Paralelización de Bases de Datos Relacionales

- El objetivo es enseñar la paralelización de bases de datos relacionales.
- Para esto primero se describe un modelo de computación paralela que facilita pensar soluciones en paralelo en el esquema SPMD (single program multiple data, explicado más abajo).
- Este modelo, llamado BSP, también sirve de base para entender sistemas de procesamiento de paralelo de grandes volúmenes de datos distribuidos presentes en distintas tecnologías para el modelo de cómputo paralelo Map-Reduce, el cual a su vez puede ser vista como un caso particular del modelo BSP.
- Tecnologías de bases de datos No-SQL tales como MongoDB traen incluido internamente comandos para realizar procesamiento de datos utilizando el modelo Map-Reduce.

Modelo de computación paralela BSP (bulk-synchronous parallel model)

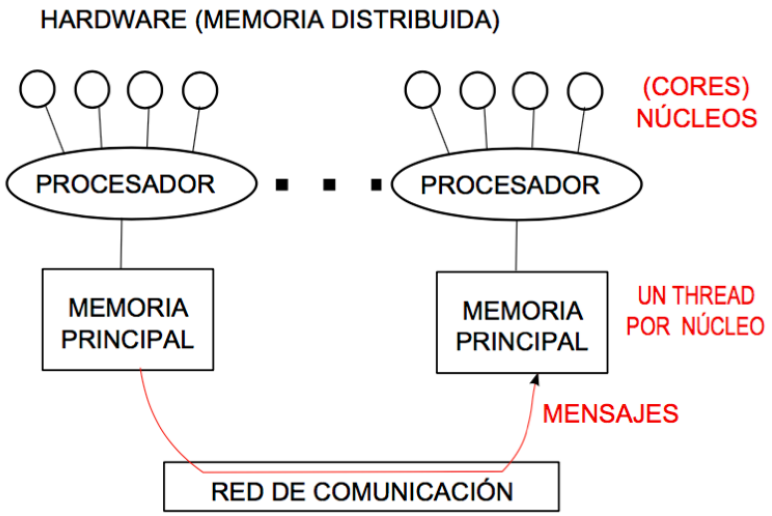
- Es una representación de la realidad que considera las características relevantes del costo de algoritmos paralelos sobre memoria distribuida, es decir

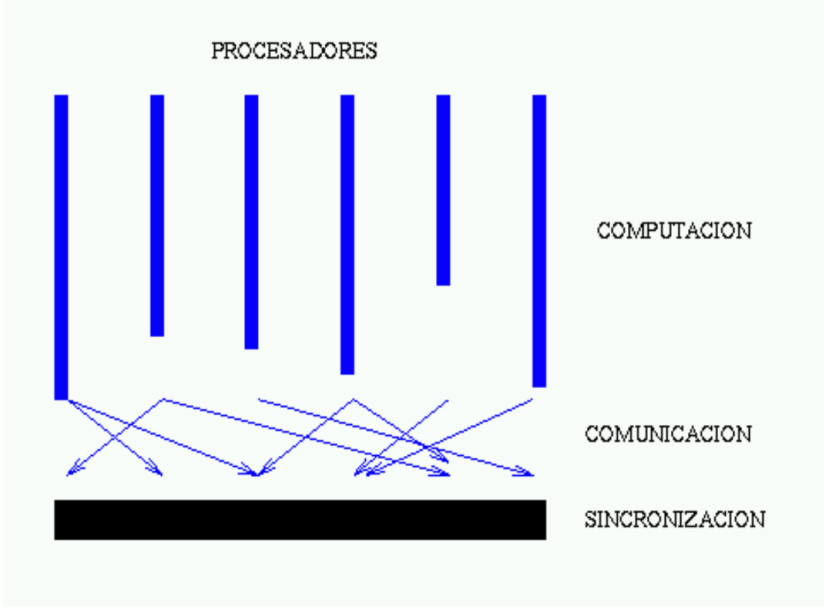
    - Costo de computación (incluido el balance de carga de procesadores).
    - Costo de comunicación de datos entre procesadores.
    - Costo de sincronización de procesadores.

- El modelo impone una manera estructurada de organizar el cómputo lo cual permite asociar un esquema que permite estimar el costo del algoritmo, lo cual permite comparar distintos diseños algoritmicos para un mismo problema.

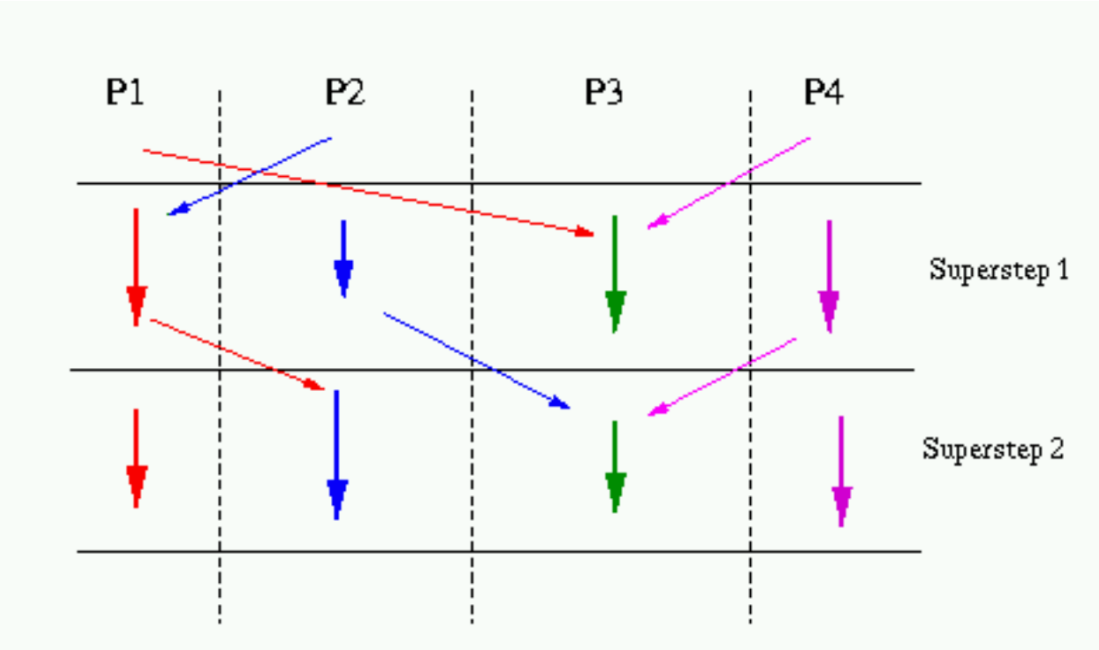
- Se asume una arquitectura de memoria distribuida para el conjunto de procesadores.

- También existe una extensión de BSP para hacer el diseño y análisis de algoritmos para memoria compartida (multithreading), en este caso el modelo recibe el nombre de Multi-BSP.





En general, un programa en BSP estará compuesto de una secuencia de supersteps, cada uno iniciado luego de finalizar el anterior como se muestra en la siguiente figura para cuatro procesadores *P1*, *P2*, *P3*, y *P4*.

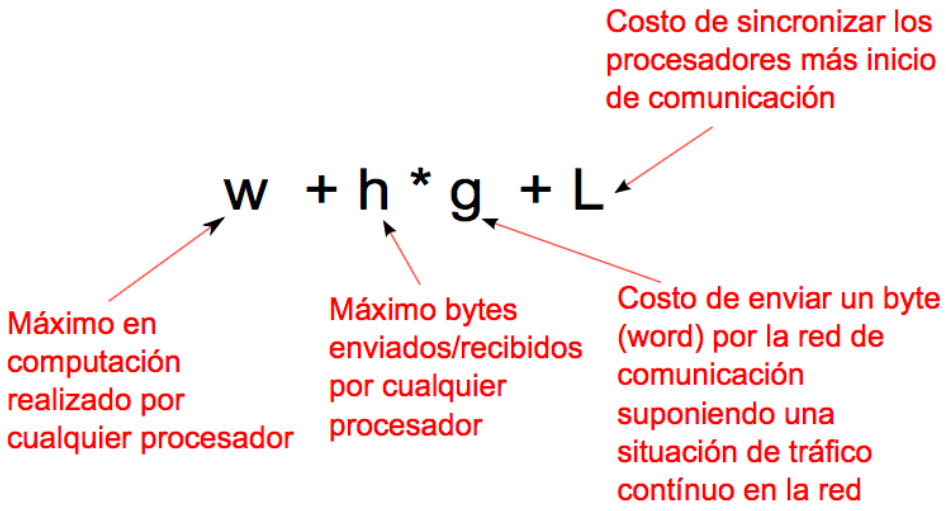


Note que la estructura del modelo facilita la predicción del desempeño de programas BSP. El costo de un programa esta dado por la suma del del costo de todos sus supersteps, donde el costo de cada superstep esta dado por

- la suma del costo originado por las computaciones sobre datos locales (el máximo sobre los procesadores),
- el costo de las comunicaciones de mensajes entre procesadores (una función del máximo enviado/recibido sobre los procesadores), y
- el costo asociado a la sincronización de los procesadores.

- El modelo de costo tiene tres parámetros:
  - P = Numero de procesadores.
  - G o g = Latencia de la red, la cual representa el costo de enviar un mensaje entre dos procesadores.
  - L o ℓ = Costo de sincronizar los procesadores e iniciar la comunicación entre ellos.

El modelo de costo de un superstep es:



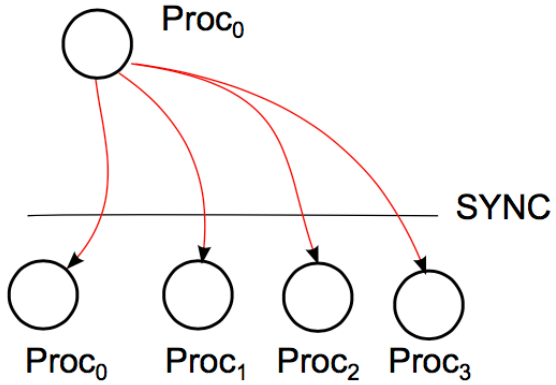
- El costo total de un programa BSP es la suma del costo de todos los supersteps.
- Para la evaluación de los algoritmos BSP suponemos costo asintótico $O(P-1) = O(P)$.
- Para simplificar la explicación de los algoritmos suponemos que a veces los procesadores se envían mensajes as si mismos.
- Generalmente la letra "N" denota el tamaño del problema. El costo de un algoritmo es una función de N, P, g y L.

$$Costo_{BSP} = f(N, P, g, L).$$

- Los parámetros g y L representan el costo del hardware. Crece con P.
- Un computador real puede ser visto como un punto en el espacio (P, g, L).
- Los valores de g y L se determinan mediante benchmark. Ejecutando programas de prueba sobre el hardware. Generalmente $g = O(\log P)$ y $L = O(\log P)$.

Algoritmos Fundamentales

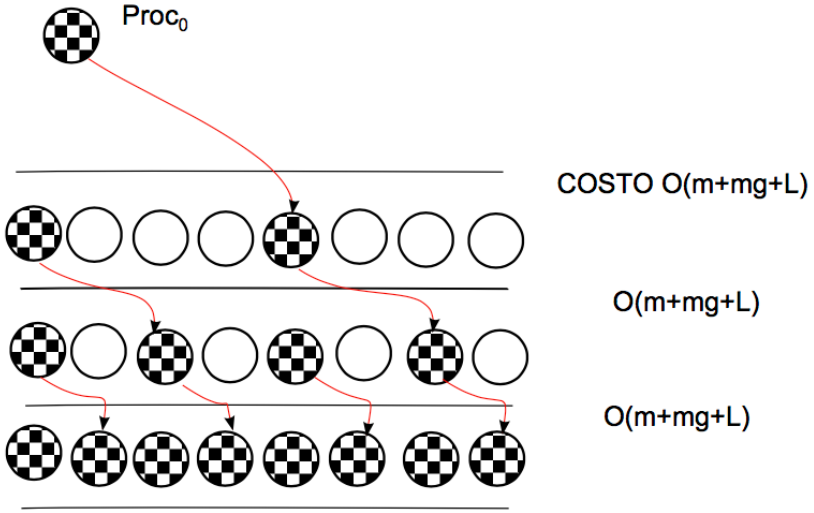
Broadcast I: Un procesador le envía un dato a todos los otros procesadores.



Dato de tamaño **m.**
El procesador P_0 le envía una copia del mensaje de tamaño **m** a los P procesadores (inclusive a sí mismo)

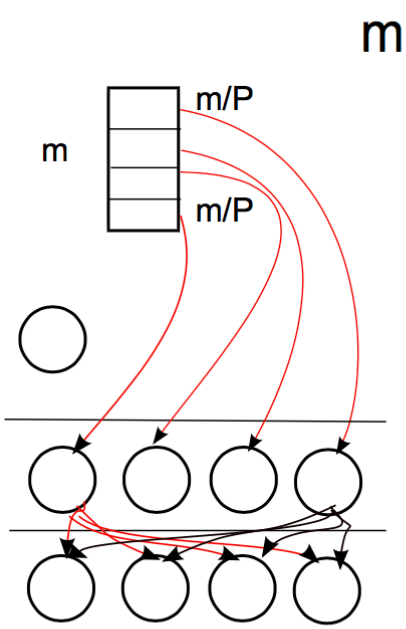
COSTO = mP + mPg + L = O(mP(1+g)+L)
ESCALABILIDAD --> O(P)

Broadcast II: Escalabilidad $O(\log P)$. Se forma un árbol balanceado de procesadores.



COSTO TOTAL O(log(P)(m+mg+L))

Broadcast III: Escalabilidad $O(cte)$.

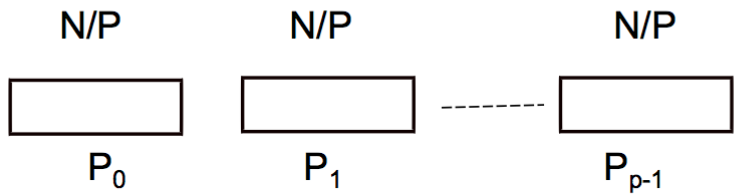


El mensaje se divide en P partes de tamaño m/P. Cada parte se envía a un procesador y luego cada procesador envía su parte a todos los otros.

COSTO = (m/P)P+(m/P)Pg+L

COSTOT TOTAL O(m(1+g)L)

Cálculo del mínimo/máximo/suma/etc.: De un conjunto de N enteros uniformemente distribuidos en P procesadores.



SStep 1

- Cada procesador calcula el mínimo local en paralelo. COSTO O(N/P)
- Cada procesador envía su mínimo local al procesador P_0 COSTO O(Pg+L)

SYNC

SStep 2 — PID = Identificador de procesador

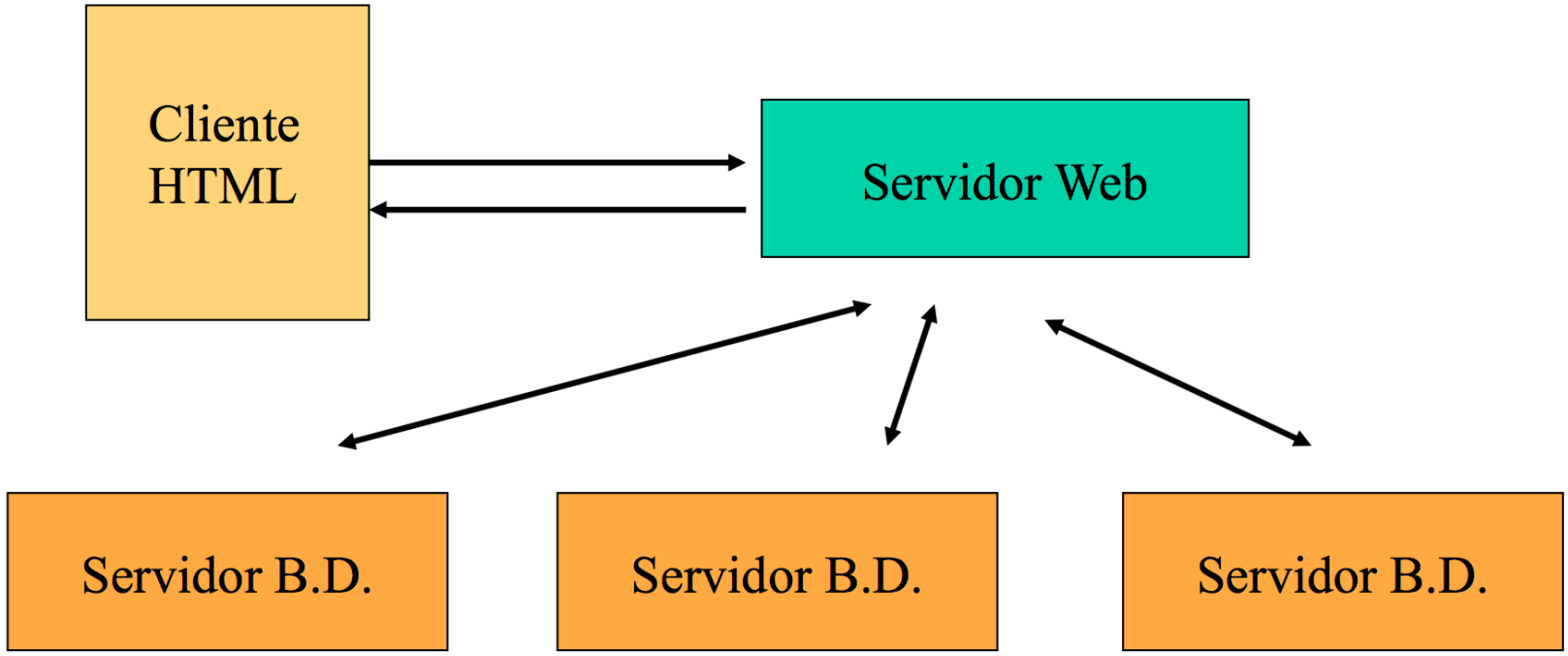
```
if (PID==0)
{
    • Recibe P mensajes                O(P)
    • Calcula el mínimo Global
}
```

SYNC

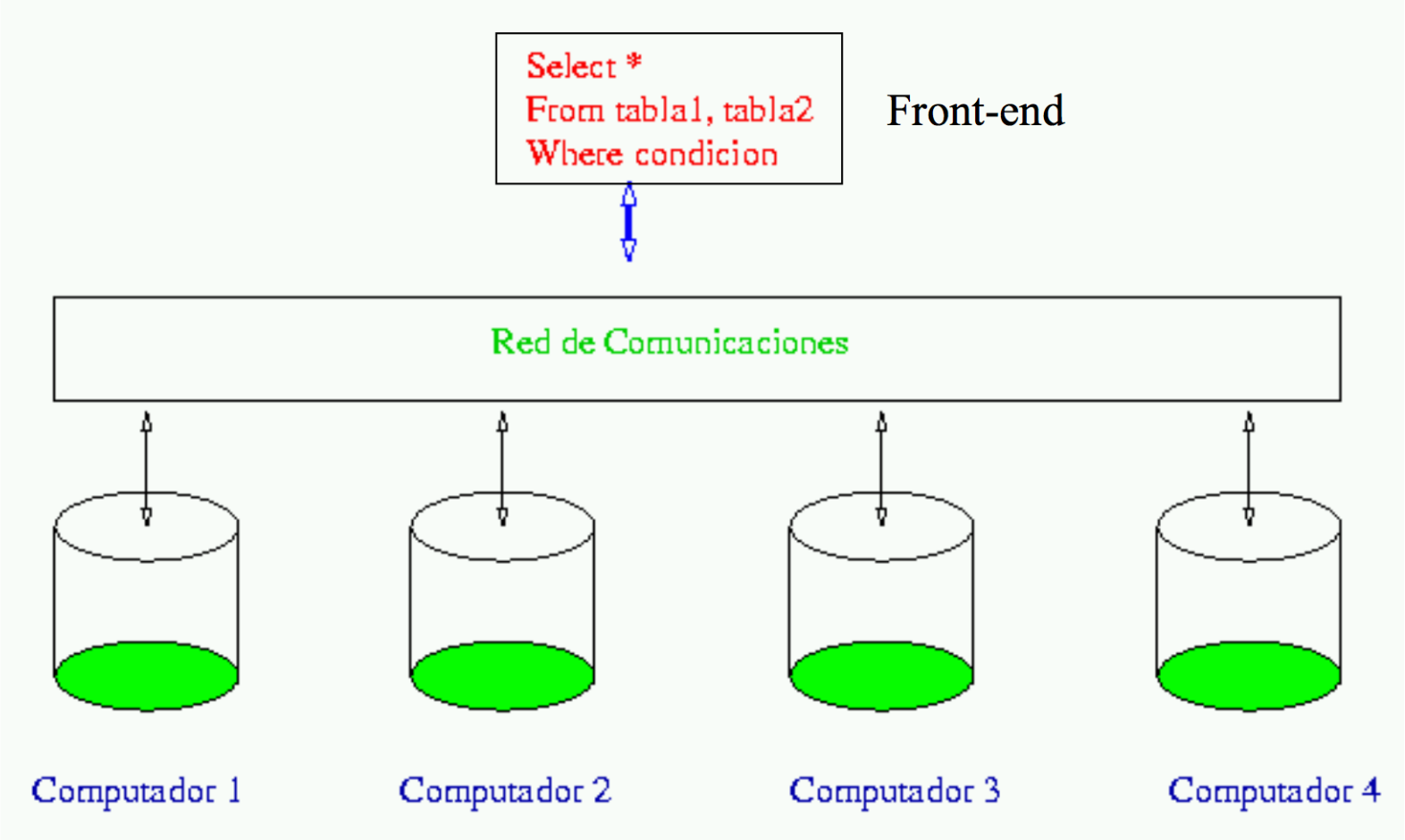
COSTO TOTAL O(N/P + P + Pg + L)

SELECCIÓN DEL MÍNIMO

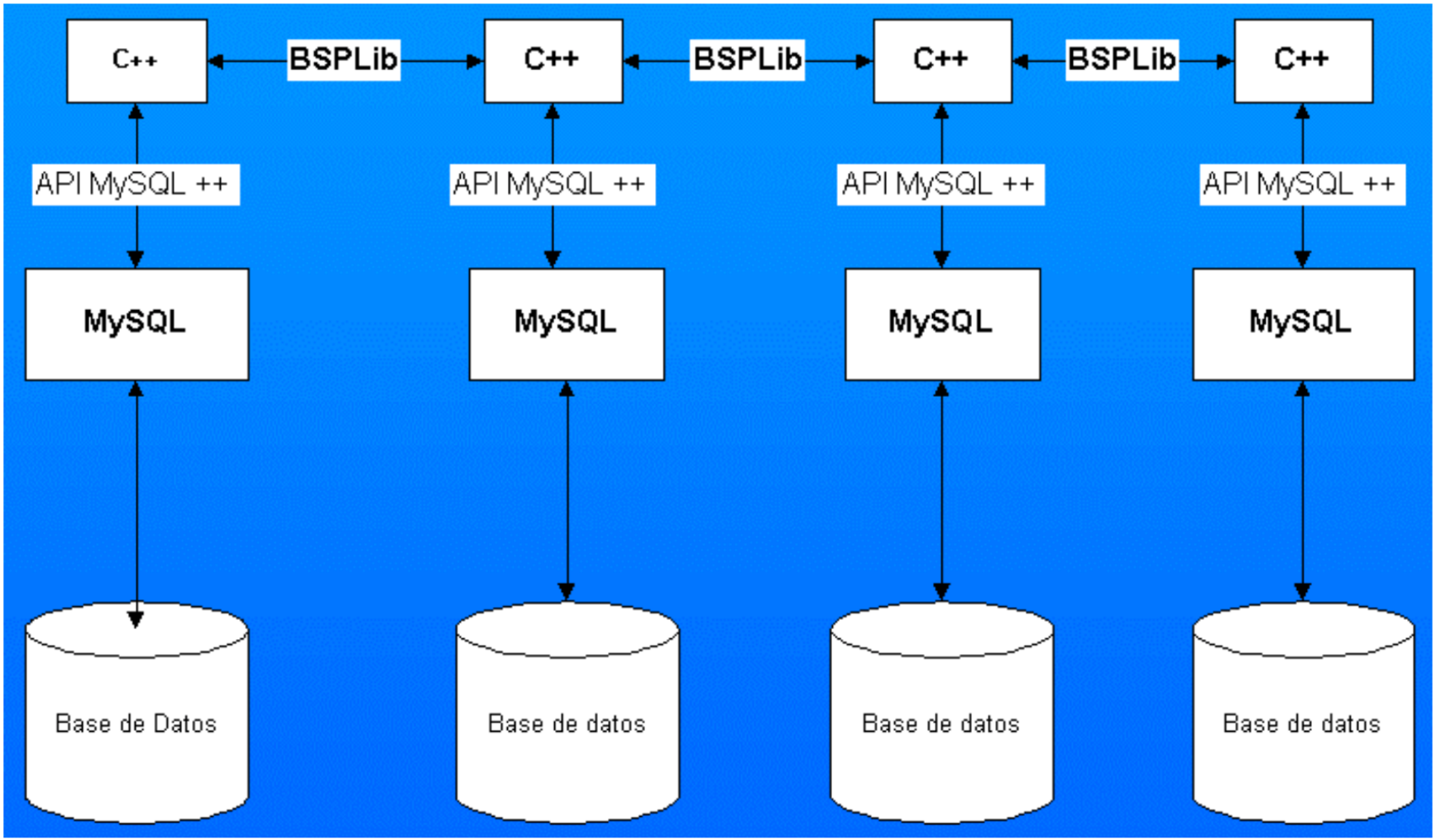
Bases de Datos Relacionales



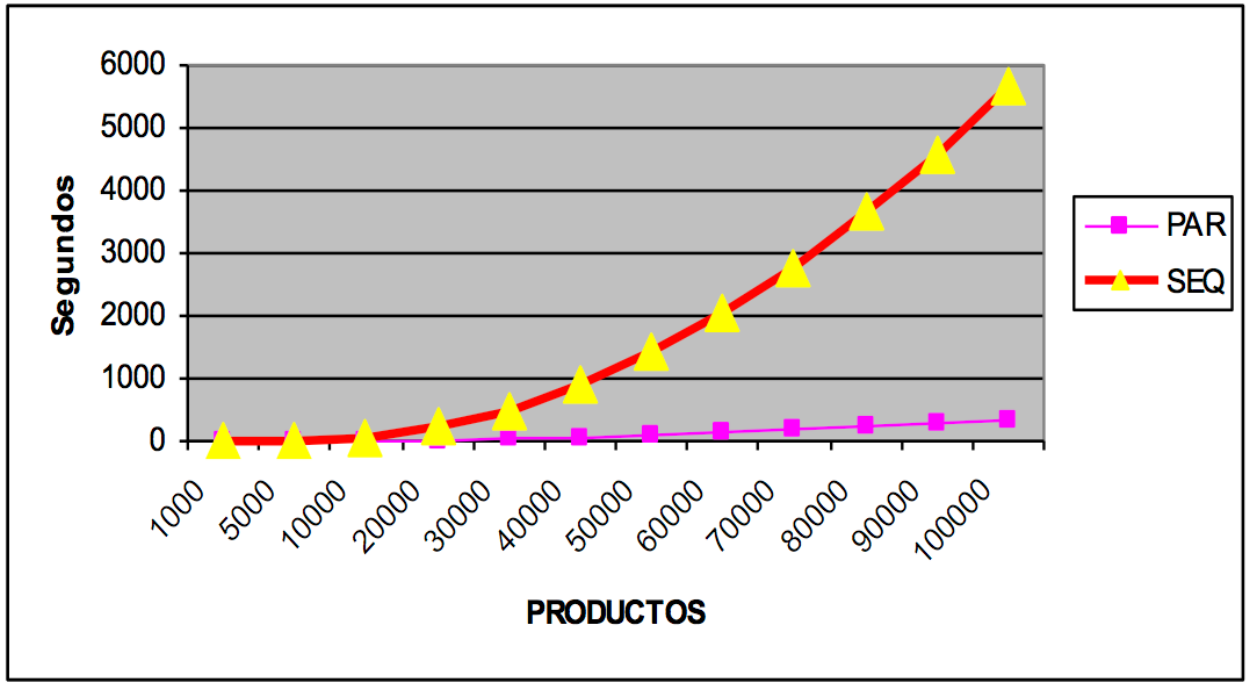
La base de datos esta distribuida en varios computadores



Solución basada en software de dominio público



Consultas a un servidor de Libros
"Cantidad vendida por cada tema"



350 versus 5600 Seg => 16 veces más rápido con 4 máquinas

Sin pérdida de generalidad, explicaremos las estrategias para responder consultas SQL en paralelo mediante ejemplos.

Se asume que cada procesador mantiene el mismo esquema de tablas y son las tuplas las que se distribuyen en los P procesadores (pid=0, pid=1, ..., pid= P-1).

En general son las consultas las que definen la mejor manera de distribuir las tuplas en los P procesadores.

Algunas decisiones de distribución de tuplas pueden afectar significativamente el costo de las consultas, especialmente en el componente de costo de comunicación (incluso puede ser necesario transferir desde un procesador a otro tablas completas).

Para distribuir las tuplas en los procesadores necesitamos una regla que permita localizar (pid) cualquier tupla sin ambiguedad. Pueden haber varias soluciones dependiendo de los objetivos del diseñador de algoritmos paralelos.

En nuestros ejemplos vamos a suponer la existencia de una función de hashing que dado valor de entrada entrega como salida valores entre 0 y P-1, es decir entrega el pid del procesador donde se encuentra la tupla. Como valor de entrada utilizaremos el valor de la llave de la tabla. La función tiene la “habilidad” de distribuir uniformemente las tuplas en los procesadores.

Supongamos las siguientes tablas de una BD relacional

EMPLEADOS(rut, nom, dir)
PERTENECE(rut, depto)
DEPARTAMENTOS(depto, nom, ubicacion)

Consulta: "Departamento donde trabaja Juan Perez”

Solución secuencial

```
select      D.nom
from
            EMPLEADOS  E,  PERTENECE  P,  DEPARTAMENTOS  D
where
            E.nom = "Juan Perez" and
            E.rut = P.rut and
            P.depto = D.depto
```

Estrategia de paralelización: Cada procesador mantiene los mismos esquemas de tablas y las tuplas se distribuyen uniformemente al azar mediante la función de hashing

Hash(llave, P) { return *transformaEntero*(llave) % P ; }

Ejemplo para 2 procesadores, P=2 (pid=0, pid=1)

Pid=0

EMPLEADOS(rut, nom, dir)
(2222-2, N2, R2)
(4444-4, N4, R4)
(6666-6, N6, R6)
(8888-8, N8, R8)

DEPTOS(depto, nom, ubic)
(D2,N2,U2)
(D4,N4,U4)

PERTENECE(rut*, depto)
(2222-2, D1)
(4444-4, D4)
(6666-6, D1)
(8888-8, D3)

Pid=1

EMPLEADOS(rut, nom, dir)
(1111-1, N1, R1)
(3333-3, N3, R3)
(5555-5, N5, R5)
(7777-7, N7, R7)

DEPTOS(depto, nom, ubic)
(D1,N1,U1)
(D3,N3,U3)

PERTENECE(rut*, depto)
(1111-1, D2)
(3333-3, D3)
(5555-5, D2)
(7777-7, D4)

Caso en que las tuplas de PERTENECE se distribuyen mediante Hash(depto, P)

Pid=0	Pid=1
PERTENECE(rut, depto*)	PERTENECE(rut, depto*)
(1111-1, D2)	(2222-2, D1)
(5555-5, D2)	(6666-6, D1)
(4444-4, D4)	(3333-3, D3)
(7777-7, D4)	(8888-8, D3)

Solución paralela “Departamento donde trabaja Juan Perez”

Procesador Maestro:

```
        RecibeMensaje( msg ); // recibe desde front-end
        for( int i= 0;  i<P;  i++ )
            BSP_send( i, msg ) // Broadcast “Juan Perez”
```

FIN operacion

Por cada procesador BSP: solución 1, PERTENECE -> hash(depto,P)

```
BSP_receive( msg );
T = tabla{ tupla(msg.nom) };
rut =   select E.rut
        from EMPLEADOS E, $T
        where E.nom = $T.nom;
if (rut != vacio() )
  BSP_broadcast( rut );

BSP_sync(); // fin superstep

BSP_receive( msg );
T = tabla{ tupla(msg.rut) };
nom =   select  D.nom
        from DEPARTAMENTOS D, PERTENECE P, $T
        where   P.rut= $T.rut and
                D.depto = P.depto
if (nom != vacio() )
  sendMaestro( nom );

BSP_sync(); // fin superstep
```

Por cada procesador BSP: solución 2, PERTENECE -> hash(rut,P)

```
BSP_receive( msg );
T = tabla{ tupla(msg.nom) }; // Juan Perez

rut =   select E.rut
        from EMPLEADOS E, $T
        where E.nom = $T.nom;

if (rut != vacio() ) {

   T = tabla{ tupla($rut) };
   nom = select D.nom
         from DEPARTAMENTOS D, PERTENECE P, $T
         where  P.rut= $T.rut and
                D.depto = P.depto

   sendMaestro( nom );
}

BSP_sync(); // fin superstep
```

Costo BSP de las soluciones considerando P procesadores

E = número de tuplas en EMPLEADOS, E/P en cada procesador.
D = número de tuplas en DEPARTAMENTOS, D/P en cada procesador.
R = número de tuplas en PERTENECE, R/P en cada procesador.

Costo procesador maestro = O(P + P*g + L)

Solución 2 (procesadores esclavos)

costo asintótico: E/P + ((D/P)+(R/P)) + g + L

El término "((D/P)+(R/P))" es debido a que el DBMS posee un optimizador de consultas, en este caso el optimizador reduce el producto cartesiano "PERTENECE x $T" a una sola tupla debido a "P.rut= $T.rut", lo cual requiere de O(R/P) comparaciones dado que consiste en buscar un rut dentro de PERTENECE. Algo similar ocurre con la condición "D.depto = P.depto", lo que equivale a buscar un valor de depto dentro de la tabla DEPARTAMENTOS, a un costo de O(D/P) comparaciones. Además si las tablas poseen índices en los respectivos campos, el costo total mejora a

costo asintótico: log(E/P) + log(D/P) + log(R/P) + g + L

Solución 1 (procesadores esclavos)

log(E/P) + (P + P*g + L) + log(D/P) + log(R/P) + g + L

Ejemplo 2 BDR paralela

Considere una base de datos para gestión de voluntarios donde se realiza la asignación de misiones a voluntarios, y donde cada misión tiene una descripción y un coordinador responsable de liderar la misión.

Para el siguiente esquema de tablas para la base de datos de voluntarios se pide formular consultas en paralelismo BSP-SQL, donde todos los procesadores mantienen los mismos esquemas de tablas y las tuplas están distribuidas en P procesadores mediante pid= Hash(clave, P) tal que "clave" es el primer campo de cada tabla.

```
PERSONAS( rut, nombre, correo, rol, cod )
CONVOCATORIAS( cod, nomMisión, descripción, estado )
INTERES( rut, cod, rol )
MISIONES(rut, cod, rol )
```

PERSONAS(rut, nombre, correo, rol, cod)
CONVOCATORIAS(cod, nomMisión, descripción, estado)
INTERES(rut, cod, rol)
MISIONES(rut, cod, rol)

Las tablas están diseñadas para mantener el registro de las misiones donde participan los voluntarios y coordinadores, y formar grupos por cada misión. El sistema de software de gestión de misiones realiza las siguientes operaciones:

1. Llamado a una convocatoria para una misión a voluntarios y coordinadores enviando un correo de invitación a participar si están disponibles. Luego los interesados postulan.
2. El sistema de gestión recibe las manifestaciones de interés por parte de los interesados en participar en las misiones. Una persona es seleccionada como coordinador si ya está registrado en la base de datos con ese rol. En caso contrario se asume que es un voluntario.
3. El sistema de gestión crea un grupo coordinador-voluntarios para atender una misión que ha sido convocada. Cada misión puede tener un coordinador y hasta *N* voluntarios.

Existe un administrador del sistema quien determina el momento en que se realizan las operaciones de creación del equipo que va a realizar una determinada misión.

1. Llamado a una convocatoria para una misión a voluntarios y coordinadores enviando un correo de invitación a participar si están disponibles. Luego los interesados postulan.

Respuesta pregunta 1

```
Procesador Maestro_Operacion_1:
        RecibeMensaje( msg ); // recibe convocatorias desde front-end
        for( int i= 0;  i<P;  i++ )
            BSP_send( i, msg )
FIN operacion

Por cada procesador Esclavo_Operacion_1:

        BSP_receive( msg ); // recibe nueva convocatoria
        tupla= { msg.cod, msg.nomMision, msg.descripcion, estado="Llamado" };

        if ( Hash(msg.cod, P) == BSP_pid() ) // inserta convocatoria
                SQL::insert into CONVOCATORIAS values ($tupla);
```

Continuación respuesta pregunta 1

```
        tabla = SQL::select rut, nombre, correo, rol
                            from PERSONAS
                            where cod == null // no tiene mision asignada.
                            limit N; // máximo N invitaciones por procesador.


        foreach( tupla T in tabla ){


           if ( T.rol == "Coordinador" )
                texto = formaCuerpoCorreo( msg, "Coordinador" );
           else texto = formaCuerpoCorreo( msg, "Voluntario" );

           head = formaEncabezadoCorreo( T );
           sendCorreo( T.correo, head, texto );
        }
```

FIN operacion

2. El sistema de gestión recibe las manifestaciones de interés por parte de los interesados en participar en las misiones. Una persona es seleccionada como coordinador si ya está registrado en la base de datos con ese rol. En caso contrario se asume que es un voluntario.

Respuesta pregunta 2

Procesador Maestro_Operacion_2:

```
        RecibeMensaje( msg ); // recibe interés en misión desde front-end
        BSP_send( Hash(msg.rut,P), msg );
```

FIN operacion

Por cada procesador Esclavo_Operacion_2:

```
        n=BSP_receive( msg );//recibe código de misión, rut y rol (viene desde
                             //la respuesta del usuario al correo (link)
        if (n==0) return;

        tupla= { msg.rut, msg.cod, msg.rol };
        SQL::insert into INTERES values( $tupla.rut, $tupla.cod, $tupla.rol );
```

FIN operacion

3. El sistema de gestión crea un grupo coordinador-voluntarios para atender una misión que ha sido convocada. Cada misión puede tener un coordinador y hasta *N* voluntarios.

Respuesta pregunta 3

(el administrador inicia desde front-end la formación de grupos para una misión)

Procesador Maestro_Operacion_3:

```
        RecibeMensaje( msg ); // recibe código misión desde front-end
        for( int i=0; i<P; i++ )
          BSP_send( i, msg );
```

FIN operacion

Procesador Esclavo_Operacion_3:

```
// supone que ya existe al menos 1 coordinador y N voluntarios para la misión

        BSP_receive( msg ); // recibe código de misión
        codigo= msg.cod;

        if ( Hash(codigo, P) == BSP_pid() )
                SQL::update CONVOCATORIAS
                     set estado="Asignada"
                     where cod==$codigo;
```

```
        todos_proc = SQL::select rut, rol, cod
                           from INTERES where cod==$codigo;
        SQL::delete from INTERES where cod==$codigo;

        // cuenta el total de coordinadores y voluntarios en el procesador
        cuentaTotalVC( todos_proc, &nv, &nc );

        BSP_send( 0, new Msg( nv, nc ) );

BSP_sync();

        if ( BSP_pid() != 0 ) BSP_sync();

        array sub_totales = vacio();
        for( int i= 0;  i<P;  i++ ){
           BSP_receive( msg );
           sub_totales.insert( msg.nv, msg.nc );
        }

// determina aplicando una regla de distribución la cantidad de voluntarios
// que aporta cada procesador y el procesador donde está el coordinador.

        distribuyeTotalesVC( sub_totales );

        for( int i=0; i<P; i++) //envía cada par (nv,nc) a cada procesador
           BSP_send( i, new Msg( sub_totales[i].nv, sub_totales[i].nc ) );

BSP_sync();
```

```
// ultimo superstep

        BSP_receive( msg );
        nv= msg.nv; nc= msg.nc;

        tabla = SQL::select rut, rol, cod from $todos_proc
                    where rol=="Voluntario"
                    limit $nv;
        if (nc==1)
          tabla.insert( SQL::select rut, rol, cod from $todos_proc
                              where rol=="Coordinador" limit 1 );

        SQL::insert into MISIONES select rut, rol, cod from $tabla;

        SQL::update PERSONAS
             set cod= $codigo
             where rut in (select rut from $tabla);
```

FIN operacion