RESOLUCION DEL PROBLEMA DE LA MOCHILA 0/1 USANDO ESQUELETOS DIVIDE-Y- VENCERAS Y RAMIFICACION-Y-ACOTACION

I. Dorta[1], C. León[2], C. Rodríguez[3] y A. Rojas
Departamento de Estadística, Investigación Operativa y Computación
Universidad de La Laguna, Edificio Física/Matemáticas, La Laguna, Tenerife, España

RESUMEN
En este artículo se presenta la librería **MaLLBa**. Esta librería proporciona esqueletos para la resolución de problemas de optimización combinatoria usando técnicas exactas, heurísticas e híbridas. El usuario ha de elegir un paradigma y para el ha de especificar el tipo del problema, el tipo de la solución y las características específicas de la técnica usando el lenguaje de programación C++. Esta información se combina con los esqueletos de resolución que proporciona la librería para obtener un programa secuencial y un programa paralelo. Para explotar el paralelismo en redes de estaciones de trabajo bajo Linux, **MaLLBa** utiliza el paradigma de paso de mensajes. Nos centraremos en la presentación de los esqueletos Divide y Vencerás y Ramificación y Acotación. Concretamente se aplicaran ambos, de forma integrada, en la resolución del problema de la Mochila Entera. Finalmente se presentarán los resultados obtenidos.

ABSTRACT
This paper presents the **MaLLBa** library. This library provides skeletons to solve combinatorial optimization problems using exact, heuristic and hybrid techniques. The user must choose a paradigm and for it must establish the problem and solution types and the specific characteristics of the technique using the C++ language. This information is combined with the skeletons provided by the library to obtain two programs: one sequential and other parallel. To exploit parallelism on Linux workstations, **MaLLBa** uses the Message Passing paradigm. This work is centered to present the Divide an Conquer and Branch and Bound skeletons. Concretely both of them will be applied to solve the Integer Knapsack Problem. Finally, the obtained computational results will be presented.

Key words: Algorithmic Techniques, Algorithmic Skeletons, Branch and Bound, Divide and Conquer.

MSC: 68R99

1. INTRODUCCION

La solución exacta de la mayoría de los problemas combinatorios implica la enumeración de los elementos de un espacio de soluciones exponencial. Fijado un problema, la búsqueda exhaustiva de una solución se puede acelerar a través de técnicas exactas como Divide y Vencerás, Ramificación y Acotación 0 Programación Dinámica. El esquema de trabajo de estas técnicas se puede generalizar para desarrollar herramientas software que permitan el análisis, diseño e implementación de resolutores para problemas concretos. Desafortunadamente, en muchas aplicaciones aparecen casos de tamaño irresoluble, esto es, no es posible encontrar la solución óptima en tiempo razonable. Sin embargo, es posible incrementar el tamaño de los problemas que se pueden abordar por medio de técnicas de paralelización.

En este trabajo se presentan los esqueletos orientados a objeto **MaLLBa**::**DnC** y **MaLLBa::BnB** para la resolución de problemas de optimización combinatoria con las técnicas Divide y Vencerás y Ramificación y Acotación respectivamente. La implementación de los esqueletos se ha realizado en C++. Se proporciona el código secuencial y el código paralelo de la parte invariante de los resolutores para ambos paradigmas. Además se pone a disposición del usuario la plantilla de la parte que ha de rellenar. Las clases que componen dicha plantilla sirven para establecer la relación entre el resolutor principal y el problema instanciado. Una vez que el usuario ha instanciado su problema, obtiene gratis un resolutor paralelo de su algoritmo, sin realizar ningún esfuerzo adicional.

E-mail: [1]isadorta@ull.es
[2]cleon@ull.es
[3]crguezl@ull.es

Los esqueletos proporcionan modularidad al diseño de algoritmos exactos, lo que supone una gran ventaja respecto a la implementación directa del algoritmo desde el principio, no sólo en términos de reutilización de código sino también en metodología y claridad de conceptos.

Se han encontrado en la revisión bibliográfica varias herramientas para la implementación paralela de algoritmos generales de Ramificación y Acotación PPBB (Portable Parallel Branch-and-Bound Library) (Tschöke-Polzer (1995)) y PUBB (Parallelization Utility for Branch and Bound algorithms) (Shinano **et al**., (1995)) proponen implementaciones en el lenguaje de programación C. BOB (Le Cun-Roucairol (1999)) y PICO (An Object-Oriented Framework for Parallel Branch-and-Bound) (Eckstein **el** al., (2000)) están desarrollados en C++ y en ambos casos se proporciona una jerarquía de clases que el usuario ha de extender para resolver su problema particular. En cuanto a los sistemas de programación paralelos orientados a la metodología Divide y Vencerás podemos citar los siguientes: Cilk (2000) basado en el lenguaje C y Satin (2000) basado en Java. Sin embargo, no hemos encontrado ninguna herramienta que permita trabajar de forma integrada con más de una técnica algorítmica, y esta es una de las principales aportaciones de nuestra propuesta.

MaLLBa::DnC y **MaLLBa::BnB** forman parte del proyecto **MaLLBa** (Alba **et al**. (2001)) y Dorta **et al**. (2001), cuyo objetivo final es proporcionar esqueletos exactos, heurísticos e híbridos. En la realización de **MaLLBa** participan las Universidades españolas de La Laguna, Politécnica de Cataluña y Málaga. Existe más información acerca del proyecto disponible en la dirección web: http://www.lsi.upc.es/~mallba.

El resto del artículo se organiza como sigue. En la segunda sección se presenta la arquitectura **MaLLBa** y las interfaces particulares de **MaLLBa::DnC** y **MaLLBa::BnB**. A continuación, en el tercer epígrafe, se describen los esqueletos secuenciales y paralelos. El uso de **MaLLBa::DnC** y **MaLLBa::BnB** de forma integrada se aplica a la resolución del Problema de la Mochila Entera en la sección cuarta. En la sección quinta se presenta una comparativa entre implementaciones con **MaLLBa** de dicho problema e implementaciones ad-hoc en C.

Finalmente, se presentan las conclusiones y los trabajos futuros.

2. DESCRIPCION DE LA ARQUITECTURA MaLLBa

Se denomina *esqueleto algorítmico* a un conjunto de procedimientos que constituyen el armazón con el que desarrollar programas para resolver un problema dado utilizando una técnica particular. Este software presenta declaraciones de clases vacías que el usuario ha de rellenar para adaptarlo a la resolución de un problema concreto.

La librería **MaLLBa** se ha diseñado de forma que la tarea de la persona que adapta el problema real al esqueleto sea lo más simple posible. En un esqueleto **MaLLBa** se distinguen dos partes principales: una parte que implementa el *patrón de resolución* y que es completamente proporcionada por la librería, y una parte que el usuario ha de acabar de completar con las características particulares del *problema a resolver* y que será utilizada por el patrón de resolución.

En el diseño de los esqueletos se ha utilizado la Orientación a Objetos (00). La facilidad de interpretación del esqueleto es una de las ventajas que aporta esta filosofía, además de las de modularidad, reusabilidad y modificabilidad. Existe una relación bastante intuitiva entre las entidades participantes en el patrón de resolución y las clases que ha de implementar el usuario.

La parte proporcionada por el esqueleto, esto es, los patrones de resolución, se implementan mediante clases en las que explícitamente se indica que son definidas por el esqueleto. A estas clases se les da el nombre de clases *proporcionadas* y aparecen en el código con el calificativo *provides*.

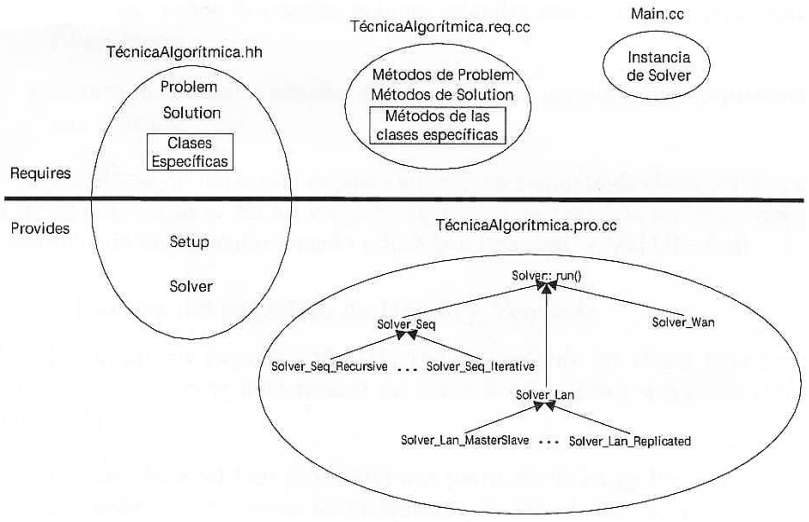

Figura 1.
Arquitectura de los Esqueletos **MaLLBa**.

La parte que ha de completar el usuario con la instanciación de su problema particular se implementa mediante clases marcadas con el calificativo *requires*, se las denominará clases *requeridas*. La adaptación que ha de realizar el usuario consiste en representar mediante las estructuras de datos necesarias su problema e implementar (en función de dicha representación) las funcionalidades requeridas por los métodos de las clases. Estas clases serán invocadas desde el patrón de resolución particular (por que conoce la interfaz para dichas clases) de forma que, cuando la instanciación se ha completado, se obtienen las funcionalidades esperadas aplicadas al problema concreto del usuario.

La Figura 1, presenta la arquitectura del esqueleto. En el fichero con extensión **.hh** se definen las cabeceras de las clases requeridas y proporcionadas. El fichero **.pro.cc** contiene las implementaciones C++ de los patrones de resolución, mientras que el fichero **.req.cc** es el que ha de implementar el usuario.

2.1. Clases Requeridas

La solución de un problema de optimización combinatoria en general 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. Partiendo de esta descripción, se abstrae que en un esqueleto deben representarse mediante clases tanto el *problema* como la *solución*.

Las clases requeridas se utilizan para almacenar los datos básicos del algoritmo. Con ellas se representan el problema, la solución, los estados del espacio de búsqueda y la entrada/salida. Todos los esqueletos **MaLLBa** han de definir las siguientes clases:

- **Problem**: define la interfaz mínima estándar necesaria para representar un problema.
- **Solution**: define la interfaz mínima estándar necesaria para representar una solución.

Cada patrón de resolución requiere además un conjunto de clases propias de la técnica algorítmica. En los epígrafes siguientes se describen las clases que el usuario ha de implementar cuando utilice **MaLLBa::DnC** y **MaLLBa::BnB**.

2.1.1 Interfaz del esqueleto de Divide y Vencerás

Para instanciar un esqueleto **MaLLBa::DnC** además de las clases **Problem** y **Solution**, es necesario implementar las clases **SubProblem** y **Auxiliar** (véase la Figura 2).

- La clase **SubProblem**: representa una partición de un problema en partes disjuntas. Esta clase se ha introducido en aras de una mayor eficiencia en la implementación. La alternativa hubiera sido, representar a los sub-problemas para que fueran del mismo tipo que los problemas. Esta clase debe proporcionar las siguientes funcionalidades:

  - **initSubProblem(pbm)**: Inicializa el subproblema de partida a partir del problema original.
  - **easy (pbm)**: Determina si un subproblema es lo suficientemente pequeño para ser considerado simple.
  - **solve(pbm, sol)**: Proporciona el algoritmo básico de resolución para subproblemas fáciles.
  - **divide (pbm, sps, aux)**: Genera una lista de subproblemas.

- La clase Auxiliar: en algunos casos es necesario el uso de una clase auxiliar. Su necesidad se hace patente cuando como resultado de dividir el problema se obtiene una partición que no constituye un subproblema.

- Además la clase Solution debe proporcionar la siguiente funcionalidad:

  - **combine(pbm, sols, aux)**: dado un conjunto de soluciones parciales de subproblemas, este método ha de combinarlas para crear una nueva solución parcial más general.

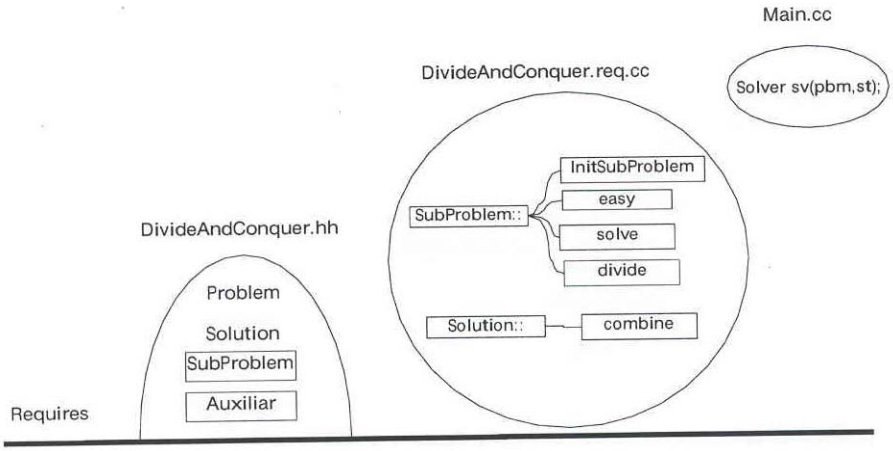

Figura 2. Clases requeridas por el esqueleto de Divide y Vencerás

2.1.2. Interfaz del esqueleto de Ramificación y Acotación

Los algoritmos de Ramificación y Acotación dividen el área de soluciones paso a 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 es necesario seguir explorando esta parte del árbol. Por lo general, el cálculo de cotas se combina con un recorrido en anchura o en profundidad. Para representar este proceso en un esqueleto, se utilizará la clase **Subproblem**. Así pues, el esqueleto **MaLLBa::BnB** a diferencia de **MaLLBa::DnC** sólo requiere que el usuario le suministre dicha clase (véase la Figura 3). Además el usuario debe especificar en la definición de la clase **Problem** si el problema a resolver es de máximo o de mínimo.

- La clase **Subproblem**: representa el área de soluciones no exploradas. Esta clase debe proporcionar las siguientes funcionalidades:

  - **initSubProblem (pbm)**: inicializa el subproblema de partida a partir del problema original.

  - **solve(pbm)**: este método booleano devuelve cierto cuando el área de soluciones factibles a ser explorada es vacía.

  - **branch (pbm, sps, sol)**: ha de proporcionar un algoritmo para dividir el área de soluciones factibles, esto implica, generar a partir del subproblema actual un subconjunto de problemas a ser explorados.

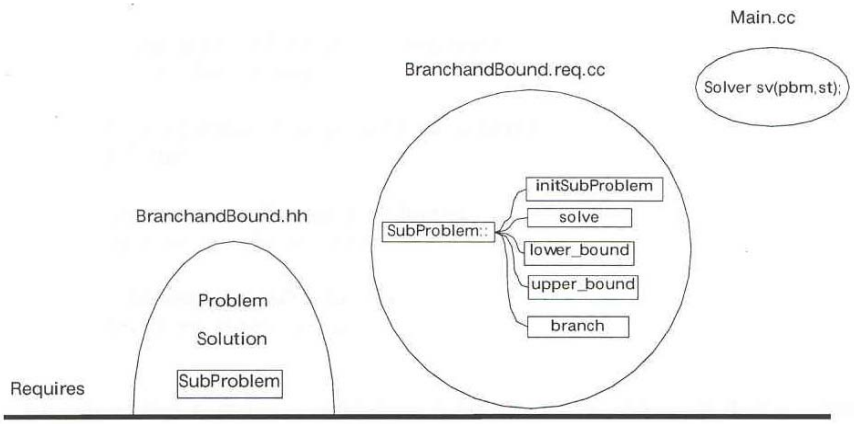

Figura 3. Clases requeridas por el esqueleto de Ramificación y Acotación.

- **lower_bound (pbm, sol)**: este método calcula una cota inferior del mejor valor de la función objetivo que podría ser obtenido para un subproblema dado.

- **upper_bound(pbm,sol)**: esta función calcula una cota superior del mejor valor de la función objetivo que podría ser obtenido para un subproblema dado.

2.2. Clases Proporcionadas

Las clases proporcionadas son las que el usuario ha de instanciar cuando utilice un esqueleto, esto es, sólo ha de utilizarlas. Dichas clases son:

- **Setup**: agrupa los parámetros de configuración del esqueleto. Por ejemplo, en esta clase en el esqueleto de Ramificación y Acotación se especifica si la búsqueda en el área de soluciones será en profundidad, primero el mejor, etc.
- **Solver**: esta clase está comprendida por ella misma y sus hijas **Solver_Seq, Solver_Lan** y **Solver_Wan**. Implementa la estrategia a seguir: Divide y Vencerás, Ramificación y Acotación, etc., y mantiene información actualizada del estado de la exploración durante la ejecución. La ejecución se lleva a cabo mediante una llamada al método **run ()**.

Para elegir un patrón de resolución determinado, el usuario debe instanciar en el método main () la clase correspondiente. El siguiente código muestra un ejemplo de instanciación de un resolutor secuencial, para el esqueleto de Ramificación y Acotación.

```
int main (int argc, char** argv) {
 using skeleton BranchAndBound;

 Problem pbm;
 Solution sol;
 Bound bs;

 //instacia del esqueleto secuencial
 Solver_Seq sv(pbm);

 //se ejecuta el esqueleto secuencial
 sv.run();

 //se obtiene la mejor solución
 bs = sv.bestSolution();

 //se obtiene la solución
 sol = sv.solution();
}
```

En primer lugar, se indica que se va a utilizar un esqueleto de Ramificación y Acotación. A continuación se crea una instancia del esqueleto secuencial **(Solver_Seq)**. Para que se ejecute, es necesario realizar una llamada al método **run( )**. Se invoca al método **bestSolution( )** para obtener la mejor solución. La solución se obtiene utilizando el método **solution( )** que proporciona el esqueleto.

3. IMPLEMENTACION DE LOS PATRONES DE RESOLUCION

En esta sección se describe la forma en la que se han diseñado e implementado los patrones de resolución generales. Se presenta en detalle el esqueleto **MaLLBa::DnC**, es decir, el patrón de resolución para la técnica Divide y Vencerás. La estrategia Divide y Vencerás consiste en descomponer un problema en subproblemas más simples del mismo tipo, resolverlos de forma independiente y una vez obtenidas las soluciones parciales combinarlas para obtener la solución del problema original. El siguiente código refleja la estructura general de un algoritmo basado en la estrategia Divide y Vencerás.

```
procedure DandC(pbm. sol)
local var aux;
begin
    if easy(pbm) then
      sol ve (pbm)
   else
      begin
      divide (pbm, subpbm, aux);
      for i := 1 to numProblem() do
         DandC(subpbm[i] , subsol[i]);
      combine (subsol , aux, sol);
      end
end;
```

Una posible aproximación paralela para el paradigma Divide y Vencerás consiste en hacer que el bucle en el que recursivamente se resuelven cada uno de los subproblemas generados se realice en paralelo. Utilizando notación PRAM (Sande (1998)), el código que se muestra a continuación implementa dicho algoritmo paralelo recursivo.

```
 1 procedure DandC(pbm, sol)
 2 local var aux;
 3 begin
 4   if easy (pbm) then
 5    solve(pbm)
 6   else
 7    begin
 8      Divide(pbm, subpbm, aux)
 9      Parallel i in 1. .numProblem() do
10        DandC(subpbm[i] , subsol[i]);
11      Combine(subsol, aux, sol);
12   end
13 end;
```

La semántica del bucle **parallel** (línea 9), indica lo siguiente:

(a) Asignar un procesador a cada elemento i en {1..*numProblem*()} .

(b) Ejecutar, en paralelo y para cada procesador asignado, la llamada recursiva a la función **DandC**.

La ejecución de la sentencia paralela termina cuando todos los procesadores implicados completan sus computaciones individuales. Hemos de tener en cuenta que en la línea 10 se están realizando llamadas paralelas y recursivas. Ningún lenguaje de programación paralelo para los que existen compiladores en la actualidad ejecuta eficientemente el algoritmo anterior, aunque permiten expresarlo de forma simple, ni siquiera HPF, High Performance Fortran Forum (1993).

Las implementaciones de **MaLLBa::DnC** y **MaLLBa::BnB** se han realizado mediante un esquema *Maestro-Esclavo*. En el patrón maestro-esclavo un procesador *Maestro (Master)* controla la actividad de un grupo de procesadores *Esclavos (Slaves)*, asignando el trabajo que se ha de realizar en paralelo y ocupándose también de las operaciones de entrada/salida. Dado un problema de grandes dimensiones, se divide y se distribuye sobre los esclavos disponibles. Cada Esclavo calcula los resultados parciales de manera independiente y se los envía al Maestro (véase la Figura 4).

Las tareas del Maestro consisten en distribuir trabajo a los componentes Esclavos, arrancar la ejecución de los esclavos y generar el resultado final usando los resultados de los diferentes esclavos. Un esclavo ha de implementar la tarea que le indique el maestro. En el Caso de **MaLLBa::DnC** se ha utilizado el patrón dos veces, uno para la *Fase de División* del problema original en subproblemas y una segunda vez en la *Fase de Combinación* de soluciones parciales para obtener la solución del problema original. El siguiente código muestra las tareas que realizan el Maestro y los Esclavos en la *Fase de División*.

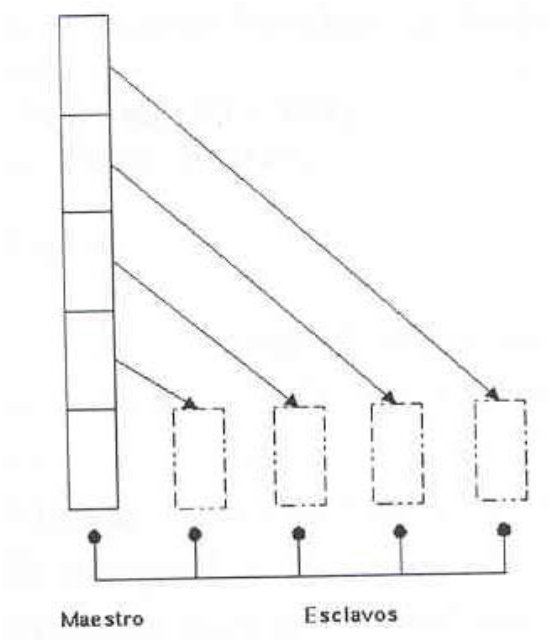

Figura 4. Esquema Maestro-Esclavo Genérico.

```
   Master: //Fase de división
 1     do i = 1,P busy[i] = false;
 2     idle = P;
 3     working = 0;
 4     Quee = {Problem}
 5     while ( (Queue != empty) or (working > 0) ){
 6       while ( (Queue != empty) and (idle> 0)) {
 7         pop(Problem);
 8         send Problem to firstidle;
 9         busy[firstIdle] = true;
10         idle--;
11         working++;
12       }
13       while (there is a message for any) {
14         receive Problems from sender;
15         push Problems in the Queue;
16         idle++;
17         Ilorking--;
18         busy[sender] = false;
19       }
20     }
21     do i=1, P send finish to i

   Slave: //Fase de división
23     do {
24       receive from Master Problem or finish
25       if (!finish) {
26         divide(Problem, P1, P2);
27         send P1, P2 to Master;
28       }
29     } while (!finish)
```

La división del trabajo entre las subtareas y la evaluación conjunta de los resultados de las subtareas están completamente separados del procesamiento individual de cada subtarea. Existe una coordinación entre subtareas que corre a cargo del Maestro. El Maestro cuenta con una estructura de datos, *busy* en la que registra el estado de ocupación de cada uno de los esclavos (línea 1), en principio todos los esclavos están ociosos. Además, también se ha de mantener una cola de problemas, *Queue*, sin resolver. El Maestro, va sacando problemas de esta cola y asignándoselos a cada uno de los esclavos, mientras haya problemas y esclavos libres (líneas 6-12). El Maestro espera recibir de sus esclavos, los nuevas problemas que se han generado, para insertarlos de manera adecuada en la cola (líneas 13-19). Cuando acaba el proceso de división el Maestro se lo indica a todos los esclavos. La tarea a realizar por los esclavos en esta fase, consiste en recibir problemas del maestro (línea 24), dividirlos utilizando la función de división dada y devolverle al maestro los nuevos subproblemas generados (líneas 25-28).

Siguiendo el mismo patrón, la *Fase de Combinación* parte de la estructura de datos generada en la fase anterior y obtiene la solución:

```
  Master: //Fase de combinación
1   do i = l,P busy[i] = false;
2   idle = P;
3   working = 0;
4   Quee = {leaves}
5   while ( (Queue != empty) or (Ilorking > 0) ) {
6     while«Queue != empty) and (idle> 0» {
7       pop(list of results of brothers);
8       send Problem to firstidle;
9       busy[firstIdle] = true;
10      idle--;
11      working++;
12    }
13    while (there is a message for any) {
14      receive Result from sender;
```

```
15        update node for Result;
16        father(node).childrensolved--;
17        if (father(node).children.solved == 0) {
18          push(children(father(node»)
19        idle++;
20        working--;
21        busy [sender] = false;
22      }
23    }
24    do i=l, P send finish to i

    Slave: //Fase de combinación
25    do {
26      receive from Master List of result or finish
27      if (!finish) {
28        result = combine(list of results);
29        send result to Master;
30      }
31    } while (!finish)
```

En este caso se ha de mantener una cola de soluciones parciales, *Queue*. El Maestro, va sacando soluciones de nodos hermanos de esta cola y asignándoselas a cada uno de los esclavos, mientras haya problemas y esclavos libres (líneas 6-12). El Maestro espera recibir de sus esclavos, las soluciones parciales de todos los nodos hermanos, y cuando se han recibido todas, insertar la solución del nodo padre en la cola (líneas 13-22). La tarea a realizar por los esclavos en esta fase, consiste en recibir un conjunto de soluciones del maestro (línea 26), combinarlos utilizando la función de combinación dada y devolverle al maestro la nueva solución generada (líneas 27-30).

La implementación de **MaLLBa::BnB** se realiza siguiendo el mismo esquema que el de la *Fase de División*. Se ha de tener en cuenta que la aproximación paralela que se ha seguido es la más simple. De los experimentos que se han hecho sobre el cluster de PCs en Linux se desprende que necesita ser mejorada, puesto que aún se invierte mucho tiempo en comunicaciones. Sin embargo, la posibilidad de utilizar de forma integrada ambos esqueletos, así como el sólo tener que programar la versión secuencial y obtener la paralela sin ningún esfuerzo adicional ya constituye una ventaja. Siempre se puede partir del código obtenido para mejorar la versión paralela.

4. INTEGRACION DE LOS ESQUELETOS MALLBA::DNC Y MALLBA::BNB

Se ha implementado utilizando los esqueletos Divide y Vencerás y Ramificación y Acotación el algoritmo que describen Martelo y Thoth (1990) para la resolución del problema de la Mochila 0/1 clásico. En dicho algoritmo, es necesario que los elementos estén ordenados de forma ascendente mediante la relación peso/beneficio. Haciendo uso de este requisito se ha implementado el m étodo de ordenación Quicksort, Hoare (1961), con **MaLLBa::DnC** para ordenar los elementos a insertar en la Mochila. A continuación mediante el uso de **MaLLBa::BnB** se ha implementado el algoritmo de Ramificación y Acotación, lo que nos ha permitido utilizar de forma integrada ambos esqueletos. El siguiente código muestra cómo se integran los esqueletos.

```
int main (int argc, char** argv) {
   Knapsack: : Problem pbm;
   Knapsack: :Solution ksol;
   Knapsack: : Bound bs;
   ...
   Knapsack::Problem opbm; //problema ordenado
   MergeSort::Solution ssol;
   ...
   MergeSort::Solver_Seq svs(pbm);
   svs.run();
   ssol = svs.solution();
   ...
   opbm.setN(pbm.N);
   opbm.setCapacity(pbm.C);
   for (Knapsack: : Number i = 0; i < n ; i++)
   { //actualizar los pesos y beneficios }
```

```
    Knapsack: :Solver_Seq svk(opbm, st);
    svk.run();
    bs = svk.bestSolution();
    ksol = svk.solution();
    ...
}
```

5. RESULTADOS COMPUTACIONALES

Las graficas que se muestran en esta sección se han obtenido ejecutando el Problema de la Mochila 0/1 descrito en la sección anterior. En la implementación sólo se calcula el valor de la función objetivo, no se devuelve como solución la lista de los elementos que se ha de insertar en la mochila. Los experimentos se han realizado sobre un Pentium III/600 Mhz con 256 Mb de memoria.

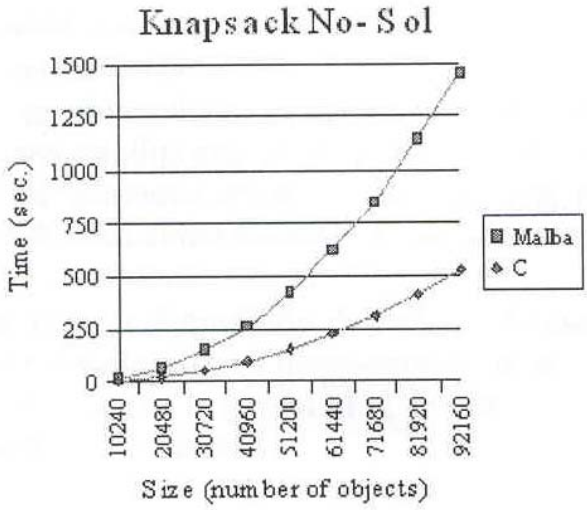

Figura 5. Mochila-MaLLBa vs. Mochila C sin solución.

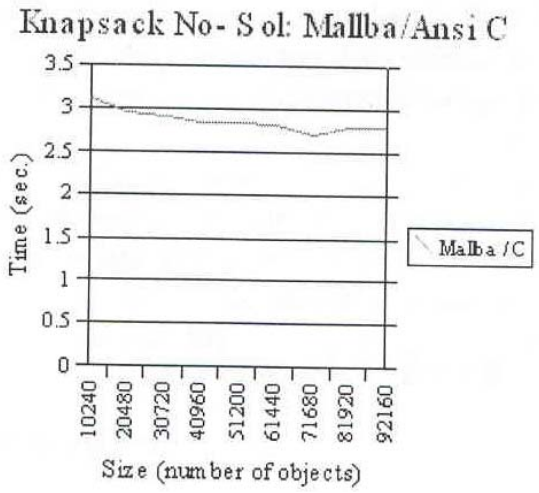

Figura 6. Cociente Mochila-MaLLBa vs. Mochila C sin solución.

Las Figuras 5 y 6 muestran una comparativa entre la implementación **MaLLBa** del problema y una implementación del mismo, sin usar esqueletos en C. Como se puede apreciar la diferencia entre ambas implementaciones no es significativa.

6. CONCLUSIONES Y TRABAJOS FUTUROS

En este artículo se han presentado los esqueletos de la librería **MaLLBa** para la estrategia Divide y Vencerás (**MaLLBa::DnC**) y para el paradigma de Ramificación y Acotación (**MaLLBa::BnB**). La herramienta **MaLLBa** ofrece al usuario libertad para implementar las estructuras de datos que representan su problema, pero proporciona unos patrones de resolución que controlan el flujo de la ejecución.

El principal objetivo de **MaLLBa** es simplificar la tarea de los investigadores o usuarios que han de implementar un algoritmo usando una técnica algorítmica particular. **MaLLBa** proporciona al usuario un valor añadido no sólo en términos de la cantidad de código que ha de escribir, que es mucho menor, sino también en modularidad y claridad conceptual. Si se utiliza **MaLLBa**, o cualquier otro sistema orientado a objetos, el usuario ha de colocar cada pieza de código en la posición correcta.

Otra de las características de **MaLLBa** es la modularidad: una vez que las estructuras de datos básicas y las funcionalidades se han definido e insertado en el código, el esqueleto proporciona una implementación gratuita en paralelo.

MaLLBa hace un uso equilibrado de la programación orientada a objetos. En el sentido de que proporciona las mínimas plantillas necesarias para conseguir una eficiencia computacional buena y al mismo tiempo hacer el proceso de compilación seguro.

Además el sistema pone a disposición del usuario la posibilidad de generar y experimentar con nuevos algoritmos que permitan la integración de técnicas. En este trabajo se ha mostrado la posibilidad de esta integración mediante el ejemplo del Problema de la Mochila.

Se obtienen buenos resultados computacionales para los esqueletos secuenciales. Son ligeramente menos eficientes que los que se obtienen programado una solución en lenguaje C, sin embargo, creemos que cuando mejore el código generado por C++, mejorarán también los tiempos de nuestros algoritmos. Además, se ha propuesto una implementación simple de esqueletos paralelos para cada una de las técnicas. Estos esqueletos pueden servir como punta de partida para desarrollar implementaciones paralelas particulares para cada problema.

Con respecto al paradigma divide y vencerás hemos implementado como ejemplos de uso de la herramienta los siguientes problemas: el Producto de Matrices de Strassen, el producto de entero grandes, la fft y la convex Hull. Para el paradigma de la ramificación y acotación disponemos de una implementación del TSP y se trabaja actualmente en implementaciones de la Mochila Multidimensional.

En la actualidad se está trabajando en la implementación del algoritmo paralelo que mejora la eficiencia del que se ha presentado. Como línea de investigación futura se abordará la implementación de otros esquemas paralelos para los patrones de resolución como: una aproximación replicada (Sande (1998)) y una que utilice memoria compartida (OpenMP Architecture Review Board (1998)).

7. AGRADECIMIENTOS

Este trabajo ha sido parcialmente subvencionado por el proyecto CICYT Español TIC99-0754-C03-01 y por el proyecto del Gobierno de Canarias PI2001/60. El Trabajo de C. León ha sido subvencionado parcialmente por el proyecto Europeo HPRI-CT-1999-00026) en el programa TRACS del EPCC (Edinburgh Parallel Computing Center).

REFERENCIAS

ALBA, E. **et al**. (2001): “**MaLLBa**: Towards a Combinatorial Optimization Library for Geographically Distributed Systems", **Actas de las XII Jornadas de Paralelismo**, Valencia, 105-110.

DORTA, I.; A. ROJAS; C. LEON y P. DORTA (2001): “Utilización de software en la docencia de técnicas algorítmicas", **Actas de las VII Jornadas de Enseñanza Universitaria de la Informática**, Palma de Mallorca, 190-195.

ECKSTEIN, J.; C.A. PHILLIPS and W.E. HART (2000): "PICO: An Object-Oriented Framework for Parallel Branch and Bound", **Rutcor Research Report**, USA.

HIGH PERFORMANCE FORTRAN FORUM (1993): High Performance Fortran Specification. http://www.crpc.rice.edu/HPFF/home.html.

HOARE, C.A.R. (1961): "Algorithm 64: Quicksort", **Communications of the ACM**, 4, 321.

KIELMANN, T.; R. NIEUWPOORT and H. BAL (2000): "Cilk-5.3 Reference Manual", Super-computing Technologies Group. MIT.

_____ (2000): "Satin: Efficient Parallel Divide-and-Conquer in Java", **Euro-Par 2000**, Springer-Verlag, 690-699.

LE CUN, B.; C. ROUCAIROL and THE PNN Team (1999): "BOB: a Unified Platform for Implementing Branch-and-Bound like Algorithms", **Rapport de Recherche** 95/16, Universitè de Versailles, France.

MARTELLO, S. and P. TOTH (1990): "KNAPSACK PROBLEMS Algorithms and Computer Implementations", John Wiley & Sons Ltd.

OPENMP ARCHITECTURE REVIEW BOARD (1998): "OpenMP C and C++ Application Program Interface. Version 1.0.", http: / /www. openmp. org.

SANDE, F. (1998): "El Modelo de Computación Collectiva", Memoria de Tesis Doctoral. Universidad de La Laguna.

SHINANO, Y.; M. HIGAKI and R. HIRABAYASHI (1995): "A Generalized Utility for Parallel Branch and Bound Algorithms", IEEE Computer Society Press, 392-401.

TSCHÖKE, S. and T. POLZER (1995): "Portable Parallel Branch-and-Bound Library", **User Manual Library Version 2.0**, Paderborn.