Clase Auxiliar I

Prof: Tomas Barros
Aux: Victor Ramiro

16 de abril de 2011

1. P1. Busqueda por fila en una Matriz

Si utilizamos un solo thread, buscar un elemento en el interior de una matriz de nxn tiene complejidad $O(n^2)$. Suponiendo la fortuna de tener n procesadores, se le pide a Ud. determinar si una matriz tiene o no un elementeo en $O(n)$ utilizando tareas en `nSystem`.

Para esto, implemente los siguientes métodos:

- `int BuscarMatrix(int num, int a[][MAX]);` Es el método que crea una tarea por fila para realizar la búsqueda. Además debe retornar verdadero si alguna tarea encontró el elemento buscado `num`.
- `int BuscarFila(int num, int a[][MAX], int k);` Este método corresponde al `main` de cada tarea, y debe buscar el elemento `num` en la fila `k` de la matriz `a[][]`.

Suponga que la matriz `a[][]` ya fue inicializada con valores utilizando el método `int initMatrix(int a[][MAX]);` .

2. P2. Búsqueda en un Árbol Binario

Suponiendo la siguiente estructura de datos Node y utilizando `nSystem` implemente:

```
typedef struct Node{
        int inf;
        struct Node *left, *rigth;
} Node;
```

1. `int BuscarSeq(Node *node, int inf);` Creando tantas tareas como nodos para recorrer el árbol en paralelo.
2. `int BuscarSeq(Node *node, int inf, int level);` Creando tareas mientras la profunidad del árbol no supere algún H, y luego recorriendo el árbol recursivamente.
3. `int BuscarSeq(Node *node, int inf, int level, int *pFOUND);` Como el anterior, pero con una optimización: La ejecución de las demás tareas termina cuando alguna encuentra el elemento buscado.

Observación I: Un árbol binario $\neq>$ de búsqueda binaria!

3. P1. Solución

```
#include "nSystem.h"
#define MAX 1000

int initMatrix(int a[][MAX]){
    int i,j;
    for(i=0;i<MAX;i++){
        for(j=0;j<MAX;j++){
            a[i][j]=i+j;
        }
    }
}

int BuscarFila(int  num, int a[][MAX], int k){
    int i;
    for(i=0;i<MAX; i++){
        if(a[k][i]==num) return TRUE;
    }
    return FALSE;
}

int BuscarMatrix(int num, int a[][MAX]){

    nTask T[MAX];
    int rc[MAX];
    int found=FALSE;
    int i;
    for(i=0;i<MAX;i++){
        T[i]=nEmitTask(BuscarFila,num,a,i);
    }

    for(i=0;i<MAX;i++){
        rc[i]=nWaitTask(T[i]);
        found = found || rc[i];
    }
    return found;
}

int nMain(int argc, char **argv){

    int a[MAX][MAX];

    initMatrix(a);
    if(BuscarMatrix(atoi(argv[1]),a)) {
        nPrintf("Found: %s\n",argv[1]);
    }
    else{
        nPrintf("Not Found: %s\n",argv[1]);
    }
}
```

4. P2. Solución

```
/* Creo tantas tareas como nodos para recorrer el arbol en paralelo */
int BuscarSeq(Node *node, int inf) {
  if (node==NULL) return FALSE;
  else if (inf == node->inf) return TRUE;
  else {
    nTask task1 = nEmitTask(BuscarSeq, node->left, inf);
    nTask task2 = nEmitTask(BuscarSeq, node->right, inf);
    return nWaitTask(task1) || nWaitTask(task2);
  }
}

/* Creo tantas tareas como procesadores. Una vez que complete mi nivel
 * de procesadores, recorro secuencialmente */
int BuscarSeq(Node *node, int inf, int level) {
  if (node==NULL) return FALSE;
  else if (inf == node->inf) return TRUE;
  else {
    if (level > 6) {
      return BuscarSeq(node->left, inf, level+1) ||
             BuscarSeq(node->right, inf, level+1);
    }
    else {
      nTask task1 = nEmitTask(BuscarSeq, node->left, inf, level+1);
      nTask task2 = nEmitTask(BuscarSeq, node->right, inf, level+1);
      int r1 = nWaitTask(task1);
      int r2 = nWaitTask(task2);
      return r1 || r2;
    }
  }
}

/* Cuando encuentre el elemento las demas tareas terminan su ejecucion.  */
int BuscarSeq(Node *node, int inf, int level, int *pFOUND) {
  if (node==NULL) return FALSE;
  else if (*pFOUND) return TRUE;
  else if (inf == node->inf) {
    *pFOUND = TRUE;
    return TRUE;
  }
  else {
    if (level > 6) {
      return BuscarSeq(node->left, inf, level+1, pFOUND) ||
             BuscarSeq(node->right, inf, level+1, pFOUND);
    }
    else {
      nTask task1 = nEmitTask(BuscarSeq, node->left, inf, level+1, pFOUND);
      nTask task2 = nEmitTask(BuscarSeq, node->right, inf, level+1, pFOUND);
      int r1 = nWaitTask(task1);
      int r2 = nWaitTask(task2);
```

```
      return r1 || r2;
} } }
```