Paralelismo y Concurrencia en Sistemas

Work pool en memoria compartida

Dr. Alejandro J. García

e-mail: agarcia@cs.uns.edu.ar

http:// cs.uns.edu.ar /~ajg

Departamento de Ciencias e Ingeniería de la Computación

Universidad Nacional del Sur

Bahía Blanca - Argentina

Carga balanceada

- Un tema importante a tener en cuenta, cuando se realiza cómputo en paralelo, es el balance de la carga que se le asigna cada procesador.
- Si la cantidad de tareas a realizar se conoce de antemano, entonces es posible distribuir las tareas a computar en paralelo en forma balanceada entre los procesadores disponibles.
- En la materia ya hemos visto varios ejemplos de este tipo.

Carga balanceada

Ejemplos:

- Rank-sort: como se conoce de antemano la long del arrego, cada procesador puede ocuparse de un elemento o un grupo de elementos. Ej: con 100 elementos y 20 procesadores
- Mult-matrices: cada procedasor puede ocuparse de una celda, una fila o un grupo de filas
- Jacovi: cada procesador puede ocuparse de una o un grupo de filas.

Carga balanceada

- Sin embargo, existen problemas donde la cantidad de tareas no se conoce de ante mano, porque cada proceso puede crear una nueva tarea dinámicamente.
- Ejemplo: búsqueda (en un file system, en una red, o en páginas vinculadas por "links",etc)
- Las tareas deben entonces ser asignadas dinámicamente a los procesadores, lo cual puede desbalancear la carga.
- Sin embargo, lo que sí se conoce de ante mano es la cantidad de procesadores del sistema …

Carga balanceada

- Una forma de lograr un buen balance de carga, cuando la cantidad de tareas no se conoce de antemano, es tener un trabajador idéntico en cada procesador esperando por tareas.
- Las tareas son tomadas de un "pool" (pileta) de tareas dinámicamente por cada trabajador.
- Se necesita de una estructura de datos auxiliar llamada "work pool" donde se almacenan dinámicamente las tareas a ser resueltas.

Trabajadores replicados (Replicated Workers)

- El work pool es una colección de descriptores de tareas a ser resueltas.
- Cuando un trabajador (*worker*) está libre, saca una tarea del pool y se pone a trabajar en ella.
- Cuando termina el *worker* vuelve a estar libre.
- Al procesar una tarea cada trabajador puede crear nuevas tareas que coloca en el pool.
- Cada trabajador es consumidor y puede ser productor de tareas.
- Esta técnica puede emplearse tanto en Multiprocessors y como en Multicomputers.

El uso total o parcial de este material está permitido siempre que se haga mención explícita de su fuente:
"*Paralelismo y Concurrencia en Sistemas. Notas de Clase*". **Alejandro J. García. Universidad Nacional del Sur. (c) 2002-2010.**

Implementación en multiprocessors

Puntos a considerar:

1) cómo sacar y poner tareas en el pool
2) cómo detectar la terminación
3) contención de memoria

- En multiprocessors el work pool estará en memoria compartida.
- Para sincronizar la toma de tareas se puede usar un CHANNEL (cola).

```
TYPE task_descriptor = .....
VAR workpool: CHANNEL of task_descriptor;
```


Implementación en multiprocessors

```
VAR workpool: CHANNEL of task_descriptor;
  procedure worker(w);
  get_work(w);
  while not terminar do
     procesar(w);
     if (......)
      then put_work(new);
     get_work(w);
  end;
.....
put_work(inical);
FORALL i:=1 TO cantProc DO worker(i);
.....
```

Idea base:

```
Proc get_work(var w)
     w:=workpool;
end;

Proc put_work(w);
   workpool:= w;
end;
```


Condiciones de terminación

- El proceso en paralelo termina cuando:

1) no hay mas tareas en el work pool y
2) ningún worker está trabajando

- Para (1) se puede tener un contador de tareas en el pool: get_work lo decrementa y put_work lo incrementa
- Para (2) hay que asegurarse que no hay workers trabajando, ie, todos están ociosos, esto ocurre cuando todos ejecutaron get_work sobre un workpool vacio.

Cuando "contador = - cant_workers" entonces se envía un mensaje de terminación a todos.

Implementación en multiprocessors

Asumiendo una variable global en el programa principal:

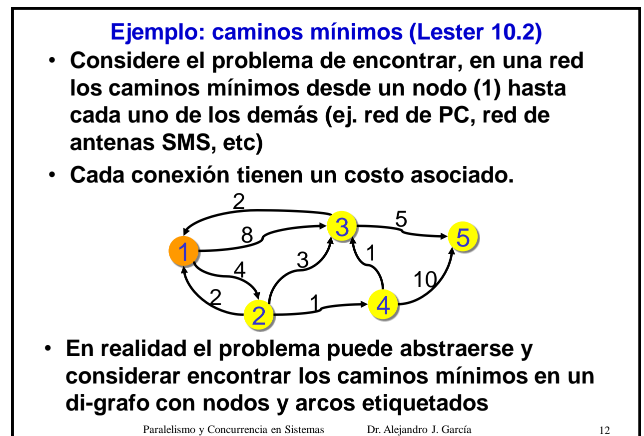
```
VAR Lcount: SPINLOCK;


PROCEDURE PutWork (w:integer; item:integer);
BEGIN
 lock(Lcount);  (*incrementa el contador de works*)
    count:= count + 1;
 unlock(Lcount);
 workpool:=item;
END;
```


Implementación en multiprocessors

```
PROCEDURE GetWork(w:integer; VAR item:integer);
VAR workcount:integer;
BEGIN
lock(Lcount);  workcount:= count - 1;   count:= workcount; unlock(Lcount);
IF workcount = -numworkers (* si todos estan ociosos *)
THEN begin    writeln('soy',w, ' y me di cuenta que todo termino');
      item:=-1; (* no hay trabajo para hacer *)
      for i:=1 to numworkers-1
       do workpool:=item; (* envia los  msg de terminación *)
     end
 ELSE begin
     writeln('soy',w, ' y me tiro a la pileta');
     item:=workpool; (* toma un trabajo de la cola *)
     writeln('soy',w, ' y encontre', item);
   end;
END;
```


Ejemplo: caminos mínimos (Lester 10.2)

- Considere el problema de encontrar, en una red los caminos mínimos desde un nodo (1) hasta cada uno de los demás (ej. red de PC, red de antenas SMS, etc)
- Cada conexión tienen un costo asociado.

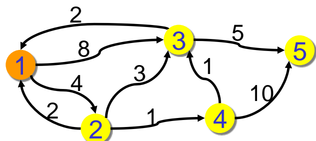


- En realidad el problema puede abstraerse y considerar encontrar los caminos mínimos en un di-grafo con nodos y arcos etiquetados

El uso total o parcial de este material está permitido siempre que se haga mención explícita de su fuente:
"*Paralelismo y Concurrencia en Sistemas. Notas de Clase*". **Alejandro J. García. Universidad Nacional del Sur. (c) 2002-2010.**

Ejemplo: caminos mínimos (Lester 10.2)

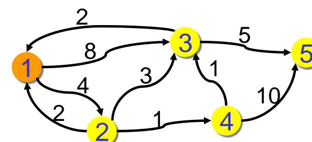
- Supongamos una representación con matriz G de adyacencia para el grafo, que contiene las etiquetas de los arcos (en memoria compartida).
- Ej: G[1,2]=4 G[1,3]=8 G[4,5] = 10 G[1,5]=inf



- mindist[X] tendrá la distancia mínima para llegar desde (1) al vértice X. Está inicializado con un número lo suficientemente grande (“infinito”)

caminos mínimos secuencial

```
Mindist[1]:=0; cola:=1;
while not cola vacia do
   x:= saca de cola
   for w:=1 to N do
      Newd:=mindist[x]+G[x,w];
      if newd < mindist[w]
      then mindist[w]:=newd;
            if w not en cola,
            then poner en cola
   endfor;
endwhile;
```



X	mindist
?	[0, inf, inf, inf,inf]
1	[0, 4, 8, inf, inf]
2	[0, 4, 7, 5, inf]
3	[0, 4, 7, 5, 12]
4	[0, 4, 6, 5, 11]
5	[0, 4, 6, 5, 11]

Referencias

– The Art of Parallel Programming.
Capítulo 10
Bruce P. Lester. 1993. Prentice Hall

El uso total o parcial de este material está permitido siempre que se haga mención explícita de su fuente:
“*Paralelismo y Concurrencia en Sistemas. Notas de Clase*”. Alejandro J. García. Universidad Nacional del Sur. (c) 2002-2010.