Complejidad Algorítmica: de la Teoría a la Práctica

I. Dorta, C. León, C. Rodríguez, G. Rodríguez, A. Rojas
Universidad de La Laguna
Edificio de Física y Matemáticas
c/ Astrofísico Fco. Sánchez s/n
38271 La Laguna. Tenerife

Resumen

En este trabajo se presentan las herramientas CALL y LLAC que facilitan el análisis de complejidad de aplicaciones tanto secuenciales como paralelas. La descripción del modo de uso se realiza mediante el estudio de un caso práctico. El ejemplo elegido ha sido el Problema de la Mochila 0/1. La resolución del mismo se aborda mediante la técnica de Ramificación y Acotación, presentando dos opciones de implementación: recursiva y paralela.
La principal aportación del uso de estas herramientas en la docencia es establecer un puente entre el análisis teórico de la complejidad de los algoritmos y el análisis práctico de los parámetros de rendimiento de los programas, simplificando la labor de ejecución, recolección y estudio de resultados computacionales.

1. Introducción

En los planes de estudio vigentes en la Universidad de La Laguna se cuenta con asignaturas de contenidos relacionados con la Programación Paralela y Distribuida [4] que se concretan en las asignaturas optativas: *Programación en Paralelo I* y *Programación en Paralelo II* y *Programación Distribuida*.

En ellas, los alumnos deben realizar prácticas de laboratorio que implican el uso de los dos paradigmas de programación de máquinas paralelas estándar: el Paso de Mensajes y la Memoria Compartida. Ello supone la realización de experimentos que permitan confirmar que el algoritmo paralelo que se ha diseñado proporciona alguna ventaja frente al algoritmo secuencial. El uso de las herramientas CALL y LLAC facilita esta labor.

En este trabajo, se presenta un ejemplo para ilustrar cómo se usarían las herramientas CALL y LLAC en la docencia de "*Programación en Paralelo*". El problema que se ha considerado es el Problema de la Mochila 0/1 [3]. Se ha optado por un problema de optimización combinatoria porque tienen una gran trascendencia en todos aquellos sectores productivos y usuarios de software que se dediquen al desarrollo de aplicaciones de ayuda a la toma de decisiones y optimización de recursos. Dado que el objetivo de este trabajo es mostrar la utilidad del software en la docencia de optimización matemática, sólo se incluyen los conocimientos elementales necesarios para empezar a trabajar con ambos paquetes.

En el apartado siguiente se presentan las herramientas CALL y LLAC. En la tercera sección se define el caso de estudio y se presentan dos aproximaciones para su resolución: secuencial y paralela, se describe cómo se instrumenta el código y cómo interpretar los resultados. Finalmente se exponen las conclusiones y trabajos futuros.

2. Las herramientas CALL y LLAC

El análisis de complejidad de un algoritmo produce como resultado una "*función de complejidad*" que da una aproximación del número de operaciones que realiza. La herramienta CALL [5] permite anotar con dicha fórmula de complejidad el código C que implanta el algoritmo. Por ejemplo, para el clásico producto de matrices cuadradas C = A×B, donde A y B son matrices de dimensión N×N, el código de la llamada al procedimiento se anotaría de la siguiente forma:

```
#pragma cll mp mp[0] + mp[1]*N + \
                mp[2]*N*N + mp[3]*N*N*N
  MatrixProduct(A, B, C, N);
#pragma cll end mp
```

Donde C_0 = mp[0], C_1 = mp[1], C_2 = mp[2] y C_3 = mp[3] son las constantes asociadas a la fórmula de complejidad del algoritmo ($C_0 + C_1 N + C_2 N^2 + C_3 N^3$).

Los valores de dichas constantes, para una arquitectura dada, se calculan mediante regresión lineal por LLAC. LLAC es una herramienta estadística basada en R [2] que analiza los datos aportados por la ejecución del código instrumentado mediante CALL.

3. Caso de estudio: El Problema de la Mochila 0/1

Consideremos la siguiente definición del Problema de la Mochila 0/1:

"Se dispone de una mochila de capacidad C y de un conjunto de N objetos. Se supone que los objetos no se pueden fragmentar en trozos más pequeños, así pues, se puede decidir si se toma un objeto o si se deja, pero no se puede tomar una fracción de un objeto. Supóngase además, que el objeto k tiene beneficio b_k y peso p_k, para $k=1,2,...,N$. El problema consiste en averiguar qué objetos se han de insertar en la mochila sin exceder su capacidad, de manera que el beneficio que se obtenga sea máximo".

Su formulación como un problema de optimización es:

$$max \sum_{k=1}^{N} b_k x_k$$

sujeto a: $$\sum_{k=1}^{N} p_k x_k \leq C$$

$$x_k \in \{0,1\} \quad k \in \{1,...,N\}$$

La Ramificación y Acotación [1] es un método general que permite resolver un amplio rango de problemas de optimización combinatoria. La solución de un problema consiste en un vector de enteros que cumple un número de restricciones y optimiza una función objetivo. La función objetivo debe ser maximizada o minimizada.

Dado que el área de soluciones contiene un número de elementos exponencial, es imposible explorar todas las soluciones en un tiempo razonable para problemas grandes. La técnica de Ramificación y Acotación intenta reducir el número de soluciones factible por exploración sistemática del área de solución. Los algoritmos de Ramificación y Acotación dividen el área de solución paso por paso y calculan una cota del posible valor de aquellas soluciones que pudieran encontrarse más adelante. Si la cota muestra que cualquiera de estas soluciones tiene que ser necesariamente peor que la mejor solución hallada hasta el momento, entonces no necesitamos seguir explorando esta parte del árbol. Por lo general, el cálculo de cotas se combina con un recorrido en anchura o en profundidad.

En los párrafos siguientes se presentan dos implementaciones, una secuencial en C y una paralela con MPI, de un algoritmo que resuelve el Problema de la Mochila mediante la técnica de Ramificación y Acotación.

3.1. Implementación Secuencial

El Algoritmo 1 muestra el código para resolver el problema de forma recursiva. El número de objetos a insertar en la mochila se almacena en la variable N, en las variables w y p se guardan los pesos y los beneficios de cada uno de ellos, mientras que M representa la capacidad. Puesto que se trata de un problema de maximización se da el valor inicial de $-\infty$ a la variable que almacena la mejor solución encontrada hasta el momento *bestSol* (líneas 1 a 4).

Entre las líneas 8 y 18 se define la función de acotación (*lowerUpper*). Se utiliza la misma función tanto para calcular la cota inferior como la cota superior. La cota inferior se define como el máximo beneficio que se puede obtener para un subproblema dado. Mientras que la cota superior incluye la parte proporcional del beneficio del último objeto que no se pudo insertar en la mochila.

La función *knap* (líneas 20-35) implementa el algoritmo de Ramificación y Acotación de forma recursiva. En la línea 29 se realiza la llamada que estudia la posibilidad de insertar el objeto k, mientras que en la línea 30 se considera su no inclusión. De las líneas 22 a la 26 se implementa la condición que actualiza los valores de la mejor solución (*bestSol*) encontrada hasta el momento.

Estamos interesados en conocer el comportamiento del algoritmo, concretamente la pregunta que nos gustaría responder es ¿cuántos nodos del espacio de búsqueda se visitan para encontrar la mejor solución?.

```
1     /* …ficheros de cabecera */
2     number N, M;
3     number w[MAX], p[MAX];
4     number bestSol = -INFINITY;
5
6     #pragma cll code double numvis;
7
8     void lowerUpper(number k, number C, number P,number *L, number *U) {
9       number i, weig, prof;
10      if (C < 0) {*L = -INFINITY; *U = -INFINITY;  }
11      else {
12        for(i=k, weig = 0, prof = P; weig <= C; i++)
13        {weig += w[i]; prof += p[i];}
14        i--;
15        weig -= w[i]; prof -= p[i];
16        *L = prof;  *U = prof+(p[i]*(C-weig))/w[i];
17      }
18    }
19
20    number knap(number k, number C, number P) {
21      number L, U, next;
22      if (k < N) {
23        lowerUpper(k,C,P,&L,&U);
24        if (bestSol < L) {
25          bestSol = L;
26        }
27        if (bestSol < U) { /* L <= bestSol <= U */
28          next = k+1;
29          knap(next, C - w[k], P + p[k]);
30          knap(next, C, P);
31    #pragma cll code numvis += 2 ;
32        }
33      }
34      return bestSol;
35    }
36
37    int main(int argc, char ** argv) {
38      number sol;
39      readKnap(data);
40    #pragma cll code double numvis = 0.0;
41    #pragma cll kps kps[0]*unknown(numvis) posteriori numvis
42                  /* obj. sig., capacidad rest., beneficio */
43      sol = knap(   0,          M,                0);
44    #pragma cll end kps
45      printf("\nsol = ", sol);
46    #pragma cll report all
47      return 0;
48    }
```

Algoritmo 2. Implementación Secuencial recursiva anotada

Para ello anotamos el código con directivas de CALL. La línea 6 le dice a CALL que se va a definir una variable de tipo `double` para almacenar el número de nodos visitados, *numvis*. A dicha variable se le da inicialmente el valor cero (línea 40). La directiva *code* de CALL proporciona la posibilidad de insertar código fuente al programa anotado. Este código no modifica el programa original, sólo se utiliza en las sentencias CALL. La cadena *cll* que se coloca después de la palabra reservada `#pragma` le indica al compilador de C que se trata de una directiva para el compilador de CALL. Si no se dispone del mismo, simplemente se interpreta como un comentario.

La línea 41 crea un experimento CALL típico. El identificador *kps* especifica su nombre. Este experimento mide el tiempo de ejecución de las sentencias entre las directivas de inicio y finalización (línea 44). El `pragma cll end` va seguido del nombre del experimento y hace que el cronómetro de CALL se pare, se guarden los tiempos y se prepare para la próxima ejecución.

La fórmula que sigue al identificador del experimento *kps* (línea 41) ha de sintetizar la

fórmula de complejidad que nosotros creemos que describe el comportamiento del tiempo. Asociadas a la fórmula se tienen las constantes ksp[0],…,kps[i]. La sintaxis de CALL requiere que se utilice el nombre del experimento para indexar las constantes asociadas con su fórmula de complejidad. Estas constantes pueden ser evaluadas con la herramienta LLAC.

Nuestra intuición nos dice que el tiempo invertido por el algoritmo de Ramificación y Acotación está relacionado con el número de nodos del espacio de búsqueda que se visitan. Así pues, nuestra fórmula se ha de escribir en términos de la variable *numvis* - número de nodos visitados. Sin embargo, el valor final de *numvis* sólo se conocerá una vez resuelto el problema, esto es cuando acabe la llamada a *knap(0,M,0)*. Por lo tanto, en la fórmula de complejidad se indica que dicho valor es desconocido (`unknown`) y que se evaluará al final (`posteriori`).

El número de nodos visitados (*numvis*) se incrementa cuando se tratan los dos nuevos subproblemas que se obtienen a partir del que se esté estudiando en cada momento, esto es, cuando recursivamente se resuelven los problemas que "si" incluyen al siguiente objeto (línea 29) y "no" lo incluyen (línea 30). Este incremento hemos de indicárselo a CALL insertando la directiva correspondiente en la línea 31.

Finalmente, se ha de añadir la directiva `report` después de la especificación de todos los experimentos. Esta directiva le dice al compilador de CALL que genere un fichero con todos los resultados obtenidos durante la ejecución. En nuestro caso, la colocamos justo antes de la sentencia `return` de la función *main* (línea 46). Se ha utilizado el calificador `all` que guarda los resultados de todos los experimentos definidos en el programa. Alternativamente, se puede especificar una lista con los identificadores de los experimentos de los cuales se quieren almacenar los resultados.

Una vez anotado el código fuente con las directivas de CALL se puede compilar con cualquier compilador de C y se seguirá ejecutando exactamente como si no se hubiera modificado. Esto es debido a que el compilador ignora todas las directivas CALL tratándolas simplemente como comentarios.

Supongamos que el Algoritmo 1 está almacenado en el fichero *kpr.c*. Procedemos a procesarlo con el compilador de CALL utilizando el comando:

```
> call kpr.c
```

Como resultado se producen dos ficheros de salida: *kpr.cll.c* y *kpr.cll.h*. Para saber qué código añade CALL al programa original sólo hay que echarle un vistazo a estos ficheros. Ahora se procede a compilar estos ficheros con un compilador de C, indicándole dónde están los ficheros de CALL que es necesario incluir (`/usr/local/CALL/include`):

```
>cc -o kpr kpr.cll.c
```

Al ejecutar el programa resultante (`kpr`) se obtiene un fichero con los resultados del experimento definido: *kpr.c.dat*. La Figura 1 muestra el contenido de este fichero. El programa se ejecutó sobre una mochila de capacidad M = 1.254.202 y número de objetos N = 5000. Fijemos nuestra atención en las dos últimas líneas del fichero. En ellas aparecen los resultados de nuestro experimento. Puesto que se trata de una ejecución secuencial, el número de CPUS usadas es 1. Por defecto, el nombre de la CPU es 0. El número de nodos visitados (*numvis*) ha sido de 261.134 y el tiempo en el que se ha ejecutado la llamada a la función *knap()* ha sido de 0,16 segundos. En el manual de usuario de CALL [5] se puede encontrar una descripción detallada del resto de la información que aparece en el fichero.

```
EXPERIMENT: "kps"
BEGIN_LINE: 115
END_LINE: 119
FORMULA: p 0 p 1 v 0  * +
INFORMULA: kps[0]+kps[1]*numvis
MAXTESTS: 131072
DIMENSION: 2
PARAMETERS:
NUMIDENTS: 1
IDENTS: numvis
OBSERVABLES: CLOCK
COMPONENTS: 1 numvis
POSTFIX_COMPONENT_0: 1
POSTFIX_COMPONENT_1: v 0
NUMTESTS: 1
SAMPLE:
  CPU  NCPUS  numvis    CLOCK
    0      1  261134.0  0.16491100
```

Figura 1. Contenido del fichero knr.c.dat

```
1        busy[0] = 1; for i = 1, nProcs { busy[i] = 0;}

2        idle = nProcs - 1;
3        //Send initial subproblem to first idle slave
4        auxSp = sp.initSubProblem();
5        outputPackect.send(firstIdle,
6                           auxSp,   // initial subproblem
7                           bestSol, // bestSolution
8                           sol);    // current solution
9        idle--;
10       IDLE2WORKING(busy,firstIdle);  // mark this slave like working
11       while (idle < (groupSize-1)) { // while there are working slaves
12          recv(source, flag);
13          while(flag) {
14              if (SOLVE_TAG) {        // receive the final solution
15                inputPacket.recv(source,
16                                 bestSol,  // best solution
17                                 sol);     // current solution
18              }
19              if (BnB_TAG) {              // receive a slave request
20                inputPacket.recv(source,
21                                 high,     // upper bound
22                                 nSlaves); // num. of reuiered slaves
23                if ( high > bestSol){      // problem to branch
24                  total= ((nSlaves <= idle)?nSlaves:idle);
25                  for i = 1, total { idle--; IDLE2WORKING(busy,i); }
26                  outputPacket.send(source,
27                                    total,  // num. of assigned slaves
28                                    bestSol,     // best Solution
29                                    1,..., total // slaves identifiers
30                                   );
31                }
32                else { // the problem must be bounded
33                  outputPackted.send(source, DONE);
34                }
35              }
36              if (IDLE_TAG) {     // signal of an idle slave
37                inputPacket.recv(source, IDLE);
38                idle++;
39                WORKING2IDLE(busy,source); // mark this slave like idle
40              }
41              recv(source, flag);
42          } // while (flag)
43       } // while (idle < (groupSize-1))
44       // Send the ending message
45       for i = 1, groupSize { outputPacket.send(i, END); }
```

Algoritmo 1. Implementación del Maestro

3.2. Resolución Paralela

La versión paralela del algoritmo de Ramificación y Acotación para resolver el problema de la mochila se ha implementado mediante Paso de Mensajes [6] siguiendo un esquema *Maestro/Esclavo*.

El *Maestro* (véase el Algoritmo 2) es el responsable de la coordinación entre tareas, para ello, cuenta con la estructura de datos *busy* donde registra el estado de ocupación de cada uno de los esclavos (línea 1). Al comienzo de la computación todos los esclavos se marcan como ociosos.

El subproblema inicial (*auxSp*), el mejor valor de la función objetivo (*bestSol*) y el mejor vector solución hasta el momento se envía al primer esclavo ociosos (líneas 3-10). Mientras existan esclavos libres el *Maestro* recibe información de ellos y decide la siguiente acción a ejecutar dependiendo de si el problema está o no resuelto (línea 14), si hay una solicitud de esclavos (línea 19) o si los esclavos no tienen nada que hacer (línea 36). Si el problema está resuelto se reciben

```
1    while (1) {
2      recv(source, flag);
3      while (flag) {
4        if (END_TAG){                  // receive the finishing message
5          inputPacket.recv(MASTER, END); return;
6        }
7        if (PBM_TAG){                  // the problem to be branched
8           inputPacket.recv(source,    // source = slave or master:
9                            auxSp,     //   the initial subproblem
10                           bestSol,   //   the best solution value
11                           sol);      //   the current solution
12          auxSol = sol;
13          bqueue.insert(auxSp);       // insert in the local queue
14          while(!bqueue.empty()) {
15            auxSp = bqueue.remove();  // pop from the local queue
16   #pragma cll code numvis++;
17            high = auxSp.upper_bound(pbm,auxSol);    // upper bound
18            if ( high > bestSol ) {
19              low = auxSp.lower_bound(pbm,auxSol);   // lower bound
20              if ( low > bestSol ) {
21                bestSol = low;
22                sol = auxSol;
23                outputPacket.send(MASTER,    // send to the Master:
24                                  SOLVE_TAG,    // problem solved
25                                  bestSol,      // best solution value
26                                  sol);         // solution vector
27              }
28              if ( high != low ) {
29                rSlaves = bqueue.getNumberOfNodes();
30                op.send(MASTER,
31                        BnB_TAG,
32                        high,                 // upper bound
33                        rSlaves);             // num. of slaves req.
34                inputPacket.recv(MASTER,
35                                 nfSlaves,   // num. of slaves assig.
36                                 bestSol,    // updated best solution
37                                 rank {1,..., nfSlaves});
38                if ( nfSlaves >= 0) {
39                  auxSp.branch(pbm,bqueue);   // branch
40                  for i=0, nfSlaves{          // send problems to slaves
41                    auxSp = bqueue.remove();
42   #pragma cll code numvis++;
43                      outputPacket.send(rank,  // send to the slave:
44                                        PBM_TAG, //   tag
45                                        auxSp,   //   problem
46                                        bestSol, //   best solution value
47                                        sol);    //   the solution vector
48                  }
49                } // if nfSlaves == DONE the problem is bounded (cut)
50          } } }
51          outputPacket.send(MASTER, IDLE_TAG);  /idle slave
52        }
53        recv(source, flag);
54      } // while (flag)
55   } // while(1)
```

Algoritmo 3. Implementación del Esclavo anotada

el mejor valor de la función objetivo y el vector solución y se almacenan (líneas 15-18). Cuando el *Maestro* recibe una solicitud de "*nSlaves*" esclavos libres, viene acompañada del valor de la cota superior (*high*). Si el valor de la cota superior es mayor que el valor actual de la mejor solución (*bestSol*) la respuesta al esclavo incluye el número de esclavos libres que pueden ayudar a resolver el problema (*total*) - líneas 26-30. En otro caso, la respuesta indica que no es necesario trabajar en ese subárbol de búsqueda (línea 33). Cuando el número de esclavos libres es igual al valor inicial, el proceso de búsqueda finaliza y el Maestro

notifica a todos los esclavos que dejen de trabajar (línea 45).

Cada *esclavo* (Algoritmo 3) trabaja acotando los problemas que recibe (líneas 8-12). Se generan nuevos subproblemas mediante llamadas a la función de ramificación *branch()* (línea 39). El *esclavo* pide información sobre esclavos libres al *Maestro* (líneas 30-33). Si no hay otros esclavos libres que le ayuden en su tarea, el *esclavo* continúa trabajando localmente. En otro caso, elimina subproblemas de su cola local y los envía directamente a los otros esclavos que le hayan asignado (líneas 39-47).

Al igual que en el caso secuencial, queremos estudiar el número de nodos visitados (*numvis*). Ahora este valor está distribuido entre los *esclavos*, porque el *Maestro* no realiza labores de acotación. Por lo tanto, anotaremos el código paralelo exactamente igual que el secuencial, cambiando sólo el lugar donde se incrementa el número de nodos visitados. Las directivas CALL que lo hacen se han añadido en los puntos en los que se extraen problemas de la cola local de cada esclavo ya sea para estudiarlo (línea 16) o para enviárselo a otro esclavo para que lo resuelva (línea 42).

3.3. Estudio de los resultados

La herramienta LLAC al ser una extensión de R, proporciona la posibilidad de hacer un análisis de las muestras que se obtienen al ejecutar los experimentos generados con CALL. Con las mismas muestras se pueden realizar distintos tipos de representaciones para estudiar cuanto se ajustan la fórmula con la que habíamos anotado el programa y los resultados obtenidos.

Las ejecuciones secuenciales del problema de la mochila se ejecutaron sobre un procesador AMD-Duron a 800 MHz y 256 Mb de memoria. El experimento consistió en generar aleatoriamente diez problemas de la mochila con capacidad en el rango [500, 5000]. La Figura 2, generada con LLAC, muestra los resultados obtenidos. Cada uno de los diez puntos redondos asociado a cada tamaño (500-5000) representa el número de nodos que se visitó para resolver ese problema. Las "x" unidas con una línea punteada representan la media de nodos visitados. La línea continua representa a un polinomio de segundo grado, lo que nos indica que el número medio de nodos visitados se aproxima a una parábola. De la gráfica también se concluye que a mayor número de objetos mayor dispersión en el número de nodos visitados. Este comportamiento creemos que está ligado al generador de problemas aleatorio que se está utilizando.

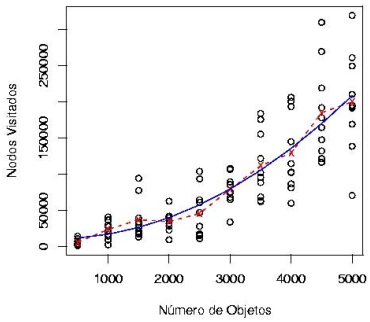

Figura 2. Resultados del Secuencial

Un parámetro muy interesante de estudiar en las implementaciones paralelas de los algoritmos de ramificación y acotación es el "equilibrado de la carga de trabajo" entre los distintos procesadores que intervienen en la ejecución del algoritmo.

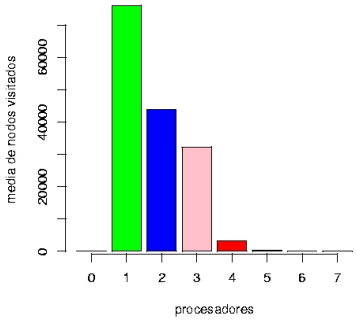

Figura 3. Resultados ejecución paralela

La Figura 3 muestra la distribución media de nodos visitados entre ocho procesadores de cinco ejecuciones de un problema de la mochila generado aletoriamente de tamaño 4000. Se trata de un conjunto de máquinas heterogéneo. Las dos primeras son AMD-Duron a 800 MHz y el resto a 500 MHz, todas con 256 Mb de memoria. Nótese que el procesador cero no visita ningún nodo puesto que se trata del *Maestro*. Creemos que el primero de los esclavos explora la mayor parte del espacio de búsqueda, porque se trata de un procesador más rápido y debido al tamaño del problema, no queda mucho trabajo para los últimos. También hemos notado una gran diferencia entre el número de nodos visitados de una ejecución a otra. La Figura 4 muestra los resultados de los cinco experimentos que dan lugar a la media de la Figura 3. Creemos que el incremento de nodos en la segunda ejecución se debe a que existía algún otro proceso en la máquina uno que consumía muchos recursos. Esto implicaría que la recepción de las cotas que permitirían podar se realiza más tarde, por lo que se explora un espacio de búsqueda mayor.

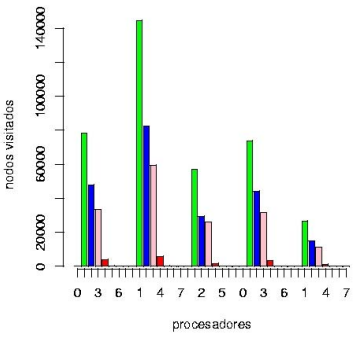

Figura 4. Cinco Ejecuciones Paralelas para N=4000

4. Conclusiones

Se ha presentado a través de un ejemplo el modo de uso de las herramientas CALL y LLAC para el análisis de complejidad de algoritmos.

El ejemplo utilizado forma parte de las prácticas de laboratorio de las asignaturas *Programación en Paralelo I y II* que se imparten en la Ingeniería Superior en Informática en la Universidad de La Laguna. Sin embargo, consideramos que estas herramientas serían de gran utilidad en la docencia de asignaturas que incluyan en sus contenidos la descripción de técnicas algorítmicas. Las herramientas proporcionan la posibilidad no sólo de estudiar implementaciones secuenciales de algoritmos, sino también paralelas.

Además, el algoritmo paralelo propuesto para resolver el problema de la Mochila 0/1, se puede generalizar para resolver mediante Ramificación y Acotación otro tipo de problemas.

Agradecimientos

Este trabajo ha sido financiado parcialmente por los proyectos del ministerio de Ciencia y Tecnología: TIC2002-04498-C05-05 (TRACER) y TIC2002-04400-C03-03 (PELICAN).

Referencias

[1] Dorta I., León C., Rodríguez C., Rojas A. *Parallel Skeletons for Divide-and-Conquer and Branch-and-Bound techniques*. In Proceeding of 11th Euromicro Conference on Parallel, Distributed and Network based Processing, 2003.

[2] Ihaka R., Gentleman R. *R, A language for Data Analysis and Graphics.* Journal of Computational and Graphical Statistics, 5 (3), pp. 299-314, 1996.

[3] Martello S., Toth P. *Knapsack Problems: Algorithms and Computer Implementations*. Jonh Wiley & Sons Ltd, 1990.

[4] Planes de Estudio del CSI. http://www.csi.ull.es

[5] Rodríguez, G. *CALL: a complexity Analysis Tool*. Proyecto Fin de Carrera, Centro Superior de Informática, Universidad de La Laguna, Junio 2002. http://nereida.deioc.ull.es/~call.

[6] Snir M., Otto S., Huss S., Walker D. and Dongarra J. *MPI-the Complete Reference*. 2nd Edition, MIT Press, 1998.