ejecutan al mismo tiempo. Se considera como un tipo particular de concurrencia.

¿Qué es política de scheduling? Describa fairness y tipos de fairness.

Una política de scheduling es el conjunto de reglas o criterios que utiliza el sistema operativo o sistema concurrente para decidir qué acción atómica de qué proceso o hilo se ejecutará a continuación, cuando existen múltiples acciones atómicas o hilos listos para ejecutarse.

El scheduler define el orden de ejecución, la asignación de tiempo de CPU, la interrupción o reanudación de procesos, gestión de prioridades, tiempos de espera, etc.

El concepto de fairness describe el nivel de equidad con el que el scheduler permite que los procesos o hilos avancen.

Su objetivo es asegurar que todos los procesos tengan la oportunidad de avanzar y realizar sus operaciones, evitando situaciones no deseadas como deadlock o inanición. Este mecanismo garantiza que todos tengan oportunidades razonables de ejecutar sus acciones atómicas.

Hay distintos tipos:

- **Fairness incondicional:** Garantiza que toda acción atómica incondicional que es elegible eventualmente es ejecutada.
- **Fairness débil:** Además de ser incondicionalmente fair, toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada, asumiendo que su condición se vuelve true y permanece true hasta que es vista por el proceso que ejecuta la acción atómica condicional.
- **Fairness fuerte:** Además de ser incondicionalmente fair, toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada pues su guarda se convierte en true con infinita frecuencia.

Ejemplos de código:

Incondicional: ambas acciones están listas para ejecutarse todo el tiempo, el fairness garantiza que ambos procesos no quedarán eternamente sin ejecutar.

Proceso P1:

```
a1: print("P1 ejecuta") // acción incondicional
```

Proceso P2:

```
a2: print("P2 ejecuta") // acción incondicional
```

Débil: la condición debe quedarse en true para que eventualmente las operaciones del proceso P2 se ejecuten.

variable compartida x = false

Proceso P1:

```
a1: x := true // incondicional
```

Proceso P2:

```
a2: if (x) then print("P2 ejecuta") // acción atómica condicional
```

Fuerte: garantiza que en alguna de las infinitas veces que x se vuelva true, ejecutará P2

variable compartida x = false

Proceso P1:

```
a1: x := not x // alterna true / false infinitamente
```

Proceso P2:

```
a2: if (x) then print("P2 ejecuta") // acción condicional
```

¿Qué es deadlock? ¿Cuáles son las condiciones para que ocurra?

Deadlock es un bloqueo en la que dos o más procesos o hilos quedan atrapados en un estado en el que ninguno puede continuar su ejecución debido a que cada uno está esperando que el otro libere un recurso que necesita, es decir, se provoca un loop infinito.

Las condiciones para que ocurra son:

1. Recursos reusables serialmente: los procesos comparten recursos que pueden usar con exclusión mutua.
2. Adquisición incremental: los procesos mantienen los recursos que poseen mientras esperan adquirir recursos adicionales.
3. No-preemption: una vez que son adquiridos por un proceso, los recursos no pueden quitarse de manera forzada sino que solo son liberados voluntariamente.
4. Espera cíclica: existe una cadena circular de procesos tal que cada uno tiene un recurso que su sucesor en el ciclo está esperando adquirir.

Con evitar alguna de las cuatro condiciones, se evita el deadlock.

¿Cuál es el problema de la sección crítica? ¿Cuáles son las cuatro propiedades que deben cumplir las soluciones para este problema?

El problema de la sección crítica surge cuando varios procesos comparten memoria y necesitan acceder o modificar los mismos datos. Esa porción del código donde se accede a datos compartidos se conoce como sección crítica. Si dos procesos ingresan a esta sección al mismo tiempo, pueden producirse resultados incorrectos o estados inconsistentes. Para mitigar este problema se utilizan mecanismos de coordinación que controlan el acceso mediante exclusión mutua, asegurando que solo un proceso a la vez ejecute su sección crítica y que todos los procesos puedan avanzar sin quedar bloqueados indefinidamente.

Las cuatro propiedades que debe satisfacer las soluciones al problema de la sección crítica se dividen entre propiedades de seguridad y propiedades de vida, y estas son:

1. Exclusión mutua: como máximo un proceso está en su sección crítica en un momento dado, el objetivo es evitar que dos o más procesos puedan encontrarse en la misma sección crítica al mismo tiempo.
2. Ausencia de Deadlocks: si dos o más procesos intentan entrar en sus secciones críticas, al menos uno de ellos progresará, el objetivo es evitar que los procesos queden bloqueados en un punto muerto y no puedan avanzar.
3. Ausencia de demora innecesaria: si un proceso intenta entrar en su sección crítica y los otros procesos están en secciones no críticas o ya han terminado, el primer proceso no debe ser impedido de entrar en su sección crítica, el objetivo es evitar la espera innecesaria de un recurso ya disponible.
4. Eventual entrada: garantiza que un proceso que intenta entrar en su sección crítica tiene la posibilidad de hacerlo en algún momento eventualmente lo hará, el objetivo es evitar la inanición garantizando que un proceso no sea postergado indefinidamente.

Compare los algoritmos para resolver el problema de la sección crítica (spin locks, tie breaker, ticket, bakery) marcando ventajas y desventajas de cada uno.

Los Spin locks son una técnica cuyo objetivo es hacer atómico el await de grano grueso utilizando instrucciones atómicas que están disponibles en la mayoría de los procesadores.

Los procesos se quedan iterando spinning mientras esperan que el lock se libere.

- Ventajas: cumple las cuatro propiedades de la sección crítica sólo si el scheduling es fuertemente fair, ya que no impone ningún orden, aunque una política de scheduling débilmente fair es aceptable. Es sencillo de implementar.
- Desventajas: puede ocurrir inanición (es posible que algún proceso no entre nunca a la sección crítica si la política de scheduling no es fuertemente fair, ya que este algoritmo no impone un orden). Implica busy waiting, lo cual es inefficiente en sistemas de multiprogramación, aunque aceptable si cada proceso se ejecuta en su propio procesador.

El objetivo de Tie-Breaker es el mismo que el de los Spin locks, pero con la característica distintiva de establecer un orden lógico para romper empates.

El algoritmo no usa instrucciones atómicas especiales. Cada proceso utiliza una variable para indicar que comenzó su protocolo de entrada, y una variable compartida adicional se usa para romper empates. El algoritmo demora (quita prioridad) al último en comenzar su protocolo de entrada.

- Ventajas: evita empates, y solo requiere una política de scheduling débilmente fair para asegurar la eventual entrada. No usa instrucciones especiales.
- Desventajas: Es más complejo, costoso en tiempo especialmente cuando hay N procesos, ya que requiere más verificaciones.

El objetivo del algoritmo Ticket es el mismo que el de Tie-Breaker y spin locks. Funciona repartiendo números o turnos.

Los procesos toman un número que es amyor que el de cualquier otro proceso que esté esperando ser atendido, y luego esperan hasta que todos los procesos con números más pequeños hayan sido atendidos

- Ventajas: respeta el orden, la eventual entrada es asegurada por una política de scheduling débilmente fair. Garantiza la ausencia de deadlock y la ausencia de demora innecesaria debido a los valores de turnos únicos.
- Desventajas: requiere de la instrucción atómica Fetch-and-Add (FA). Si no existe, debe ser simulada utilizando una sección crítica grande, lo que puede resultar en que la solución no sea fair.

El Algoritmo Bakery se utiliza como alternativa cuando no existe la instrucción atómica Fetch-and-Add (FA). Su objetivo es igual al algoritmo de Ticket, siendo una variación de este último.

Cada proceso que intenta ingresar recorre los números de los demás procesos y se autoasigna uno mayor. Luego espera a que su número sea el menor de todos los que están esperando.

- Ventajas: es fair (garantiza una espera limitada). No requiere instrucciones atómicas especiales.
- Desventajas: es más complejo que otros algoritmos. La solución de grano grueso no es implementable directamente.

Defina el problema general de alocación de recursos. Defina la política de alocación de recursos SJN. ¿Es fair?

El problema general de alocación de recursos se enmarca en la administración de recursos compartidos dentro de los sistemas concurrentes. El problema consiste en cómo gestionar el acceso a estos recursos de manera que se logre un equilibrio entre el acceso por parte de todos los procesos.

El objetivo central de la solución a este problema es evitar dos situaciones no deseadas que surgen debido a la competencia por los recursos, que son el deadlock y la inanición.

La política de alocación de recursos SJN (Shortest Job Next) es un mecanismo de asignación de recursos que prioriza a los procesos que menos tiempo utilizarán el recurso. Para esto, cada proceso envía su identificador junto a su tiempo estimado de utilización del recurso, y el scheduler elige al que menos tiempo lo utilizará.

La política SJN minimiza el tiempo promedio de ejecución, pero es unfair, ya que a los procesos que deban utilizar por mucho tiempo el recurso quizás nunca se les sea asignado, esto incumple una de las propiedades del problema de sección crítica que es la eventual entrada.

¿En qué consiste la comunicación guardada y cuál es su utilidad? Describa cómo es la ejecución de sentencias de alternativa e iteración que contienen comunicaciones guardadas.

La comunicación guardada es un mecanismo introducido en CSP (communicating sequential processes) y los modelos de PMS (pasaje de mensajes síncrono). sirve para manejar la comunicación de manera no determinista, permitiendo que un proceso elija entre varias comunicaciones posibles según cuales están listas para ejecutarse.

Una sentencia de comunicación guardada tiene la forma general: B; C → S.

- B es una condición booleana (guarda booleana) si se omite, se asume que es verdadera.
- C es una sentencia de comunicación, la cual puede ser de envío o recepción de mensajes.
- S es un conjunto de sentencias a ejecutar.
- B y C forman la guarda.

Las guardas pueden tener los siguientes estados:

- Éxito: si la condición B es verdadera y la ejecución C no causa demora (puede ejecutarse inmediatamente porque hay mensajes o la comunicación está lista).
- Falla: si la condición B es falsa.
- Bloqueo: si la condición B es verdadera pero la ejecución C causa demora (no puede ejecutarse inmediatamente)

Su utilidad principal es que soporta la comunicación no determinista entre procesos. Evita la limitación del pasaje de mensaje sincrónico, que obligan a un proceso a comunicarse en un orden fijo. Al aplicar comunicación guardada, un proceso puede interactuar con otros sin importar el orden en que los demás quieran comunicarse con él, evitando bloqueos en la comunicación.

La ejecución de sentencias de alternativa e iteración que contienen comunicaciones guardadas son

Alternativa:

1. Se evalúan todas las guardas.
2. Si todas fallan, la estructura finaliza sin ejecutar nada.
3. Si alguna tiene éxito, se elige una de forma no determinista.
4. Se ejecuta la comunicación C y luego S.

5. Termina la estructura.

Iteración:

1. Se evalúan todas las guardas
2. Si todas fallan, la estructura finaliza sin ejecutar nada.
3. Si alguna tiene éxito, se elige una de forma no determinista.
4. Se ejecuta la comunicación C y luego S.
5. Se vuelven a evaluar todas las guardas y se repite mientras al menos una guarda pueda tener éxito.
6. Finaliza cuando todas las guardas fallas.

Caso de bloqueo: si ninguna guarda tiene éxito pero alguna está bloqueada, el proceso espera hasta que alguna puede ejecutarse.

Puede producirse deadlock si todas las guardas quedan en bloqueo infinito.

Describa el concepto de sincronización barrier y cuál es su utilidad. ¿Qué es y cómo funciona una “butterfly barrier”? Ejemplifique gráficamente el funcionamiento de una “butterfly barrier” para 16 procesos.

El concepto de sincronización barrier (barrera) es un mecanismo utilizado para coordinar la ejecución de múltiples procesos en sistemas concurrentes y paralelos.

Una barrera es un punto del programa al que todos los procesos deben llegar antes de que cualquiera pueda continuar. Constituye una forma de sincronización por condición, donde la condición es que todos hayan arribado.

Su utilidad principal es coordinar el avance por fases o iteraciones, asegurando que ningún proceso avance a la siguiente etapa hasta que todos hayan completado la actual.

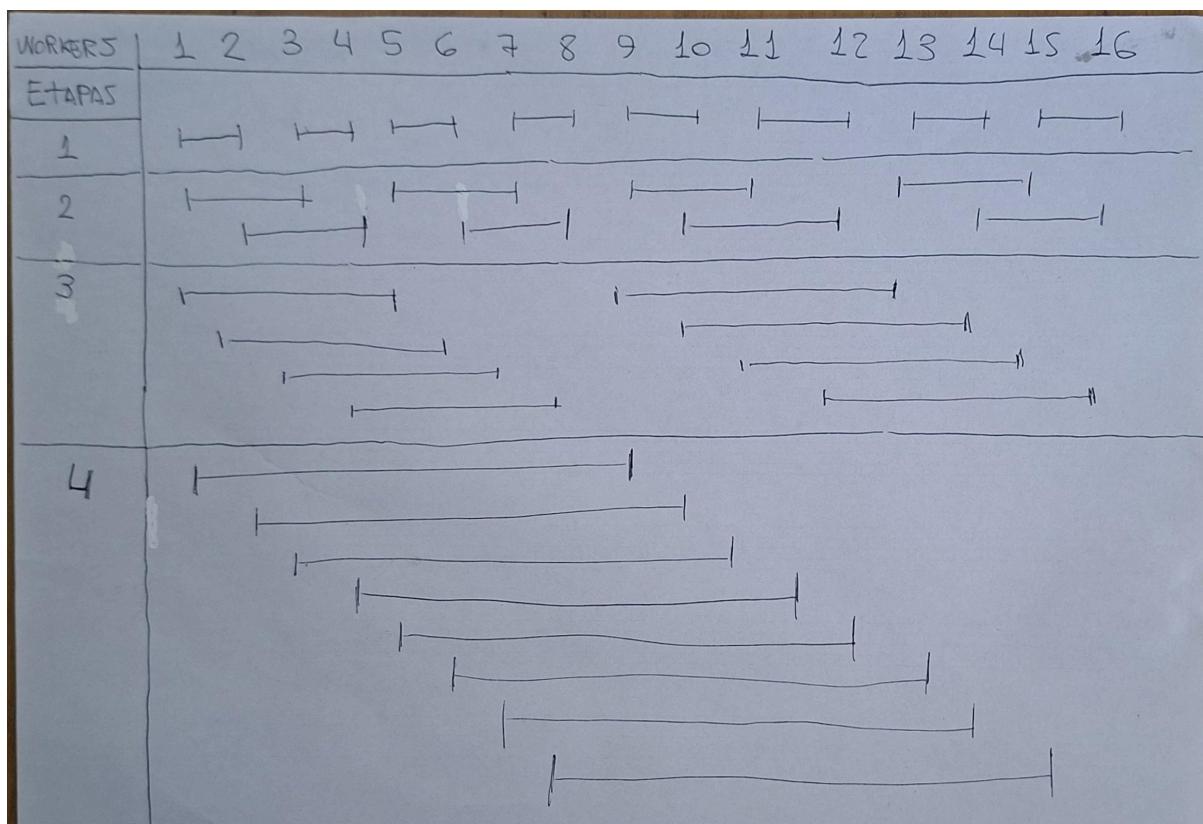
Esto evita inconsistencias y permite mantener el progreso sincronizado entre procesos que cooperan.

La butterfly barrier es un algoritmo diseñado para implementar una barrera simétrica entre N procesos (Siendo N potencias de 2). Su funcionamiento se basa en dividir la sincronización en $\log_2(N)$ etapas.

En cada etapa, cada proceso se sincroniza con un proceso distinto, determinado según un patrón que replica la estructura en una red.

Luego de completar las $\log_2(N)$ etapas, todos los procesos han quedado sincronizados directa o indirectamente con los demás, por lo que la barrera se cumple y pueden continuar su ejecución.

A diferencia de una barrera tradicional, que normalmente requiere un proceso coordinador (o juez) y cuyo tiempo de finalización crece linealmente con la cantidad de procesos ($O(N)$), la butterfly barrier elimina la necesidad de un coordinador central y reduce el tiempo total a $O(\log_2(N))$, logrando una sincronización más eficiente y escalable.



Defina los paradigmas de interacción entre procesos distribuidos heartbeat, servidores replicados y token passing. Marque ventajas y desventajas en cada uno de ellos cuando se utiliza comunicación por mensaje sincrónicos o asincrónicos.

Los paradigmas de interacción entre procesos son modelos que combinan los esquemas básicos (productor/consumidor, cliente/servidor y pares que interactúan) para organizar la comunicación y coordinación entre procesos.

El paradigma Heartbeat (latido) implica que los procesos deben intercambiar información periódicamente utilizando mecanismos de tipo send/receive. Es un modelo útil para paralelizar soluciones iterativas.

Se basa en el esquema de divide y vencerás donde la carga se distribuye entre los workers y cada uno actualiza una parte de los datos. Los nuevos valores que un worker calcula dependen de los valores mantenidos por él mismo o por sus vecinos inmediatos. Cada paso o iteración avanza hacia la solución.

Mensajes asincrónicos:

- Ventajas: el proceso que envía la información no se bloquea, lo que evita demoras innecesarias del emisor y permite a los procesos ejecutarse a su propia velocidad.
- Desventajas: La sincronización requerida para la barrera debe ser programada explícitamente mediante un receive bloqueante o busy waiting si fuera el caso.

Mensajes sincrónicos:

- Ventajas: el proceso que envía la información se bloquea hasta que el mensaje es recibido, lo que puede alinear la comunicación y la necesidad de sincronización del heartbeat.
- Desventajas: El bloqueo del emisor puede reducir el grado de concurrencias además de aumentar las probabilidades de deadlock.

Los servidores replicados se utilizan para manejar recursos compartidos a través de múltiples instancias. Los clientes tienen la ilusión de estar utilizando un único recurso a pesar de la existencia de varias instancias del mismo.

Mensajes asincrónicos:

- Ventajas: los procesos operan a su propia velocidad debido al buffering implícito.
- Desventajas: Para la comunicación bidireccional se requiere especificar múltiples canales, al menos uno de requerimiento y uno de puesta por cliente, lo que resulta no óptimo.

Mensajes sincrónicos:

- Ventajas: el cliente realiza una llamada que demora hasta obtener la respuesta del servidor, lo que se alinea con el requerimiento bidireccional.
- Desventajas: las posibilidades de deadlock son mayores en la comunicación sincrónica. Se necesita programar un proceso buffer si se requiere amortiguación de mensajes.

En el paradigma Token Passing, se utiliza un mensaje especial llamado “token” que puede otorgar permisos o recopilar información global en arquitecturas distribuidas. Este mecanismo es fundamental para la toma de decisiones distribuidas.

El token se usa, por ejemplo, para controlar la exclusión mutua distribuida para detectar la terminación en un cómputo distribuido. El proceso que posee el token tiene el permiso para acceder al recurso compartido.

Mensajes asincrónicos:

- Ventajas: el proceso que posee el token puede enviarlo a su sucesor y continuar inmediatamente con otras tareas ya que el send no es bloqueante.
- Desventajas: puede haber problemas si la transferencia requiere una confirmación estricta de que el siguiente proceso lo ha recibido y está listo para actuar, haciendo la coordinación más compleja de programar.

Mensajes sincrónicos:

- Ventajas: como el send es bloqueante, se asegura una sincronización estricta sobre la transferencia del token, confirmando que el receptor ha tomado el control.
- Desventajas: el bloqueo impide que el proceso que envía el token quede ocioso esperando la recepción, lo cual puede ser ineficiente si la única acción es pasar el token.

Defina las métricas de speedup y eficiencia. ¿Cuál es el significado de cada una de ellas (qué miden)? ¿Cuál es el rango de valores posibles de cada uno? Ejemplifíque.

Las métricas de speedup y eficiencia son nociones que se utilizan para determinar si una solución paralela tiene mejores prestaciones que su análogo secuencial.

El speedup evalúa el tiempo total de ejecución de un programa. Mide la ganancia en performance que se puede alcanzar con un algoritmo paralelo, se calcula como la razón entre el tiempo de una solución secuencial y el tiempo de una solución paralela.

$$S = \frac{T_1}{T_p}$$

T₁: es el tiempo que tarda en ejecutarse la mejor solución secuencial sobre la mejor máquina.

Tp: es el tiempo desde que el primer hilo comienza a ejecutarse hasta que termina el último en la solución paralela con P CPUs.

El rango de valores de S está entre 0 y el S óptimo.

- Speedup lineal ($S = P$): ocurre cuando S es igual a la cantidad de procesadores P. Significa que la arquitectura se está utilizando de forma perfecta.
- Speedup sublineal ($S < P$): ocurre cuando el speedup es menor al óptimo.
- Speedup superlineal ($S > P$): ocurre cuando el speedup es mayor al óptimo.

El speedup depende del número de procesadores, el tamaño de los datos y el algoritmo utilizado. Además, para todo algoritmo existe un speedup máximo alcanzable. el cual está postulado por la Ley de Amdahl.

La eficiencia es una métrica que permite conocer que tan bien aprovechará los procesadores extra un programa paralelo. Esta métrica permite independizarse de la arquitectura.

$$E = \frac{\text{Speedup}}{p}$$

o

$$E = \frac{S}{S_{\text{optimo}}}$$

Speedup: el resultado de la métrica anterior.

p: cantidad de procesadores.

El rango de valores de E está entre 0 y 1.

- Eficiencia perfecta ($E = 1$): corresponde al speedup perfecto. Significa que se está aprovechando perfectamente toda la capacidad de cómputo.
- Eficiencia menor a 1 ($E < 1$): indica que se está desaprovechando la capacidad de cómputo.

La eficiencia perfecta es teórica y nunca podrá ser alcanzada por diversos factores como el alto porcentaje de código secuencial, el desbalance de carga que produce esperas ociosas e algunos procesadores, la contención excesiva de memoria, el tamaño del problema que es pequeño o fijo y no crece con P, etc.

Definir escalabilidad en sistemas paralelos

La escalabilidad es una métrica utilizada para evaluar el rendimiento de un programa paralelo.

Un programa paralelo se considera escalable si su eficiencia se mantiene constante para un rango amplio de CPUs

- 3- Dado el siguiente bloque de código, indique para cada uno de los ítems si son equivalentes o no. Justificar cada caso (de ser necesario dar ejemplos).

	Segmento 1	Segmento 2
	<pre> ... int cant=1000; While (true) { IF (cant > 15); datos?(cant) → Sentencias1 □ (cant < 5); datos?(cant) → Sentencias2 □ (INCÓGNITA); datos?(cant) → Sentencias3 END IF ... </pre>	<pre> ... int cant=1000; DO (cant > 15); datos?(cant) → Sentencias1 □ (cant < 5); datos?(cant) → Sentencias2 □ (INCÓGNITA); datos?(cant) → Sentencias3 END DO ... </pre>
a)	INCÓGNITA equivale a: $(cant = 5) \text{ or } (cant = 15)$	
b)	INCÓGNITA equivale a: $(cant > 0)$	
c)	INCÓGNITA equivale a: $(cant \geq 2) \text{ and } (cant \leq 20)$	
d)	INCÓGNITA equivale a: $((cant > 5) \text{ and } (cant \leq 15))$	
e)	INCÓGNITA equivale a: $((cant > 5) \text{ and } (cant < 15))$	

Para que sean equivalentes se requiere que ninguna de las guardas del do falle para que se itere de manera infinita como sucede en el segmento 1, donde en el caso de que falle, vuelve a iterar por el while(true).

- Caso (cant = 5) or (cant = 15): no es equivalente ya que si el valor de cant = 6 el do no determinístico fallaría y continuaría la ejecución del programa, mientras que el if volvería a ejecutarse.
- Caso (cant > 0): es equivalente, ya que se cubren todos los valores posibles sin que falle.
- Caso ((cant >= 2) and (cant <= 20)): es equivalente, ya que se cubren todos los valores posibles sin que falle.
- Caso ((cant > 5) and (cant <= 15)): no es equivalente ya que si el valor de cant = 5 el do no determinístico fallaría y continuaría la ejecución del programa, mientras que el if volvería a ejecutarse.
- Caso ((cant > 5) and (cant < 15)): no es equivalente ya que si el valor de cant = 5 o cant = 15 el do no determinístico fallaría y continuaría la ejecución del programa, mientras que el if volvería a ejecutarse.

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

La técnica de passing the baton es un mecanismo utilizado para garantizar exclusión mutua y, al mismo tiempo, controlar con precisión cuál proceso bloqueado es el próximo en continuar su ejecución. Su utilidad principal es permitir implementar sincronización por condición arbitraria, ya que posibilita elegir exactamente qué proceso debe despertarse cuando una condición se vuelve verdadera.

El funcionamiento puede describirse así:

Mientras un proceso está dentro de su sección crítica, se dice que posee el bastón (baton). Esto significa que tiene permiso exclusivo para ejecutar. Cuando finaliza, el proceso no libera simplemente el recurso, sino que pasa explícitamente el bastón al proceso que corresponda. Si existe un proceso que está esperando por una condición que acaba de volverse verdadera, el bastón se le asigna directamente a él. Ese proceso ejecuta su sección crítica y, al finalizar, vuelve a pasar el bastón a otro proceso.

Si no hay procesos esperando, el bastón se entrega al siguiente proceso que intente ingresar a la sección crítica.

La técnica de passing the condition aparece en el contexto de monitores. Consiste en que un proceso “habilita” una condición (por ejemplo, actualizando un estado compartido) que otro proceso necesita para continuar. Es decir, un proceso pasa la condición necesaria para que otro proceso pueda ejecutar su guarda y avanzar.

Ambas técnicas comparten su propósito: imponer un orden preciso sobre qué proceso continúa su ejecución, especialmente cuando existen condiciones de sincronización.

La diferencia fundamental es el ámbito en el que se aplican:

- Passing the baton se utiliza en implementaciones basadas en semáforos, donde el control del próximo proceso se maneja de manera explícita.
- Passing the condition se utiliza en monitores, donde la habilitación de una condición permite que otro proceso retome su ejecución cuando la guarda se torna verdadera.

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 # Primera fila del strip
    int last = first + n/P - 1; # Última fila del strip to last
    for [i = first to last] {
        for [j = 1 to n] {
            c[i,j] = 0.0;
            for [k = 1 to n]
                c[i,j] = c[i,j] + a[i,k]*b[k,j];
        }
    }
}
```

Suponga n=512 y cada procesador capaz de ejecutar un proceso.
Calcular cuántas asignaciones, sumas y productos se hacen secuencialmente (caso en que P=1), y cuantas se realizan en cada procesador en la solución paralela P=8.

Secuencial:

First = 0
Last = 0 + 511

primer for de 0 a 511, o sea, 512 iteraciones
segundo for de 1 a 512, o sea, 512 iteraciones
tercer for de 1 a 512, o sea, 512 iteraciones

se van a hacer $512 \times 512 \times 512$ sumas y productos de manera secuencial 134.217.728
se van a hacer $512 \times 512 \times 513$ asignaciones 134.479.872

Paralelo:

First = $(w-1) * 64 + 1$
Last = First + 64 - 1

First	Resultado	Last	Resultado
$(1 - 1) * 64 + 1$	0	$0 + 63$	63
$(2 - 1) * 64 + 1$	65	$65 + 63$	128
$(3 - 1) * 64 + 1$	130	$130 + 63$	193

$(4 - 1) * 64 + 1$	195	$195 + 63$	258
$(5 - 1) * 64 + 1$	260	$260 + 63$	323
$(6 - 1) * 64 + 1$	325	$325 + 63$	388
$(7 - 1) * 64 + 1$	390	$390 + 63$	453
$(8 - 1) * 64 + 1$	455	$455 + 63$	518

por procesador

primer for de 0 a 63, o sea, 64 iteraciones

segundo for de 1 a 512, o sea, 512 iteraciones

tercer for de 1 a 512, o sea, 512 iteraciones

se van a hacer $64 * 512 * 512$ sumas y productos de manera paralela 16.777.216

se van a hacer $64 * 512 * 513$ asignaciones de manera paralela 16.809.984

Si los procesadores P1 a P7 son idénticos, con tiempos de asignación 1, de suma 2 y de producto 3, y si el procesador P8 es 3 veces más lento, calcule cuánto tarda el proceso total concurrente

Asignaciones P1 a P7

$$16.809.984 * 1 = 16809984$$

Sumas y productos P1 a P7

$$16.777.216 * 2 = 33554432 \rightarrow \text{Sumas}$$

$$16.777.216 * 3 = 50331648 \rightarrow \text{Producto}$$

Asignaciones P8

$$16.809.984 * 3 = 50429952$$

Sumas y productos P8

$$33554432 * 3 = 100663296 \rightarrow \text{Sumas}$$

$$50331648 * 3 = 150994944 \rightarrow \text{Producto}$$

En un sistema concurrente, el tiempo total está determinado por el procesador que más tarde en finalizar, es decir, el procesador P8 que tarda $50.429.952 + 100.663.296 + 150.994.944 = 302.088.192$ unidades de tiempo

¿Cuál es el valor del speedup?

$$T1 = (134.217.728 * 2 + 134.217.728 * 3) + 134.479.872 = 805.568.512$$

$$T8 = 302.088.192$$

$$\text{SpeedUp} = (805.568.512 / 302.088.192) = 2,666666667$$

¿Cómo modificaría el código para lograr un mejor speedup?

Si P8 no trabaja 3 veces más lento que los procesadores P1, P2, P3, P4 , P5, P6 y P7, se podría obtener un Speedup = 8, es decir, un speedup perfecto.

Sino, se podría dar menor cantidad de cálculos al procesador P8 para que tenga una carga de tareas más balanceada.

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

Describa conceptualmente las soluciones posibles con memoria distribuida para arquitecturas en estrella (centralizada), anillo circular, totalmente conectada y árbol. Nombré ventajas y desventajas.

Arquitectura en estrella.

Un proceso central actúa como el coordinador o centralpeer mientras que los N-1 procesos restantes actúan como peers.

1. Cada uno de los N-1 procesos envían su valor inicial al proceso central.
2. El proceso central recibe los N valores, calcula la suma total y el promedio.
3. Una vez calculado el promedio, el proceso central lo envía de vuelta a cada uno de los N-1 procesos.

Ventajas: es eficaz para la toma de decisiones centralizadas. El cálculo se realiza de forma secuencial en el nodo central, ya que toda la carga recae en él. El número de mensajes enviados es de $2(N-1)$.

Desventajas: el nodo central puede convertirse en un punto único de fallo o cuello de botella a medida que N aumenta.

Arquitectura anillo circular.

Los procesos forman un anillo, cada proceso se comunica únicamente con su sucesor.

El proceso $P[i]$ recibe mensajes de $P[i - 1]$ y envía mensajes a $P[i + 1]$, siendo $P[i - 1]$ sucesor de $P[i]$.

1. El proceso P inicia la primera iteración enviando su valor inicial a su sucesor.
2. Cada proceso recibe una suma parcial, agrega su valor y la reenvía.
3. Cuando la suma vuelve a P, este calcula el promedio.
4. P envía a su sucesor el promedio final, este lo almacena y lo reenvía a su sucesor hasta que todos conozcan el resultado.

Ventajas: permite el cómputo paralelo durante la agregación. El número total de mensajes es de $2(N - 1)$.

Desventajas: el tiempo de cómputo depende linealmente de N. Un fallo en un proceso rompe el anillo completo.

Arquitectura totalmente conectada

Cada proceso tiene un canal de comunicación directo con todos los demás. Todos los nodos tienen roles y responsabilidades similares, cada proceso opera de forma independiente para recopilar toda la información necesaria para el cálculo.

1. Cada proceso envía su valor local a los otros N-1 procesos a través de su enlace directo.
2. Todos los procesos reciben los valores locales de los demás por lo que se calcula el promedio de manera local, de manera independiente y en paralelo.

Ventajas: ofrece la mayor velocidad de comunicación ya que en un solo paso se resuelve el problema. Ofrece un alto grado de paralelismo de computo. Es descentralizada y tolerable a fallos.

Desventajas: El número de mensajes es $N(N-1)$ lo que implica un costo alto.

Arquitectura de árbol

Los procesos se organizan en una estructura jerárquica de tipo árbol. Se utiliza un patrón similar a una barrera de árbol combinado.

El proceso se divide en fases ascendentes y descendentes, distribuyendo la carga de la coordinación y el cómputo entre los procesos.

Fase ascendente:

1. Las hojas del árbol inicián el proceso enviando sus valores a sus padres.
2. Cada parente acumula las sumas parciales de sus hijos, agrega su valor y envía la suma hacia arriba.
3. Este proceso continúa recursivamente hacia arriba hasta que el proceso raíz recibe la suma total de todos los valores y el conteo total.

Fase descendente:

1. El nodo raíz calcula el promedio final.
2. Cada nodo distribuye el valor recibido a sus hijos hasta llegar a las hojas.

Ventajas: el rendimiento mejora en sistemas con gran número de procesos, ya que el tiempo de ejecución es proporcional a la altura del árbol.

Desventajas: Al tener gran cantidad de proceso requiere un diseño más elaborado que la centralizada.

Implemente todas las soluciones.

Arquitectura en estrella (centralizada).

```
channel values[n] (int)
channel average[n] (int)

Process Peer[i = 1 to n-1]
    int v = random(1..100)
    send values[i]()
    receive average[i](v)
End.

Process CentralPeer[0]
    int temp
    int sum = 0
    int calculated_average

    for (i = 1 to n-1)
        receive values[i](temp)
        sum = sum + temp
    end

    calculated_average = sum / (n - 1)

    for (i = 1 to n-1)
        send average[i](calculated_average)
    end
End.
```

Arquitectura anillo circular

```
channel values[n] (int, int)

Process Initial[0]
    int v = random(1..100)
    int sum, cant

    # Primera interacion
    send values[1] (v, 1)
    receive values[0] (sum, cant)

    calculated_average = sum / cant

    # segunda interacion
    send values[1] (calculated_average, cant)
End.

Process Peer[i = 1 to n-1]
    int v = random(1..100)
    int sum, cant

    receive values[i](sum, cant)

    sum = sum + v
    cant = cant + 1

    if (i < n - 1) then
        | send values[i + 1](sum, cant)
    else
        | send values[0](sum, cant)
    end
End.
```

Arquitectura totalmente conectada

```
channel values[n] (int)

Process Peer[i = 0 to n-1]
    int v = random(1..100)
    int sum = v
    int temp, calculated_average

    for (j = 0 to n-1 st j != i)
        | send values[j](v)
    end

    for (j = 0 to n-1 st j != i)
        | receive values[j](temp)
        | sum = sum + temp
    end

    calculated_average = sum / n
End.
```

Arquitectura de árbol

```
channel up[n](int, int)      # mensajes ascendentes (suma, cantidad)
channel down[n](real)       # mensajes descendentes (promedio)

Process Peer[i = 0 to N-1]
    int v = random(1..100)
    int sum = v
    int count_p = 1
    real avg

    list children = calculate_children(i, N)
    int parent = calculate_parent(i)

    # ===== FASE ASCENDENTE =====
    # Recibir contribuciones de los hijos
    for (child in children)
        int child_sum, child_count
        receive up[child](child_sum, child_count)
        sum = sum + child_sum
        count_p = count_p + child_count
    end

    if (parent != -1)
        # Enviar contribución al padre
        send up[i](sum, count_p)

        # Esperar el promedio desde arriba
        receive down[i](avg)
    else
        # Soy la raíz, calculo el promedio
        avg = (real) sum / count_p
    end

    # ===== FASE DESCENDENTE =====
    # Enviar promedio a los hijos
    for (child in children)
        send down[child](avg)
    end
End.
```

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

```

chan token[N] (bool)

process Worker[i = 0 to N-1]
    bool have_token = false

    if (i == 0) then
        # El primer proceso inicia con el token
        have_token = true
    end

    while (true)
        if (!have_token) then
            # Esperar el token
            receive token[i](have_token)
        end

        # -----
        # Sección crítica
        # -----

        if (i < n - 1) then
            send token[i+1](true)
        else
            send token[0](true)
        end

        have_token = false
    end
End.

```

Dado los siguientes 3 ejemplos de soluciones al problema de sección crítica, para cada uno indicar si es correcto, y en caso contrario indicar cual/es propiedad/es no cumple. ¿Por qué?

Ejemplo 1	Ejemplo 2	Ejemplo 3
<pre> bool aviso[100] = ([100] false) bool libre = true process Worker[id: 0..99] while (true) # SNC aviso[id] = true while (aviso[id]) skip # # SC # libre = true end End. process Admin int i = 0 while (true) while (true) while (not (aviso[i])) i = (i + 1) mod 100 aviso[i] = false libre = false end end End. </pre>	<pre> bool aviso[100] = ([100] false) bool libre = true process Worker[id: 0..99] while (true) # SNC aviso[id] = true while (aviso[id]) skip # # SC # libre = true end End. process Admin int i = 0 while (true) while (not (aviso[i])) i = (i + 1) mod 100 aviso[i] = false libre = false while (not (libre)) skip i = (i + 1) mod 100 end End. </pre>	<pre> bool aviso[100] = ([100] false) bool libre = true process Worker[id: 0..99] while (true) # SNC aviso[id] = true while (aviso[id]) skip # # SC # libre = true end End. process Admin int i = 0 while (true) while (not (aviso[i])) skip aviso[i] = false libre = false while (not (libre)) skip i = (i + 1) mod 100 end end End. </pre>

En el ejemplo 1 la solución no es correcta porque no cumple la propiedad de exclusión mutua porque el administrador puede habilitar a un proceso y antes de que salga de su sección crítica habilitar a otro, permitiendo que ambos estén dentro de la sección crítica simultáneamente. Tampoco se cumple la propiedad de ausencia de demoras innecesarias, porque si un proceso solicita la sección crítica el administrador no lo atiende hasta que su recorrido vuelva a su índice, generando esperas arbitrarias.

En el ejemplo 2 la solución si es correcta ya que cumple las 4 propiedades: ausencia de deadlocks, ausencia de demora innecesaria, eventual entrada y exclusión mutua.

En el ejemplo 3 la solución no es correcta porque no cumple la propiedad de ausencia de deadlock y por consiguiente no cumple la propiedad de eventual entrada. El administrador queda esperando que el proceso libere, mientras el proceso sigue esperando que su aviso cambie, ambos quedan en bucle infinito.

Explicar las 3 formas diferentes en que se pueden usar las sentencias AWAIT. Ejemplifíquese.

Existen tres formas diferentes de usar la sentencia await.

- **Acción atómica incondicional:** <S; >, se ejecutan las sentencias S de forma atómica, sin esperar ninguna condición previa, se utiliza cuando solo se necesita exclusión mutua.
 - Ejemplo: < x = x +1; y = y + 1; >
- **Acción atómica condicional sin cuerpo:** < await(B) >, el proceso se demora hasta que la condición booleana B sea verdadera, no ejecuta ninguna sentencia atómica adicional, se utiliza cuando se requiere sincronización por condición.
 - Ejemplo: < await(count > 0) >
- **Acción atómica condicional completa:** < await(B) S; >, si B es falsa, el proceso se bloquea, cuando B es verdadera, S se ejecuta de forma atómica, se utiliza cuando una operación implica esperar y luego realizar cambios coherentes en variables compartidas.
 - Ejemplo: < await(s > 0) s = s - 1; >

Cuál es la diferencia entre los semáforos generales (los usados en la práctica) y los binarios en cuanto a su funcionalidad. Como se define el funcionamiento de ambos por medio de las sentencias AWAIT.

La diferencia entre los semáforos generales y los semáforos binarios radica en el rango de valores que pueden tomar su contador interno y en su aplicación funcional.

Los semáforos generales son adecuados cuando los procesos compiten por recursos de múltiples unidades. Por ejemplo, para contar el número de unidades libres de un recurso.

Los semáforos binarios son ideales para implementar exclusión mutua. También para señalar procesos individuales, un valor de 1 indica que el recurso está disponible y 0 que está ocupado.

Implementación por medio de sentencias AWAIT:

- < await (s > 0) s = s -1 >

Explicar la diferencia entre los protocolos de señalización para monitores: signal and wait y signal and continue.

La diferencia entre ambos procesos de señalización es que proceso conserva el control del monitor en el momento de la señalización y cual ejecuta a continuación.

En signal and continue, el proceso que ejecuta la señalización (signal) mantiene el control del monitor y continúa su ejecución dentro de él. El proceso que es despertado debe esperar a que el monitor quede libre para poder ingresar nuevamente, compitiendo como cualquier otro proceso.

En signal and wait, el proceso que realiza el signal cede el control del monitor de forma inmediata. El proceso que fue despertado entra directamente al monitor y continua su ejecución desde el punto en el que había quedado bloqueado. El proceso que hizo el signal pasa a esperar para volver a entrar cuando el monitor quede disponible.

Indicar al menos 2 diferencias entre ambos PMA y PMS (relacionado con su funcionalidad, no con su sintaxis). Indicar la principal ventaja de PMS sobre PMA. Indicar cual es LA PRINCIPAL característica común de RPC y Rendezvous que lo diferencian de los Pasajes de Mensajes.

La comunicación mediante pasaje de mensajes se clasifica en asincrónica (PMA) y sincrónica (PMS), diferenciándose principalmente en cómo manejan el envío y el flujo de control del proceso emisor.

En PMA, el envío es asincrónico y no bloqueante: una vez que el proceso emisor envía el mensaje, continúa su ejecución sin esperar al receptor. Los canales funcionan como una cola de mensajes enviados pero aún no recibidos, lo que implica la existencia de buffering implícito. Esto permite que los procesos avancen a su propio ritmo sin necesidad de estar sincronizados temporalmente.

En PMS, el envío es sincrónico y bloqueante: después de enviar un mensaje, el proceso emisor se detiene hasta que el receptor lo recibe. Esto reduce el grado de concurrencia en comparación con PMA, ya que el emisor queda obligado a esperar. Además, la cola asociada a un send se reduce a un solo mensaje; si se necesitara buffering adicional, se debe implementar explícitamente mediante un proceso intermediario.

La principal ventaja de PMS sobre PMA es la reducción del uso de memoria: al no existir colas largas de mensajes pendientes, los canales consumen menos espacio y el sistema evita acumulaciones de mensajes.

La principal característica común entre RPC y Rendezvous, que los diferencia de las primitivas estándar de pasaje de mensajes, es que combinan una invocación de operación estructurada (similares a los monitores) con comunicación sincrónica bidireccional. Tanto RPC como Rendezvous integran en una única operación el envío de la solicitud y la recepción de la respuesta, lo que los vuelve mecanismos ideales para modelos cliente/servidor y los distingue de los pasajes de mensajes unidireccionales tradicionales.

Sea la siguiente solución a un problema de matrices de nxn con P procesos en paralelo con variables compartidas. Suponga n = 1000 y cada procesador capaz de ejecutar un proceso. Nota: para hacer los cálculos solo tenga en cuenta las operaciones realizadas en las instrucciones dentro del for Y.

int a[n,n], b[n,n]; – Datos cargados

```

int d[n,n], e[n,n];

process worker [w: 0..P-1] {
    int x, y;
    int primera = w * (n/P);
    int ultima = primera + (n/P) - 1;

    for (x = primera; x <= ultima; x++) {
        for (y = 0; y < n; y++) {
            d[x,y] = a[x,y] + b[x,y];
            e[x,y] = a[x,y] * b[x,y];
        };
    };
}

```

Calcular cuántas asignaciones, sumas y productos se hacen secuencialmente (caso en que P=1) y cuantas se realizan en cada procesador en la solución paralela con P=10.

Solución secuencial

$$\begin{aligned} \text{primera} &= 0 * (1000/1) = 0 \\ \text{ultima} &= 0 + (1000/1) - 1 = 999 \end{aligned}$$

Primer for ejecuta desde 0 hasta = 999 es decir 1000 iteraciones
 Segundo for ejecuta desde 0 hasta 1000 es decir 1000 iteraciones

Se hacen $1000 * 1000 * 2 = 2,000,000$ asignaciones
 Se hacen $1000 * 1000 * 1 = 1,000,000$ sumas
 Se hacen $1000 * 1000 * 1 = 1,000,000$ productos

Solución paralela

Primera	Resultado	Última	Resultado
$0 * (1000/10)$	0	$0 + (1000/10) - 1$	99
$1 * (1000/10)$	100	$100 + (1000/10) - 1$	199
$2 * (1000/10)$	200	$200 + (1000/10) - 1$	299
$3 * (1000/10)$	300	$300 + (1000/10) - 1$	399
$4 * (1000/10)$	400	$400 + (1000/10) - 1$	499
$5 * (1000/10)$	500	$500 + (1000/10) - 1$	599
$6 * (1000/10)$	600	$600 + (1000/10) - 1$	699

$7 * (1000/10)$	700	$700 + (1000/10) - 1$	799
$8 * (1000/10)$	800	$800 + (1000/10) - 1$	899
$9 * (1000/10)$	900	$900 + (1000/10) - 1$	999

Primer for ejecuta desde 0 hasta = 99 es decir 100 iteraciones

Segundo for ejecuta desde 0 hasta 1000 es decir 1000 iteraciones

Se hacen $100 * 1000 * 2 = 200,000$ asignaciones

Se hacen $100 * 1000 * 1 = 100,000$ sumas

Se hacen $100 * 1000 * 1 = 100,000$ productos

Dado que los procesadores de P1 A P4 son idénticos, con tiempos de asignación 2, de suma 4 y de producto 8; los procesadores de P5 a P8 son la mitad de potentes que los anteriores (tiempos de 4, 8 y 16 para asignaciones, sumas y productos respectivamente); y los procesadores P9 y P10 son el doble de potentes que los primeros (tiempo 1, 2 y 4 para asignaciones, sumas y productos respectivamente). Calcular cuánto tarda el programa paralelo y secuencial.

Programa secuencial

(Tomó los tiempos de asignación, suma y producto de P1 a P4)

Se hacen $1000 * 1000 * 2 * 2 = 4,000,000$ unidades de tiempo en asignaciones

Se hacen $1000 * 1000 * 1 * 4 = 4,000,000$ unidades de tiempo en sumas

Se hacen $1000 * 1000 * 1 * 8 = 8,000,000$ unidades de tiempo en productos

Tiempo total $4,000,000 + 4,000,000 + 8,000,000 = 16,000,000$ unidades de tiempo

Programa paralelo

P1 a P4

Se hacen $100 * 1000 * 2 * 2 = 400,000$ unidades de tiempo en asignaciones

Se hacen $100 * 1000 * 1 * 4 = 400,000$ unidades de tiempo en sumas

Se hacen $100 * 1000 * 1 * 8 = 800,000$ unidades de tiempo en productos

Tiempo total por procesador $400,000 + 400,000 + 800,000 = 1,600,000$ unidades de tiempo

P5 a P8

Se hacen $100 * 1000 * 2 * 4 = 800,000$ unidades de tiempo en asignaciones

Se hacen $100 * 1000 * 1 * 8 = 800,000$ unidades de tiempo en sumas

Se hacen $100 * 1000 * 1 * 16 = 1,600,000$ unidades de tiempo en productos

Tiempo total por procesador $800,000 + 800,000 + 1,600,000 = 3,200,000$ unidades de tiempo

P9 a P10

Se hacen $100 * 1000 * 2 * 1 = 200,000$ unidades de tiempo en asignaciones

Se hacen $100 * 1000 * 1 * 2 = 200,000$ unidades de tiempo en sumas

Se hacen $100 * 1000 * 1 * 4 = 400,000$ unidades de tiempo en productos

Tiempo total por procesador $200,000 + 200,000 + 400,000 = 800,000$ unidades de tiempo

El programa paralelo tardará 3,200,000 unidades de tiempo ya que se toma el tiempo del procesador que más tarda

Realizar paso a paso el cálculo del valor del speedup y la eficiencia

$$T_1 = 16,000,000$$

$$P = 3,200,000$$

$$\text{Speedup} = (16,000,000 / 3,200,000) = 5$$

Esto quiere decir que la solución paralela es 5 veces más rápida que la secuencial

$$\text{Eficiencia} = (5/10) = 0,5$$

Esto quiere decir que la solución paralela utiliza solo el 50% de la capacidad total teórica de los procesadores

Explicar detalladamente cómo modificaría la solución para lograr una mejor eficiencia. Recalculando la misma.

Para mejorar la eficiencia de la solución hay que distribuir la carga de trabajo por procesador, según su potencia relativa, es decir, los procesadores que más tarde, ejecutarán menos operaciones y los que menos tarden, ejecutarán más operaciones.

P1-P4: Potencia relativa 1 (normales)

P5-P8: Potencia relativa 0.5 (mitad de rápidos)

P9-P10: Potencia relativa 2 (doble de rápido)

En la solución se reparten 100 filas por proceso, ahora reparto en base a la potencia relativa

P1-P4: $1 * 100 = 100$ filas cada uno

P5-P8: $0,5 * 100 = 50$ filas cada uno

P9-P10 $2 * 100 = 200$ filas cada uno

De esta manera se consigue un nuevo tiempo total del programa paralelo, ahora todos los workers tardan 1,600,000

$$\text{Speedup} = (16,000,000 / 1,600,000) = 10$$

$$\text{Eficiencia} = (10/10) = 1$$

Suponga que la solución a un problema es paralelizada sobre P procesadores de dos maneras diferentes. En un caso, el speedup (S) está regido por la función $S=P-8$ y en el otro por la función $S=P/3$. ¿Cuál de las dos soluciones se comportará más eficientemente al crecer la cantidad de procesadores? Justifique claramente.

P = 10

$$S_1 = 10 - 8 = 2$$

$$S_2 = 10/3 = 3.34$$

$$E_1 = (2/10) = 0.2$$

$$E_2 = (3.34/10) = 0.334$$

P = 100

$$S_1 = 100 - 8 = 92$$

$$S_2 = 100/3 = 33,34$$

$$E_1 = (92/100) = 0.92$$

$$E_2 = (33.34/100) = 0.3334$$

Al crecer la cantidad de procesadores, la solución S1 tiende a tener una eficiencia alta, casi ideal $E_1 \rightarrow 1$, mientras que E2 tiende a valores chicos. Por lo tanto, la mejor solución paralela va a ser S1 por su grado de eficiencia.

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. ¿Cuál es el límite en la mejora que puede obtenerse paralelizando el algoritmo? Justifique claramente.

El 90% del algoritmo es paralelizable, mientras que el 10% restante no. Para determinar el límite en la mejora que puede obtenerse paralelizando el algoritmo, utilizamos la ley de Amdahl, con la fórmula $Límite = 1/(1 - Paralelizable)$ siendo "paralelizable" la fracción del tiempo de ejecución que corresponde a código que puede ejecutarse en paralelo.

Paralelizable = 0.9

No paralelizable = 0.1

$$Límite = 1/(1 - 0.9) = 10$$

Esto significa que, aun utilizando una cantidad infinita de procesadores, el algoritmo no puede acelerarse más allá de un factor de 10, porque el 10% secuencial actúa como cuello de botella.

Suponga una ciudad representada por una matriz A(nxn). De cada esquina x,y se conocen dos valores enteros que representan la cantidad de autos y motos que cruzaron en la última hora. Los valores de cada esquina son mantenidos por un proceso distinto P(x,y). Cada proceso puede comunicarse con sus vecinos izquierdo, derecho, arriba y abajo, y también con los de las 4 diagonales (los procesos de las esquinas tienen sólo 3 vecinos, y los otros en los bordes de la grilla tienen 5 vecinos).

Escriba un algoritmo Heartbeat que calcule las esquinas donde cruzaron la mayor cantidad de autos y la menor cantidad de motos respectivamente, de forma que al terminar el programa, cada proceso conozca ambos valores. Nota: Indicar qué tipo de pasajes de mensajes se va a utilizar. Justificar la elección.

Se utiliza pasaje de mensajes sincrónico porque no se necesita buffering: cada envío bloquea hasta que el vecino recibe, asegurando que todos los procesos intercambien información de manera consistente en cada ronda del algoritmo Heartbeat. Esto evita acumulación no deseada de mensajes, garantiza que los procesos no avancen a la siguiente ronda sin haber recibido todos los valores de sus vecinos y mantiene la coherencia ronda por ronda, cosa que no estaría garantizada con PMA.

```
chan valores[N,N] (int)(int)

Process P[x: 0..N-1, y: 0..N-1]
| int maxAutos = autos[x,y]
| int minMotos = motos[x,y]

| int rondas = 2 * (N - 1)
| int a, m

For [k = 1 to rondas] {
    | // Enviar mis valores a todos los vecinos
    | For [p = 1 to vecinos.count] {
    |     | send valores[vecinos[p].fila, vecinos[p].columna] (maxAutos, minMotos)
    | }

    | // Recibir los valores de todos los vecinos
    | For [p = 1 to vecinos.count] {
    |     | receive valores[x,y] (a, m)

    |         | if (a > maxAutos)
    |             |     maxAutos = a

    |         | if (m < minMotos)
    |             |     minMotos = m
    |     }
    }

}

End.
```

Analizar desde el punto de vista de la cantidad de mensajes.

El total de mensajes será aproximadamente $16 * N$ elevado a la 3

Analizar cómo podría mejorarse la cantidad de mensajes.

La cantidad de mensajes puede reducirse evitando enviar valores que no cambiaron. Si un proceso en una ronda recibe datos pero su máximo y mínimo siguen siendo los mismos, entonces no tiene sentido que vuelva a enviarlos, porque estaría reenviando información repetida.

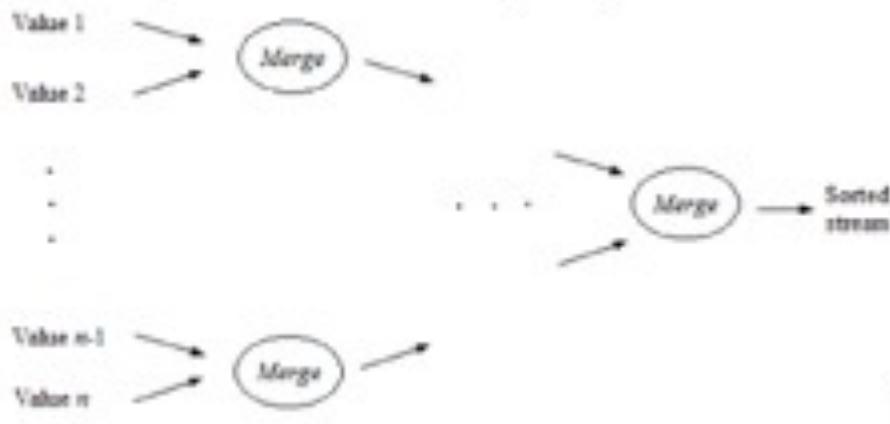
Así, solo se envían mensajes cuando hay una actualización real, y eso disminuye mucho el total de mensajes del Heartbeat.

Analizar qué pasaría si no existieran las diagonales.

Al tener menos vecinos, la información tarda más en propagarse por toda la matriz porque ahora solo puede viajar en cuatro direcciones. Esto obliga al algoritmo a usar más rondas para que los valores viajen de un extremo al otro. Aunque cada ronda tiene menos mensajes, el número total de mensajes enviados es mayor, y la convergencia se vuelve más lenta.

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

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



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

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

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

En el pipeline se necesitan N procesos, porque cada filtro conserva un único valor mínimo entre los que recibe y pasa los demás. Para obtener N valores ordenados al final, debe haber N filtros, uno por cada posición en la secuencia resultante.

En la red de procesos filtros que aparece en la imagen, los valores se combinan y ordenan progresivamente siguiendo una estructura de árbol binario.

Por cada nivel del árbol, se reduce a la mitad la cantidad de valores hasta llegar al nodo raíz. Por lo que, la red necesita $2 \log_2(N) - 1$ procesos.

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

En un pipeline, la información fluye secuencialmente a través de una serie de procesos. El primer filtro recibe los N valores, y cada filtro i -ésimo retiene el mínimo de los valores que recibe y pasa el resto a su sucesor.

Por esto, la cantidad total de mensajes enviados entre filtros es:

$$\frac{N(N - 1)}{2}$$

porque el primer filtro envía $N - 1$ valores, el segundo $N - 2$, y así sucesivamente.

En una red de procesos filtro, la estructura se organiza como un árbol binario.

La cantidad de mensajes está relacionada con el número de datos y la altura del árbol (que es $\log_2(N)$)

Cada valor atraviesa todos los niveles del árbol, por lo que la cantidad total de mensajes es:

$$N \log_2(N)$$

En **Odd/Even Exchange Sort**, en cada ronda ciertos pares de procesos intercambian valores.

En el caso general con k procesos y n valores distribuidos, pueden enviarse hasta:

$$O\left(k^2 \cdot \frac{n}{k}\right)$$

mensajes en el peor caso.

Pero en el caso habitual donde $k = n$ (es decir, un proceso por valor), el algoritmo necesita N rondas, y en cada ronda se envían aproximadamente N mensajes, por lo que:

$$N \cdot N = N^2$$

mensajes totales.

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

En un pipeline de filtros, los mensajes pueden enviarse en paralelo solo cuando el pipeline se llena. En ese momento, cada filtro trabaja simultáneamente procesando un valor distinto al mismo tiempo que su vecino, de modo que existen varios mensajes circulando en paralelo. Sin embargo, el recorrido de un mismo dato dentro del pipeline sigue siendo estrictamente secuencial, porque debe pasar por cada filtro en orden, uno tras otro. El

parallelismo se da entre valores diferentes, mientras que el procesamiento interno de un valor siempre requiere una secuencia estricta de envíos.

En la red de procesos filtro (o merge network), la estructura está organizada por niveles en forma de árbol: todos los procesos dentro de un mismo nivel pueden enviar mensajes al mismo tiempo, lo que permite un parallelismo muy marcado dentro de cada capa del árbol. El avance entre niveles no puede hacerse en paralelo: un nodo solamente puede comunicarse con el nivel siguiente cuando ya recibió y procesó toda la información proveniente del nivel anterior. La ejecución ocurre en fases, cada una paralela, pero las fases deben desarrollarse de manera secuencial.

En el caso del Odd/Even Exchange Sort, el algoritmo también permite comunicaciones paralelas, pero por pares. Durante cada ronda, varios pares de procesos intercambian valores simultáneamente (por ejemplo, en las rondas impares interactúan los procesos 1–2, 3–4, 5–6, etc., todos al mismo tiempo). Sin embargo, dentro de cada uno de esos pares el intercambio es necesariamente secuencial, porque el proceso debe recibir el valor del vecino, compararlo y eventualmente actualizarlo antes de seguir. Además, las rondas mismas no pueden ejecutarse en paralelo: cada ronda depende de que todos los intercambios de la ronda anterior hayan finalizado, por lo que el algoritmo avanza estrictamente ronda por ronda.

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

En un pipeline de filtros, la estructura obliga a que los datos avancen linealmente de un proceso al siguiente. En el peor caso, el primer dato atraviesa $n-1$ filtros, el segundo $n-2$, y así sucesivamente, hasta que el último ya no se mueve. La suma de todos esos desplazamientos es $n(n-1)/2$, y como cada paso implica una comparación y un eventual envío, el tiempo total resulta ser:

$$T = n(n-1) \text{ UTs.}$$

En una red de procesos filtro organizada como Sort-Merge Network, las comunicaciones se estructuran en niveles cuya profundidad es $\log_2(n)$. Cada nivel realiza comparaciones y envíos que involucran a todos los procesos, y por eso el costo global es proporcional a n en cada altura del árbol. Como nuevamente cada operación consume una unidad de tiempo y se contabilizan tanto comparaciones como envíos, el tiempo se expresa como:

$$T = 2n \log_2(n) \text{ UTs.}$$

En el Odd/Even Exchange Sort, el funcionamiento por rondas impone un límite de n rondas en el peor caso, aun cuando dentro de cada ronda todas las parejas trabajen en paralelo. La cantidad total de comparaciones y envíos acumulados a lo largo de todas las rondas forma un patrón cúbico que, al sumar todos los mensajes involucrados, produce un costo total equivalente a $(n+1)n^2/2$. Dado que cada uno de esos eventos insume una unidad de tiempo, el tiempo de ejecución final queda determinado por:

$$T = [(n+1)/2] \cdot n^3 \text{ UTs.}$$