Programación Concurrente 2019

Clase 10

Facultad de Informática
UNLP

Resumen de la clase anterior

Bolsa de tareas

RPC – Java (RMI)

Rendezvous – ADA

Primitivas múltiples

Paradigmas para la interacción entre procesos

3 esquemas básicos de interacción e/ procesos: productor / consumidor, cliente / servidor e interacción entre pares.

Se pueden combinar de muchas maneras, dando lugar a **7 paradigmas** o modelos de interacción entre procesos.

Paradigma 1: algoritmos heartbeat

Los procesos periódicamente deben intercambiar información con mecanismos tipo send/receive.

Ejemplo: Modelos biológicos / Problema de los N Cuerpos / Modelos de simulación paramétrica

Paradigmas para la interacción entre procesos

Paradigma 2: algoritmos pipeline

La información recorre una serie de procesos utilizando alguna forma de receive/send.

Se supone una arquitectura de procesos/procesadores donde la salida de uno es entrada del siguiente.

Los procesadores pueden distribuirse DATOS o FUNCIONES.

Ejemplo: Redes de Filtros, Tratamiento de Imágenes.

Paradigma 3: token passing

En muchos casos la arquitectura distribuida recibe una información global a través del viaje de tokens de control o datos.

La arquitectura puede ser un anillo, caso en el cual el manejo se asemeja a un pipeline, pero también puede ser cualquier topología (tipo objetos distribuidos).

Los tokens normalmente habilitan el control para la toma de decisiones distribuídas.

Paradigmas para la interacción entre procesos

Paradigma 4: servidores replicados

Los servidores manejan (mediante múltiples instancias) recursos compartidos tales como dispositivos o archivos.

Ej: File servers con múltiples clientes y una instancia de servidor por cliente (o por File, o por Unidad de almacenamiento)

Paradigma 5: probes (send) y echoes(receive)

La interacción entre los procesos permite recorrer grafos o árboles (o estructuras dinámicas) diseminando y juntando información.

Un ejemplo clásico es recuperar la topología activa de una red móvil o hacer un broadcast desde un nodo cuando no se “alcanzan” o “ven” directamente todos los destinatarios.

Paradigmas para la interacción entre procesos

Paradigma 6: algoritmos broadcast

Permiten alcanzar una información global en una arquitectura distribuida. Sirven para toma de decisiones descentralizadas.

En general en sistemas distribuidos con múltiples procesadores, las comunicaciones colectivas representan un costo crítico en tiempo.

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

Paradigma 7: manager/workers

Implementación distribuida del modelo de *bag of tasks*.

Un procesador controla los datos y/o procesos a ejecutarse y múltiples procesadores acceden a él para acceder a datos/procesos y ejecutar las tareas distribuidas.

Algoritmos heartbeat

Paradigma *heartbeat* $\Rightarrow$ útil para soluciones iterativas que se quieren paralelizar.

Usando un esquema "*divide & conquer*" se distribuye la carga e/ los workers; c/u es responsable de actualizar una parte.
Los nuevos valores dependen de los mantenidos x los workers o sus vecinos inmediatos
Cada "paso" debiera significar un progreso hacia la solución

Ej: grid computations (imágenes o PDE), autómatas celulares (simulación de fenómenos como incendios o crecimiento biológico)

```
process worker [i =1 to numWorkers]   {
    declaraciones e inicializaciones locales;
    while (no terminado)   {
        send valores a los workers vecinos;
        receive valores de los workers vecinos;
        Actualizar valores locales;       }
}
```

Algoritmos heartbeat. Labeling de regiones en imágenes

• Si tenemos una imagen representada en una matriz Imagen[mxn], en muchos casos es importante separarla en zonas que se identifiquen con un label (ej: conteo de células, reconocimiento tumores).

• Un modo natural de paralelizar este tipo de aplicaciones es dividir la Imagen en P “strips” o conjunto de filas en las que un proceso trata de poner un label a cada pixel.

• En c/ ciclo los resultados deben ser trasmitidos entre los procesos vecinos porque las “zonas” pueden ser compartidas. Esto establece una especie de “barrera” entre los workers vecinos.

• Las iteraciones pueden ser fijas o tener un proceso coordinador o manager que detecte cuando ninguno de los P procesos workers encontró cambios en el labeling.

•El resultado final es la imagen descompuesta en K “regiones” cada una con un label. Luego se puede avanzar en el tratamiento de cada región

Algoritmos heartbeat. Labeling de regiones en imágenes

```
chan first[1:P](int edge[n]);     # for exchanging edges
chan second[1:P](int edge[n]);
chan answer[1:P](bool);           # for termination check

process Worker[w = 1 to P] {
  int stripSize = m/W;
  int image[stripSize+2,n];     # local values plus edges
  int label[stripSize+2,n];     #   from neighbors
  int change = true;
  initialize image[1:stripSize,*] and label[1:stripSize,*];

  # exchange edges of image with neighbors
  if (w != 1)
    send first[w-1](image[1,*]);        # to worker above
  if (w != P)
    send second[w+1](image[stripSize,*]);   # to below
  if (w != P)
    receive first[w](image[stripSize+1,*]); # from below
  if (w != 1)
    receive second[w](image[0,*]);    # from worker above

  while (change) {
    exchange edges of label with neighbors, as above;
    update label[1:stripSize,*] and set change to true if
      the value of the label changes;
    send result(change);            # tell coordinator
    receive answer[w](change);      # and get back answer
  }
}
```

Figure 9.3 (a) Region labeling: Worker processes.

Algoritmos heartbeat. Labeling de regiones en imágenes

```
chan result(bool);  # for results from workers

process Coordinator {
  bool chg, change = true;
  while (change) {
    change = false;
    # see if there has been a change in any strip
    for [i = 1 to P] {
      receive result(chg);
      change = change or chg;
    }
    # broadcast answer to every worker
    for [i = 1 to P]
      send answer[i](change);
  }
}
```

Figure 9.3 (b) Region labeling: Coordinator process.

Algoritmos heartbeat. Autómatas celulares: el Juego de la Vida

Muchas aplicaciones de sistemas físicos o biológicos pueden modelizarse como una colección de cuerpos que repetidamente interactúan y evolucionan en el tiempo.

Algunos fenómenos pueden modelizarse con algún tipo de autómata celular. La idea es dividir el espacio en un conjunto de células donde c/u es un autómata (máquina de estado) finito

Cada transición de estado de una célula se basa en su estado y el de sus células vecinas.

Un ejemplo es *el Juego de la Vida*: grilla bidimensional donde c/ nodo es una célula que puede estar viva o muerta. Cada célula (excepto en los bordes) tiene 8 vecinas que influyen en ella.

Algoritmos heartbeat. Autómatas celulares: el juego de la vida

Primero el tablero es inicializado al azar.
Luego, c/ célula examina su estado y el de sus vecinos, y realiza una transición de estado de acuerdo a las siguientes reglas:

1- Una célula viva rodeada por 0 o 1 células vivas, MUERE de soledad.
2- Una célula viva rodeada por 2 o 3 células vivas, SOBREVIVE para otra generación.
3- Una célula viva rodeada por 4 o más células vivas, MUERE por superpoblación.
4- Una célula muerta rodeada por exactamente 3 células vivas, REVIVE.

⇨ Los procesos (células) interactúan con un paradigma tipo heartbeat, enviando y recibiendo mensajes en cada ciclo.
⇨ El “juego” dura un número predeterminado de ciclos, por lo que no haría falta coordinador.

Algoritmos heartbeat. Autómatas celulares: el juego de la vida

Sin analizar bordes y extremos, suponiendo 1 proceso Celula[i,j] por célula y considerando *send* no bloqueante tenemos:

```
chan intercambio[1:n, 1:n] (INT fila, columna, estado);
process celula [I = 1 to n, j = 1 to n]   {
    int estado;     # inicializado en viva o muerta
    declaraciones de variables;
    for [k =1 to cantGeneraciones]    {
            # intercambiar info con los 8 vecinos
            for [p = i-1 to i+1, q = j-1 to j+1]
                if p <> q send intercambio[p,q]  (i, j, estado);
            for [p =1 to 8]  {
                receive intercambio[i,j] (fila, columna, valor);
                guardar el valor de las células vecinas;
              }
            Actualizar el estado local usando las reglas;
        }
}
```

Paradigma de pipelining

Un pipeline es un arreglo lineal de procesos “filtro” que reciben datos de un puerto (canal) de E/ y entregan resultados por un canal de S/.

Estos procesos (“workers”) pueden estar en procesadores que operan en paralelo, en un primer esquema *a lazo abierto* (W_1 en el INPUT, W_n en el OUTPUT).

Un segundo esquema es el pipeline *circular*, donde W_n se conecta con W_1. Estos esquemas sirven en procesos iterativos o bien donde la aplicación no se resuelve en una pasada por el pipe.

En un tercer esquema posible (*cerrado*), existe un proceso coordinador que maneja la “realimentación” entre W_n y W_1.

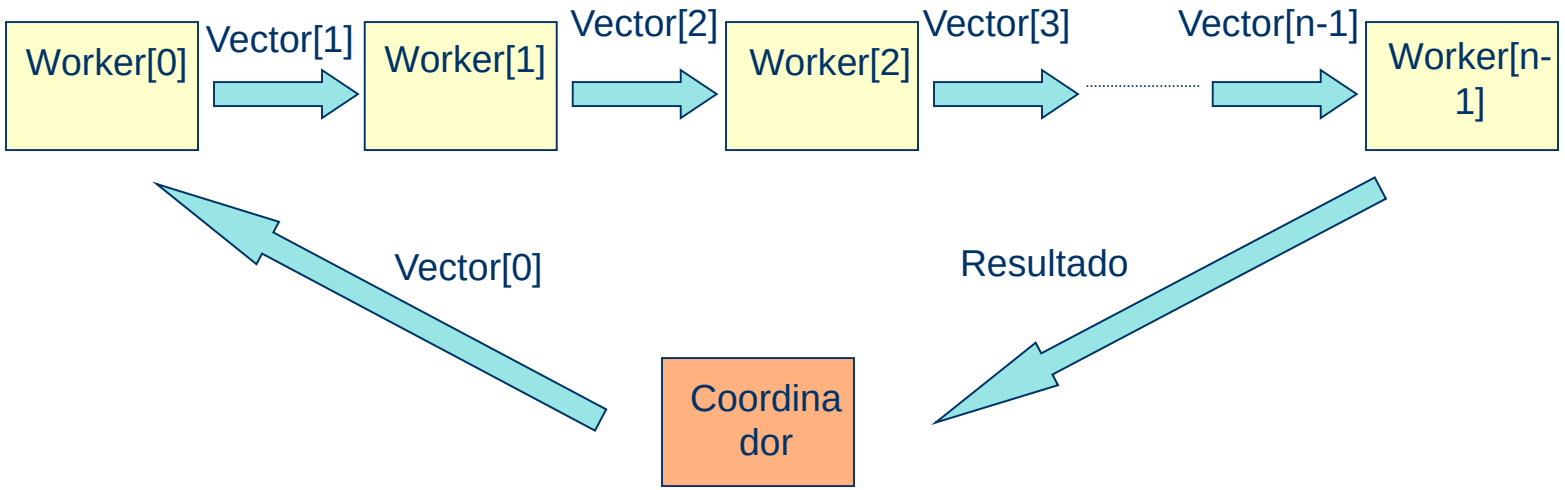
Paradigma de pipelining. Multiplicación de matrices

La multiplicación distribuida de matrices A y B de nxn usando un pipe no parece una aplicación “natural”, pero puede resolverse con un *pipe cerrado* con coordinador. El coordinador pasa las filas de A y luego las columnas de B por el canal de E/ al Worker 0, y recibe en orden inverso las filas del producto C desde el último worker.

C/ Worker ejecuta en tres fases: 1ro recibe las filas de A, guardando la 1ra y pasando las restantes. Así, Wi tendrá la fila i de A.
En la 2da fase realiza n ciclos de recibir una columna de B, pasarla y calcular un producto interno. Luego de n ciclos, Wi tiene C[i, *].
En la 3ra fase todas las filas de C fluyen entre los Workers, terminando en el coordinador.

La solución es interesante por el gran flujo de mensajes (casi continuo) en el pipe, y por la importancia del orden en que se envían los datos iniciales y finales.

Paradigma de pipelining. Multiplicación de matrices



Qué tipo de mensajes es más adecuado utilizar? Por qué?

Paradigma de pipelining. Multiplicación de matrices: Coordinador

```
chan vector[n] (double v[n]);   # mensajes a los Workers
chan resultado (double v[n]);   # filas de C para el coordinador

process Coordinador {
    double A[n,n], B[n,n], C[n,n];
    Inicializar A y B;

    for [i=0 to n-1]
        send vector[0] (A[i, *]);          # envía las filas de A
    for [i=0 to n-1]
        send vector[0] (B[*, i]);          # envía las columnas de B
    for [i=n-1 to 0]
        receive resultado (C[i, *]);       # recibe C en orden inverso
  }
```

Paradigma de pipelining. Multiplicación de matrices: Workers

```
process Worker [w=0 to n-1]    {
    double A[n], B[n], C[n];     # fila y columna con las que trabaja
    double temp[n];              # usado para pasar vectores
    double total;                # usado para calcular el producto interno
    # Recibe las filas de A, almacena la primera y pasa las otras
    receive vector[w]  (A);
    for [i= w+1  to n-1]   {
            receive vector[w] (temp);
            send vector[w +1] (temp);       }
    # 2da FASE. Recibe las columnas de B y calcula el producto interno
    for [j=0 to n-1]  {
        receive vector[w]  (B);     # Obtiene 1 columna de B
        IF ( w  < n-1)              # NO es el último Worker
            send vector[w+1] (B);
        # Calcula el producto interno que le corresponde
        total=0;
        for [k= 0  to n-1] total += A[k] * B[k];
        C[j] = total;         }
```

Paradigma de pipelining. Multiplicación de matrices: Workers

```
    # 3ra FASE. Envía su columna de C al next Worker o coordinador
    if ( w  < n-1)              #NO es el último Worker
          send vector[w+1] (C);
      else send result(C);
    # Recibe y pasa filas anteriores de C
    for [j=0 to w-1]  {
          receive vector[w]  (temp);    # Obtiene 1 fila resultado de C
          IF ( w  < n-1) send vector[w+1] (temp);
            else send result(temp);      }
}
```

No hay delay e/ el tiempo en que un worker recibe un msg y lo pasa $\Rightarrow$ los mensajes fluyen continuamente. Cuando un worker calcula un producto interno, ya pasó la columna que está usando.

Una vez que el pipe está lleno, los productos internos pueden calcularse casi tan rápido como fluyen los mensajes.

Puede modificarse la solución para usar menos workers

Algoritmos Token-Passing

Un paradigma de interacción muy usado se basa en un tipo especial de mensaje (“token”) que puede usarse para otorgar un permiso (control) o recoger información global de la arquitectura distribuida.

Un ejemplo del 1er tipo de algoritmos es el caso de tener que controlar *exclusión mutua distribuida*
Ejemplos de recolección de información de estado son los algoritmos de detección de terminación en computación distribuida.

- Aunque el problema de la SC se da principalmente en pgms de MC, puede encontrarse en programas distribuidos cuando hay algún recurso compartido que puede usar un único proceso a la vez.
- Generalmente es una componente de un problema más grande, tal como asegurar consistencia en un sistema de BDD.
- Soluciones posibles: Monitor activo que da permiso de acceso (ej: locks en archivos), semáforos distribuidos (usando broadcast, con gran intercambio de mensajes), o *token ring* (descentralizado y fair)

Algoritmos Token-Passing. Exclusión mutua distribuida

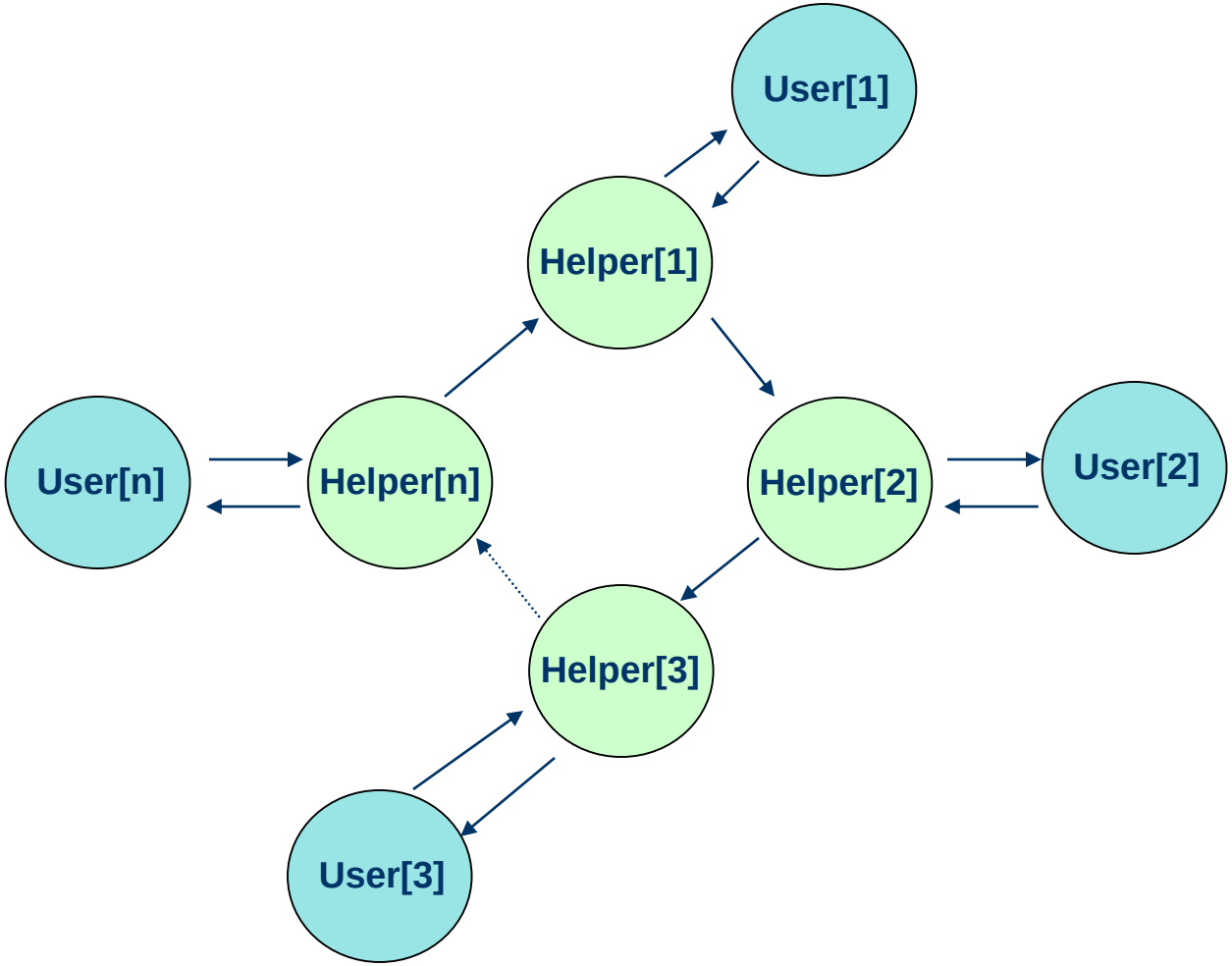
User[1:n] procesos de aplicación con SC y SNC.
La EM distribuida consiste en desarrollar protocolos de interacción (E/S a las SC), asegurando EM, no deadlock, evitar demoras innecesarias y eventual entrada (fairness).

Para no ocupar los User en el manejo de los tokens, usamos un proceso auxiliar (Helper) por c/ User, para manejar la circulación de los tokens en el anillo que forman los helpers.

Cuando Helper[i] tiene el token, User[i] tendrá prioridad para acceder a la SC. Sino, pasa el token al siguiente.
El algoritmo es fair si los procesos devuelven el token.

Esquema muy usado en redes conectadas por bus, donde todos los nodos tienen el token de acceso al bus cíclicamente. La circulación del token puede hacerse más lenta.

Algoritmos Token-Passing. Exclusión mutua distribuida



Algoritmos Token-Passing. Exclusión mutua distribuida

```
chan token[1:n] ( ), enter[1:n] ( ), go[1:n] ( ), exit[1:n] ( );
process Helper[i=1 to n]  {
    while (true)   {
            receive token[i] ( );                  # Espera el token
            if (not empty (enter[i]))   {          # User[i] lo necesita ?
                receive enter[i] ( )
                send go [i] ( );                    # Otorga el permiso
                receive exit[i] ( );                # Wait por el exit
             }
            send token [ i+1 MOD n] ( );       # Pasa el token al siguiente
        }
}
process User[i=1 to n]  {
    while (true)   {
            send enter[i] ( );                     # Protocolo de E/ a la SC
            receive go[i] ( );
            SECCION CRITICA;
            send exit[i] ( );                      # Protocolo de S/ de la SC
            SECCION NO CRITICA;      }
}
```

Paradigma de servidores replicados. Ejemplo: filósofos

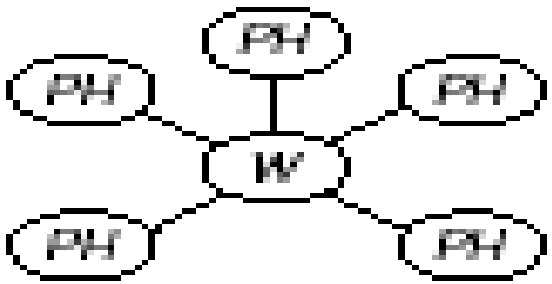
Un server puede ser replicado cuando hay múltiples instancias de un recurso: c/ server maneja una.
También puede usarse para darle a los clientes la sensación de un único recurso cuando en realidad hay varios.

Un posible modelo de solución al problema de los filósofos es el ***centralizado*****:** los procesos Filósofo se comunican con ***UN*** proceso Mozo que decide el acceso o no a los recursos.

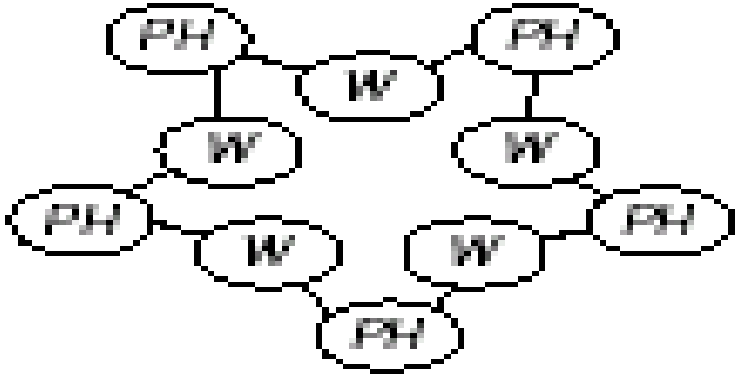
Una 2da solución (***distribuida***) supone **5 procesos Mozo** c/u manejando un tenedor. Un Filósofo puede comunicarse con **2** Mozos (izquierdo y derecho), solicitando y devolviendo el recurso. **Los Mozos NO se comunican entre ellos**.

En la 3ra solución (***descentralizada***)**,** c/ Filósofo ve **un único** Mozo. Los Mozos se comunican entre ellos (cada uno con sus **2** vecinos) para decidir el manejo del recurso asociado a "su" Filósofo.

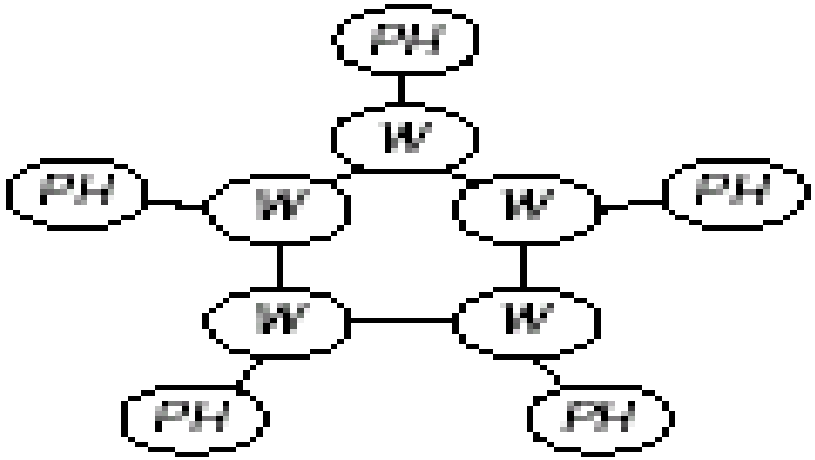
Paradigma de servidores replicados. Ejemplo: filósofos



(a) Centralized



(b) Distributed



(c) Decentralized

Recordar: Filósofos centralizado (sin deadlock – no fair – cuello de botella)

```
module Mesa
   op tomar_ten(int), liberar_ten (int);
body
   process Mozo {
            bool comiendo[5] =([5] false);
            while (true)
               in tomar_ten(i) and not (comiendo[izq(i)] and
                        not coimiendo[der(i)]  → comiendo[i] = true;
               □ liberar_ten(i) →
                        comiendo[i] = false;
               ni       }
end Mesa
Process Filosofo [i = 0 to 4] {
   while (true) {
            call tomar_ten(i);
            come;
            call liberar_ten(i);
            piensa;
     }
  }
```

Filósofos Distribuido (fair – sin deadlock – no hay cuello de botella – más mensajes)

```
Process Filosofo[i = 0 to 4]  {
      INT primero = i, segundo = i+1;
      IF (i == 4)   { primero=0; segundo=4;}
      WHILE (true)  {
            Call Mozo[primero]. TomarTen( );
            Call Mozo[segundo]. TomarTen( );
            COMER;
            Send Mozo[primero]. LiberarTen ( );
            Send Mozo[segundo].LiberarTen ( );
            PENSAR;          }
}
Module Mozo[5]
    op TomarTen( ), LiberarTen( );
BODY
    Process El_Mozo   {
          WHILE (true) {
                receive TomarTen ( );
                receive LiberarTen ( );       }       }
End Mozo;
```

Filósofos Descentralizado

La solución descentralizada tiene un mozo por filósofo. El algoritmo usado por los mozos es *token passing*, donde tokens = tenedores.

La solución puede adaptarse para coordinar acceso a archivos replicados, o dar solución eficiente al problema de EM distribuida.

Cada tenedor es un token que tiene uno de los dos mozos o está en tránsito entre ellos.

Cuando el filósofo quiere comer, le pide a su mozo que adquiera dos tenedores. Si el mozo no los tiene, interactúa con sus mozos vecinos para obtenerlos. Luego mantiene el control mientras el filósofo come.

Debe evitarse el deadlock, que podría darse si un mozo necesita dos tenedores y no los puede lograr.

Filósofos Descentralizado

Cuando su filósofo no está comiendo, el mozo debe poder ceder sus tenedores, pero debe evitar pasar ida y vuelta un tenedor entre mozos sin que sea usado.

Para evitar deadlock, el mozo debe ceder un tenedor que ya fue usado, pero debe mantener uno que adquirió y no usó.

Cuando un Phil empieza a comer, su mozo marca los tenedores como “sucios”. Cuando otro quiere un tenedor, si está sucio y no se usa, lo limpia y lo cede. El que lo recibe, mantiene el tenedor limpio hasta que sea usado.

Un tenedor sucio puede ser reusado hasta que lo necesite otro. (*Filósofos “higiénicos”*).

Inicialmente los tenedores se distribuyen asimétricamente y todos están sucios.

La solución evita la inanición.

Filósofos Descentralizado

```
Module Waiter[ t = 0 to 4]
    op getforks(int), relforks(int);        #usadas por los filósofos
    op NeedL( ), NeedR( ),                  # para los Waiters
       PassL ( ), PassR( );                 # para los Waiters
    op Forks (bool, bool, bool, bool);      # para inicializar

body
    op Hungry( ), Eat ( ) ;                 # operaciones locales
    bool haveL, dirtyL, haveR, dirtyR;      # status de los tenedores
    int left = (t-1) MOD 5;                 # vecino a izquierda
    int right = (t+1) MOD 5;                # vecino a derecha

    proc getforks ( )   {
          send hungry ( );                  #dice que Phil tiene hambre
          receive eat ( );                  # espera permiso p/ comer
     }
```

Filósofos Descentralizado

```
process the_Waiter   {
    receive forks (haveL, dirtyL, haveR, dirtyR);
    while (true)  {
          in hungry ( ) →
              # preguntar por los tenedores que no se tienen
              if (not HaveR) send Waiter[right].needL( );
              if (Not HaveL) send Waiter[left].needR( );
              # esperar hasta tener ambos tenedores
              while ( not haveL  OR  not haveR)
                    in passR( ) → haveR=true; dirtyR=false;
                    [] passL( ) → haveL=true; dirtyL=false;
                    [] needR( ) st  dirtyR → haveR=false; dirtyR=false;
                              send waiter[rigth].passL( );
                              send waiter[rigth].needL( );
                    [] needL( ) st  dirtyL → haveL=false; dirtyL=false;
                              send waiter[left].passR( );
                              send waiter[left].needR( );
                    ni
```

Filósofos Descentralizado

```
                # Al salir del while el Waiter tiene los dos tenedores para
                # su filosofo. Debe dejarlo comer y esperar que los libere
                send eat ( );
                dirtyL=true; dirtyR=true;
                receive relforks( );
            [] needR ( ) →
                #Waiter vecino requiere mi tenedor derecho (su
izquierdo)
                haveR=false; dirtyR=false;
                send waiter[rigth].passL( );
            [] needL ( ) →
                #Waiter vecino requiere mi tenedor izquierdo (su
derecho)
                haveL=false; dirtyL=false;
                send waiter[left].passR( );
            ni
    }
end Waiter
```

Filósofos Descentralizado

```
process Philosopher [i=0 to 4]   {
    while (true)
            Call Waiter[i].Getforks ( );
            COMER;
            Call Waiter[i].Relforks( );
            PENSAR;
     }
}

process Main   {
    send Waiter[0].forks(true, true, true, true);
    send Waiter[1].forks(false,false, true, true);
    send Waiter[2].forks(false, false, true, true);
    send Waiter[3].forks(false, false, true, true);
    send Waiter[4].forks(false, false, false, false);
}
```

Paradigma de Prueba-Eco (*Probe-Echo)*

Arboles y grafos: utilizados en muchas aplicaciones distribuidas (búsquedas en la WEB, BD, sistemas expertos, juegos).
Las arq. distribuidas se pueden asimilar a los nodos de grafos y árboles, con canales de comunicación que los vinculan.
DFS es uno de los paradigmas secuenciales clásicos para visitar todos los nodos en un árbol o grafo.

Prueba-eco se basa en el envío de un msg ("probe") de un nodo al sucesor, y la espera posterior del "eco" (mensaje de respuesta).

Los *probes* se envían **en paralelo** a todos los sucesores.
Este paradigma es el análogo concurrente de DFS

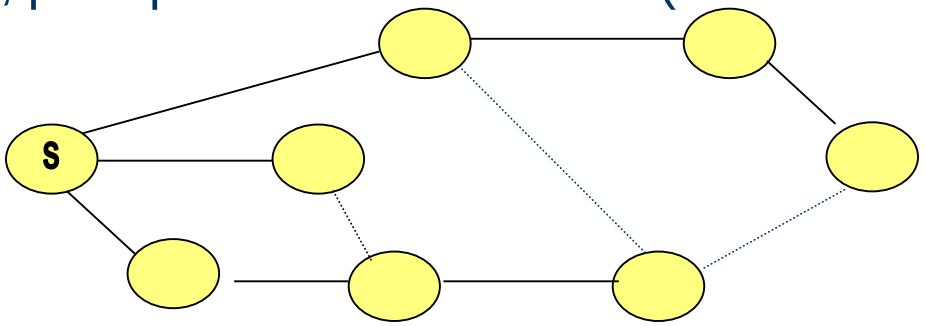
Los algoritmos de prueba-eco son particularmente interesantes cuando se trata de recorrer redes donde no hay (o no se conoce) un n° fijo de nodos activos (ejemplo: redes móviles).

Paradigma de Prueba-Eco. Broadcast en una red

El problema de hacer un **broadcast** a todos los nodos de una red es clásico en procesamiento distribuido. Un primer enfoque es suponer que algún nodo tiene la topología de la red en alguna forma de matriz donde **la entrada [i, j] es true si los nodos están conectados.**

Resolver el broadcast con un spanning tree supone **un nodo (el origen)** que tiene decidido el routing y lo envía a sus sucesores, conjuntamente con el mensaje m (actúa como raíz del árbol).

C/ nodo a su vez recibe el mensaje m y el spanning tree t y visualiza cuales son los “hijos” a los que debe reenviar el mensaje. El proceso es asincrónico, pero podría ser sincrónico (menor eficiencia)



Paradigma de Prueba-Eco. Broadcast usando un spanning tree

```
type grafo = BOOL [n, n];
chan prueba[n] (grafo spanningTree; mensaje m);

process Nodo [p = 0 to n-1]   {
    grafo t; mensaje m;
    receive prueba[p] (t,m);
    for [q = 0 to n-1  st  q es “hijo” de p en t]
          send prueba[q] (t, m);
 }
```

```
PROCESS Iniciador   {
      grafo topologia =  topología de la red;
      grafo t = spanning tree de topology;
      mensaje m = mensaje a enviar por broadcast;
      send prueba[S] (t,m);
}
```

Paradigma de Prueba-Eco. Broadcast SIN spanning tree

```
# El broadcast se complica cuando NO se conoce la topología.
# Algoritmo simétrico: cada nodo al recibir un msg lo reenvía a todos sus vecinos, incluyendo el emisor. Después recibe copias que ignora.

chan prueba[n] (mensaje m);
process Nodo [p = 1 to n]   {
    bool vecinos[n] = vecinos del nodo p;
    int num = nro. de vecinos;
    mensaje m;
    receive prueba[p] (m);
    # enviar el mensaje m a todos los vecinos.
    for [q = 0 to n-1  st  vecinos[q]] send prueba[q] (m);
    # recibir num -1 copias redundantes de m
    for [q = 1 to num-1] receive prueba[p] (m);
  }
process Iniciador   {   # ejecutado sobre el nodo S
    mensaje m = mensaje a enviar por broadcast;
    send prueba[S] (m);
  }
```

Paradigma de Prueba-Eco. Cómo obtener la topología de la red?

La solución que construye el spanning tree es más eficiente (n-1 mensajes), pero requiere conocer la topología de la red (sea un grafo con ciclos o un árbol sin ellos).

La solución *de vecinos* no necesita que el iniciador conozca la topología o calcule un spanning tree (se construye dinámicamente, y consta de los links por los que se envían las primeras copias de *m*). Además, usa mensajes más cortos pues no se envía el árbol.

La idea para obtener la topología de la red puede ser partir de un algoritmo similar al anterior (SIN spanning tree) pero pidiendo que en el ECO cada nodo envíe la información que tiene de sus vecinos.

Así, el nodo iniciador en un tiempo puede reconstruir las relaciones entre los nodos y armar el grafo de la topología. Posteriormente puede calcular el spanning tree asociado.
(VER LA SOLUCION EN EL LIBRO DE ANDREWS)

Paradigma de Prueba-Eco. Cómo obtener la topología de la red? Caso 1 – árbol (grafo sin ciclos)

```
type graph = bool [n,n];
chan probe[n](int sender);
chan echo[n](graph topology)     # parts of the topology
chan finalecho(graph topology)   # final topology

process Node[p = 0 to n-1] {
  bool links[n] = neighbors of node p;
  graph newtop, localtop = ([n*n] false);
  int parent;    # node from whom probe is received
  localtop[p,0:n-1] = links;    # initially my links

  receive probe[p](parent);
  # send probe to other neighbors, who are p's children
  for [q = 0 to n-1 st (links[q] and q != parent)]
    send probe[q](p);

  # receive echoes and union them into localtop
  for [q = 0 to n-1 st (links[q] and q != parent)] {
    receive echo[p](newtop);
    localtop = localtop or newtop;  # logical or
  }
  if (p == S)
    send finalecho(localtop);     # node S is root
  else
    send echo[parent](localtop);
}

process Initiator {
  graph topology;
  send probe[S](S)    # start probe at local node
  receive finalecho(topology);
}
```

Figure 9.11 Probe/echo algorithm for gathering the topology of a tree.

Paradigma de Prueba-Eco. Cómo obtener la topología de la red? Caso 2 – grafo con ciclos

Luego de recibir un *probe*, un nodo se lo envía a sus otros vecinos, y luego espera un eco de esos vecinos.

A causa de los ciclos, y dado que los nodos ejecutan concurrentemente, 2 vecinos podrían enviarse mutuamente probes casi al mismo tiempo.

Los probes que no son el 1ro pueden contestarse con un eco inmediatamente.

En particular, si un nodo recibe un probe mientras espera por ecos, inmediatamente envía un eco conteniendo una topología nula (los links de ese nodo estarán en el eco al primer probe).

Eventualmente un nodo recibirá un eco en respuesta a cada probe.

En este punto, envía un eco al nodo del que recibió el primer probe; este eco contiene la unión de los links del propio nodo con todos los conjuntos de links recibidos

Se usa el mismo canal para los probes y ecos, para que no sea necesario usar empty o hacer polling. Alternativamente podría usarse rendezvous, siendo probe y ecos dos operaciones diferentes.

Paradigma de Prueba-Eco. Cómo obtener la topología de la red? Caso 2 – grafo con ciclos

```
type graph = bool [n,n];
type kind = (PROBE, ECHO);
chan probe_echo[n](kind k; int sender; graph topology);
chan finalecho(graph topology);

process Node[p = 0 to n-1] {
  bool links[n] = neighbors of node p;
  graph newtop, localtop = ([n*n] false);
  int first, sender; kind k;
  int need_echo = number of neighbors - 1;
  localtop[p,0:n-1] = links;   # initially my links

  receive probe_echo[p](k, first, newtop);  # get probe
  # send probe on to to all other neighbors
  for [q = 0 to n-1 st (links[q] and q != first)]
    send probe_echo[q](PROBE, p, ∅);
  while (need_echo > 0) {
    # receive echoes or redundant probes from neighbors
    receive probe_echo[p](k, sender, newtop);
    if (k == PROBE)
      send probe_echo[sender](ECHO, p, ∅);
    else  # k == ECHO {
      localtop = localtop or newtop;  # logical or
      need_echo = need_echo-1;
    }
  }
  if (p == S)
    send finalecho(localtop);
  else
    send probe_echo[first](ECHO, p, localtop);
}

process Initiator {
  graph topology;   # network topology
  send probe_echo[source](PROBE, source, ∅);
  receive finalecho(topology);
}
```

Figure 9.12 Probe/echo algorithm for computing the topology of a graph.

Paradigma de Broadcast

En la mayoría de las LAN los procesadores comparten un canal común (Ethernet o token ring) $\Rightarrow$ c/ procesador se conecta directamente con los otros.

Estas redes normalmente soportan una primitiva especial *broadcast* que transmite un mensaje de un procesador a todos los otros:

broadcast ch(m);

Los msgs broadcast de un proceso se encolan en los canales en el orden de envío, pero *broadcast* no es atómico y los msgs enviados por procesos A y B podrían ser recibidos por otros en distinto orden.

Se puede usar *broadcast* para diseminar información (ej: intercambiar info de estado de procesadores en LAN), o p/ resolver problemas de sincronización distribuida (Ej: semáforos distribuidos).
La base es un ***ordenamiento total de eventos de comunicación*** mediante el uso de ***relojes lógicos***.

Relojes lógicos y ordenamiento de eventos

Las acciones de los procesos en un programa distribuido pueden dividirse en:
- *acciones locales* (R/W), que no afectan la ejecución de otros procesos
- *acciones de comunicación,* que afectan a otros procesos ya que transmiten información y son sincronizadas

Evento $\rightarrow$ ejecución de *send* o *receive*

- Si *A* y *B* ejecutan acciones locales, no se sabe el orden relativo
- Si *A* envía un msg a *B*, el *send* en *A* sucede antes del *receive* en *B*
- Si *B* luego envía msg a *C*, el *send* en *B* debe ocurrir antes del *receive* en *C*

$\Rightarrow$ "*Ocurre antes*" es una relación transitiva entre eventos relacionados

Aunque hay un ordenamiento total entre las cuatro acciones de comunicación, hay sólo orden parcial entre todos los eventos en un programa distribuido: las secuencias no relacionadas de eventos (ej comunicaciones entre distintos conjuntos de procesos) pueden ocurrir antes, después o concurrentemente con otras.

Relojes lógicos y ordenamiento de eventos

Si hubiera un único reloj central podríamos ordenar completamente los eventos dando a c/u un timestamp único.

En una LAN, cada procesador tiene su propio reloj.

Si estuvieran sincronizados, entonces podríamos usar los relojes locales para los timestamps. Esto no ocurre $\rightarrow$ necesitamos una manera de simular relojes físicos

reloj lógico $\rightarrow$ contador entero incrementado en cada evento

Cada proceso tiene un reloj lógico init 0 y cada mensaje un timestamp

Reglas de Actualización de relojes. Sea *lc* un reloj lógico en el proceso *A*. El proceso *A* actualiza el reloj lógico de la siguiente manera:

(1) Cuando *A* envía o broadcast un mensaje, setea el timestamp del mensaje al valor de *lc* y luego lo incrementa en 1.

(2) Cuando *A* recibe un mensaje con timestamp *ts*, setea *lc* al máximo de *lc* y *ts*+1 y luego incrementa *lc* en 1.

Relojes lógicos y ordenamiento de eventos

Usando relojes lógicos, podemos asociar un valor de reloj con cada evento:

- send: valor del reloj = timestamp del msg (valor local de *lc* al inicio del envío)

- receive: valor del reloj = *lc* después de ser seteado al máximo de *lc* y *ts*+1 pero antes de que sea incrementada por el receptor

Asegura que si el evento *a* ocurre antes que el *b*, el valor de reloj asociado con *a* es menor que el asociado con *b*.

$\Rightarrow$ Da un orden parcial en el conjunto de eventos relacionados

Si cada proceso tiene identidad única, podemos tener un ordenamiento total usando la identidad menor como tie-breaker

Semáforos distribuidos

Semáforos: implementados normalmente usando VC.

Podríamos implementarlos en un programa distribuido usando un proceso server (monitor activo).

También podemos implementarlos en forma distribuida sin usar un coordinador central.

En un semáforo, en todo momento, el número de ops **P** completadas es a lo sumo el número de **V** completadas más el valor inicial.

$\Rightarrow$ P/ implementar semáforos, necesitamos contar los **P** y **V** y una manera de demorar las operaciones **P**.

Además, los procesos que "comparten" un semáforo necesitan cooperar para mantener el invariante aunque el estado del programa esté distribuido.

Podemos cumplir esto haciendo que los procesos broadcast msgs cuando quieren ejecutar **P** y **V,** y que examinen los msgs que reciben p/ determinar cuándo continuar.

Cada proceso tiene una cola de mensajes local *mq* y un reloj lógico *lc*.

Semáforos distribuidos

Para simular P o V, un proceso broadcast un mensaje a todos los procesos usuario, incluyendo él mismo.

El mensaje contiene la identidad del emisor, un tag (P o V) y un timestamp. El timestamp en cada copia del mensaje es el valor corriente de *lc*, el cual es modificado de acuerdo a las reglas.

Cuando un proceso recibe un mensaje P o V, almacena el mensaje en su cola de mensajes *mq* (ordenada en orden creciente de los timestamps de los mensajes)

Si cada proceso recibe todos mensajes en el mismo orden en que se enviaron y en orden creciente de timestamp, c/u podría saber exactamente el orden en que se enviaron los mensajes y podría contar la cantidad de P y V para mantener el invariante.

Pero, el broadcast no es atómico: dos mensajes broadcast por procesos distintos podrían ser recibidos por otros en distinto orden (incluso uno con timestamp menor podría recibirse después que uno con ts mayor)

Pero, distintos mensajes broadcast por un proceso serán recibidos por los otros procesos en el orden en que fueron enviados, y tendrán valor creciente de ts.

Semáforos distribuidos

Sup. que la cola *mq* de un proceso tiene un mensaje *m* con timestamp *ts*

Cuando el proceso recibió un mensaje con timestamp mayor de c/u de los otros, nunca verá un mensaje con timestamp menor

⇒ *m* se dice que está *totalmente reconocido* (*fully acknowledged*)

Además, cuando *m* está fully acknowledged todos los otros msgs anteriores en mq estarán FA xq tienen ts menor

La parte de *mq* que contiene mensajes *FA* es un *prefijo estable*: no se agregarán más mensajes en él

Cada vez que un proceso recibe un P o V, broadcast un ACK

Los ACKs tienen timestamps, pero no son almacenados en las colas de mensajes y no son acknowledged a sí mismos

Se usan para determinar cuándo un mensaje en *mq* está *FA*

Semáforos distribuidos

C/ proceso usa una variable local *s* para el valor del semáforo

Cuando un proceso recibe un ACK, actualiza el prefijo estable de *mq*.

Para cada msg V, el proceso incrementa *s* y borra el mensaje. Luego examina los msgs P en orden de ts, Si *s* > 0, el proceso decrementa *s* y borra el mensaje P.

Los msgs P se procesan en el orden en que aparecen en el prefijo estable, por lo que cada proceso toma la misma decisión sobre el orden en que se completan los P.

Aunque los procesos pueden estar en distintas etapas manejando los msgs P y V, cada uno manejará los mensajes FA en el mismo orden

El algoritmo utiliza un helper por cada proceso usuario, y los helpers interactuan para implementar los P y V.

Un proceso usuario inicia una operación P o V comunicándose con su helper; si es un P, el usuario espera a que el helper le diga que puede continuar. Cada helper hace broadcast de los msgs P, V y ACK a los otros y maneja la cola local.

Cada proceso participa en todas las decisiones, por lo que la escalabilidad es baja.

Semáforos distribuidos

```
type kind = enum(reqP, reqV, VOP, POP, ACK);
chan semop[n](int sender; kind k; int timestamp);
chan go[n](int timestamp);

process User[i = 0 to n-1] {
  int lc = 0, ts;
  ...
  # ask my helper to do V(s)
  send semop[i](i, reqV, lc); lc = lc+1;
  ...
  # ask my helper to do P(s), then wait for permission
  send semop[i](i, reqP, lc); lc = lc+1;
  receive go[i](ts); lc = max(lc, ts+1); lc = lc+1;
}
```

Semáforos distribuidos

```
process Helper[i = 0 to n-1] {
  queue mq = new queue(int, kind, int);    # message queue
  int lc = 0, s = 0;                       # logical clock and semaphore
  int sender, ts; kind k;                  # values in received messages
  while (true) {    # loop invariant DSEM
    receive semop[i](sender, k, ts);
    lc = max(lc, ts+1); lc = lc+1;
    if (k == reqP)
      { broadcast semop(i, POP, lc); lc = lc+1; }
    else if (k == reqV)
      { broadcast semop(i, VOP, lc); lc = lc+1; }
    else if (k == POP or k == VOP) {
      insert (sender, k, ts) at appropriate place in mq;
      broadcast semop(i, ACK, lc); lc = lc+1;
    }
    else {  # k == ACK
      record that another ACK has been seen;
      for (all fully acknowledged VOP messages in mq)
        { remove the message from mq; s = s+1; }
      for (all fully acknowledged POP messages in mq st s > 0) {
        remove the message from mq; s = s-1;
        if (sender == i)     # my user's P request
          { send go[i](lc); lc = lc+1; }
      }
    }
  }
}
```

Manager/Workers: Bag of Tasks Distribuido

- El concepto de ***bag of tasks*** usando VC supone que un conjunto de workers comparten una "bolsa" con tareas o procesos independientes. Los workers sacan una tarea de la bolsa, la ejecutan, y posiblemente crean nuevas tareas que ponen en la bolsa. (Ejemplo en LINDA manejando un espacio compartido de tuplas).

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

- Analizaremos la implementación de este paradigma con mensajes en lugar de MC. Para esto un proceso *manager* implementará la "bolsa" manejando las tasks, comunicándose con los workers y detectando fin de tareas. **Se trata de un esquema C/S**.

Manager/Workers. Multiplicación de Matrices Ralas

Producto de dos matrices A y B de n x n $\Rightarrow$ n^2 productos internos de c/ fila de A por las columnas de B a fin de obtener C[nxn].

C/ producto interno es la multiplicación elemento a elemento de dos vectores de dimensión n y su suma para obtener un único valor.

Si tenemos matrices “ralas” (con gran número de ceros), podemos aprovechar p/ optimizar el producto (Cada 0 en A o B no contribuye al resultado de n de los productos internos).

Si una fila de A contiene todos 0, la fila correspondiente de C también será de todos 0)

Puede ahorrarse espacio almacenando sólo las entradas no cero:

INT lengthA[n]; # lenghtA[i] = n° de no ceros en la fila i de A
pair *elementsA[n]; # elementsA[i] apunta a lista de entradas no cero en fila i. C/ E/ es un par: índice de columna y valor.

Manager/Workers. Multiplicación de Matrices Ralas

La matriz C se representa como A, pero B se representa por columnas $\Rightarrow$ **lenghtB** indica el número de elementos no cero en cada columna de B y **elementsB** contiene pares de índice de fila y valor de datos.

Calcular AxB requiere examinar los n^2 pares de filas y columnas.
Con matrices ralas, el tamaño más natural para una tarea es una fila de la matriz **C**, porque una fila entera está representada por un único vector de pares (columna, valor).

$\Rightarrow$ Hay tantas tareas como filas de **A**. Puede optimizarse salteando las filas de A con todos ceros, ya que las filas correspondientes en **C** serán de todos ceros (no es una aplicación realista).

Manager/Workers. Multiplicación de Matrices Ralas. Proceso Manager.

```
Module Manager
    type pair = (INT index, DOUBLE value);
    op getTask(result INT row, len; result pair [*] elems);
    op putResult(INT row, len; pair [*] elems);
body Manager
    int lengthA[n], lengthC[n];    pair *elementsA[n], *elementsC[n];
    # Se supone A inicializada
    int nextRow=0, tasksDone=0;
process Manager {
    while (nextRow < n  OR  tasksDone <n) {
            # Hay más tareas para hacer o se necesitan más resultados
            in getTask(row, len, elems) →
                row=nextRow;
                len = lengthA[i];
                copiar los pares de A[i] en elems
                nextRow++;
            [] putResult (row, len, elems) →
                lengthC[row]=len;
                copiar los pares de elems a *elementsC[row];
                taskDone++;                }
    }
```

Manager/Workers. Multiplicación de Matrices Ralas. Workers.

```
process Worker [w = 1 to numWorkers]  {
    int lengthB[n];
    pair *elementsB[n];
    # se supone inicializada B
    int row, lengthA, lengthC;
    pair *elementsA, *elementsC;
    int r, c, na, nb;
    double sum;
    while (true)    {
            #Obtener una fila de A y calcular la fila de C
            call getTask(row, lengthA, elementsA);
            lengthC=0;
            for [ i=0 to n-1] INNER_PRODUCT (i);
            send putResult(row, lengthC, elementsC);
          }
 }
```

Manager/Workers. Multiplicación de Matrices Ralas. Código de INNER_PRODUCT.

```
sum = 0; na = 1; nb = 1;
c = elementsA[na] -> index;            # columna en la fila de A
r = elementsB[i] [nb] -> index;        # fila en la columna de B
while (na <= lenghtA and nb <= lengthB)    {
     if (r ==c) {
            sum += elementsA[na] -> value *
                      elementsB[i] [nb] -> value;
         na++; nb++;
         c = elementsA[na] -> index;
         r = elementsB[i] [nb] -> index;
      }  else if (r<c) {
            nb++; r = elementsB[i] [nb] -> index;
      }  else {
            na++; c = elementsA[na] -> index;
   }
 }
If (sum != 0) {
    elementsC[lengthC] = pair(i,sum);
    lengthC++;
 }
```

Casos problemas de interés

-Quicksort usando manager-workers

-*N* reinas con manager-workers

-Modificar el Juego de la vida para que cada worker maneje un strip o un bloque de células

-Modificar el Juego de la vida para simular alguna otra clase de interacción (tiburones y peces, conejos y zorros, coyotes y correcaminos, ...) o incendio de árboles y maleza en un bosque. Las reglas también podrían no ser fijas sino probabilísticas ...

Casos problemas de interés

-**Generalización de *Drinking Philosophers*.**
Se tiene un grafo no dirigido G.
Los Phil se asocian a nodos del grafo y pueden comunicarse sólo con sus vecinos.
Una botella se asocia con cada arista de G.
Cada Phil cicla entre tres estados: tranquilo, sediento y bebiendo.
Un Phil tranquilo se puede volver sediento.
Antes de beber, el Phil debe adquirir la botella asociada con cada arista conectada a su nodo.
Luego de beber, un filósofo nuevamente se vuelve tranquilo

Diseñe una solución fair y sin deadlock, donde cada Phil ejecute el mismo algoritmo.

Use tokens para representar las botellas.
A un Phil tranquilo se le permite responder a pedidos de vecinos por cualquier botella que él tenga.

Implementaciones

El capítulo 10 del libro de Andrews contiene una descripción de posibles implementaciones de los mecanismos de comunicación y sincronización en programas distribuidos.

También incluye el tema de Memoria Compartida Distribuida

Programación Paralela

Programa Concurrente: múltiples procesos

Programa Distribuido: pgm concurrente en el cual los procesos se comunican y sincronizan por PM, RPC o Rendezvous

Programa Paralelo: pgm concurrente escrito para resolver un problema en menos tiempo que el secuencial.
El objetivo ppal es reducir el tiempo de ejecución, o resolver problemas + grandes o con > precisión en el mismo tiempo.
Ejemplos.

Un programa paralelo puede escribirse usando VC o PM. La elección la dicta el tipo de arquitectura.

Computación Científica

Los dos modos tradicionales del descubrimiento científico son *teoría* y *experimentación*.

El 3er modo es la *modelización computacional*, que usa computadoras para simular fenómenos y tratar cuestiones del tipo "*what if?*"

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

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

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

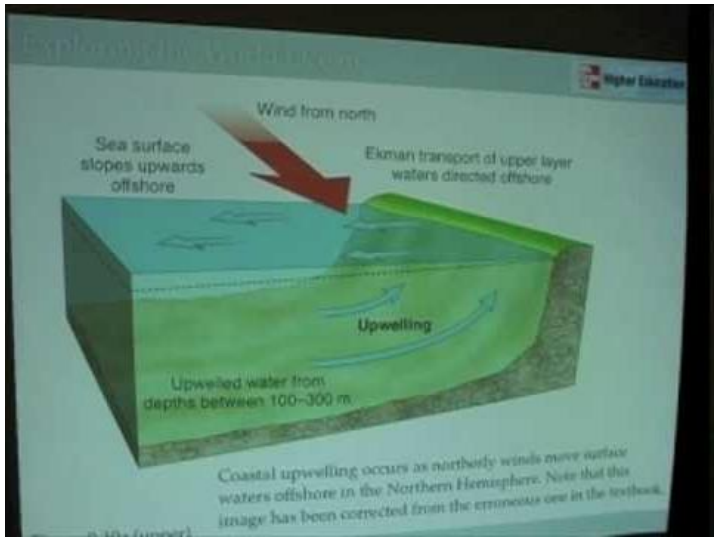
(3) Computación de matrices (sistemas de ecuaciones simultáneas)

Necesidad del paralelismo

Problemas “grand challenge”. Abarcan química cuántica, mecánica estadística, cosmología y astrofísica, dinámica de fluidos computacional, diseño de materiales, biología, farmacología, secuencia genómica, ingeniería genética, medicina y modelización de órganos y huesos humanos, pronóstico del tiempo, sensado remoto, física de partículas etc.

Necesidad del paralelismo

Ej: *simulación de circulación oceánica*. División del océano en 4096x1024 regiones, y c/u en 12 niveles. Aprox. 50 millones de celdas 3D. Una iteración del modelo simula la circulación por 10 minutos y requiere alrededor de 30 billones de cálculos de punto flotante. Se intenta usar el modelo para simular la circulación en un período de años...



Necesidad del paralelismo

Ej: *pronóstico del clima*.
Modelización de la atmósfera en celdas 3D.
Ecuaciones complejas p/ capturar diversos efectos atmosféricos.

Celdas de 1 milla^3, hasta 10 en altura $\Rightarrow$ 5x10^8 celdas.
C/cálculo requiere 200 op pto flotante $\Rightarrow$ 10^{11} op pto flotante en un paso.
Simular una semana usando intervalos de un minuto $\Rightarrow$ 10^{15} op pto flotante en total
1 máquina de 1Gflops (10^9 op por segundo) tomaría 10^6 segundos (más de 10 días).
Para hacer el cálculo en 5 min, 3.4 Tflops (3.4 x 10^{12} op pto flotante por segundo)

Diseño de algoritmos paralelos

No se reduce a simples recetas, sino que es necesaria la *creatividad.*

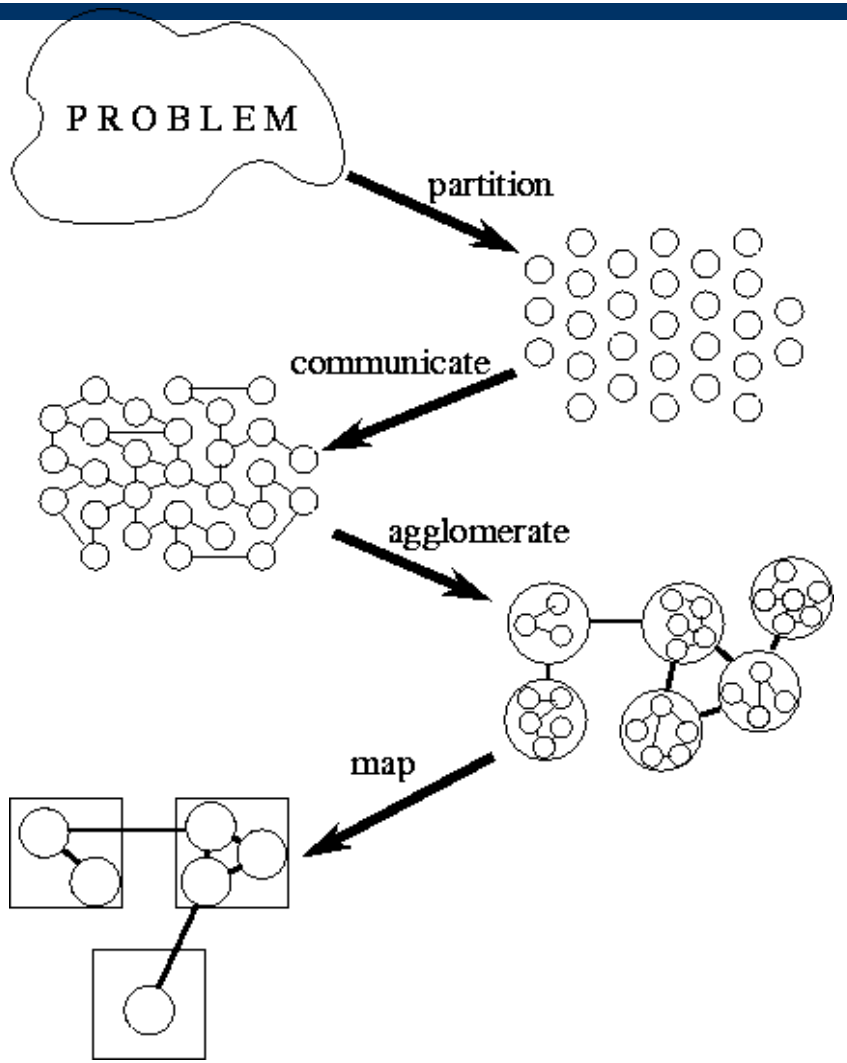
La mejor solución puede diferir totalmente de la sugerida por los algoritmos secuenciales existentes

Puede darse un enfoque metódico p/ maximizar el rango de opciones, brindar mecanismos para evaluar las alternativas, y reducir el costo de *backtracking* por malas elecciones

⇒ Metodología de diseño que da un enfoque exploratorio en el cual aspectos independientes de la máquina tales como la concurrencia son considerados temprano, y los aspectos específicos de la máquina se demoran.

Diseño de algoritmos paralelos

4 etapas:
- *particionamiento* (descomposición en tareas)
- *comunicación* (estructura necesaria para coordinar la ejecución)
- *aglomeración* (evaluación de tareas y estructura con respecto a performance y costo, combinando tareas para mejorar)
- *mapeo* (asignación de tareas a procesadores)



Diseño de algoritmos paralelos

Particionamiento

Intenta exponer oportunidades para la ejecución paralela

Se trata de definir un gran n° de pequeñas tareas para obtener una descomposición *de grano fino*, para brindar la mayor flexibilidad a los algoritmos paralelos potenciales

En etapas posteriores, la evaluación de los requerimientos de comunicación, arquitectura de destino, o temas de IS pueden llevar a descartar algunas posibilidades detectadas en esta etapa, revisando la partición original y aglomerando tareas para incrementar su tamaño o granularidad.

Una buena partición divide *computación* y *datos*. Son técnicas complementarias.

Diseño de algoritmos paralelos

Particionamiento

Descomposición de dominio: determinar una división de los datos (en muchos casos, de igual tamaño) y luego asociarle la computación (típicamente, cada operación con los datos con que opera)

Esto da un n° de tareas, donde c/u comprende algunos datos y un conjunto de operaciones sobre ellos.
Una operación puede requerir datos de varias tareas, y esto llevará a la comunicación.

Son posibles distintas particiones, basadas en diferentes estructuras de datos.Algunas reglas indican tratar primero la estructura de datos más grande, o la más accedida.

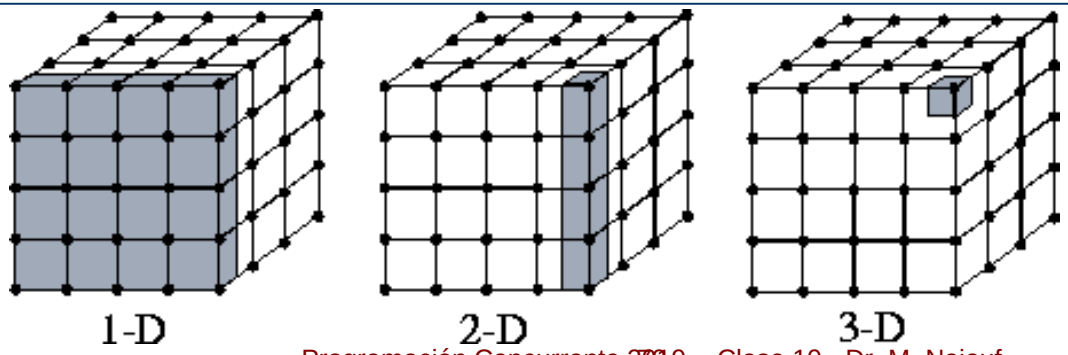
Ej: Binaria, RCB, RGB, RSB, ORB, cíclica, greeedy

Diseño de algoritmos paralelos

Particionamiento

Ej: descomposición para un problema que incluye grilla 3D (ej, estado de atmósfera, o espacio 3D en imágenes). Cómputo repetidamente en c/ punto de la grilla. Son posibles descomposiciones en *x*, *y*, y/o *z*. En las primeras etapas se favorecen las descomposiciones más agresivas, en este caso una tarea por cada punto.

Cada tarea mantiene los valores asociados con sus puntos, y es responsable del cómputo para actualizar esos valores.



Diseño de algoritmos paralelos

Particionamiento

Descomposición funcional: primero descompone la computación en tareas disjuntas y luego trata los datos.

Los requerimientos de datos pueden ser disjuntos (partición completa) o superponerse significativamente (necesidad de comunicación para evitar replicación de datos). En el 2do caso, probablemente convenga descomponer el dominio.

Al apuntar a las computaciones, se puede descubrir la estructura del problema y posibilidades de optimización no obvias en el análisis de datos (ej: búsqueda de soluciones en un árbol)

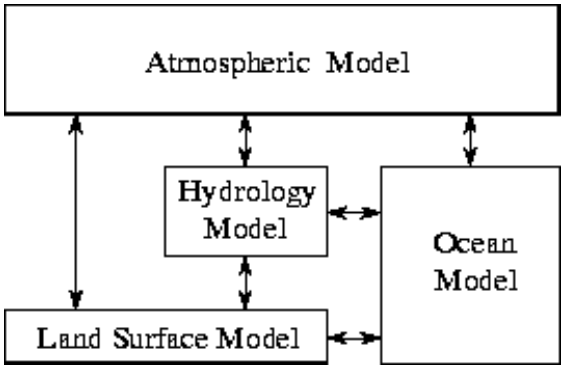
Inicialmente se busca no replicar computación y datos. Esto puede revisarse luego p/ reducir costos

Diseño de algoritmos paralelos

Particionamiento

La descomposición funcional tiene un rol importante como técnica de estructuración del programa, p/ reducir la complejidad del diseño general. Caso de modelos computacionales de sistemas complejos, que pueden estructurarse como conjuntos de modelos más simples conectados por interfases.

Ej: simulación climática con componentes representando atmósfera, océano, hidrología, hielo, etc. C/ componente puede paralelizarse usando DD, pero el algoritmo paralelo completo es más simple si el sistema se descompone por DF



Diseño de algoritmos paralelos

Comunicación

Marca las transferencias de datos entre tareas

Pueden marcarse dos etapas: definir una estructura de canales que enlacen las tareas que cooperan y especificar los mensajes que viajan sobre ellos

En problemas de DD puede ser difícil de determinar. En algoritmos obtenidos por descomposición funcional es casi directo: corresponde a flujos de datos entre tareas.

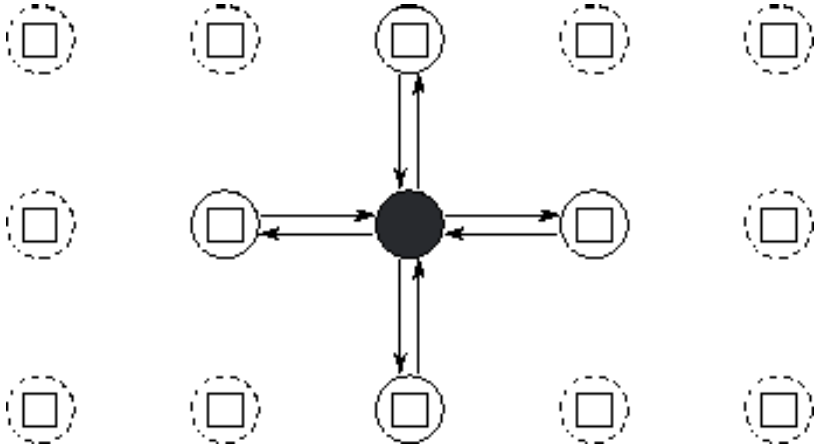
Distintas clasificaciones para patrones de comunicación:
- local/global
- estructurada/no estructurada
- estática/dinámica
- sincrónica/asincrónica

Diseño de algoritmos paralelos

Comunicación

Comunicación local. Ej: comunicación en una computación numérica (método de diferencias finitas de Jacobi).

$$x_{i,j}^{(t+1)} = \frac{4x_{i,j}^{t} + x_{i-1,j}^{t} + x_{i+1,j}^{t} + x_{i,j-1}^{t} + x_{i,j+1}^{t}}{8}$$

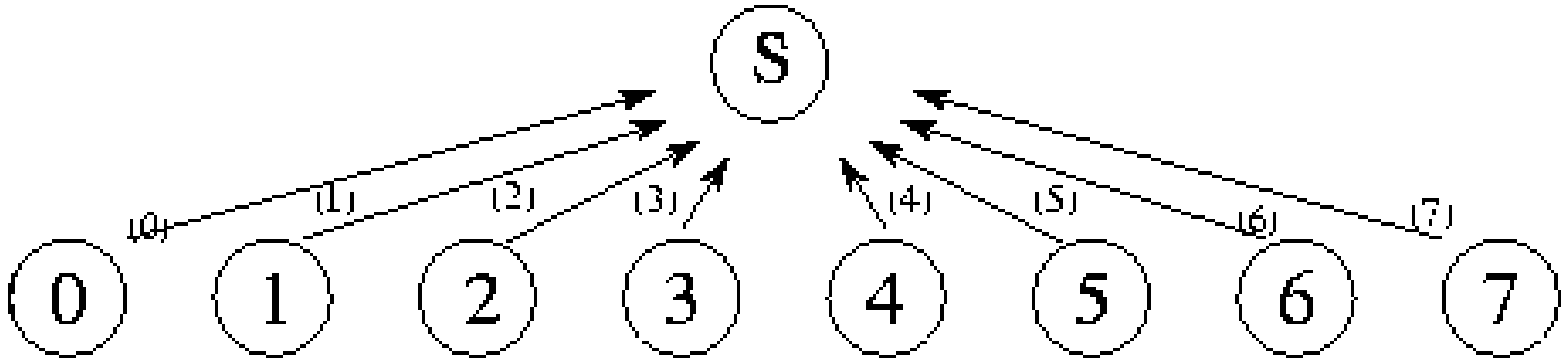


Diseño de algoritmos paralelos

Comunicación

Comunicación global. Ejemplo: operación de reducción paralela

$$S = \sum_{i=0}^{N-1} x_i$$

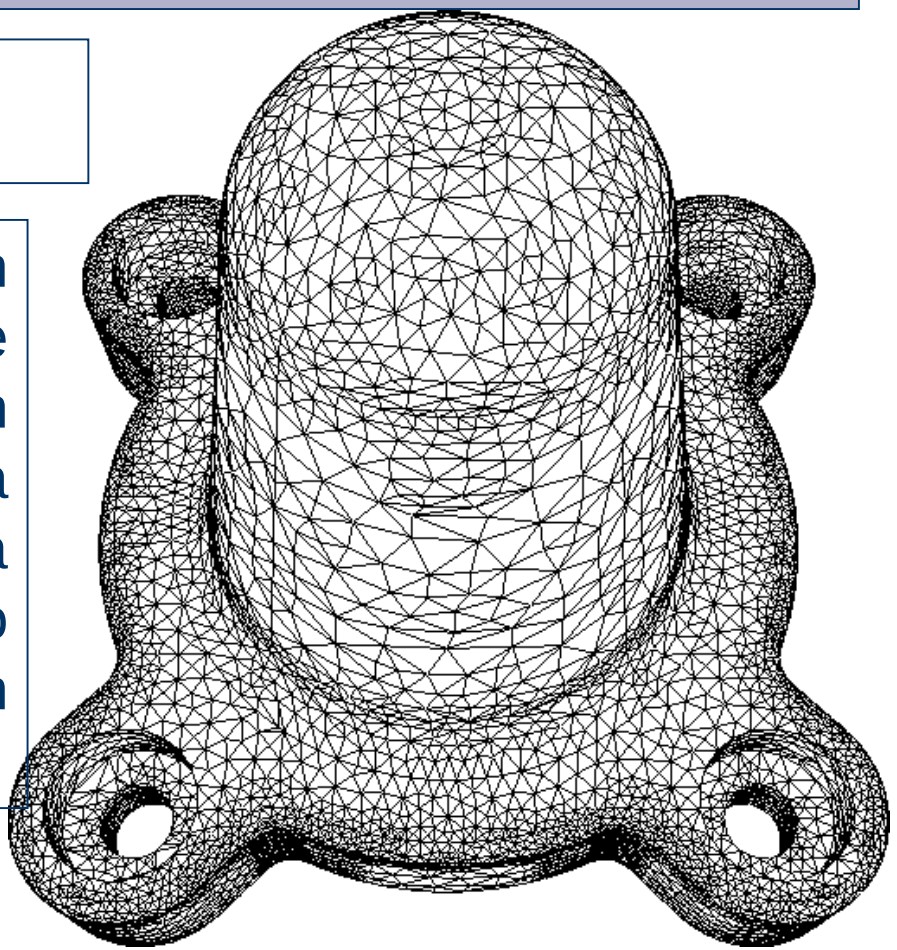


Diseño de algoritmos paralelos

Comunicación

Comunicación no estructurada y dinámica.

Patrones de comunicación complejos. Ej: métodos de elementos finitos usados en cálculos de ingeniería. La grilla puede cambiar de forma para “seguir” un objeto irregular o para lograr mayor resolución en regiones críticas



Diseño de algoritmos paralelos

Aglomeración

El algoritmo resultante de las etapas anteriores es abstracto en el sentido de que no es especializado para ejecución eficiente en una máquina particular

Esta etapa revisa las decisiones tomadas con la visión de obtener un algoritmo que ejecute en forma eficiente en una clase de máquina real

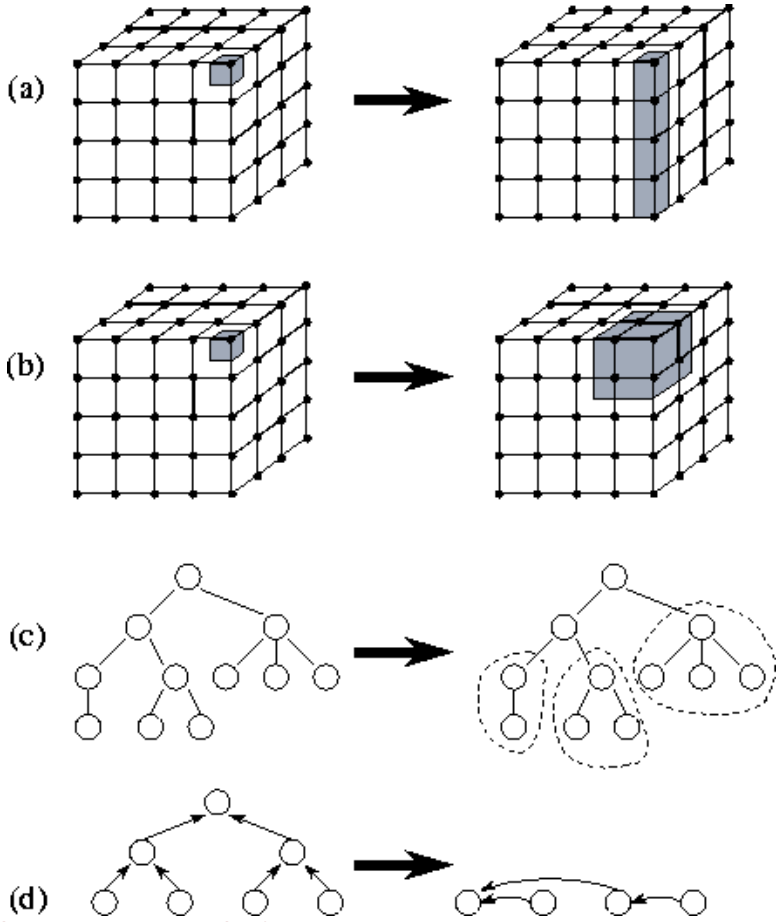
En particular, se considera si es útil combinar o *aglomerar* las tareas para obtener otras de mayor tamaño
También se define si vale la pena *replicar* datos y/o computación

Diseño de algoritmos paralelos

Aglomeración

Al final, la cantidad de tareas puede ser mayor que la cantidad de procesadores

En este caso, el diseño sigue siendo algo abstracto, dado que aún no se resolvió el problema del mapeo de tareas a procesadores.



Diseño de algoritmos paralelos

Aglomeración

3 objetivos, a veces conflictivos, que guían las decisiones de aglomeración y replicación:

Incremento de la granularidad: intenta reducir la cantidad de comunicaciones combinando varias tareas relacionadas en una sola

Preservación de la flexibilidad: al juntar tareas puede limitarse la escalabilidad del algoritmo. La capacidad para crear un número variante de tareas es crítica si se busca un programa portable y escalable.

Reducción de costos de IS: se intenta evitar cambios extensivos, por ejemplo, reutilizando rutinas existentes.

Diseño de algoritmos paralelos

Mapeo

Se especifica dónde ejecuta cada tarea

Este problema no existe en uniprocesadores o máquinas de MC con scheduling de tareas automático

Objetivo: minimizar tiempo de ejecución. 2 estrategias, que a veces conflictúan: ubicar tareas que pueden ejecutar concurrentemente en ≠ procesadores p/ mejorar la concurrencia o poner tareas que se comunican con frecuencia en = procesador p/ incrementar la localidad

El problema es NP-completo: no ∃ un algoritmo de tiempo polinomial tratable computacionalmente para evaluar tradeoffs e/ estrategias en el caso general. Existen heurísticas para clases de problema.

Métricas del paralelismo

En el mundo serial la performance con frecuencia es medida teniendo en cuenta los requerimientos de tiempo y memoria de un programa

En un algoritmo paralelo para resolver un problema interesa saber cuál es la ganancia en performance.

Hay otras medidas que deben tenerse en cuenta siempre que favorezcan a sistemas con mejor tiempo de ejecución

A falta de un modelo unificador de cómputo paralelo, el tiempo de ejecución depende del tamaño de la entrada y de la arquitectura y número de procesadores (*sistema paralelo* = *algoritmo* + *arquitectura sobre la que se implementa*)

Métricas del paralelismo

La diversidad torna complejo el análisis de performance...

Qué interesa medir??

Qué indica que un sistema paralelo es mejor que otro??

Qué sucede si agrego procesadores??

En la medición de performance es usual elegir un problema y testear el tiempo variando el n° de procesadores. Aquí subyacen las nociones de speedup y eficiencia, y la *ley de Amdahl*. Los *modelos de tiempo fijo* son una alternativa.

Otro tema de interés es la *escalabilidad,* que da una medida de usar eficientemente un número creciente de procesadores

Las medidas de performance standard (MIPS, Mflops) se refieren a valores pico, difíciles de alcanzar o mantener en 1 aplicación...

Métricas del paralelismo

Tamaño del problema (W)

Función del tamaño de la entrada. Está dado por el número de operaciones básicas necesarias para resolver el problema en el algoritmo secuencial más rápido.

Es incorrecto pensar, por ej, que en problemas con matrices de *n*x*n* el tamaño de problema es *n* pues la interpretación cambiaría de un problema a otro.

Por ej, duplicar el tamaño de la E/ resulta en un incremento de 8 veces en el tiempo de ejecución serial p/ la multiplicación y de 4 veces p/ la suma.

El *tiempo de ejecución paralelo,* para un sistema paralelo dado, es función del tamaño del problema y el n° de procesadores ($T_p(W,p)$)

Métricas del paralelismo

Speedup (S)

S es el cociente entre el tiempo de ejecución serial del algoritmo serial conocido más rápido (T_s) y el tiempo de ejecución paralelo del algoritmo elegido (T_p) $S = T_s / T_p$

Las diferentes definiciones de tiempo de ejecución serial llevan a distintas definiciones de speedup.

Rango de valores: en general entre 0 y p

La gráfica (*curva de speedup*) refleja *p* en las abcisas y S en las ordenadas

Speedup lineal o perfecto, sublineal y superlineal

Métricas del paralelismo

Eficiencia (E)

Cociente entre speedup y número de procesadores

$E = T_s / pT_p$

Mide la fracción de tiempo en que los procesadores son *útiles* para el cómputo

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

No debieran usarse speedup y eficiencia como métricas independientes del tiempo de corrida.

Métricas del paralelismo

Amdahl: "para un dado problema existe un máximo speedup alcanzable independiente del número de procesadores"

Ejemplo: Sort.

# Proc.	Tiempo	T ideal	Speedup	Eficiencia
1	86.7	86.7	1	100%
2	49.7	43.35	1.94	97%
4	27.8	21.68	3.13	78%
8	17.0	10.84	5.09	64%
16	10.6	5.42	8.19	51%
32	7.5	2.71	11.65	36%

Por qué? $T1 = Tpar + Tsec$

$Tn = Tpar/n + Tsec$

Si Tpar=90 y Tsec=10, y n=90, Tn = 1 + 10 = 11
y el Speedup alcanzado 100/11 = 9.1

Métricas del paralelismo

Factores que limitan el Speedup (Overhead paralelo)

-entrada/salida (el % es alto respecto de la computación)
- algoritmo (no adecuado, necesidad de rediseñar)
- excesiva contencíón de memoria (es necesario rediseñar el código p/tener localidad de datos)
- tamaño del problema (puede ser chico, o fijo y no crecer con *p*)
- desbalance de carga (produciendo esperas ociosas en algunos procesadores)
- alto porcentaje de código secuencial (*Amdahl*)
- overhead paralelo (ciclos adicionales de CPU para crear procesos, sincronizar, etc)

***Función de overhead*: $T_o(W,p) = pT_p - W$**

Suma todos los overheads en que incurren todos los procesadores debido al paralelismo

Métricas del paralelismo

Costo

El costo de un sistema paralelo es el producto de T_p y p.

Refleja la suma del tiempo que cada procesador utiliza en la resolución del problema

Puede expresarse la eficiencia como el cociente entre el tiempo de ejecución del algoritmo secuencial conocido más rápido y el costo de resolver el problema en *p* procesadores

También suele referirse como *trabajo* o *producto procesador-tiempo*.

Métricas del paralelismo

Grado de concurrencia o paralelismo

$C(W)$ es el número máximo de tareas que pueden ejecutarse simultáneamente en cualquier momento del algoritmo paralelo

Para un W dado, el algoritmo paralelo no puede usar más de $C(W)$ procesadores.

$C(W)$ depende sólo del algoritmo, no de la arquit.

Es una función discreta del tiempo. La gráfica de $C(W)$ vs. tiempo es el *perfil de concurrencia*.

Supone un número ilimitado de procesadores y otros recursos, lo que no siempre es posible de tener

Noción de granularidad

Granularidad

Cuando el nro de procesadores crece, normalmente la cantidad de procesamiento en c/u disminuye y las comunicaciones aumentan. Esta relación se conoce como *granularidad*

Puede definirse la granularidad de una aplicación o una máquina paralela como la relación e/ la cantidad mínima o promedio de operaciones aritmético-lógicas con respecto a la cantidad mínima o promedio de datos que se comunican

La relación cómputo/comunicación impacta en la complejidad de los procesadores: a medida que son más independientes y realizan más operaciones A-L e/ comunicaciones, también deben ser más complejos

Si la granularidad del algoritmo es diferente a la de la arquitectura, normalmente se tendrá pérdida de rendimiento

Otros temas de interés

Modelos de speedup

Escalabilidad y balance de carga.

Migración de datos y procesos

Modelos de computación paralela

Computación móvil / ubicua / pervasiva / elástica / …

Cloud computing

Asignaturas relacionadas →

Sistemas paralelos / Programación Distribuida y Tiempo Real / Cloud Computing y Cloud Robotics / Taller de Programación sobre GPU / Conceptos y Aplicaciones de Big Data