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

Segmento 1

```
...
int cant=1000;
DO  (cant < -10);  datos?(cant) → Sentencias1
  □ (cant > 10);  datos?(cant) → Sentencias2
  □ (INCOGNITA);  datos?(cant) → Sentencias3
END DO
...
```

Segmento 2

```
...
int cant=1000;
While (true)
{ IF  (cant < -10);  datos?(cant) → Sentencias1
    □ (cant > 10);  datos?(cant) →Sentencias2
    □ (INCOGNITA);  datos?(cant) → Sentencias3
  END IF
}
...
```

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 reallzando.

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 = $16 * 128 + 16 * 128^2 = 2048 + 262144 = 264192$
Sumas = $16 * 128^2 = 262144$
Productos = $16 * 128^2 = 262144$

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   B¹ ; C¹ → S¹ ;
     B² ; C² → s² ;
fi
```

```
do B¹ ; C¹ → S¹ ;
     B² ; C² → s² ;
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.